Simple Distributed Programming
for
Scalable Software-Defined Networks

by

Soheil Hassas Yeganeh

A thesis submitted in conformity with the requirements
for the degree of Doctor of Philosophy
Graduate Department of Computer Science
University of Toronto

Copyright © 2015 by Soheil Hassas Yeganeh
Abstract

Simple Distributed Programming for Scalable Software-Defined Networks

Soheil Hassas Yeganeh
Doctor of Philosophy
Graduate Department of Computer Science
University of Toronto
2015

Software Defined Networking (SDN) aims at simplifying the process of network programming, by decoupling the control and data planes. This is often exemplified by implementing a complicated control logic as a simple control application deployed on a centralized controller. In practice, for scalability and resilience, such simple control applications are implemented as complex distributed applications, that demand significant efforts to develop, measure, and optimize. Such complexities obviate the benefits of SDN.

In this thesis, we present two distributed control platforms, Kandoo and Beehive, that are straightforward to program. Kandoo is a two-layer control plane: (i) the bottom layer is a colony of controllers with only local state, and (ii) the top layer is a logically centralized controller that maintains the network-wide state. Controllers at the bottom layer run only local control applications (i.e., applications that can function using the state of a single switch) near datapath. These controllers handle frequent events and shield the top layer. Using Kandoo, programmers flag their applications as either local or non-local, and the offloading process is seamlessly handled by the platform. Our evaluations show that a network controlled by Kandoo has an order of magnitude lower control channel consumption compared to centralized controllers. We demonstrate that Kandoo can be used to adapt and optimize Transform Control Protocol (TCP) in a large-scale High-Performance Computing (HPC) datacenter with a low network overhead.

Beehive is a more generic proposal built around a programming model that is similar
to centralized controllers, yet enables the platform to automatically infer how applications maintain their state and depend on one another. Using this programming model, the platform automatically generates the distributed version of each control application. With runtime instrumentation, the platform dynamically and seamlessly migrates applications among controllers aiming to optimize the control plane. Beehive also provides feedback to identify design bottlenecks in control applications, helping developers enhance the performance of the control plane. Implementing a distributed controller and a simple routing algorithm using Beehive, we demonstrate that Beehive is as simple to use as centralized controllers and can scale on par with existing distributed controllers. Moreover, we demonstrate that Beehive automatically improves control plane’s latency by optimizing the placement of control applications without any intervention by network operators.
Dedication

To Mom, Dad, and Shabnam.
Acknowledgements

Above all, I would like to sincerely thank my advisor, Yashar Ganjali, for his support, for his dedication, and for being an amazing source of encouragement and inspiration. I consider myself very fortunate to have him as my supervisor, mentor and friend. I am also grateful to the members of my supervisory committee Eyal Delara and David Lie for their continuous guidance, and to my committee members Nick Feamster and Angela Demke Brown for their insightful and invaluable feedbacks on my research.

I would also like to thank Renée Miller for her strong support and continuous research advice. I cannot go without mentioning my friends and co-authors Oktie Hassanzadeh, Kaveh Aasaraai, and Ken Pu who have been extremely helpful during my five years at the University of Toronto. I am also grateful to the SciNet team, especially Chris Loken, Daniel Gruner, and Ching-Hsing Yu, for trusting and supporting me during my experiments.

This thesis is dedicated to my wife and my parents for their unconditional love, constant encouragement and support.
Contents

1 Introduction .................................................................................................................. 1
  1.1 Problem Statement ................................................................................................. 3
  1.2 Objectives ............................................................................................................... 8
  1.3 Overview ............................................................................................................... 9
  1.4 Thesis Statement .................................................................................................... 14
  1.5 Contributions ....................................................................................................... 14
  1.6 Dissertation Structure ......................................................................................... 15

2 Background ................................................................................................................ 17
  2.1 Precursors ............................................................................................................ 17
    2.1.1 ForCES: An initial proposal ............................................................................. 18
    2.1.2 4D: Scalable Routing .................................................................................... 19
    2.1.3 Ethane: Centralized Authentication ............................................................... 21
    2.1.4 OpenFlow: Convergence of Proposals .......................................................... 22
  2.2 Forwarding Abstractions ....................................................................................... 23
  2.3 Control Plane Scalability ..................................................................................... 26
    2.3.1 In-Data-Plane Proposals ................................................................................. 27
    2.3.2 Distributed Control Planes ........................................................................... 28
  2.4 Resilience to Failures ............................................................................................ 31
  2.5 Programming Abstractions .................................................................................... 34
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.6</td>
<td>Correctness and Consistency Tools</td>
<td></td>
<td>35</td>
</tr>
<tr>
<td>2.7</td>
<td>Monitoring and Measurements</td>
<td></td>
<td>40</td>
</tr>
<tr>
<td>2.8</td>
<td>Network Optimization</td>
<td></td>
<td>42</td>
</tr>
<tr>
<td>2.9</td>
<td>Discussion</td>
<td></td>
<td>43</td>
</tr>
<tr>
<td>3</td>
<td>Kandoo</td>
<td></td>
<td>44</td>
</tr>
<tr>
<td>3.1</td>
<td>Introduction</td>
<td></td>
<td>44</td>
</tr>
<tr>
<td>3.2</td>
<td>Design and Implementation</td>
<td></td>
<td>47</td>
</tr>
<tr>
<td>3.3</td>
<td>Evaluation</td>
<td></td>
<td>51</td>
</tr>
<tr>
<td>3.4</td>
<td>Discussion</td>
<td></td>
<td>55</td>
</tr>
<tr>
<td>3.5</td>
<td>Remarks</td>
<td></td>
<td>58</td>
</tr>
<tr>
<td>4</td>
<td>Beehive</td>
<td></td>
<td>59</td>
</tr>
<tr>
<td>4.1</td>
<td>Introduction</td>
<td></td>
<td>59</td>
</tr>
<tr>
<td>4.2</td>
<td>Programming Model</td>
<td></td>
<td>62</td>
</tr>
<tr>
<td>4.3</td>
<td>Control Platform</td>
<td></td>
<td>67</td>
</tr>
<tr>
<td>4.3.1</td>
<td>Primitives</td>
<td></td>
<td>67</td>
</tr>
<tr>
<td>4.3.2</td>
<td>Fault-Tolerance</td>
<td></td>
<td>74</td>
</tr>
<tr>
<td>4.3.3</td>
<td>Live Migration of Bees</td>
<td></td>
<td>79</td>
</tr>
<tr>
<td>4.3.4</td>
<td>Runtime Instrumentation</td>
<td></td>
<td>79</td>
</tr>
<tr>
<td>4.3.5</td>
<td>Automatic Placement</td>
<td></td>
<td>81</td>
</tr>
<tr>
<td>4.3.6</td>
<td>Application Composition</td>
<td></td>
<td>83</td>
</tr>
<tr>
<td>4.4</td>
<td>Applications</td>
<td></td>
<td>84</td>
</tr>
<tr>
<td>4.5</td>
<td>Discussion</td>
<td></td>
<td>87</td>
</tr>
<tr>
<td>4.6</td>
<td>Remarks</td>
<td></td>
<td>90</td>
</tr>
<tr>
<td>5</td>
<td>Use Cases</td>
<td></td>
<td>91</td>
</tr>
<tr>
<td>5.1</td>
<td>OpenTCP</td>
<td></td>
<td>91</td>
</tr>
<tr>
<td>5.1.1</td>
<td>OpenTCP Architecture</td>
<td></td>
<td>93</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
<td>Page</td>
<td></td>
</tr>
<tr>
<td>---------</td>
<td>--------------------------------------------</td>
<td>------</td>
<td></td>
</tr>
<tr>
<td>5.1.2</td>
<td>OpenTCP Operation</td>
<td>94</td>
<td></td>
</tr>
<tr>
<td>5.1.3</td>
<td>Evaluation</td>
<td>97</td>
<td></td>
</tr>
<tr>
<td>5.2</td>
<td>Beehive SDN Controller</td>
<td>107</td>
<td></td>
</tr>
<tr>
<td>5.3</td>
<td>Simple Routing in Beehive</td>
<td>113</td>
<td></td>
</tr>
<tr>
<td>5.4</td>
<td>Remarks</td>
<td>116</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Conclusion and Future Directions</td>
<td>117</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Bibliography</td>
<td>120</td>
<td></td>
</tr>
</tbody>
</table>
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>Sample use-cases implemented in Beehive and their code size</td>
<td>84</td>
</tr>
<tr>
<td>5.1</td>
<td>The approximate OpenTCP overhead for a network with $\sim 4000$ nodes and $\sim 100$ switches for different Oracle refresh cycles</td>
<td>106</td>
</tr>
</tbody>
</table>
List of Figures

1.1 Software-Defined Networking: A generic scheme. .............................. 2
1.2 Using an eventually consistent state, two controllers can steer traffic (e.g., the green and the red flows) onto the same link, which results in an unnecessary oversubscription in the network. ......................................................... 4
1.3 In a distributed routing application that tries to avoid loops in a network, we need coordination and leader election mechanisms to avoid service disruption. Without such mechanisms, an eventually consistent state may lead to faults in the network. .............................................................................. 5
1.4 When a virtual network migrates, the control plane can migrate the respective virtual networking application instance. Without fine-grained control on the external datastore, however, such a migration is ineffective. ............... 7
1.5 Kandoo has two layers of controllers: Local controllers run local applications and handle frequent events, while the root controller runs non-local applications processing infrequent events. ........................................ 11
1.6 Beehive’s control platform is built based on the intuitive primitives of hives, bees, and cells. ................................................................. 13
2.1 ForCES Components. ........................................................................ 19
2.2 4D Four Planes: (i) data, (ii) discovery, (iii) dissemination, and (iv) decision planes 20
2.3 OpenFlow Switch: An OpenFlow 1.1 switch stores multiple flow tables each consisting of multiple flow-entries. ........................................ 23

x
ONIX maintains the network-wide state in a distributed fashion.

ONOS maintains the network-wide state using an external database system.

The five steps when converging upon a link failure.

Kandoo's Two Levels of Controllers. Local controllers handle frequent events, while a logically centralized root controller handles rare events.

Toy example for Kandoo's design: In this example, two hosts are connected using a simple line topology. Each switch is controlled by one local Kandoo controller. The root controller controls the local controllers. In this example, we have two control applications: $\text{App}_{\text{detect}}$ is a local control application, but $\text{App}_{\text{reroute}}$ is non-local.

Kandoo's high level architecture.

Kandoo in a virtualized environment. For software switches, we can leverage the same end-host for local controllers, and, for physical switches, we use separate processing resources.

Experiment Setup. We use a simple tree topology. Each switch is controlled by one local Kandoo controller. The root Kandoo controller manages the local controllers.

Control Plane Load for the Elephant Flow Detection Scenario. The load is based on the number of elephant flows in the network.

Control Plane Load for the Elephant Flow Detection Scenario. The load is based on the number of nodes in the network.

A simple learning switch application in Beehive.

The LearningSwitch application (Figure 4.1) learns the MAC address to port mapping on each switch and stores them in the Switches dictionary.
4.3 This simple elephant flow routing application stores flow periodically collects stats, and accordingly reroutes traffic. The schematic view of this algorithm is also portrayed in Figure 4.4.

4.4 The schematic view of the simple elephant flow routing application presented in Figure 4.3.

4.5 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.

4.6 Automatically generated Map functions based on the code in Figure 4.4.

4.7 Life of a message in Beehive.

4.8 Throughput of a queen bee for creating new bees, using different batch sizes. Here, we have used clusters of 1, 3, 5, 7, 9 and 11 hives. For larger batch sizes, the throughput of a queen bee is almost the same for all clusters.

4.9 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.

4.10 The latency and throughput of the sample key-value store application: (a) Throughput using replication factors of 1 to 5, and batch sizes of 128 to 8k. (b) The PDF of the end-to-end latency for GET and PUT requests in a cluster of 20 hives and 80 clients.

4.11 Beehive failover characteristics using different Raft election timeouts. In this experiment, we use the key-value store sample application with a replication factor of 3 on a 3-node cluster. Smaller election timeouts result in faster failover, but they must be chosen to avoid spurious timeouts based on the network characteristics.
4.12 The PDF of a bee's throughput in the sample key-value application, with and without runtime instrumentation. ................................................................. 81
4.13 Inter-hive traffic matrix of the elephant flow routing application (a) with an ideal placement, and (b) when the placement is automatically optimized. (c) Bandwidth usage of the control plane in (b). ................................................................. 82
4.14 A centralized application maps all messages to the same cell. This results in processing all messages in the same bee. ................................................................. 85
4.15 A local application maps all messages to a cell local to the hive (e.g., the switch's ID). ................................................................. 86
4.16 To manage a graph of network objects, one can easily map all messages to the node's ID. This way nodes are spread among hives on the platform. ........... 86
4.17 For virtual networks, messages of the same virtual network are naturally processed in the same bee. ................................................................. 87

5.1 Using Kandoo to implement OpenTCP: The Oracle, SDN-enabled switches, and Congestion Control Agents are the three components of OpenTCP. ........... 93
5.2 The Oracle collects network state, and statistics through the SDN controller and switches. Combining those with the CCP defined by the operator, the Oracle generates CUEs which are sent to CCAs sitting on end-hosts. CCAs will adapt TCP based on the CUEs. ................................................................. 95
5.3 The topology of SciNet, the datacenter network used for deploying and evaluating OpenTCP. ................................................................. 98
5.4 SciNet traffic characterization. The x-axis is in log scale. (a) CDF of flow sizes excluding TCP/IP headers. (b) CDF of flow completion times (right). ........... 99
5.5 The locality pattern as seen in matrix of log(number of bytes) exchanged between server pairs. ................................................................. 100
5.6 CDF of flow sizes in different benchmarks used in experiments. The x-axis is in log scale. ................................................................. 101
5.7 Comparing the CDF of FCT (a) and drop rate (b) of OpenTCP\textsubscript{1}, OpenTCP\textsubscript{2}, and regular TCP while running the “all-to-all” IMB benchmark. 

5.8 Congestion window distribution for OpenTCP\textsubscript{1} and unmodified TCP.

5.9 Comparing the CDF of FCT (a) and drop rate (b) of OpenTCP\textsubscript{1}, OpenTCP\textsubscript{2}, and regular TCP while running the “Flash” benchmark.

5.10 Beehive’s SDN controller suite.

5.11 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.

5.12 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.

5.13 Throughput of Beehive’s learning switch application and ONOS’s forwarding application in different cluster sizes. The number of nodes (i.e., the x-axis) in this graph represents the nodes on which we launch cbench. For Beehive, we always use a 5-node cluster. For ONOS, the cluster size is equal to the replication factor since all nodes in the cluster will participate in running the application.

5.14 A distributed shortest path algorithm.

5.15 Convergence time of the flat routing algorithm in a k-ary fat-tree topology.
Chapter 1

Introduction

Computer networks have traditionally been designed based on a plethora of protocols implemented by autonomous networking elements. In such a setting, controlling the network is difficult and network programmability is limited since these network protocols are focused on specific functionalities implemented using a specific mechanism. As networks and their applications evolve, the need for fine-grained control and network programming becomes more pronounced. Without programmability, implementing new ideas (e.g., network virtualization and isolation) and adapting new changes (e.g., new routing mechanisms) are quite challenging and tedious [88].

Software-Defined Networking (SDN) emerged with the goal of lowering the barrier for innovation in computer networks. There are numerous SDN proposals, ranging from ForCES [128] to 4D [49] to OpenFlow [88]. Although, these proposals are different in scope, moving the control functions out of data-path elements is the common denominator among them. These proposals essentially decouple control and data planes, and provide a standard API using which the control plane can configure the data plane. In SDN, as shown in Figure 1.1, data-path elements (such as routers and switches) are high-performance entities implementing a set of forwarding primitives (mostly matched-action forwarding rules). The control plane, on the other hand, is a cohort of programs (i.e., the control applications) that collectively control
the data-path elements using the exposed forwarding primitives. These control applications are developed in general-purpose programming languages and are hosted on top of a middleware called the network controller. The network between the data plane and the control plane is called the control channel, which can be either in-band (i.e., shared with the actual network flows) or out-of-band (i.e., a dedicated and isolated network).

Advantages of SDN. The separation of control from data path elements brings about numerous advantages such as high flexibility, programmability, and the possibility of realizing a centralized network view; with the promise of simplifying network programming. After all, it is significantly easier to implement complex policies with arbitrarily large scopes on a decoupled, centralized controller. For example, consider a simple shortest-path routing application. In traditional networks, one needs to implement the shortest path algorithm in a highly distributed, heterogeneous environment and should define boilerplates including the serialization protocol, the advertisement primitives and the coordination mechanisms, in addition to the core of the shortest path routing algorithm. In contrast, on a centralized controller, such a routing application is implemented as finding shortest paths using the centralized network-wide view.

Despite all its simplicity, centralization is not feasible in large scale networks due to concerns
about scalability and resilience [60]. In such environments, network programmers need to implement distributed control applications. As we explain next, implementing distributed control applications is quite challenging without proper platform-level support.

1.1 Problem Statement

In large-scale networks, to overcome scalability and performance limitations, control applications are usually implemented on distributed control platforms. Implementing a distributed control application is clearly more difficult than implementing its centralized version. Some of these complexities (e.g., synchronization, and coordination) are common among all distributed applications, whereas some are stemmed from the application’s design (e.g., parallelization).

Existing distributed control platforms either (i) expose the complexities of distributed programming to control applications (e.g., ONIX [75]), or (ii) delegate distributed state management to external distributed systems (e.g., ONOS [17] and OpenDayLight [4]). In such environments, designing a distributed control application comprises significant efforts beyond the design and implementation of the application’s core: one needs to address timing, consistency, synchronization, coordination, and other common complications of distributed programming. More importantly, network programmers have to develop tools and techniques to measure the performance of their control applications in order to identify design bottlenecks, and need to manually optimize the control plane to come up with a scalable design. This directly obviates the promised simplicity of SDN.

Issues of Exposing Complexities. The most notable distributed control platform is ONIX [75], which is built using an eventually consistent representation of the network graph (i.e., Network Information Base). Providing eventual consistency greatly simplifies the control plane’s design, as there is no need to deal with timing complications that can happen as a result of delays and failures in the network. Yet, such a design simplification, admittedly, pushes a lot of complexities into control applications and hinders the promised simplicity of SDN [75, 86].

Chapter 1. Introduction
Installing Flows in Parallel

Figure 1.2: Using an eventually consistent state, two controllers can steer traffic (e.g., the green and the red flows) onto the same link, which results in an unnecessary oversubscription in the network.

This becomes more pronounced when control applications need to implement coordination and synchronization mechanisms that are mostly overlooked in existing control platforms.

For instance, consider a routing application designed to prevent link oversubscription by steering flows along paths with sufficient vacant capacity, shown in Figure 1.2. This application is simple to implement on a centralized controller, but it can easily oversubscribe a link when deployed in a distributed controller using an eventually consistent state. For instance, due to latency in updating the state and lack of proper synchronization, two distinct instances of the application can route two elephant flows (e.g., the green and the red flows) to the same link, leading to an unnecessary oversubscription. To overcome this issue, the network programmer needs to implement/integrate complex coordination mechanisms, which requires more efforts than implementing the core of the routing functionality.

In addition to such suboptimalities, eventual consistency may result in invalid control. For example, consider a naive minimum spanning tree (MST) routing application, depicted in Figure 1.3, that tries to prevent loops by keeping one outgoing port up per neighbor switch. When two neighbor switches are controlled by two physically separate controllers, each
controller immediately blocks a few ports, but will eventually notify the neighbor controller. Without a proper coordination mechanism, this can result in obscure connectivity problems even in extremely simple situations. This problem can be overcome by employing a leader election mechanism, which inevitably complicates the control application’s logic.

Moreover, in scenarios where control applications require a subset of network events (e.g., network events related to a specific tenant in a multi-tenant datacenter), it would be futile to keep a copy of the complete network state in each controller instance. This is especially important when network policies are naturally enforced using the state of a small partition. For instance, to enforce isolation and to provide network statistics and billing for each tenant, the control application needs the network events of that specific tenant in a consistent way, not a holistic view of the entire network. This suboptimal behavior can potentially become a scalability bottleneck by unavailingly consuming resources. To prevent unnecessary duplications of state, different instances of an application must explicitly specify what parts of the network state they need to access. This requires proper partitioning that, if not facilitated by the platform, can easily dwarf the main part of the control logic.

**Issues of Delegating to External Datastores.** An alternative design for a distributed control plane is to delegate state management to an external distributed datastore. Although using
an external datastore can simplify distributed state management, it has several important drawbacks. Delegating SDN state management to an external system is shown to have considerable overheads [76]. Further, such an over-delegation of the control plane functionality does not resolve the problem of distributed programming, such as concurrent updates and consistency [123]. More importantly, it is the external datastore that dictates the architecture of the control plane: if the datastore forms a ring, we cannot form an effective hierarchy in the control plane.

ONOS [17] is a distributed control plane that delegates all the core functions of a distributed control application to distributed state storage systems (e.g., ZooKeeper [64], Casandra [77], or RAMCloud [99]). This simplifies ONOS’s design but inherits all the complications of distributed programming with one layer of indirection. In such a setting, network programmers need to design and optimize their control application as well as the external distributed system since the logic is implemented in one environment and the state is stored in another system. As a result, using an external system does not allow the network programmer to design the functions and the respective state in accordance. At least, one needs to design and measure two separate systems with different characteristics and fundamentally different designs.

For instance, suppose we have a network managed by multiple controllers that store their state in a distributed datastore. Despite sharing a common datastore (even if the datastore is strongly consistent), the control applications on these controllers still need to coordinate to correctly manage the network. For example, if they see packets of the same flow in parallel (say due to a multi-path route), they need to coordinate with one another to avoid black holes and routing loops. In general, they need to synchronize on any decision that might lead to conflicts. Such synchronization problems require orchestration facilities such as Corybantic [90], Athens [15] and STN [23]. It is important to note that providing such facilities in the control plane would eliminate the need for an external state management system.

Relying on an external datastore has strong performance implications. Consider a simple learning switch application, as another example, that stores the state of each switch and forwards
Figure 1.4: When a virtual network migrates, the control plane can migrate the respective virtual networking application instance. Without fine-grained control on the external datastore, however, such a migration is ineffective.

packets accordingly. To store its state, this application will have to save the state of all switches on the external distributed database that ONOS supports. This learning switch application should be high-throughput, and the overheads of using an external system can be prohibitive for this simple control application. This makes network programming unnecessarily complicated as one needs to not only make their learning switch application scale, but also to come up with a scalable design for the external state management system.

Using an external datastore also limits the control plane’s ability to self-optimize. For example, consider an instance of a virtual networking application that controls the ports and the flows of a virtual network in a virtualized environment. In such a setting, virtual machines have mobility and can migrate. Thus, it is important that the control plane places the virtual networking application instance close to the virtual network upon migration. As shown in Figure 1.4, even if we implement an optimized placement mechanism for the control application, the external state storage system may not move the state according to the application placement. That necessitates changes in the external system which is quite challenging.

Motivation. These simple examples demonstrate that, without a proper control plane abstraction, control applications have to deal with practical complications of distributed systems,
which can be a huge burden if not impossible to deal with. Pushing such complications (e.g., synchronization, coordination, and placement) into control applications clearly obviates many benefits of SDN. A viable way to provide an easy-to-use network programming paradigm is to hide these generic complications and boilerplates from control applications behind a suitable abstraction. Such an abstraction must be simple and, more importantly, powerful enough to express essential control functions.

1.2 Objectives

Simplicity and scalability are not mutually exclusive. Map-Reduce [35] and Spark [132] are the infamous programming models that hide the complexities of batch data processing. Storm [84] and D-Stream [133] are successful examples that hide the boilerplates of distributed stream processing. In essence, such systems find the sweet spot between what they hide from and what they expose to the distributed applications. The key to finding a proper programming model is an easy-to-use abstraction that is generic enough to cover all important use cases in its target domain. We believe similar approaches are viable for SDN.

The main objective of this thesis is a new distributed programming platform that can simplify network programming in SDN. By programming platform, we mean a composition of (i) a programming model that provides the APIs to develop control applications and (ii) a distributed runtime that automatically deploys, runs, and optimizes the applications written in the proposed programming model.

Such a platform must have the following characteristics:

1. **Simple**: We aim at creating a programming paradigm that is as simple to program as existing centralized controllers. That is, the efforts required to develop a control application (e.g., line-of-code and cyclomatic complexity [85]) using the distributed programming model should be almost the same as implementing it on a centralized controller. More importantly, the efforts required to instrument and measure the
distributed control application should be comparable to existing centralized controllers.

2. **Generic**: The programming model should be sufficiently expressive for essential control functionalities. For example, one should be able to implement forwarding, routing, and isolation using this programming model. Moreover, one should be able to implement and emulate existing control platforms using the proposed programming model. This ensures that the programming model is generic enough to accommodate existing proposals and legacy control applications.

3. **Scalable**: We need a programming model that has no inherent scalability bottleneck. It is important to note that the scalability of any programming model depends heavily on the applications. For instance, one can create an inefficient and unscalable Map-Reduce application. Such scalability bottlenecks that reside in applications cannot be automatically resolved. Having said that, our objective is to propose a programming paradigm that scales well given a control application design that is not inherently centralized. In other words, our proposal by itself should not be scalability bottleneck.

4. **Autonomous**: There are many commonalities among distributed control applications. One of the most important example of such commonalities is optimizing the placement of control applications. Our goal is to develop an environment in which such commonalities are automated with no programmer intervention. Having such an autonomous system simplifies network programming and operation. More importantly, it results in an elastic control plane that can automatically adapt to the network characteristics and the workload.

1.3 **Overview**

We first present a comprehensive overview of existing SDN proposals and research. To gain a better perspective on the topic of this dissertation, we start with earlier proposals from
the academia and the industry that have emerged before SDN. Then, we present high-level architectural proposals, as well as latest research on SDN control plane, SDN data plane, SDN abstractions, and SDN tools.

In our background research, we deconstruct what are the main scalability concerns in SDN and discuss how they compare with traditional networks. We argue that some of these scalability concerns are not unique to SDN and coexist in traditional networks [60]. Moreover, we explore scalability and performance bottlenecks (such as scarce control channels and resilience to failure) in different settings for different control applications.

These bottlenecks reside in the data plane, in the control plane, or both; hence, there are numerous places (e.g., the forwarding fabric, the controller, and the control applications) to tackle them. We enumerate such different alternatives and trade-offs in the design space to address the scalability and performance bottlenecks. Furthermore, we compare these alternatives and analyze the cons and pros of each approach. This analysis forms the foundation of our distributed control plane proposals: Kandoo and Beehive. Both of our proposals scale well and are simple to program.

**Kandoo.** One of the main scalability bottlenecks of the control plane in SDN is the scarce control channels that can be easily overwhelmed by frequent events. Even when one proactively installs flow-entries, collecting flow statistics to implement adaptive networking algorithms can overwhelm the control channels and consequently the control plane. Kandoo [57] is a distributed programming model that tries to address this scalability bottleneck in environments where computing resources are available close to the data path elements (e.g., datacenters and campus networks). In contrast to previous proposals such as DIFANE [131] and DevoFlow [34], Kandoo limits the overhead of frequent events on the control plane without any modifications in the data plane and without hindering the visibility of the control applications.

As shown in Figure 1.5, Kandoo has two layers of controllers: (i) the bottom layer is a group of controllers with no interconnection and no knowledge of the network-wide state, and (ii) the top layer is a logically centralized controller that maintains the network-wide state. Controllers
Figure 1.5: Kandoo has two layers of controllers: Local controllers run local applications and handle frequent events, while the root controller runs non-local applications processing infrequent events.

at the bottom layer run only local control applications (i.e., applications that can function using the state of a single switch) close to the datapath elements. These controllers handle most of the frequent events and effectively shield the top layer.

Local control applications have implicit parallelism and can be easily replicated since they access only their local state. As such, Kandoo’s design enables network operators to replicate local controllers on demand without any concerns about distributed state replication. These highly replicated local controllers process frequent events and relieve the load on the top layer. Our evaluations show that a network controlled by Kandoo has an order of magnitude lower control channel consumption compared to centralized OpenFlow controllers. Moreover, the only extra information we need from network programmers is a flag that states whether a control application is local or non-local. As a result, control applications written in Kandoo are as simple as their centralized versions.

**Beehive.** Beehive [59], the second programming paradigm presented in this dissertation, is more generic and more powerful than Kandoo. Beehive consists of a programming model
and a distributed control plane. Using Beehive's programming model, network programmers implement control applications as asynchronous message handlers storing data in their local dictionaries \((i.e., \text{associative arrays or maps})\). Based on this programming model, Beehive's distributed control platform seamlessly partitions the dictionaries of each application, and assigns each partition exclusively to one leader thread. Beehive's partitioning mechanism guarantees that each message can be processed solely using the entries in one partition. That is, for each message, there is one and only one partition (and accordingly there is one thread managing that partition) that is used to serve the message. To that end, Beehive automatically infers the \textit{mapping} between each message to the keys required to process that message, and sends the message to the leader thread which exclusively owns those keys \((i.e., \text{the thread that owns the partition containing those keys})\). It is important to note that all these functionalities \((e.g., \text{inferring partitions, parallelization through threads, and the platform-wide coordination})\) are \textit{internal} to our control platform and are \textit{not exposed} to network programmers.

Beehive provides seamless fault-tolerance, runtime instrumentation, and optimized placement of control applications without developer intervention. To provide such autonomous functionalities and to remain generic, Beehive is not built on top of existing external datastores \((e.g., \text{memcached} [43] \text{ or } \text{RAMCloud} [99])\) since we need fine-grained control over all aspects of the control plane, including but not limited to the placement of state and the replication mechanism.

As depicted in Figure 1.6, Beehive's control platform is internally built based on the intuitive primitives of \textit{hives}, \textit{cells}, and \textit{bees}. A hive is basically an instance of the controller running on a physical machine, a container, or a virtual machine. A cell is an entry in the dictionary of a control application. A bee, on the other hand, is a light-weight thread of execution that exclusively manages its own cells and runs the message handlers of a control application. The platform guarantees that each message is processed by the bee that exclusively owns the cells required to process that message. The interplay of these primitives results in concurrent and consistent state access in a distributed fashion.
Figure 1.6: Beehive’s control platform is built based on the intuitive primitives of hives, bees, and cells.

The Beehive control platform is fault-tolerant. Each bee forms a colony (i.e., a Raft [96] quorum) with follower bees on other hives to consistently replicate its cells. Upon a failure, the colony reelects its leader and continues to process messages in a consistent manner. Another important capability of Beehive is runtime instrumentation of control applications. This enables Beehive to automatically live migrate the bees among hives aiming to optimize the control plane. The runtime instrumentation, as a feedback provided to network programmers, can also be used to identify bottlenecks in control applications, and can assist developers in enhancing their designs.

It is important to note that Beehive hides many commonalities of distributed programming (such as coordination, locking, concurrency, consistency, optimization and fault-tolerance), while providing sufficient knobs for network programmers to implement a wide range of application designs. In this dissertation, we demonstrate that Beehive is generic enough for implementing existing distributed control platforms (including Kandoo) as well as important control applications (such as routing and traffic engineering) with ease and at scale.

Use Cases. We present how important use cases can be implemented using Kandoo and Beehive. We first present OpenTCP [46], our proposal to optimize and adapt TCP using the centralized network-wide view available in SDN, and how one can use Kandoo to implement OpenTCP at scale. Furthermore, we present a distributed SDN controller and a simple shortest-path routing
algorithm implemented using Beehive to showcase the expressiveness of our proposal. We demonstrate that Beehive's automatic optimizer can be advantageous for SDN applications in real-world settings. Evaluating these use-cases, we demonstrate that Beehive and Kandoo are simple to program and can scale.

1.4 Thesis Statement

We present expressive and extensible distributed programming paradigms for SDN that are easy to use, straightforward to instrument, and autonomous in optimizing the control plane.

1.5 Contributions

This dissertation makes the following contributions:

1. We present Kandoo [57], a two-layer hierarchy of controllers that is as simple to program as centralized controllers, yet scales considerably better by offloading local control applications onto local computing resources close to data-path elements to process frequent events. The key differentiation of Kandoo is that, unlike previous proposals such as DIFANE [131] and DevoFlow [34], it does not require modifications in the forwarding hardware nor to the southbound protocols (e.g., OpenFlow [88]).

2. As a use case of Kandoo, we present OpenTCP [46], which can adapt and optimize TCP by exploiting SDN’s network-wide view. By deploying OpenTCP in SciNet (the largest academic HPC datacenter in Canada), we demonstrate that OpenTCP can achieve considerable improvements in flow completion time with simple optimization policies, at scale. Such a deployment is quite challenging using centralized controllers.

3. We propose Beehive which provides a programming model (i.e., a set of APIs) that is intentionally similar to centralized controllers to preserve simplicity. Applications developed using this programming model are seamlessly transformed into distributed
applications by our distributed control platform. Using Beehive, we have implemented our own SDN controller as well as existing distributed controllers such as Kandoo. To demonstrate the generality of our proposal, we have implemented several non-SDN use cases on top of Beehive including a key-value store, a message queue, and graph processing algorithms. Using these SDN and non-SDN use cases, we demonstrate that our proposal is simple to program, scales considerably well, can automatically tolerate faults, provides low-overhead runtime instrumentation, and can optimize itself upon changes in the workload.

To preserve simplicity, Beehive applications are written in a generic programming language with an API similar to centralized controllers. To that end, our proposal does not introduce a new domain specific language (DSL), such as BOOM’s disorderly programming [13] and Fabric’s DSL for secure storage and computation [80]. Moreover, unlike existing distributed SDN controllers, such as ONIX [75] and ONOS [17], Beehive hides the boilerplates of distributed programming and does not expose them to programmers. Beehive is generic: it does not impose a specific distribution model (e.g., a hierarchy or a ring), and is not tied to a specific control application (e.g., forwarding or routing) or a specific southbound protocol (e.g., OpenFlow). As such, it can accommodate a variety of designs for control applications, including hierarchies, rings and pub-sub. Similarly, its self-optimization mechanism is generic and applies to all types of control applications, in contrast to proposals such as ElasticCon [37] which has the specific goal of optimizing the workload of switches among several controllers.

1.6 Dissertation Structure

This dissertation is organized as follows:

- In Chapter 2, we present an overview of the state-of-the-art SDN research from both the academia and the industry. We present SDN precursors (which includes the proposals
that led to what we know as SDN today), SDN proposals in the design space, SDN tools, and novel applications of SDN. Moreover, we deconstruct challenges in SDN, and discuss different alternatives to address these challenges.

- In Chapter 3, we present Kandoo. Kandoo is one of the two programming abstraction presented in this dissertation. We explain Kandoo’s design philosophy in detail and compare it with other alternative proposals in the same domain. Using a sample elephant flow detection application, we demonstrate that Kandoo can result in a considerably more scalable control plane.

- In Chapter 4, we present our second programming paradigm, Beehive. We present the details of Beehive’s design, and demonstrate its programming model. Moreover, we explain the internals of the Beehive distributed control plane and how it acts as a scalable runtime for the proposed programming model. We also explain fault-tolerance, runtime instrumentation, and automatic placement optimization in Beehive.

- We present important use cases of our proposals in Chapter 5. We first present OpenTCP, and explain how it can be implemented using Kandoo (and consequently Beehive). Then, we present and evaluate a real-world distributed controller and a distributed shortest path algorithm implemented using Beehive.

- We summarize and conclude this dissertation in Chapter 6, and enumerate all the potential directions to follow up this research in the future.
Chapter 2

Background

SDN is a broad area and has a vast body of research ranging from low-level hardware designs to high-level abstractions to real-world network applications. Covering all SDN research is beyond the scope of this dissertation. Thus, in this chapter, we limit the scope of our literature review to fundamental SDN research and the state-of-the-art proposals related to the topic of this dissertation. We start with precursors and important fundamental SDN abstractions. Then, we enumerate and analyze proposals that address scalability, resilience, correctness and simplicity concerns in SDN. Moreover, we present important measurement techniques and optimization applications using SDN.

2.1 Precursors

Traditional networks are built as a fragile assemblage of networking protocols (*i.e.*, slightly less than 7000 RFCs), which are instrumental for building the Internet, but inessential for orchestration within a single networking domain. These protocols often solve very straightforward problems but are unnecessarily complicated since they are designed to operate in highly heterogeneous infrastructures with no orchestration mechanism. For that reason, to enforce a simple network policy (*e.g.*, a simple isolation rule) in traditional networks, we either have to use a set of complicated signaling protocols supported by all networking elements, or
build an overlay network to steer traffic through middleboxes.

There are various architectural proposals to address this inherent problem of traditional network architectures, and SDN is the result of their convergence. These proposals, although different in scope and mechanisms, agree upon the separation of control functions from the data plane: data-path elements support addressing protocols and respective forwarding primitives, and all other problems are tackled in the control plane which communicates with data-path elements using a standard API\(^1\).

In this chapter, we present an overview of proposals which profoundly influence SDN’s design principles.

### 2.1.1 ForCES: An initial proposal

In most operating systems, networking applications are designed as separate fast-path and slow-path components. The former provides line-rate packet processing functionality, needs to be extremely efficient, and is usually implemented within the Kernel, whereas the second component is a user-space software that provides higher level functionality, and manages its kernel-level counterparts via signaling mechanisms (\(e.g.,\) NetLink sockets).

Routers and switches are internally designed the same way. Their fast path (\(i.e.,\) the forwarding plane) is usually implemented in hardware (using either ASIC, programmable hardware or network processors) separated from the slow path (\(i.e.,\) the controller) which is usually implemented in software running on a general purpose processor. Observing this analogy\(^2\), Forwarding and Control Element Separation (ForCES) [128] proposes to use a standard signaling mechanism in between control and forwarding elements which effectively decouples control plane from forwarding elements.

As portrayed in Figure 2.1, ForCES have two types of networking entities: \((i)\) Forwarding Elements (FE) that forward packets, and \((ii)\) Control Elements (CE) that control FEs. A

---

\(^1\)It is important to note that traditional network equipments are implemented in a similar fashion except for the fact that their control plane is highly coupled with the data plane, and thus is inflexible.

\(^2\)ForCES is proposed by designers of Linux NetLink sockets.
(a) Forwarding Elements (FE) are configured as a set of interconnected Logical Function Blocks (LFB).

(b) Control Elements (CE) can associate with any Forwarding Element (FE), configure and query its Logical Function Blocks.

Figure 2.1: ForCES Components.

FE is configured as a topology of Logical Function Blocks (LFB) which define high level forwarding functions (Figure 2.1a). As shown in Figure 2.1b, each CE can associate with any FE and configure, query, and subscribe to its LFBs. LFBs are designed to be fine-grained, and thus almost all sorts of forwarding functionality can be implemented in ForCES’s FEs. This advantage, however, is also the main disadvantage of ForCES. This makes the model too generic and underspecified, and for that reason ForCES is not deployed in practice.

2.1.2 4D: Scalable Routing

RCP. Routing Control Platform (RCP) [22] aims at addressing scalability issues in traditional routing protocols, specifically iBGP, by delegating routing decisions to a control platform. To centralize routing decisions, RCP gathers internal network state (by monitoring link-state advertisements from IGP protocols) as well as candidate routes to external networks (by establishing iBGP connection to network routers). Utilizing this centralized information, RCP selects and assigns routes for each router, essentially preventing the protocol oscillation and persistent forwarding loops. To be resilient to failures, RCP realizes multiple replicas and guarantees that different replicas will have identical routing decisions in steady state.

4D. Extending RCP, 4D [49] proposes a clean-slate approach which extends RCP beyond routing protocols into all kinds of decision logics in networks. 4D has three major design objectives: First, network requirements should be expressed as a set of network-level objectives that should
be independent of data plane elements, and control decisions should satisfy and optimize those objectives. Second, decisions are made based on a timely, accurate network-wide view of the data plane’s state. Consequently, routers and switches should provide the information required to build such a view. Third, instead of implementing decision logic inside network protocols, the data plane elements should only use the output of decision logic to operate. This ensures direct control over the data plane elements. In other words, we can directly enforce decisions and make sure they are obeyed by switches and routers.

To achieve its design objectives, 4D (as implied by its name) partitions network functionality into four planes (Figure 2.2): (i) data, (ii) discovery, (iii) dissemination, and (iv) decision planes. The decision plane makes all control decisions, such as load balancing, access control, and routing. The data plane provides packet processing functions such as forwarding, filtering, scheduling, measurements, and handles packets according to the state enforced by the decision plane. Discovering physical elements in the data plane is the responsibility of the discovery plane. This plane merely discovers physical networking elements and represents them as logical entities encapsulating their capabilities and adjacency information. The dissemination plane, on the other hand, is a passive communication medium that relays decisions to the data plane, and provides the discovered state to the decision plane.

Figure 2.2: 4D Four Planes: (i) data, (ii) discovery, (iii) dissemination, and (iv) decision planes
Tesseract. 4D is a generic architectural template and does not enforce any implementation details for the planes, nor any protocols and APIs for communicating between them. Tesseract [127] is an experimental implementation of 4D that embeds the four planes into two components: (i) Decision Element (DE) is the centralized entity that makes networking decisions and has direct control over forwarding elements. DE is basically an implementation of 4D’s decision plane that uses the dissemination service to communicate to switches, and is replicated using a simple master-slave model to ensure resilience. (ii) Switch is the component residing in the forwarding elements that provides packet processing, discovery functions, and the dissemination service endpoint. In essence, the dissemination plane is partly deployed on switches and partly on DE to maintain a robust communication channel. Due to the lack of standard protocols in 4D, Tesseract uses its own node configuration API to configure forwarding elements. Evaluated in Emulab, Tesseract is shown to be resilient to failures and sufficiently scalable to handle a thousand nodes.

2.1.3 Ethane: Centralized Authentication

Traditional networks “retrofit” security using auxiliary middleboxes and network processors. This approach does not scale as it is not an integral part of the network. It is difficult for network operators to maintain policies over time, and it is straightforward for attackers to find a way through the maze of broken policies. To resolve this problem, Casado et al. proposed clean-slate approaches, SANE [26] and then Ethane [25], incorporating a centralized authority to authenticate flows in the network.

SANE. SANE [26] aims at transforming the permissive network into a secure infrastructure using a centralized authorization mechanism. In SANE, a centralized controller (DC) is responsible for authentication, service management and authorization. DC assigns capabilities to authorized flows and if a flow does not have a valid capability switches drop it. Upon initiating a new flow, end-hosts need to ask for capability grants from a centralized domain controller (DC). When capabilities are granted, packets are tagged with the capability and sent out from
the end-hosts.

**Ethane.** Ethane [25] extends SANE such that no modifications are required on end-hosts. In contrast to SANE, Ethane switches have an initially empty flow table that holds active flow-entries \((i.e., \text{forwarding rules})\). When a packet with no active flow-entry arrives at a switch, it is sent to the controller. Then, it is the controller’s responsibility to either setup proper flow-entries to forward the packet, or drop the packet. This ensures centralized policy enforcement while minimizing the modifications on the end-hosts.

### 2.1.4 OpenFlow: Convergence of Proposals

Inspired by 4D, OpenFlow [88] generalizes Ethane for all network programming applications. In contrast to 4D and other clean-slate designs, OpenFlow takes a pragmatic approach with a few compromises on generality. OpenFlow adds a standard API between the data plane and the control plane. In contrast to providing a fully programmable datapath, OpenFlow imposes minimal changes to switches for programming their forwarding state \((i.e., \text{how incoming packets are forwarded in a switch})\). This is to enable high-performance and low-cost implementations in new hardware, as well as straightforward implementation of OpenFlow in existing switches.

**Flow-entries.** OpenFlow models the forwarding state as a set of matched-action rules, called flow-entries, composed of a set of conditions over packet header, and a set of actions. The conditions can match information such as ingress port, mac addresses, IPv4/IPv6 fields, TCP ports and outermost VLAN/MPLS fields to name a few. These conditions are designed to be easily implementable in ternary content-addressable memories (TCAMs). Once all the conditions match a packet the set of respective actions are executed. These actions range from sending packets out to a port to pushing/popping tags and QoS actions.

In addition to forwarding rules, OpenFlow adds minimal functionalities for collecting network statistics. In addition to generic statistics per port, OpenFlow maintains aggregate
metrics (e.g., number of packets, size of packets, etc.) for each flow-entry. This uniquely enables adaptive forwarding.

**Flow-tables.** In OpenFlow 1.0 (and prior versions), all flow-entries are stored in a single table. Each flow-entry had a priority for disambiguation. The problem with this approach is space explosion: we are storing the Cartesian product of all rules in flow-entries. In OpenFlow 1.1, this problem is resolved by replacing the single table with a pipeline of flow-tables. In addition to normal actions, each flow-entry can point to the next flow-table in the pipeline (note that no back reference is allowed). This process is presented in Figure 2.3.

### 2.2 Forwarding Abstractions

Separating the control and data planes is the high-level design approach advocated by all SDN proposals, but they are considerably different in how they draw the border between the responsibilities of the control plane and the functions of the data plane. Such fundamental differences result in different forwarding abstractions for which there are three major schools of thought.
**Programmable Abstractions.** Programmable abstractions view the data plane as a set of programmable entities with an instruction set. Similar to general-purpose processors, forwarding elements do not provide any predefined network function but instead one can offload code into these elements to get the desired functionalities. Active networks, the pioneer of these abstractions, allow programmability at packet transport granularity. Active networks can be implemented in software running on general-purpose processors, or using programmable hardware, such as NetFPGAs\[94\] or BEE \[29\] to achieve line-rate processing.

Active network proposals can be categorized into two categories: Some proposals, such as ActiveIP \[126\], Smart Packets \[107\] and ANTS \[125\] advocate encapsulating code inside packets. The proposals assume that the forwarding elements are stateless, generic processing elements that execute a code accompanying a packet on a per packet basis. This code can either modify the packet itself or result in a forwarding decision. Other proposals, such as Switchware \[114\] and DAN \[36\], advocate offloading code into switches. They assume a switch is a stateful and programmable processing element. The offloaded code is executed per incoming packet and can result in different forwarding decisions.

We also note that there are extensible network programming platforms such as XORP \[55\] and Click \[73\] that must be deployed on generic x86 platforms. These platforms suite well for implementing routing protocols but they are not suitable as a forwarding abstraction.

**Generic Abstractions.** Proposals in this class keep the data plane as generic as possible essentially defining data-path functions as a set of black boxes with no predefined characteristics. In such abstractions, the data plane is responsible for providing a formal specification of its functionalities, and it is the control plane's job to compose these black boxes to program the network. In other words, the control plane merely composes a set of predefined functions to modify a packet or reach a forwarding decision.

ForCES \[128\] is the most notable proposal in this category. In ForCES, Logical Functional Blocks (LFBs) represent the generic networking functions available in forwarding elements. For instance, there are LFBs for address translation, packet validation, IP unicast, and queueing.
Composing these LFBs, the control elements can program forwarding elements.

SwitchBlade \cite{14} is another example in this category which exposes predefined networking functions as blackboxes which can be composed to perform a cohesive behavior. SwitchBlade slices a switch into one or more Virtual Data Planes (VDPs) each having their own isolated pipeline. Each VDP is composed of a set of modules (note that modules can be shared among different VDPs). The set of modules of each VDP is dynamic and configurable.

SwitchBlade processes incoming packets in four stages: (i) It selects the appropriate VDP(s) based on the first few bytes of the packet. (ii) Once the VDPs are selected, the packet enters a shaping module for rate limiting. (iii) Then, packets are preprocessed through VDP's respective preprocessing modules to calculate meta-data about packets. (iv) Finally, packets are forwarded using an output port lookup module which can use longest-prefix matching, exact matching, or forward packets to the CPU.

**Reconfigurable Abstractions.** The other school of thought, led by OpenFlow \cite{88}, views the data plane as a minimal set of functions that can be directly implemented in hardware and pushes all other functions to the control plane. These proposals define the data plane as a set of packet processing elements providing basic forwarding and queuing primitives. The state of these forwarding elements is configured by the control plane. For that reason, reconfigurable abstraction are not completely programmable. For instance, OpenFlow allows the control plane to add, remove and modify flow-entries (§2.1.4) in switches. Some proposals (e.g., Fabric \cite{27}) even push the abstraction further towards a model in which edge switches are programmable, and the core uses a high throughput label switching protocol (e.g., MPLS \cite{105}).

P4 \cite{20, 67} is another proposal that aims at providing more programmability and more flexibility in reconfigurable networks. In essence, the goal of P4 is to bring the benefits of programmable abstractions into reconfigurable abstractions. P4 has a high-level programming language using which one can specify packet fields, how the packets are parsed, and what are the actions to apply on the packets. To be target independent, P4 automatically compiles this high-level program onto the actual target.
Remarks. Programmable and generic abstractions are more evolvable, and accommodate a broader range of data plane functions (including the reconfigurable abstractions) at the cost of simplicity since, to use a specific data plane function, one would need to define several programming contracts; effectively specifying a custom southbound API. Reconfigurable abstractions, on the other hand, enforce a well-defined southbound protocol that both the data and control planes agree upon. This results in a relatively limited data plane, but simplifies network control. More importantly, it prevents API fragmentation for control programs and favors a vendor agnostic ecosystem. For that reason, the later model is well adopted by the research community as well as the industry. In essence, reconfigurable abstractions are the most pragmatic abstraction and thus most SDN proposals have converged on that.

2.3 Control Plane Scalability

Decoupling control from the data plane leads to several scalability concerns. Moving traditionally local control functionalities to a remote controller can potentially result in new bottlenecks. The common perception that control in SDN is centralized leads to concerns about SDN scalability and resiliency. It can also lead to signaling overheads that can be significant depending on the type of network and associated applications.

We argue that there is no inherent bottleneck to SDN scalability, i.e., these concerns stem from implicit and extrinsic assumptions [60]. For instance, early benchmarks on NOX (the first SDN controller) showed it could only handle 30k flow initiations per second [51] while maintaining a sub-10ms flow install time. Such a low flow setup throughput in NOX is shown to be an implementation artifact [118]; or, the control plane being centralized was simply due to the historical evolution of SDN. Without such assumptions, similar to any distributed system, one can design a scalable SDN control plane.

We believe there are legitimate concerns for SDN scalability, but we argue that these scalability limitations are not restricted to SDN, i.e., traditional control protocol design also
facing the same challenges. While this does not address these scalability issues, it shows that we do not need to worry about most scalability issues in SDN more than we do for traditional networks. In this section, we present recent proposals to address these scalability concerns.

2.3.1 In-Data-Plane Proposals

Datapath elements sit at the base layer of SDN with significant influence on the whole architecture. One way to address scalability concerns in the control plane is delegating more responsibilities to the data plane. This simply reduces the load on the control plane and can eliminate bottlenecks.

**Decision Caching.** One way to eliminate the load on the control plane is to cache the decision logic (*i.e.*, the function that maps events into concrete rules) inside the datapath elements. The most notable work pursuing this idea is DIFANE [131]. DIFANE delegates forwarding logic from the controller to a special kind of switches called Authority Switches in order to shield the controller from packet-in events. The controller offloads forwarding rules to Authority Switches, and Authority Switches, on behalf of the controller, install rules in other switches. In other words, Authority Switches act as a proxy that caches simple forwarding decisions. Although considerably efficient, DIFANE is only applicable to forwarding rules that can be implemented in a switch, and cannot help for complex control logics.

**Aggregate Visibility.** Another way to reduce the control plane’s load is to handle frequent events (such as short-lived flow arrivals) within the datapath, and selectively send aggregate events to the control plane. This approach is pioneered by DevoFlow [34] which uses wildcarded flows to forward traffic, and sends only elephant flows (*e.g.*, flow with more than a certain amount of data, bandwidth consumption, etc.) to the controller. Although this approach would result in scalable SDN, it comes at the cost of controller visibility: the controller never gets to see the short-lived flows. Some advantages of SDN would be hindered when the controller does not have the power to visit all flows.
Remarks. These in-data-plane proposals are mostly motivated by early benchmarks on NOX [51] that are shown to be fallacies rooted in a naive IO implementation, which can be simply resolved by using asynchronous IO mechanisms [118] or by exploiting parallelism [134]. Newer implementations of NOX and Beacon [39] can easily cope with hundreds of thousands of flows on commodity machines. We therefore believe that controller’s throughput is not an inherent bottleneck in SDN and there is no need to push functionalities back to the data plane. Moreover, delegating more to the data plane would result in less visibility in control applications that can obviate the benefits of SDN.

2.3.2 Distributed Control Planes

The controller plays a central role in SDN: it programs forwarding elements, maintains the network state, and runs the control applications. In essence, it is the network operating system, and thus must be scalable, reliable, and efficient. Led by NOX [51], SDN controllers are traditionally designed as a centralized middleware that communicates with switches and hosts control applications. These controllers mostly use an abstract object model to represent the network and provide an event-based programming model for control applications. These controllers all had centralized architectures with inefficient implementations. These limitations have recently been the topic of extensive research that we present in this section.

Regardless of the controller’s capability, a centralized controller does not scale as the network grows (increasing the number of switches, flows, bandwidth, etc.) and will fail to handle all the incoming requests while providing the same service guarantees. This necessitates scalable distributed control platforms that can cope with arbitrary loads in different network environments.

ONIX. ONIX [75] is the successor of NOX that maintains the centralized network state in a distributed fashion. As shown in Figure 2.4, ONIX models the network using an abstract data structure called Network Information Base (NIB). NIB represents a network as a graph in which nodes represent networking elements and edges represents logical connections. NIB is
maintained in a distributed fashion and is eventually consistent. Within ONIX, each controller has its own NIB, and when a part of the NIB is updated in one of the controllers it will be eventually propagated to the whole cluster. All other aspects of state distribution (e.g., partitioning and consistency) are handled by control applications.

**HyperFlow.** HyperFlow [116] is another distributed controller built based on NOX. It partitions the network, and assigns a controller to each partition. In order to synchronize the network state, HyperFlow logs events on each controller and propagates them on the cluster. Within HyperFlow, consistency is preserved by managing each switch only by the local controller. Whenever a remote application requires to modify a switch, the message is sent to the controller that manages that switch, and replies are routed back to the application accordingly. In contrast to ONIX, HyperFlow does not require any change in control applications but is not as scalable as ONIX.

**DISCO.** DISCO [103] is a distributed controller focusing on scalable overlay networks. DISCO partitions the network into multiple domains, each controlled by its own controller. In contrast to HyperFlow, DISCO focuses on providing end-to-end connectivity among nodes in the network, not detailed path setup. As such, neighbor controllers form pub-sub relationships with one another. Using this pub-sub system, which is built on top of AMQP [95], network controllers share aggregated network-wide views and establish end-to-end connections.

**ONOS.** ONOS [17] is another distributed controller that tries to address the shortcomings of

![Figure 2.4: ONIX maintains the network-wide state in a distributed fashion.](image-url)
ONIX. In contrast to ONIX, the goal of ONOS is to provide a strongly consistent network-wide state. To that end, as shown in Figure 2.5, ONOS stores the network state in an external distributed datastore; Cassandra and ZooKeeper in its first version and RAMCloud in its second version. As a result, the control platform consists of controllers that synchronize their state using the external state management system. Delegating state management to an external system simplifies ONOS’s design, yet would lead into suboptimalities, scalability issues, and maintenance difficulties.

B4. B4 [66] is Google’s production wide-area network (WAN) that consists of multiple WAN sites. Each site in B4 is controlled by a distributed control plane, called Network Control Servers (NCS), that manage the SDN switches in that site. NCS internally uses ONIX NIBs [75] to implement OpenFlow, and Quagga [6] to implement routing protocols. At a higher level, a logically-centralized global view of the network is used by a centralized Traffic Engineering (TE) application to optimize traffic across B4’s planet-scale network.

ElastiCon. ElastiCon [37] is a proposal on how to distribute the load of the network among controllers in a distributed control plane. The main contribution of ElastiCon is a four phase protocol that controllers use to elect an apt master for a new switch. Moreover, it proposes an algorithm to migrate a switch from one controller to another. ElastiCon is tackling an important
problem and the proposal is orthogonal to all distributed controllers. That is, ElastiCon can be easily adopted in other distributed controllers.

Remarks. An interesting observation is that control plane scalability challenges in SDN (e.g., simplicity and consistency requirements) are not inherently different than those faced in traditional network design. SDN, by itself, is neither likely to eliminate the control plane design complexity nor make it more or less scalable.\(^3\)

SDN, however, (a) allows us to rethink the constraints traditionally imposed on control protocol designs (e.g., a fixed distribution model) and decide on our own trade-offs in the design space; and (b) encourages us to apply common software and distributed systems development practices to simplify development, verification, and debugging. Unlike traditional networks, in SDN, we do not need to address basic but challenging issues like topology discovery, state distribution, and resiliency over and over again. Control applications can rely on the control platform to provide these common functions; functions such as maintaining a cohesive view of the network in a distributed and scalable fashion.

### 2.4 Resilience to Failures

Resilience to failures and convergence time after a failure have always been a key concern in network performance. SDN is no exception, and, with the early systems setting an example of designs with a single central control, resiliency to failures has been a major concern. A state-synchronized slave controller would be sufficient to recover from controller failures, but a network partition would leave half of the network brainless. In a multi-controller network, with an appropriate controller discovery mechanism, switches can always discover a controller if one exists within their partition. Therefore, given a scalable discovery mechanism, controller failures do not pose a challenge to SDN scalability.

Link Failures. Let us decompose the process of repairing a broken link or switch to see how it

---

\(^3\)After all, one can replicate a traditional network design with SDN by co-locating equal number of forwarding and control elements. Even though this obviates the benefits of SDN, it is technically possible.
is different from the traditional networks. As shown in Figure 2.6, convergence in response to a link failure has five steps. The switch *detects* a change. Then, the switch *notifies* the controller. Upon notification, the control program *computes* the repair actions and *pushes* updates to the affected datapath elements which, in turn, *update* their forwarding tables.\(^4\) In traditional networks, link failure notifications are flooded across the network, whereas with SDN, this information is sent directly to a controller. Therefore, the information propagation delay in SDN is no worse than traditional networks. Also, as an advantage for SDN, the computation is carried out on more capable controller machines as opposed to weak management CPUs of all switches, regardless of whether they are affected by the failure or not.

Note that the above argument was built on the implicit assumption that the failed switch or link does not affect the switch-controller communication channel. The control network itself needs to be repaired first, if a failed link or switch was part of it. In that case, if the control network – built with traditional network gear – is running an IGP, the IGP needs to converge first before switches can communicate with the controller to repair the data network. In this corner case therefore convergence may be slower than in traditional networks. If this proves to be a problem, the network operator should deploy an *out-of-band* control network to alleviate this issue.

**Control Plane Failure.** To tolerate a controller failure, controllers have to ensure that the controller’s state is persistent and the state is properly replicated. For a centralized controller, this means persistent state with master-slave replication. For distributed controllers, on the other hand, this depends on the type of consistency guarantees they provide. Existing distributed controllers either provide eventually consistent replications (e.g., ONIX [75]), or rely on an external distributed datastore to replicate the state for them (e.g., ONOS [17]). They handle network partitions in the control plane similarly. It is important to note that beyond the distributed system mechanisms, one can use the state stored in switches to preserve consistency upon failure [100].

\(^4\)For switch failures, the process is very similar with the exception that the controller itself detects the failure.
Control applications can also result in failures in the control plane. For example, when a control application fails to handle an event or when it crashes, it can result in a control plane failure (e.g., loss of connectivity and a transient high loss episode). LegoSDN [28] aims at isolating application failures using a sandbox called AppVisor which contains applications and, according to the operator policies, can stop applications or revert their transactions upon an application error. Approaches like LegoSDN can be very useful to improve the reliability of the control plane.

Remarks. Overall, the failure recovery process in SDN is no worse than traditional networks. Consequently, similar scalability concerns exist and the same techniques used to minimize downtime in traditional networks are applicable to SDN. For instance, SDN design can and should also leverage local fast failover mechanisms available in switches to transiently forward traffic towards preprogrammed backup paths while a failure is being addressed. We stress that control platforms must provide the essential failover and recovery mechanisms that control applications can reuse and rely upon.
2.5 Programming Abstractions

Almost all SDN controllers provide APIs that expose the details embedded in the southbound interface between the data and control planes. That level of details is essential for implementing a controller, but not generally required for managing a network. Network operators and programmers instead require an abstraction through which they can specify the required behavior from the network with minimal effort. To that end, network programming boilerplates and complexities should be handled by the controllers and be hidden from management interfaces. Such an interface will accommodate network growth and will ease management at scale. This is a major challenge for the future of SDN to preserve its simplicity. In this section, we present the proposals that aim at simpler programming abstractions.

**Frenetic.** The Frenetic project \[45, 91\] is the foremost initiative aiming at raising the abstraction level for network programming. Frenetic provides a declarative language (inspired from SQL) for querying network state, and a reactive functional language for programming the network. Frenetic provides functional composition by design and facilitates easy re-use. Frenetic languages run atop a runtime environment that manages all low-level semantics such as installing and updating rules and gathering network statistics. Moreover, the Frenetic runtime provides optimizations and consistency guarantees (both at packet and flow levels).

**Nettle and Procera.** Nettle \[120\] and Procera \[121\] is a high level functional abstraction for network programming on top of OpenFlow, very similar to Frenetic. The only difference is that Nettle and Procera support all networking events while Frenetic is limited to packet level events. On the other hand, Frenetic provides a declarative language for stat collection which neither Nettle nor Procera provide.

**Kinetic.** Kinetic \[72\] is another programming abstraction, built on top of Pyretic, with the goal of creating a domain-specific language (DSL) for dynamic network configurations. In contrast to Frenetic, which provides a static network programming abstraction, Kinetic aims at creating a network programming environment that can react to temporal changes and events in the
network. More importantly, policies implemented in Kinetic are formally verifiable. Internally, Kinetic uses a finite state machine (FSM) to present a dynamic abstraction of the policy. These state machines can be automatically composed to create larger policies and are fed into NuSMV for formal verification. Providing these features makes Kinetic simpler than Pyretic to use in real environments yet incur its own performance and scalability overheads.

**Remarks.** These high-level programming abstractions can indeed simplify network programming by hiding the internal details of SDN protocols. All these proposals, however, focus on centralized controller systems. Providing such abstractions on distributed controllers requires extensive architectural changes in both the control plane and also the abstraction.

### 2.6 Correctness and Consistency Tools

Networks are dynamic entities that constantly change state; links go up and down, elements fail, and new switches are added. Each of these events necessitates new configurations that are enforced while the network is operating. This results in transient inconsistencies, subtle errors (such as loops), and colorful failures that are difficult to detect and resolve. One of the advantages of SDN is that it paves the way for developing new tools to help in detecting and prohibiting such consistency and validation problems.

**Verification.** Verification tools mostly focus on verifying a set of invariants against the network configuration. There are static verification tools that verify network configuration offline and out of bound. Such verification tools aim at checking “what-if” rules, and usually pursue formal modeling for packets, protocols, and flows in order to check network invariants.

FlowChecker [31] and its predecessor ConfigChecker [11] use Binary Decision Diagrams (BDDs) to model the network state, and use model checking techniques to check whether the model matches a set of predefined specifications and rules. Anteater [82], similarly, models network state as a boolean satisfiability problem (SAT), and uses a SAT solver to check invariants.

NICE [24] is a similar approach that uses model checking to validate OpenFlow applications.
It emulates a simple switch for the controller and tries to explore all valid state spaces (in regards to the controller, control channels, and switches) to find bugs and inconsistencies. One limitation of NICE is that it can only check one control application at a time due to the complexity of verifying parallel control logic.

Header Space Analysis (HSA) [69], in contrast, models packets as a binary vector and formally define network configuration as a function that transforms this binary vector. Invariants are then verified based on such header transformations. These methods have interesting theoretical properties and are able to detect some configuration problems, while they fall short in modeling complex network transformations (even when it comes to modeling simple address translations).

Another category of verification tools are focused on real-time, inbound verification. The challenge here is to develop methods for (i) gathering a real-time view of the network and (ii) efficient verification. VeriFlow [70, 71], for instance, limits the search space to a subset of the network and, with that, significantly improves verification performance. VeriFlow, however, cannot detect transient inconsistencies that are caused by different synchronizations between the control plane and the datapath.

**Consistency Guarantees.** Verification mechanisms, although important, cannot detect transient inconsistencies resulting from out of sync configuration updates. For instance, to install a path, we need to push network configuration to every switch in the path. This can simply go out of sync because of networking delays, and until the system reaches a stable state, many packets are maliciously dropped or wrongly forwarded. These sort of problems are very frequent in practice (due to the nature of distributed systems) and are significantly difficult to discover.

Reitblatt et al. [104] have systematically studied these problems, classified them, and proposed update mechanisms to prevent such inconsistencies. They have identified two classes of consistency guarantees: (i) per-packet consistency which ensures each packet is processed using the same version of configuration throughout the network, and (ii) per-flow consistency which ensures packets of a flow are all forwarding using the same version of network
configuration. Note that the latter is a stronger guarantee and thus more difficult to implement.

For per-packet consistency, one can add a tag match to each rule representing the configuration version. Once all rules of a specific version are installed, packets can be labeled with the apt configuration version at ingress ports. Authors have proposed three mechanisms for per-flow consistency. The first mechanism is mixing the per-packet mechanism with flow timeouts. This ensures configurations are not applied unless all flows using a specific configuration version are ended. This approach falls short when a flow spans different ingress port. The second approach is to use wild card cloning which requires DevoFlow switches [34], and the third approach requires notification from end-hosts. This approach is further extended to provide bandwidth guarantees for virtual machine migration in [47]. The main issue of all these approaches is that, in general, they can double the size of forwarding tables during an update.

McGeer [87] has proposed an alternative approach for rule update which incorporates the controller. This approach pushes a set of intermediate rules to switches which forward affected flows to the controller. Once the controller received all packets from all affected flows, it installs the new version of rules, and then forwards all flows. This approach offers more flexibility while increasing the load on the control plane.

Customizable Consistency Generator (CCG) [135] is another proposal that aims at minimizing the relatively high performance and space overheads of the aforementioned consistency proposals. The main idea behind CCG is to minimize the network state transition delay upon a configuration update. To that end, for each update, it verifies whether the update is safe to apply based on a set of user-defined invariants. If safe, the update is installed, otherwise buffered. In the presence of a deadlock (i.e., when none of the updates can be safely applied), CCG falls back to stronger consistency methods (such as [104]) to apply the updates. The authors have shown that CCG can lower space overheads and the transition delays for consistent state update.

**Congestion-Free Updates.** Even when rules are updated consistently (say with no transient loops or dead-end paths), the network might observe transient congestion and respective high
loss episodes during an update due to asynchronous updates among switches. That is, because switches apply updates asynchronously, there is a possibility of significant transient traffic spikes in the network. This can cause high packet loss and considerable temporary congestion, which hinders the usability of interactive (i.e., latency intolerant) systems.

zUpdate [79] and SWAN [62] are both proposals aiming for congestion-free network updates. zUpdate calculates the steps required to transition from the current state to the desired state in a way to minimize congestion and traffic imbalance, and installs the configuration updates using the same method proposed in [104]. SWAN proposes a similar technique with the difference that it keeps a scratch link capacity (e.g., $\frac{1}{3}$ of the link capacity) to avoid infeasible update scenarios. Both zUpdate and SWAN perform better than distributed network protocols such MPLS-TE but are limited to specific network topologies.

**Conflict-Free Configurations.** Policies and configurations can come from different sources for different purposes, and hence can be conflicting. For example, a load balancing application naturally has a different objective than an energy optimizer. Such control applications will compete for resources of the network. When dealing with user-defined policies, one can design an autonomous system that resolves conflict (e.g., the conflict resolution mechanism used for hierarchical policies in PANE [41, 42]).

Composing control applications in a conflict-free manner is more complicated than resolving conflicts among user-defined policies since control applications are not as well-defined and limited as user-defined policies. There are two important proposals for this problem. Corybantic [90] proposes a coarse-grained system in which control modules propose their intended updates in each update round. Then, the system selects a non-conflicting set of updates to apply and notifies the control modules of the selected updates. This way the control modules will adapt their next updates according to the decisions in this round. Corybantic is a plausible solution yet it is quite coarse-grained and appears to be difficult to fully automate.

Volpano et al. [122] propose a finer-grained solution based on deterministic finite-state transducer (DFTs) [63]. DFTs are basically finite state machines that can translate an input to
an output. Volpano et al. assume that control applications are implemented or specified using DFTs. Given such DFTs, an autonomous system can compile updates in safe (i.e., conflict-free) zones. Although the proposed system is fine-grained, the main problem with this proposal is that DFTs are limited and cannot express all sorts of control applications.

**Debuggers.** Although written in software, it is not easy to debug a network application using existing software debugging tools. The reason is that network applications program a distributed system (a cluster of switches), and for debugging its behavior we need to access arbitrary switches and inspect arbitrary forwarding packets.

ndb [54] aims at providing basic debugging features (such as breakpoints and traces). To provide breakpoints, ndb install rules on switches to notify the information collector. To provide packet backtrace, ndb stores postcards (i.e., packet headers along with their flow match information on all flows). When packets traverse through the network, they generate postcards and once it hits a breakpoint the debugger can build the backtrace. ndb is built in a way that postcards and breakpoints do not hit the control plane. However, they consume a considerable amount of bandwidth on the datapath.

SDN Troubleshooting System (STS) [108] takes an alternative path and tries to find software bugs by replaying logs. Due to the probabilistic nature of SDN (e.g., asynchronous events, race conditions and distributed actors), STS cannot rely on a simple replay of logs. Instead, STS does a random walk among possible replay sequences, tries each sequence multiple times, and checks invariants to find the minimum sequence of inputs that trigger the bug. Although STS uses pruning techniques to narrow its search space, it is still unclear how it will scale for realistic networks with tens of thousands of switches.

**Remarks.** Consistency, correctness, and troubleshooting are all very important aspects of network programming. The proposals that we discussed in this chapter are all useful to implement a correct and consistent SDN. Having said that, there are still significant open problems that are not addressed by existing proposals. Specifically, correctness and consistency is still an open problem in a distributed setting. For example, when we have multiple controllers
managing the network providing a flow-level consistency is quite challenging. Another important open challenge is performance troubleshooting and debugging. That is, existing proposals cannot answer queries such as “why is my control plane slow?” or “why doesn't it scale?”

2.7 Monitoring and Measurements

Before SDN, networking applications relies on protocols such as SNMP [56] to capture network statistics and metrics. SNMP metrics, however, are coarse grained and difficult to customize. For instance, it was only possible to estimate traffic matrix using SNMP in traditional networks [89]. For that reason, one would need to use sampling mechanisms (such as sFlow [102]) or capture statistics by modifying end-hosts [46]. To that end, emergence of SDN protocols provides an opportunity for improving monitoring and measurement tools. That is because SDN protocols provide meaningful counters that are attached to individual flows, ports, queues, and switches. This powerful construct enables control applications to gather accurate traffic matrix.

**OpenTM.** OpenTM [117] was the first effort to capture traffic matrix using SDN counters. OpenTM is basically a control application running on NOX [51] that queries OpenFlow switches for flow counters in the network, and creates the traffic matrix accordingly. To lower the overhead of statistics collection, OpenTM tries to find the optimum set of switches to query. The authors have conducted experiments with different selection schemes (i.e., uniform random, non-uniform random, last switch, round-robin, and least-loaded switch) and have shown that querying the least-loaded switch will be the best switch selection scheme in OpenFlow network. Despite all the values of OpenTM, we believe in a real world setting edge switches are always the least-loaded switches and hence querying edge switches would be the simplest, most realistic, and an optimal way of gathering the traffic matrix.

**FlowSense.** FlowSense [129] has the same goal as OpenTM, but proposes an alternative approach. Instead of querying switches and polling the stat counters, FlowSense relies on Packet-
In and Flow-Removed OpenFlow messages to collect the traffic matrix. Packet-In events are fired when a packet does not match any flow in the switch, and Flow-Removed is fired when a flow is removed/expired. FlowSense uses Packet-In events as a trigger for starting a flow, and uses byte counters and duration in Flow-Removed events to estimate the traffic of the flow. The authors have shown that FlowSense has a low overhead compared to polling while maintaining the same level of accuracy. The major problem with FlowSense is that it is limited to reactive flow installation and will not work for a network that is proactively programmed. As most real-world networks are proactively managed, FlowSense would not be widely adopted in practice.

OpenSketch. In contrast to OpenTM and FlowSense, which monitor the network in the control plane, OpenSketch [130] proposes to implement measurement and monitoring facilities in the data plane. OpenSketch has a three-stage, data plane pipeline to select and count packets: (i) Hashing to summarize packet headers, (ii) Classification to classify the packets into different classes, and (iii) Counting which uses probabilistic counters (i.e., Count-Min Sketches) to store the count (for each class). In the control plane, OpenSketch provides a measurement library to assist the control logic to identify flows, store statistics, and query counters. OpenSketch is shown to be accurate while preserving low memory footprint. The major issue with OpenSketch is that it requires changes in the data plane, which results in bloated switches and fragmentation which is in contrast to SDN design principles.

DREAM. DREAM [92] is another inbound measurement system that tries to implement fine-grain measurements using existing TCAM counters in hardware switches. The main challenge of using TCAMs is their limited capacity; especially when one needs fine-grained measurements. DREAM proposes an adaptive method, trying to find a trade-off between the granularity of matches installed in TCAMs and the TCAM space. Although this method can be advantageous in existing network, it has limits in the level of visibility it provides for control applications.

NetAssay. NetAssay [38] takes a different approach for measurement and monitoring by focusing on high-level concepts (e.g., devices and applications) instead of low-level constructs (e.g., flows or addresses). To that end, NetAssay forms a dynamic mapping between higher-level
Chapter 2. Background

concepts and the flow space, and uses that mapping for translating low-level measurements to high-level monitoring. To keep NetAssay generic, these mappings can be implemented by the operators. It is important to note that NetAssay’s approach is orthogonal to other SDN measurement approaches and can complement other proposals.

2.8 Network Optimization

Network applications are traditionally built based on the end-to-end design argument [106] and traditional networks act as a black-box medium for packet transport. This resulted in suboptimal behaviors in transport and application layer [18]. This problem can only be solved when the network provides an open integration/programming point for applications. SDN provides such an open API and creates an opportunity to optimize network applications. In what follows, we briefly explain some of the most important application optimization efforts based on SDN.

Hedera and Helios. Data center networks usually have a multi-root topology (e.g., leaf-spine topology) with high aggregate bandwidth. Traditional Interior Gateway Protocols (IGPs), such as IS-IS [97] and OSPF [93], mostly yield a minimum spanning tree, and hence, quite a few underutilized links. Link aggregation protocols (e.g., MLAG or MC-MLAG), on the other hand, are passive and cannot optimize for traffic patterns in a network. For instance, it can result in congestion for elephant flows. To tackle this problem, Hedera [10] proposes a dynamic flow scheduling mechanism using SDN. The Hedera scheduler finds elephant flows (i.e., flows that need bandwidth but are network limited) and tries to find a optimal placement for elephant flows on ECMP paths.

Helios [40] focuses on hybrid datacenter fabrics that use all optical switches alongside an Ethernet network. In such networks, the optical fabric is high-throughput but does not have enough (bandwidth and switching) capacity to forward all connections in the network. Moreover, the optical fabric has a relatively high latency for switching and thus cannot efficiently react to short-lived traffic. The idea behind Helios is to forward short lived traffic on the Ethernet
fabric, and forward long-lived, elephant flows on optical paths. To that end, Helios queries switches to detect elephant flows. Once an elephant flow is detected it will be forwarded to a pre-setup optical path.

Both Hedera and Helios result in higher link utilization, lower congestion, and lower late latency.

**Hadoop Optimization.** Helios and Hedera were both application agnostic and work without the knowledge of the applications and their networking demand. Although this results in transparent network optimization, there are applications (or development frameworks) where we know about their network demand and application topology. For instance, in a Hadoop cluster, we can take advantage of our knowledge about job allocation and jobs' size, to optimize the network. Wang et al. [124] have proposed to use networking information of Hadoop clusters for flow scheduling. They focus on hybrid (optical/Ethernet) fabrics, and reconfigure the topology of the optical fabric based on Map-Reduce jobs, and HDFS demand.

## 2.9 Discussion

Previous works have addressed SDN research challenges in an innovative manner that was not feasible in traditional networks. However, we believe that there is still no clear proposal on how to resolve scalability issues in SDN without hindering its simplicity. Implementing, measuring and optimizing a distributed control application is quite difficult using existing proposals. On the other hand, using simplifying programming models results in centralization and possibly the lack of scalability. Our goal in this dissertation is to create an easy-to-use network programming paradigm that is distributed, intuitive, scalable and self-optimizing.
Chapter 3

Kandoo

Limiting the overheads of frequent events on the control plane is essential for realizing a scalable SDN. One way of limiting this overhead is to process frequent events in the data plane. This requires modifying switches and comes at the cost of visibility in the control plane. Taking an alternative route, in this chapter, we present Kandoo [57], a framework for preserving scalability without changing switches.

3.1 Introduction

Frequent and resource-exhaustive events, such as flow arrivals and network-wide statistics collection events, stress the control plane and consequently limit the scalability of OpenFlow networks [34, 116, 33]. Although one can suppress flow arrivals by proactively pushing the network state, this approach falls short when it comes to other types of frequent events such as network-wide statistics collection. Existing solutions either view this as an intrinsic limitation or try to address it by modifying switches. For instance, HyperFlow [116] can handle a few thousand events per second, and anything beyond that is considered a scalability limitation. In contrast, DIFANE [131] and DevoFlow [34] introduce new functionalities in switches to suppress frequent events and to reduce the load on the control plane.

To limit the load on the controller, frequent events should be handled in the closest vicinity
of datapaths, preferably without modifying switches. Adding new primitives to switches is undesirable. It breaks the general principles of SDN, requires changes to the standards, and necessitates the costly process of modifying switching hardware. Thus, the important question is:

How can we move control functionalities toward datapaths, without introducing new datapath mechanisms in switches?

To answer this question, we focus on: (i) environments where processing power is readily available close to switches (such as datacenter networks) or can be easily added (such as enterprise networks), and (ii) applications that are local in scope, i.e., applications that process events from a single switch without using the network-wide state. We show that, under these two conditions, one can offload local event processing to local resources, and therefore realize a control plane that handles frequent events at scale.

We note that some applications, such as routing, require the network-wide state, and cannot be offloaded to local processing resources. However, a large class of applications are either local (e.g., local policy enforcer and Link Layer Discovery Protocol [7]) or can be decomposed to modules that are local (e.g., elephant flow detection module in an elephant flow rerouting application).

Kandoo. In this chapter, we present the design and implementation of Kandoo [57], a novel distributed control plane that offloads control applications over available resources in the network with minimal developer intervention and without violating any requirements of control applications. Kandoo’s control plane essentially distinguishes local control applications (i.e., applications that process events locally) from non-local applications (i.e., applications that require access to the network-wide state). Kandoo creates a two-level hierarchy for controllers: (i) local controllers execute local applications as close as possible to switches, and (ii) a logically centralized root controller runs non-local control applications. As illustrated in Figure 3.1, several local controllers are deployed throughout the network; each of these controllers controls one or a handful of switches. The root controller, on the other hand, controls all local controllers.
It is easy to realize local controllers since they are merely switch proxies for the root controller, and they do not need the network-wide state. They can even be implemented directly in OpenFlow switches. Interestingly, local controllers can linearly scale with the number of switches in a network. Thus, the control plane scales as long as we process frequent events in local applications and shield the root controller from these frequent events. Needless to say, Kandoo cannot help any control applications that require network-wide state (even though it does not hurt them, either). In Chapter 4, we present Beehive, which provides a framework to implement such applications.

Our implementation of Kandoo is completely compliant with the OpenFlow specifications. Data and control planes are decoupled in Kandoo. Switches can operate without having a local controller; control applications function regardless of their physical location. The main advantage of Kandoo is that it gives network operators the freedom to configure the deployment model of control plane functionalities based on the characteristics of control applications.

The design and implementation of Kandoo are presented in Section 3.2. Our experiments confirm that Kandoo scales an order of magnitude better than a normal OpenFlow network and would lead to more than 90% of events being processed locally under reasonable assumptions, as described in Section 3.3. Applications of Kandoo are not limited to the evaluation scenarios
presented in this section. In Section 3.4, we briefly discuss other potential applications of Kandoo and compare it to existing solutions. We conclude our discussion of Kandoo in Section 3.5.

3.2 Design and Implementation

**Design Objectives.** Kandoo is designed with the following goals in mind:

1. Kandoo must be compatible with OpenFlow: we do not introduce any new data plane functionality in switches, and, as long as they support OpenFlow, Kandoo supports them, as well.

2. Kandoo is easy to use and automatically distributes control applications without any manual intervention. In other words, Kandoo control applications are not aware of how they are deployed in the network, and application developers can assume their applications would be run on a centralized OpenFlow controller. The only extra information Kandoo needs is a flag showing whether a control application is local or not.

In this section, we explain Kandoo's design using a toy example. We show how Kandoo can be used to reroute elephant flows in a simple network of three switches (Figure 3.2). Our example has two applications: 

(i) \textit{App$_{detect}$}, and (ii) \textit{App$_{reroute}$}. \textit{App$_{detect}$} constantly queries each switch to detect elephant flows. Once an elephant flow is detected, \textit{App$_{detect}$} notifies \textit{App$_{reroute}$}, which in turn may install or update flow-entries on network switches.

It is extremely challenging, if not impossible, to implement this application in current OpenFlow networks without modifying switches [34]. If switches are not modified, a (logically) centralized control needs to frequently query all switches, which would place a considerable load on control channels.

**Kandoo Controller.** As shown in Figure 3.3, Kandoo has a \textit{controller} component at its core. This component has the same role as a general OpenFlow controller, but it has Kandoo-specific
Figure 3.2: Toy example for Kandoo’s design: In this example, two hosts are connected using a simple line topology. Each switch is controlled by one local Kandoo controller. The root controller controls the local controllers. In this example, we have two control applications: \( \text{App}_{\text{detect}} \) is a local control application, but \( \text{App}_{\text{reroute}} \) is non-local.

functionalities for identifying local and non-local application, hiding the complexity of the underlying distributed application model, and propagating events in the control plane.

A network controlled by Kandoo has multiple local controllers and a logically centralized root controller.\(^1\) These controllers collectively form Kandoo’s distributed control plane. Each switch is controlled by only one Kandoo controller, and each Kandoo controller can control multiple switches. If the root controller needs to install flow-entries on switches of a local controller, it delegates the requests to the respective local controller. Note that for high availability, the root controller can register itself as the slave controller for a specific switch (this behavior is supported in OpenFlow 1.2).

**Deployment Model.** The deployment model of Kandoo controllers depends on the characteristics of a network. For software switches, local controllers can be directly deployed on the same end-host. Similarly, if we can change the software of a physical switch, we can deploy Kandoo directly on the switch. Otherwise, we deploy Kandoo local controllers on the processing resources closest to the switches. In such a setting, one should provision the number of local controllers based on the workload and available processing resources. Note that we can use

\(^1\)We note that the root controller in Kandoo can itself be distributed.
a hybrid model in real settings. For instance, consider a virtualized deployment environment depicted in Figure 3.4, where virtual machines are connected to the network using software switches. In this environment, we can place local controllers in end-hosts next to software switches and in separate nodes for other switches.

In our toy example (Figure 3.2), we have four Kandoo controllers: three local controllers controlling the switches and a root controller. The local controllers can be physically positioned using any deployment model explained above. Note that, in this example, we have the maximum number of local controllers required.

Control Applications. In Kandoo, control applications are implemented using the abstraction provided by the platform and are not aware of their actual placement (e.g., on local controllers or on the root controller). They are generally OpenFlow applications and can therefore send OpenFlow messages and listen on events. Moreover, they can emit application-defined events, which can be consumed by other applications, and can reply to an event emitted by another application. In Kandoo, control applications use Packet [5] to declare, serialize, and deserialize events.
Figure 3.4: Kandoo in a virtualized environment. For software switches, we can leverage the same end-host for local controllers, and, for physical switches, we use separate processing resources.

Control applications are loaded in local name spaces and can communicate only using events. This is to ensure that Kandoo does not introduce faults by offloading applications. In our example, $E_{\text{elephant}}$ is an application-defined event that carries matching information about the detected elephant flow (e.g., its OpenFlow match structure) and is emitted by $A_{\text{pp}}_{\text{detect}}$.

A local controller can run an application only if the application is local. In our example, $A_{\text{pp}}_{\text{reroute}}$ is not local, i.e., it may install flow-entries on any switch in the network. Thus, the root controller is the only controller able to run $A_{\text{pp}}_{\text{reroute}}$. In contrast, $A_{\text{pp}}_{\text{detect}}$ is local; therefore, all controllers can run it.

Event Propagation. The root controller can subscribe to specific events in the local controllers using a simple messaging channel plus a filtering component. Once the local controller receives and locally processes an event, it relays the event to the root controller for further processing. Note that all communications between Kandoo controllers are event-based and asynchronous.

In our example, the root controller subscribes to events of type $E_{\text{elephant}}$ in the local controllers since it is running $A_{\text{pp}}_{\text{reroute}}$ listening on $E_{\text{elephant}}$. $E_{\text{elephant}}$ is fired by an $A_{\text{pp}}_{\text{detect}}$.
instance deployed on one of the local controllers and is relayed to root controller. Note that if the root controller does not subscribe to $E_{elephant}$, the local controllers will not relay $E_{elephant}$ events.

It is important to note that the data flow in Kandoo is not always bottom-up. A local application can explicitly request data from an application deployed on the root controller by emitting an event, and applications on the root controllers can send data by replying to that event. For instance, we can have a topology service running on the root controller that sends topology information to local applications by replying to events of a specific type.

**Reactive vs. Proactive.** Although Kandoo provides a scalable method for event handling, we strongly recommend pushing network state proactively. We envision Kandoo to be used as a scalable, adaptive control plane, where the default configuration is pushed proactively and is adaptively refined afterwards. In our toy example, default paths can be pushed proactively, while elephant flows will be rerouted adaptively.

## 3.3 Evaluation

In this section, we present the results obtained for the elephant flow detection problem. This evaluation demonstrates the feasibility of Kandoo to distribute the control plane at scale and strongly supports our argument.

**Setup.** In our experiments, we realize a two-layer hierarchy of Kandoo controllers as shown in Figure 3.5. In each experiment, we emulate an OpenFlow network using a slightly modified version of Mininet [78] hosted on a physical server equipped with 64G of RAM and 4 Intel Xeon(R) E7-4807 CPUs (each with 6 cores). Here, we do not enforce any rate, delay or buffering for the emulated network. We use OpenVSwitch 1.4 [101] as our kernel-level software switch.

**Elephant Flow Detection.** We implemented Elephant Flow Detection applications (i.e., $App_{reroute}$ and $App_{detect}$) as described in Section 3.2. As depicted in Figure 3.5, $App_{detect}$ is deployed on all Kandoo local controllers, whereas $App_{reroute}$ is deployed only on the root
controller. Our implementation of $App_{detect}$ queries only the top-of-rack (ToR) switches to detect the elephant flows. To distinguish ToR switches from the core switch, we implemented a simple link discovery technique. $App_{detect}$ fires one query per flow per second and reports a flow as elephant if it has sent more than 1MB of data. Our implementation of $App_{reroute}$ installs new flow entries on all switches for the detected elephant flow.

**Learning Switch.** In addition to $App_{detect}$ and $App_{reroute}$, we use a simple learning switch application on all controllers to setup paths. This application associates MAC addresses to ports and installs respective high granular flow entries (i.e., destination MAC address, IP address and TCP port number) with an idle timeout of 100ms on the switch. We note that the bandwidth consumed for path setup is negligible compared to the bandwidth consumption for elephant flow detection. Thus, our evaluation results would still apply, even if we install all paths proactively.

**Methodology.** In our experiments, we aim to study how this control plane scales with respect to the number of elephant flows and the network size compared to a normal OpenFlow network (where all three applications are on a single controller and $App_{detect}$ queries switches at the same
rate). We measure the number of requests processed by each controller and their bandwidth consumption. We note that our goal is to decrease the load on the root controller in Kandoo. Local controllers handle events locally, which consume far less bandwidth compared to events sent to the root controller. Moreover, Kandoo’s design makes it easy to add local controllers when needed, effectively making the root controller the only potential bottleneck in terms of scalability.

Our first experiment studies how these applications scale with respect to the number of elephant flows in the network. In this experiment, we use a tree topology of depth 2 and fanout 6 (i.e., 1 core switch, 6 top-of-rack switches, and 36 end-hosts). Each end-host initiates one hundred UDP flows randomly to other hosts in the network over the course of four seconds. We use two types of UDP flows: (i) mouse flows that will last only 1 second, and (ii) elephant flows that will last for 4 seconds. This results in 3600 UDP flows in total. In this experiment, we measure the rate of messages sent over the control channels, averaged for 25 experiments.

To forward a UDP flow towards an end-host connected to the same ToR switch, we install only one flow-entry. Otherwise, we will install three flow-entries (one on the source ToR switch, one on the core, and one on the destination ToR switch). The probability of establishing a UDP flow from a given host to another host on the same ToR switch is only 14%. This synthetic workload stresses Kandoo since most flows are not local. As reported in [16], datacenter traffic has locality, and Kandoo would therefore perform better in practice.

As depicted in Figure 3.6, control channel consumption and the load on the central controller are considerably lower for Kandoo, even when all flows are elephant\(^2\): five times smaller in terms of messages (Figure 3.6a), and an order of magnitude in terms of bytes (Figure 3.6b). The main reason is that, unlike the normal OpenFlow, the central controller does not need to query the ToR switches. Moreover, \( App_{detect} \) fires one event for each elephant flow, which results in significantly less events.

To study Kandoo’s scalability based on the number of nodes in the network, we fix ratio of

\(^2\)When all flows are elephant, \( App_{detect} \) fires an event for each flow. Thus, it reflects the maximum number of events Kandoo will fire for elephant flow detection.
Figure 3.6: Control Plane Load for the Elephant Flow Detection Scenario. The load is based on the number of elephant flows in the network.

the elephant flows at 20% and experiment with networks of different fanouts. As illustrated in Figure 3.7, the role of local controllers is more pronounced when we have larger networks. These controllers scale linearly with the size of the network and effectively shield the control channels from frequent events. Consequently, Kandoo’s root controller handles significantly less events compared to a normal OpenFlow network.

Based on these two experiments, Kandoo scales significantly better than a normal OpenFlow network. The performance of Kandoo can be improved even further by simple optimization in Appdetect. For example, the load on the central controller can be halved if Appdetect queries only
the flow-entries initiated by the hosts directly connected to the respective switch. Moreover, the perceived functionality of our elephant flow rerouting application remains the same in both cases. That is, Kandoo does not improve nor hurt the ability to detect or reroute elephant flows. In essence, the main advantage of Kandoo is that it enables us to deploy the same application in large-scale networks by shielding the control channels from the load of frequent events.

3.4 Discussion

Datapath Extensions. The problem that we tackle with Kandoo is a generalization of several previous attempts at scaling SDN. A class of solutions, such as DIFANE [131] and DevoFlow [34], address this problem by extending data plane mechanisms of switches with the objective of
reducing the load towards the controller. DIFANE tries to partly offload forwarding decisions from the controller to special switches, called authority switches. Using this approach, network operators can reduce the load on the controller and the latencies of rule installation. DevoFlow, on the other hand, introduces new mechanisms in switches to dispatch far fewer “important” events to the control plane. Kandoo has the same goal, but, in contrast to DIFANE and DevoFlow, it does not extend switches; instead, it moves control plane functions closer to switches. Kandoo’s approach is more general and works well in datacenters, but it might have a lower throughput than specific extensions implemented in hardware.

Processing frequent events using hardware primitives in forwarding silicone can result in considerably better efficiency and lower costs at scale compared to a centralized controller. This method, however, comes with its own pitfalls [58]: (i) Designing a special purpose hardware for a new function imposes a rigid structure; hence, that function cannot evolve frequently. If we need to update the design (say to fix an error, or to improve performance) we need to wait for a considerably long time for the next development cycle. (ii) Design, development, and manufacturing special purpose hardware is quite costly (orders of magnitude more expensive than software), which increases the risk for already risk-averse ASIC manufacturers.

Because of these pitfalls, silicone is not the best way for introducing new network functions, where chances of change is not negligible. In essence, it would make sense to use silicone for implementing a specific function only when (i) the function is mature enough (i.e., applied and tested in real-world for quite some time), and (ii) we have hit a performance barrier on commodity hardware. This explains OpenFlow’s focus on forwarding as the main data path functionality.

Interestingly, we can use Kandoo to prototype and test DIFANE, DevoFlow, or other potential hardware extensions. For instance, an authority switch in DIFANE can be emulated by a local Kandoo controller that manages a subset of switches in the network. As another example, DevoFlow’s extensions can also be emulated using Kandoo controllers directly installed on switches. These controllers not only replace the functionality of DIFANE or DevoFlow, but they
Chapter 3. Kandoo

also provide a platform to run any local control application in their context.

**Distributed Controllers.** HyperFlow [116], ONIX [75], SiBF [81], and Devolved Controllers [115] try to distribute the control plane while maintaining logically centralized, eventually consistent network state. Although these approaches have their own merits, they impose limitations on applications they can run. This is because they assume that all applications require the network-wide state; hence, they cannot be of much help when it comes to local control applications. That said, the distributed controllers can be used to realize a scalable root controller, the controller that runs non-local applications in Kandoo.

**Middleboxes.** An alternative way to introduce a new network function is outside the scope of SDN in software modeled as black-boxes sitting either at the core or at the edge of the network, or software deployed on processing resources available inside switches such as co-processors and FPGAs. The common trait among these proposals is that they all rely on SDN solely for steering packets through black-boxes and packet processing units [58]. Middlebox architectures, such as FlowStream [50], SideCar [113] and CoMb [110], provide scalable programmability in the data plane by intercepting flows using processing nodes in which network applications are deployed.

Kandoo is orthogonal to these approaches in the sense that it operates in the control plane, but it provides a similar distribution for control applications. In a network equipped with FlowStream, SideCar or CoMb, Kandoo can share the processing resources with middleboxes (given that control and data plane applications are isolated) in order to increase resource utilization and decrease the number of nodes used by Kandoo.

**Active Networks.** Active Networks (AN) and SDN represent different schools of thought on programmable networks. SDN provides programmable control planes, whereas ANs allow programmability in networking elements at packet transport granularity by running code encapsulated in the packet [107, 125] or installed on the switches [114, 36]. An extreme deployment of Kandoo can deploy local controllers on all switches in the network. In such a setting, we can emulate most functionality of ANs. That said, Kandoo differs from active networks in two ways. First, Kandoo does not provide inbound packet processing; instead, it
follows the fundamentally different approach proposed by SDN. Second, Kandoo is not an all-or-nothing solution \textit{(i.e.,} there is no need to have Kandoo support on switches). Using Kandoo, network operators can still gain efficiency and scalability using commodity middle boxes, each controlling a partition of switches.

### 3.5 Remarks

Kandoo is highly configurable, quite simple and scalable. It uses a simple yet effective approach for creating a distributed control plane: it processes frequent events in highly parallel local control applications and rare events in a central location. As confirmed by our experiments, Kandoo scales remarkably better than a normal OpenFlow implementation, without modifying switches or using sampling techniques.

Kandoo can co-exist with other controllers by using either FlowVisor [112] or customized Kandoo adapters. Having said that, extra measures should be taken to ensure consistency. The major issue is that Kandoo local controllers do not propagate an OpenFlow event unless the root controller subscribes to that event. Thus, without subscribing to all OpenFlow events in all local controllers, we cannot guarantee that existing OpenFlow applications work as expected.

In the next chapter, we present another distributed control platform, Beehive, that supports a larger category of distributed control applications beyond local control applications. Using Beehive, one can create a generic hierarchy of controllers as well as other topologies for control applications.
Chapter 4

Beehive

As demonstrated in Chapter 3, Kandoo is quite effective when the control logic can be broken into local and centralized applications. However, there are applications that are neither local nor require the whole network-wide state. For example, a virtual networking application can operate using the state of a single tenant and does not require a centralized view of the network. Kandoo models such applications as centralized applications, which is unnecessarily limited. Aiming to address this issue, in this chapter, we present Beehive which is powerful enough to support a wide-range of applications from local to centralized, and a variety of distribution models including generic hierarchies.

4.1 Introduction

Distributed control platforms are employed in practice for the obvious reasons of scale and resilience [75, 61]. Existing distributed controllers are designed for scalability and availability, yet expose the complexities and boilerplates of realizing a control application to network programmers [86]. In such a setting, designing a distributed control application comprises significant efforts beyond the design and implementation of the application’s core: one needs to address consistency, concurrency, coordination, and other common hurdles 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* 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.*, a hash-map, an associative array), intentionally similar to centralized controllers to preserve simplicity. Beehive’s distributed control platform is the runtime of our programming model and is hidden from network programmers. Exploiting the proposed programming model, it seamlessly transforms the centralized application into their distributed counterparts.

Using this 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 without intervention by the programmer. 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 is used to (i) automatically optimize the control plane, and (ii) to provide feedback to network programmers. In this chapter, 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.

**Why not an external datastore?** Emerging control platforms, most notably ONOS [17], 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 overhead [76]. After all, embedding the state inside the controllers (as pioneered by ONIX [75]) has significantly lower communication overheads than using an external system. Further, creating a correct distributed control function requires coordination and synchronization.

---

1It is important to note that, 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).
among control applications. This is essentially beyond state management as demonstrated in [123, 23]. For example, even if two instances of a routing application store their state in a distributed store, they need to coordinate in order to avoid blackholes and loops. Implementing such coordination mechanisms eliminates many of the original motivations of utilizing an external datastore.

More importantly, in such a setting, it is the external datastore that dictates the placement and the topology of control functions. For instance, if the external datastore 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 [37]. 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 an external datastore is employed.

**Overview.** Our implementation of Beehive is open source and publicly available [2]. Using Beehive, we have also implemented a full featured SDN controller [3] (presented in Chapter 5) as well as important SDN and non-SDN sample applications such as Kandoo [57], traffic engineering, a key-value store (similar to memcached), a message queue, and graph processing. 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 showcase a heuristic algorithm that can optimize the placement of control applications.
App LearningSwitch:

// A message handler that handles messages of type PacketIn.
Rcv(pin PacketIn):

  switches = dict("Switches")
  // Associate the source MAC address to the input port for the switch that
  // has sent the PacketIn message.
  mac2port = switches.Get(pin.Switch)
  mac2port[pin.SrcMac] = pin.InPort
  switches.Put(pin.Switch, mac2port)
  // Lookup the output port for the destination MAC address.
  outp = mac2port.Get(pin.DstMac)
  if !outp then
    // Flood the packet if the destination MAC address is not learned yet.
    Reply(pin, PacketOut{Port: FLOOD_PORT, ...})
    return
  // Send the packet to the learned output port.
  Reply(pin, PacketOut{Port: outp, ...})
  // Install a flow on the switch to forward consequent packets.
  Reply(pin, FlowMod{...})

App OpenFlowDriver:

Rcv(pout PacketOut):

  // Send packet out to the switch using the OpenFlow connection.
  ...
Rcv(fmod FlowMod):

  // Send the flow mod to the switch using the OpenFlow connection.
  ...
...
// The IO routine.
while read data from the OpenFlow connection do

  ... // The emitted PacketIn will be sent to all applications that has a
  // handler for PacketIn, including the LearningSwitch application.
  Emit(PacketIn{...})
...

Figure 4.1: A simple learning switch application in Beehive.

4.2 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 control applications.

The Anatomy of an Application. Using Beehive’s programming model, a control application is
implemented as a set of single-threaded message handlers (denoted by \texttt{Rcv}) that are triggered by asynchronous messages and can emit further messages to communicate with other applications. For example, as shown in Figure 4.1, a sample learning switch application has a \texttt{Rcv} function for \texttt{PacketIn} messages, and an OpenFlow driver has \texttt{Rcv} functions for \texttt{PacketOut} and \texttt{FlowMod} messages. In our design, messages are either (i) emitted to all applications that have a handler for that particular type of message, or (ii) generated as a reply to another message. A reply message is processed only at its destination application. For example, the OpenFlow driver emits a \texttt{PacketIn} message whenever it receives the event from the switch. Inside the \texttt{PacketIn} message, Beehive automatically stores that this message is sent by the OpenFlow driver. The \texttt{PacketIn} message is processed in any application that has a handler for \texttt{PacketIn} messages, say the learning switch and a discovery application. The learning switch application, upon handling this \texttt{PacketIn} message, replies to the message with a \texttt{PacketOut} and/or a \texttt{FlowMod} message, and these messages are directly delivered to the OpenFlow driver, not any other application.

To preserve state, message handlers use local application dictionaries (\textit{i.e.}, maps or associative array). In Figure 4.1, the learning switch application uses its \texttt{Switches} dictionary to store the learned “MAC address to port mapping” for each switch. As shown in Figure 4.2, the \texttt{Switches} dictionary associates a switch to the “MAC address to port mapping” learned from the packets received from that switch. As explained in Section 4.3, these dictionaries are transactional, replicated, and persistent behind the scenes (\textit{i.e.}, hidden from control applications, and without network 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.

Let us demonstrate this programming model for another simple example.

\textbf{Example: Elephant Flow Routing.} As shown in Figure 4.4, an elephant flow routing application can be modeled as four message handlers at its simplest: (i) \texttt{Init}, which handles \texttt{SwitchJoin}ed messages emitted by the OpenFlow driver and initializes the \texttt{Stats} dictionary for switches. (ii) \texttt{Query}, which handles periodic timeouts and queries switches. (iii) \texttt{Collect}, which handles
Figure 4.2: The LearningSwitch application (Figure 4.1) learns the MAC address to port mapping on each switch and stores them in the `Switches` dictionary.

StatReplies (i.e., sent by the OpenFlow switch in response to stat queries), populates the time-series of flow statistics in the `Stats` dictionary, and detects and reports traffic spikes. (iv) Route, which re-steers elephant flows using `FlowMod` messages upon detecting a traffic spike. 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 elephant flow routing 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 handles `StatQuerys` and `FlowMods`.

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
**App ElephantFlowRouting:**

// Init: Handles SwitchJoined messages.

```plaintext
RCv(joined SwitchJoined):
  stats = Dict("Stats")
  switch = joined.Switch
  // Initializes the stats dictionary for the new switch.
  stats.Put(switch, FlowStat{Switch: switch})
```

// Query: Handles timeout messages. The timer emits one timeout message per second.

```plaintext
RCv(to TimeOut):
  stats = Dict("Stats")
  // Iterates over the keys in the stats dictionary. Note that these entries are initialized in Init.
  for each switch in stats:
    // Emits a StatQuery for each switch in the "stats" dictionary. Note that StatQuery is handled by the OpenFlow driver.
    Emit(StatQuery{Switch: switch})
```

// Collect: Handles StatReply messages emitted by the OpenFlow driver in response to StatQuery messages.

```plaintext
RCv(reply StatReply):
  stats = Dict("Stats")
  switch = reply.Switch
  // Compares the previous statistics with the new one, and emits a TrafficSpike message if it detects a significant change in the stats.
  oldSwitchStats = stats.Get(switch)
  newSwitchStats = reply.Stats
  if there is a spike in newSwitchStats vs oldSwitchStats then
    Emit(TrafficSpike{Stats: newSwitchStats})
  stats.Put(switch, newSwitchStats)
```

// Route: Handles messages of type TrafficSpike.

```plaintext
RCv(spike TrafficSpike):
  trafficMatrix = Dict("TrafficMatrix")
  topology = Dict("Topology")
  // Update trafficMatrix based on spike and then find the elephant flows.
  elephantFlows = findElephantFlows(trafficMatrix)
  for each flow in elephantFlows:
    // Emit flow mods to reroute elephant flows based on the topology.
    Emit(FlowMod{...})
```

// Details on handling switch and link discovery is omitted for brevity.

Figure 4.3: This simple elephant flow routing application stores flow periodically collects stats, and accordingly reroutes traffic. The schematic view of this algorithm is also portrayed in Figure 4.4.

messages 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. These utilities are as easy to use as function calls across applications.

**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 elephant flow routing example (Figure 4.4), 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.

4.3 Control Platform

Beehive’s control platform acts as the runtime environment for the proposed programming model. In other words, the programming model is the public API that is used to develop control applications, and the control platform is our implementation of that API. We have implemented Beehive’s control platform in Go. Our implementation is open source and available as a generic, distributed message passing platform with functionalities such as queuing, parallelism, replication, and synchronization, with no external dependencies. This control platform is built based on the primitives of hive, cells, and bees for concurrent and consistent state access in a distributed fashion with runtime instrumentation and automatic optimization. We note that these primitives (i.e., hives, cells and bees) are internal to the control platform and are not exposed. That is, network programmers are not expected to know or use these concepts in their control applications.

4.3.1 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: \((\text{dict}, \text{key}, \text{val})\). For instance, our elephant flow routing application (Figure 4.4) has a cell for the flow statistics of switch S in the form of \((\text{Stats}, S, \text{Stats}_S)\).
Figure 4.5: 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.

**Bees.** For each set of cells that must be co-located to preserve consistency (as discussed in Section 4.2), we create an exclusive, logical and light-weight thread of execution called a *bee*. As shown in Figure 4.5, upon receiving a message, the 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 elephant flow routing example, once the hive received the first SwitchJoined message for switch S, it creates a bee that exclusively owns the cell \((\text{Stats}, S, \text{Stats}_S)\) 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 StatReplys for S since it exclusively owns \((\text{Stats}, S, \text{Stats}_S)\). 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(A, 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 elephant flow routing example).

**Automatically Generated Map Functions.** Our control platform is capable of generating the Map functions at both compile-time and runtime. Our compiler automatically generates the missing Map functions based on the keys used by message handlers to process a message. The compiler parses the message handlers, prunes the code paths that are not reachable by a dictionary access instruction, and generates the code to retrieve respective keys to build the intended mapped cells.

```
// Init
Rcv(joined SwitchJoined):
  stats = dict("Stats")
  switch = joined.Switch
  stats.Put(switch, FlowStat{Switch: switch})

// Query
Rcv(to TimeOut):
  stats = Dict("Stats")
  for each switch in stats:
    Emit(StatQuery{Switch: switch})

// Collect
Rcv(reply StatReply):
  stats = Dict("Stats")
  switch = reply.Switch
  oldSwitchStats = stats.Get(switch)
  newSwitchStats = reply.Stats
  if there is a spike
    Emit(TrafficSpike{Stats: newSwitchStats})
  stats.Put(switch, newSwitchStats)
```

Figure 4.6: Automatically generated Map functions based on the code in Figure 4.4.
For example, as shown in Figure 4.6, the compiler generates the Map functions based on the code in Init, Query, and Collect, by parsing the Rcv functions and inferring the dictionary accesses. Processing Collect, for instance, our compiler first detects that the stats local variable is a reference to a dictionary named Stats. Then, it finds the Put and Get invocations on Stats. Parsing the parameters of those functions, it infers that the local variable switch is passed as the key to both Get and Put invocations. Thus, the Map function for Collect should return ("Stats", switch). Since switch is an expression, the compiler also needs to include all the statements that affect switch (e.g., switch = reply.Switch), while ignoring all other statements that have no effects on this expression. In other words, the compiler keeps the statements required to create the keys used to access dictionaries in the Rcv function.

Our compiler treats foreach loops in a different way. In cases that the statements inside the foreach loop access only the iterator key, it generates a local broadcast (i.e., an empty mapped cells) for the Map function. For instance, in Query, we only access switch, which is the iterator key. As such, Query accesses the dictionary on a per-key basis. Thus, if we send the TimeOut message to all bees on the current hive, each bee will call Query that iterates on all the cells of that bee. This is the expected behavior for Query.

In complicated cases where the compiler does not have enough information to infer the keys (e.g., cross-library or recursive calls), it generates a Map function that results in a centralized application. This ensures the generated Map function is correct, but not necessarily efficient. To mitigate that, 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. This method is generic and widely applicable, but it has a higher cost compared to when the Map function is generated by the compiler or when it is explicitly provided by the application.

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 4.7, a message has the following life-cycle on 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 have a handler for 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 cells 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 accordingly 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 [96] among hives. Paying one Raft consensus overhead to create a bee can be quite costly. To avoid that, queen bees batch lock proposals (used to assign cells to bees) for messages that are already enqueued with no added latency or wait. This results in paying one Raft overhead.

\(^2\)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 in cases where the default local placement is not a good fit for that application’s design.
for each batch. When the queen bee sends a batch of lock proposals, each proposal can either succeed or fail. If a lock proposal fails, a bee on another hive has already locked the cells. In such cases, the queen bee relays the message to the remote bee. As shown in Figure 4.8, the average latency of creating a new bee can be as low as $140\mu s$ 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 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.

It is important to note that, in our control platform, we use an unreliable message delivery mechanism by default. For applications that require reliable message delivery, Beehive provides lightweight reliable delivery wrappers. These wrappers have their own overheads as they store their messages in a dictionary, exchange sequenced messages with other bees, acknowledge received messages, drop duplicate messages, and retransmit messages upon timeouts.

**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 `{(Switch, 1), (MAC, A1:...)}` and `{(Switch, 1), (Port, 2)}` respectively by an application. Since these two messages share the cell (Switch, 1), the platform guarantees that both messages are
handled by the same bee that owns the 3 cells of \(\{(\text{Switch}, 1), (\text{MAC}, A: \ldots), (\text{Port}, 2)\}\). 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 performance. Even though Beehive cannot automatically redesign message handlers (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 4.3.4.

**Conflicting Mapped Cells.** It is rare\(^3\) yet possible that the platform observes conflicting mapped cells. For example, consider an application that maps two messages to \(\{(\text{Mac}, \text{FF} \ldots)\}\) and \(\{(\text{Port}, 12)\}\). For these two mapped cells, Beehive will assign two distinct bees since they are not intersecting. Now, if the application maps another message to \(\{(\text{Switch}, 1), (\text{MAC}, \text{FF} \ldots), (\text{Port}, 12)\}\), we cannot assign this mapped cell to any of the existing bee, nor can start a new one. In such conflicting cases, before processing the conflicting message, the platform stops the bees with conflicting mapped cells, consolidates their dictionaries, and allocate a new bee that owns all the three cells in \(\{(\text{Switch}, 1), (\text{MAC}, \text{FF} \ldots), (\text{Port}, 12)\}\). If the bees are on different hives, the platform uses Beehive’s bee migration mechanism that is explained shortly.

**Auxiliaries.** In addition to the functions presented here, Beehive provides 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 (i.e., continuation): (i) Applications can send message \(M\) directly to the bee responsible for cell \((D, K)\) of Application \(A\) using \(\text{SendToCell}(M, A, (D, K))\). (ii) Using \(\text{Lock}(A, (D, K))\), a bee running application \(A\) can proactively lock the cell \((D, K)\). This is very useful when a bee needs to own a cell prior to receiving any message. (iii) Using \(\text{Snooze}(M, T)\) immediately exits the message handler and schedules

---

\(^3\)We have not observed this situation in our real-world use-cases.
message \( M \) to be redelivered after timeout \( T \). \( iv \) Applications can use the `DeferReply` utility to postpone replying to a message. This utility returns a serializable object that applications can store in their dictionaries, and later use to reply to the original message. This method is Beehive's way of providing continuation.

4.3.2 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 by 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 4.9, 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.\(^4\) In a colony, only the leader bee receives and processes messages, and followers act as purely passive replicas.

---
\(^4\)Applications can choose to skip replication and persistence.
Figure 4.9: 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.

**Replicating Transactions.** Bees inside the colony form a Raft [96] 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 ($\frac{1}{2} \text{replication\_factor} + 1$) 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 for real world applications. 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.

To evaluate the impact of batching on Beehive's throughput, we have implemented a simple key-value store conceptually similar to memcached [43] 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
Figure 4.10: The latency and throughput of the sample key-value store application: (a) Throughput using replication factors of 1 to 5, and batch sizes of 128 to 8k. (b) The PDF of the end-to-end latency for GET and PUT requests in a cluster of 20 hives and 80 clients.

and 5, and measured the throughput for batch sizes from 128 to 8192 messages. In each run, we emit 10M PUT (i.e., write) messages and measure the throughput. As shown in Figure 4.10a, 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 4.10b, despite the lack of an 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.

**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 create 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 4.11, 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.

Beehive’s failover time would not be adversely affected when we increase the number of hives
Figure 4.11: Beehive failover characteristics using different Raft election timeouts. In this experiment, we use the key-value store sample application with a replication factor of 3 on a 3-node cluster. Smaller election timeouts result in faster failover, but they must be chosen to avoid spurious timeouts based on the network characteristics.

or the colony size, since Raft elections will start according to the election timeout regardless of the quorum size. Moreover, Raft adds randomness in the election timeout to avoid having multiple candidates at the same time. This reduces the probability of unsuccessful elections.

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 a 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 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.
4.3.3 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 (i.e., have replication factors of one), 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.

4.3.4 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 datacenter to another, the control functions managing that virtual network should also be placed in the new datacenter 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 message handlers, their state, and the 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 of a switch and a virtual networking application managing a particular virtual network. This metric essentially indicates the correlation of each switch to each virtual network. We also store the provenance and causation data for messages. For example, in a learning switch application, we can report that most PacketOut messages are emitted by the learning switch application upon receiving PacketIn’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 4.12, Beehive’s runtime instrumentation imposes a very low overhead (less than 3%) 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.

4.3.5 Automatic Placement

By default, a hive which receives the first message for a specific cell successfully acquires the lock and creates a new local bee for that cell. At this point, all subsequent messages for the same mapped cells, will be processed on the newly created bee on this hive. This default behavior can perform well when most messages have locality (i.e., sources are in the close vicinity), which is the case for most networks [16]. Having said that, 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\(^5\) and we consider it outside the scope of this thesis. 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 have an open socket or file cannot be migrated, since Beehive cannot move the open sockets and files along with the bee. As a result, our greedy heuristic will move other bees close to the bees performing IO, if they exchange a lot

\(^5\)By reducing the facility location problem.
of messages. For instance, as we will demonstrate in Section 5.2, 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.

![Figure 4.13: Inter-hive traffic matrix of the elephant flow routing application (a) with an ideal placement, and (b) when the placement is automatically optimized. (c) Bandwidth usage of the control plane in (b).](image)

**Effectiveness of Automatic Placement.** To demonstrate the effectiveness of our greedy placement algorithm, we have run an experiment using our elephant flow routing example shown in Figure 4.4 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 elephant flow routing application. 10% of these flows have spikes (*i.e.*, detected as an elephant flow in `Collect`).

In the ideal scenario shown in Figure 4.13a, 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 runtime 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 \((S, SW_1, Stat_{SW_1})\) next to the OpenFlow driver that controls \(SW_1\). As shown in Figures 4.13b and 4.13c, this live migration of bees localizes message processing and results in a placement similar to Figure 4.13a. Note that this is all done automatically 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: \( \text{Place}(\text{cells}, \text{live\_hives}) \). 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.

4.3.6 Application Composition

Our approach to foster code reuse in the control platform is to exchange messages among applications. Although this approach results in a high degree of decoupling, sometimes network programmers need to compose applications with an explicit, predefined ordering. For example, for security reasons, one might want to compose an access control application with a learning switch in such a way that messages are always processed by the access control application first, and then relayed to the learning switch.

For such cases, Beehive provides generic composers that compose a set of control functions into a single composed function. The composed function isolates the dictionaries of different applications. The \text{Map} function of this composition returns the union of the mapped cells of the composed functions to preserve consistency.

In Beehive, we provide two generic composers:

1. \text{All}(f_1, f_2, \ldots) composes a set of message handlers as a sequence in such a way that the incoming message is passed to the next function only if the previous function has successfully handled the message. If there is an error in any function the transaction will be aborted.
2. \text{Any}(f_1, f_2, \ldots) composes a set of message handlers in such a way that the incoming message
Table 4.1: Sample use-cases implemented in Beehive and their code size

<table>
<thead>
<tr>
<th>Use-Case</th>
<th>Code Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hub</td>
<td>16 LOC</td>
</tr>
<tr>
<td>Kandoo [57]</td>
<td>47 LOC</td>
</tr>
<tr>
<td>Learning Switch</td>
<td>69 LOC</td>
</tr>
<tr>
<td>Elephant Flow Re-Routing</td>
<td>93 LOC</td>
</tr>
<tr>
<td>Key-Value Store</td>
<td>136 LOC</td>
</tr>
<tr>
<td>Message Queue</td>
<td>158 LOC</td>
</tr>
<tr>
<td>Pub-Sub</td>
<td>162 LOC</td>
</tr>
<tr>
<td>Network Virtualization</td>
<td>183 LOC</td>
</tr>
<tr>
<td>Network Discovery</td>
<td>239 LOC</td>
</tr>
<tr>
<td>Routing</td>
<td>385 LOC</td>
</tr>
<tr>
<td>ONIX NIBs</td>
<td>471 LOC</td>
</tr>
</tbody>
</table>

is passed to the next function only if the previous function could not successfully handle the message. In other words, Any guarantees that the side effect of at most one of these functions will be applied to the composed application’s state.

### 4.4 Applications

As shown in Table 4.1, various use-cases are implemented using Beehive with an effort comparable to existing centralized controllers. Note that these numbers include the generated Map functions. These numbers indicate that the developer efforts have been devoted to the core of the use-cases not the boilerplates of distributed programming, and the end result can be run as a distributed application.

Interestingly, quite a few of these use-cases are similar to well-known distributed architectures. For example, the key-value store application behaves similar to memcached [43] and our distributed graph processing application behaves similar to Pregel [83]. This implies that Beehive’s programming model would accommodate a variety of distributed programming flavors without hindering simplicity.

It is important to note that, among these applications, ONIX NIB is the largest project. The reason is that more than 60% of our code was devoted to implementing NIB’s object model
(e.g., node, links, paths, etc.) and their APIs. We expected any centralized implementation of ONIX NIBs to have the same amount of programming complexity. Moreover, we note that the active component that manages these NIB objects is less than 100 LOCs.

**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. The \texttt{Map} function of a centralized application should return the same value (i.e., the same key) for all messages. As discussed earlier in Section 4.3, for such a message handler, Beehive guarantees that all the messages are processed in one single 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).

```plaintext
1 App Centralized:
2   ...
3   \texttt{Map}(msg):
4     // Returning key 0 in dictionary D for all messages, this application will process all messages in the same thread (i.e., in the same bee).
5     // As such, this application becomes a centralized application.
6     return \{(D, 0)\}
```

Figure 4.14: A centralized application maps all messages to the same cell. This results in processing all messages in the same bee.

**Kandoo.** At the other end of the spectrum, there are local control applications proposed in Kandoo [57] that use the local state of a single switch to process frequent events. The message handlers of a local control application maintain their state on a per switch basis: they use switch IDs as the keys in their dictionaries and, to handle messages, access their state using a single key. As such, their \texttt{Map} function would map messages to a cell specific to that switch (Figure 4.15). As such, Beehive conceives one bee for each switch. If the application uses the default placement method, the bee will be placed on the hive that receives the first message from the switch. This effectively results in a local instance of the control application.

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 [57] where the developer has to decide on function placement (e.g., local controllers close to switches). The other difference is that Beehive requires one Raft proposal to lock the local cells for the local bee, while Kandoo does not incur any consensus overhead in the local controllers. This overhead is, however, negligible since it incurs only for the first message of each local bee.

```plaintext
1 App Local:
2 ... 
3 Map(msg):
   // Returning the switch ID as the mapped cell for each message, the
   // messages of each switch will be processed in their own thread
   // (i.e., the bee that exclusively owns the cell allocated for the
   // switch). Using the default placement method, the bee will be placed
   // on the first hive that receives a message from that switch. This
   // effectively results in a local Kandoo application.
   return {(D, msg.Switch)}
```

Figure 4.15: A local application maps all messages to a cell local to the hive (e.g., the switch's ID).

ONIX's NIB [75]. NIB is basically an abstract graph that represents networking elements and their interlinking. To process a message in 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 switch) and update messages (e.g., adding an outgoing link) on a particular node in NIB will be handled by its own bee in the platform. In Chapter 5, we present Beehive SDN controller that provides an abstraction similar to ONIX’s NIBs.

```plaintext
1 App NetworkManager:
2 ... 
3 Map(msg):
   // By mapping messages to their nodes, the network manager will maintain
   // the NIB on a per-node basis.
   return {(D, msg.NodeID)}
```

Figure 4.16: To manage a graph of network objects, one can easily map all messages to the node’s ID. This way nodes are spread among hives on the platform.
Logical xBars [86]. Logical xBar represents the control plane as a hierarchy in which each controller acts as a switch to its parent controller. To emulate such an architecture (and any similar hierarchical controller), we attach a hierarchical identifier to network messages and store the state of each xBar controller using the same identifiers as the keys in the application state. This means each controller in xBar hierarchy would have its own cell and its own bee in Beehive. Upon receiving a message, Beehive Map's the message to its cell using the message's hierarchical identifier and is hence processed by its respective bee.

Network Virtualization. Typically, network virtualization applications (such as NVP[74]) 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 colocated cells in Beehive and the platform guarantees that messages of the same virtual network are handled by the same bee.

```app
NetworkVirtualization:
...

Map(create CreatePort):
  // By mapping the create port message to the entry in the Ports dictionary and the virtual network entry in the VN dictionary, this port will always be processed in the bee that owns the virtual network.
  return {(Ports, create.PortID), (VN, create.VirtualNetworkID)}

Map(update PortStatusUpdate):
  return {(Ports, update.PortID)}

Map(msg VNMsg):
  return {(VN, msg.VirtualNetworkID)}
```

Figure 4.17: For virtual networks, messages of the same virtual network are naturally processed in the same bee.

4.5 Discussion

What are the main limitations of Beehive's programming model? As we mentioned in Section 4.2, in Beehive, applications cannot store persistent data outside the application
dictionaries. Since the value of an entry in a dictionary 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, in fact, 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 implementation and performance drawbacks [21, 32].

**How does Beehive compare with existing distributed programming frameworks?** There is a significant body of research on how to simplify distributed programming. Some proposals focus on remote invocations. This goes as early as CORBA to the programming environments such as Live Objects[98]. Moreover, special-purpose programming models, such as MapReduce [35], MPI, Cilk [19], Storm [1] and Spark [132] are proposed for parallel and distributed processing. Beehive has been inspired by the approach taken by these proposals to simplify distributed programming. Having said that, the focus and the applications of these proposals are different than what we need. Therefore, they cannot be used to as a basis to implement Beehive. For example, MapReduce and Spark are suitable for batch jobs and Storm is designed for one-way stream processing. In contrast, Beehive requires bi-directional communication among generic distributed applications handling a live workload.

There are Domain-Specific Languages (DSLs), such as BOOM [13] and Fabric [80], that propose new ways of programming for the purpose of provability, verifiability or security. In contrast to proposing a new DSL, our goal here 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 the familiar APIs provided by
Beehive's programming model.

**How does Beehive compare with existing distributed controllers?** Existing control planes such as ONIX [75] and ONOS [17] 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 existing distributed controllers such as Kandoo [57], ONIX [75], and Logical xBars [86] can all be implemented using Beehive. More importantly, Beehive's automatic placement optimization leads to results similar to what ElasticCon [37] achieve by load-balancing. Our optimization, however, is generic, is not limited to balancing the workload of OpenFlow switches, applies to all control applications, and can be extended to accommodate alternative optimization methods.

**How does Beehive compare with fault-tolerance proposals?** LegoSDN [28] is a framework to isolate application failures in a controller. Beehive similarly 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 [68] is a system that provides transparent slave-master fault-tolerance for centralized controllers. As demonstrated in Figure 5.12a, 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 [23] and [90], 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 [90] or the STN Middleware [23] 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.

4.6 Remarks

In this chapter, we have presented Beehive, a distributed control platform that is easy to use and instrument. Using a simple programming model, Beehive infers the state required to process a message, and automatically compiles applications into their distributed counterparts. By instrumenting applications at runtime, Beehive optimizes the placement of control applications, and provides feedback to the developer helping them in the design process. We have demonstrated that our proposal is able to model existing distributed control planes. Moreover, our evaluations confirm that this approach can be effective in designing scalable control planes. In the next chapter, we continue our evaluations with two real-world use cases of Beehive.
Chapter 5

Use Cases

In this chapter, we present a few real-world use cases of the programming models proposed in this thesis. We start with OpenTCP which is an effort to adapt different flavors of TCP based on the scalable statistics collection provided by Kandoo. We present our real world evaluations of OpenTCP in the SciNet HPC datacenter. We continue with a distributed SDN controller we have designed and implemented based on Beehive. We demonstrate that our controller is fault-tolerant and scalable, yet is implemented in a few hundred lines of code similar to centralized controllers. We conclude this section with a sample routing algorithm that we implemented as a proof of concept to showcase the generality of Beehive.

5.1 OpenTCP

The Transmission Control Protocol (TCP) [65] is designed to fully utilize the network bandwidth while keeping the entire network stable. TCP’s behaviour can be sub-optimal and even erroneous [119, 12, 30] mainly for two reasons: First, TCP is expected to operate in a diverse set of networks with different characteristics and traffic conditions; making TCP a “Jack of all trades, master of none” protocol. Limiting TCP to a specific network and taking advantage of local characteristics of that network can lead to major performance gains. For instance, DCTCP [12] outperforms TCP in datacenter networks, even though the results might not be
applicable in the Internet. With this mindset, one can adjust TCP (the protocol itself and its parameters) to gain a better performance in specific networks (e.g., datacenters).

Second, even focusing on a particular network, the effect of dynamic congestion control adaptation to traffic pattern is not yet well understood in today’s networks. Such adaptation can potentially lead to major improvements, as it provides another dimension that today’s TCP does not explore.

Clearly, adapting TCP requires meticulous consideration of the network characteristics and traffic patterns. The fact that TCP relies solely on end-to-end measurements of packet loss or delay as the only sources of feedback from the network means that TCP, at its best, has a very limited view of the network state (e.g., the trajectory of available bandwidth, congested links, network topology, and traffic). Thus, a natural question here is:

Can we build a system that observes the state and dynamics of a computer network and adapts TCP’s behaviour accordingly?

If the answer to this question is positive, it can simplify tuning different TCP parameters. It can also facilitate dynamically changing the protocol itself.

**OpenTCP.** Here we present OpenTCP [46] as a system for dynamic adaptation of TCP based on network and traffic conditions in SDN [88]. OpenTCP mainly focuses on internal traffic in SDN-based datacenters for four reasons: (i) the SDN controller already has a global view of the network (such as topology and routing information), (ii) the controller can collect any relevant statistics (such as link utilization and traffic matrix), (iii) it is straightforward to realize OpenTCP as a control application in SDNs, and (iv) the operator can easily customize end-hosts’ TCP stack.

Our preliminary implementation of OpenTCP has been deployed in SciNet, a high-performance computing (HPC) datacenter with ~4000 hosts. We use OpenTCP to tune TCP parameters in this environment to show how it can simplify the process of adapting TCP to network and traffic conditions. We also show how modifying TCP using OpenTCP can improve the network performance. Our experiments show that using OpenTCP to adapt init_cwnd
based on link utilization leads to up to 59% reduction in flow completion times.

**OpenTCP and related work.** OpenTCP is orthogonal to previous works improving TCP’s performance. It is not meant to be a new variation of TCP. Instead, it complements previous efforts by making it easy to switch between different TCP variants automatically (or in a semi-supervised manner), or to tune TCP parameters based on network conditions. For instance, one can use OpenTCP to either utilize DCTCP or CUBIC in a datacenter environment. The decision on which variant to use is made in advance through the congestion control policies defined by the network operator.

### 5.1.1 OpenTCP Architecture

OpenTCP collects data regarding the underlying network state (e.g., topology and routing information) as well as statistics about network traffic (e.g., link utilization and traffic matrix). Then, using this aggregated information and based on policies defined by the network operator, OpenTCP decides on a specific set of adaptations for TCP. Subsequently, OpenTCP sends periodic updates to the end-hosts that, in turn, update their TCP variant using a simple kernel module. Figure 5.1 presents the schematic view of how OpenTCP works. As shown, OpenTCP’s architecture has three main components that aptly fit into Kandoo’s design ($\S3$):
1. The Oracle is a non-local SDN control application that lies at the heart of OpenTCP and is deployed on the root controller. It collects information about the underlying network and traffic. Then, based on congestion control policies defined by the network operator, it determines the appropriate changes to optimize TCP. Finally, it distributes update messages to end-hosts.

2. Collectors are typical local applications that collect statistics from switches, deployed on local controllers to scale.

3. Congestion Control Agents (CCAs) are also local applications that receive update messages from the Oracle and are responsible for modifying the TCP stack at each host using a kernel module.

5.1.2 OpenTCP Operation

OpenTCP goes through a cycle composed of three steps: (i) data collection, (ii) optimization and CUE generation, and (iii) TCP adaptation. These steps are repeated at a pre-determined interval of $T$ seconds. To keep the network stable, $T$ needs to be several orders of magnitude slower than the network RTT. In the rest of this section, we will describe each of these steps in detail.

Data Collection. OpenTCP relies on two types of data for its operation. First, it needs information about the overall structure of the network, including the topology, link properties like delay and capacity, and routing information. Since this information is readily available in the SDN controller, the Oracle can easily access them. Second, OpenTCP collects statistics about traffic characteristics by periodically querying flow-tables in SDN switches. In order to satisfy the low overhead design objective, our implementation of OpenTCP relies on the Kandoo local controllers [57] to collect summarized link level statistics at scale. The operator can choose to collect only a subset of possible statistics to further reduce the overhead of data collection. Later, we will show how this is done by using congestion control policies.
Figure 5.2: The Oracle collects network state, and statistics through the SDN controller and switches. Combining those with the CCP defined by the operator, the Oracle generates CUEs which are sent to CCAs sitting on end-hosts. CCAs will adapt TCP based on the CUEs.

**CUE Generation.** After data collection, we must determine what changes to make to TCP sources. This decision might vary from network to network depending on the network operator’s preferences. In OpenTCP, we formalize these preferences by defining a *Congestion Control Policy (CCP)*. The network operator provides OpenTCP with the CCP. At a high level, CCP defines which statistics to collect, what objective function the operator should optimize, what constraints OpenTCP must satisfy, and how the proposed updates should be mapped onto specific changes in TCP sources. Based on the CCP defined by the network operator, OpenTCP finds the appropriate changes and determines how to adapt the individual TCP sessions. These changes are sent to individual CCAs in messages we call Congestion Update Epistles (CUEs). The CCAs which are present in individual end-hosts use these CUEs to update TCP. Figure 5.2 shows how CCPs are integrated into the TCP adaptation process.

**TCP Adaptation.** The final step in OpenTCP’s operation is to change TCP based on CUEs generated by the Oracle. After solving the optimization problem, the Oracle forms CUEs and sends them directly to CCAs sitting in the end-hosts. CUEs are usually very small packets. In our implementation, each CUE is usually less than 10–20 bytes. Thus, their overhead is very small.
CCAs are responsible for receiving and enforcing updates. Upon receiving a CUE, each CCA will check the condition given in each mapping rule. If this condition is satisfied, the CCA immediately applies the command. Changes can be simple adaptations of TCP parameters or can require switching between different variants of TCP. If the CCA does not receive any CUEs from the Oracle for a long time (we use $2T$ where $T$ is the OpenTCP operation cycle), or if the mapping rule conditions are violated, the CCA reverts all changes and OpenTCP goes back to the default TCP settings.

Adaptation example. Assume the operator simply wants to switch from TCP Reno to CUBIC while keeping an eye on packet drop rates. Here, we assume that the default TCP variant running in end-hosts is TCP Reno. There is a constraint limiting the link utilization to 80%. If the network has a low load (i.e., as long as all links have a utilization less than 80%) the Oracle will send CUEs to all CCAs instructing them to switch to CUBIC. If network conditions change and utilization goes above 80% in any link, the Oracle will not be able to satisfy the optimization constraints. In this case, the CCAs will fall back to the default TCP variant, which is TCP Reno here.

OpenTCP without SDN Support. OpenTCP is expressly designed for SDN. In fact, the existence of a Kandoo root controller with access to the global state of the network, as well as local controllers which can be used to collect flow and link level statistics, make SDN the perfect platform for OpenTCP. Having said that, we can use OpenTCP on a traditional network (non-SDN) as long as we can collect the required network state and traffic statistics. Clearly, this requires more effort and might incur a higher overhead since we do not have a centralized controller or built-in statistics collection features. In the absence of SDN switches, we can use a dedicated node in the network to act as the Oracle. Moreover, we can use CCAs to collect flow statistics (e.g., number of active flows, utilization and drop rate) at each end-host using a kernel module in a distributed manner and send relevant information to the Oracle. The Oracle can query CCAs to collect various statistics and if needed to aggregate them to match the statistics available in SDN. This needs to be done with extra care, so that the overhead remains low, and
the data collection system does not have a major impact on the underlying network traffic.

**Conditions and constraints for OpenTCP.** The current implementation of OpenTCP relies on a single administrative authority to derive network and traffic statistics. To this end, we also assume no adversary in the network and that we could change the end-host stacks. Hence, OpenTCP doesn’t have to operate in a datacenter per se, and any network that satisfies the above conditions can benefit from OpenTCP.

**Changing parameters in a running system.** Depending on the TCP parameter, making changes to TCP flows in an operational network may or may not have immediate changes in TCP behaviour. For example, adapting init_cwnd only affects new TCP flows but max_cwnd change is respected almost immediately. If the congestion window size is greater than the new max_cwnd, TCP will not drop those packets, it will deliver them and afterwards respects the new max_cwnd. Similarly, enabling TCP pacing will have an immediate effect on the delivery of packets. The operator should be aware of when and how each parameter setting will affect on going TCP sessions.

### 5.1.3 Evaluation

To evaluate the performance of OpenTCP, we deploy our proposal in a High Performance Computing (HPC) datacenter. SciNet is mainly used for scientific computations, large scale data analyses, and simulations by a large number of researchers with diverse backgrounds: from biological sciences and chemistry, to astronomy and astrophysics. Through a course of 20 days, we enabled OpenTCP and collected 200TB of packet level trace as well as flow and link level statistics using our `tcp_flow_spy` kernel module [8].

**Setup.** The topology of SciNet is illustrated in Figure 5.3. There are 92 racks of 42 servers. Each server connects to a Top of Rack switch (ToR) via 1Gbps Ethernet. Each end-host in SciNet runs Linux 2.6.18 with TCP BIC [52] as the default TCP. SciNet consists of 3,864 nodes with a total of 30,912 cores (Intel Xeon Nehalem) at 2.53GHz, with 16GB RAM per node. The ToR switches are connected to a core Myricom Myri-10G 256-Port 10GigE Chassis each having a
Figure 5.3: The topology of SciNet, the datacenter network used for deploying and evaluating OpenTCP.

10Gbps connection.

**OpenTCP in SciNet.** We did not have access to a large scale SDN and could not change any hardware elements in SciNet. Therefore, we took the extra steps to deploy OpenTCP without SDN. In our experiments, the CCAs periodically collect socket-level statistics, aggregate them, and send the results to the Oracle, a dedicated server in SciNet using the `tcp_flow_spy` kernel module. The Oracle performs another level of aggregation to produce final statistics combining data from the CCAs. Throughout our experiments, we monitor OpenTCP to ensure that we do not have a major CPU or bandwidth overhead. The average CPU overhead of our instrumentation is less than 0.5%, and we have a negligible bandwidth requirement. We believe this is a reasonably low overhead, one which is unlikely to have a significant impact on the underlying jobs and traffic.

**Evaluation overview.** We begin this section by describing the workload in SciNet (§ 5.1.3). We show that, despite the differences in the nature of the jobs running in SciNet, the overall properties of the network traffic, such as flow sizes, flow completion times (FCT), traffic locality, and link utilizations are analogous to those found by previous studies of datacenter traffic [53, 12, 48]. We take this step to ensure our results are not isolated or limited to SciNet.
Traffic Characterization. At SciNet, the majority of jobs are Message Passing Interface (MPI) programs running on one or more servers. Jobs are scheduled through a queuing system that allows a maximum of 48 hours per job. Any job requiring more time must be broken into 48 hour chunks. Note that each node runs a single job at any point of time. We use OpenTCP to collect traces of flow and link level statistics, capturing a total of ∼ 200TB of flow level logs over the course of 20 days.

Workload. We recognize two types of co-existing TCP traffic in our traces (shown in Figure 5.4): (i) MPI traffic and (ii) distributed file system flows. The majority of MPI flows are less than 1MB in size and finish within 1 second. This agrees with previous studies of datacenter traffic [12, 16]. It is interesting to note that only a few MPI flows (1%) last up to the maximum job time allowance of 48 hours. Unlike the MPI flows, the distributed file system traffic ranges from 20B to 62GB in size, and most (93%) finish within 100 seconds. A few large background jobs last more than 6 days.

Locality. Figure 5.5 represents the log of the number of bytes exchanged between server pairs in 24 hours. Element $(i, j)$ in this matrix represents the amount of traffic that host $i$ sends to host $j$. 
Figure 5.5: The locality pattern as seen in matrix of log(number of bytes) exchanged between server pairs.

The nodes are ordered such that those within a rack are adjacent on the axes. The accumulation of dots around the diagonal line in the figure shows the traffic being locally exchanged among servers within a rack, a pattern that is similar to previous measurements [16, 53]. Note that this matrix is not symmetrical as it depicts the traffic from nodes on the x-axis to nodes on the y-axis. The vertical lines in the traffic matrix represent “token management traffic” belonging to the distributed file system. This type of traffic exchanges tokens between end-hosts and a small set of nodes called token managers. Finally, the rectangular regions in the figure represent traffic between the nodes participating in multi-node MPI jobs.

Utilization. We observe two utilization patterns in SciNet. First, 80% of the time, link utilizations are below 50%. Second, there are congestion epochs in the network where both edge and core links experience high levels of utilization and, thus, packet losses. Again, this observation accords with the previous studies where irrespective of the type of datacenter, link utilizations are known to be low [16, 53].

Experiment Methodology. We run a series of experiments to study how OpenTCP works in practice, using a combination of standard benchmarks and sample user jobs as the workload. During each experiment, we use OpenTCP to make simple modifications to end-hosts’ congestion control schemes, and measure the impact of these changes in terms of flow completion times and packet drop rates. Throughout our experiments, we set OpenTCP’s slow

\footnote{Virtualization is not used in this cluster.}
time scale to 1min, unless otherwise stated\(^2\) directing OpenTCP to collect statistics and send CUEs once every 1min. We also measure the overhead of running OpenTCP on CPU usage and bandwidth. In our experiments, we have used 40 nodes in 10 ToR racks and we run all the benchmarks explained above in over 10 iterations.

To evaluate the impact of the changes made by OpenTCP, we symmetrically split our nodes into two sets. Both sets run similar jobs, but we deploy OpenTCP on one of these two sets only (i.e., half the nodes). Because the two sets are comparable, we can observe the impact of OpenTCP with minimum influence from other random effects.

**Benchmarks.** In order to study different properties of OpenTCP, we use Intel MPI Benchmarks (IMB) [9] as well as sample user jobs in SciNet. These benchmarks cover a wide range of computing applications, have different processing and traffic load characteristics, and stress the SciNet network in multiple ways. We also use sample user jobs (selected from the pool of jobs submitted by real users) to have an understanding of the behaviour of the system under typical user generated workload patterns. Figure 5.6 shows the CDF of flow sizes for these benchmarks. As illustrated in the figure, flow sizes range from 50B to 1GB, much like those observed in § 5.1.3.

The results presented here are for “all-to-all” IMB benchmark and “Flash” user job. We measured similar consistent results with other IMB benchmarks as well as other sample user jobs.

\(^2\)In this environment, the fast time scale (RTT) is less than 1ms.
Chapter 5. Use Cases

Figure 5.7: Comparing the CDF of FCT (a) and drop rate (b) of OpenTCP₁, OpenTCP₂, and regular TCP while running the “all-to-all” IMB benchmark.

Experiment Results. To run our OpenTCP experiments, we first define two congestion control policies for OpenTCP: OpenTCP₁ and OpenTCP₂. We then compare the performance of OpenTCP₁, OpenTCP₂, and unmodified TCP in terms of Flow Completion Times (FCT) and packet drop rates using different benchmarks. Finally, we take a closer look at the overheads of OpenTCP and the impact of changing the refresh rate.

OpenTCP₁ – Minimize FCTs. Our first experiment with OpenTCP aims at reducing FCTs by updating the initial congestion window size, and the retransmission time-out (RTO). In SciNet, there is a strong correlation between MPI job completion times and the tail of FCT. Therefore, reducing FCTs will have a direct impact on job completion times.

OpenTCP₁ collects link utilizations, drop rate, ratio of short-lived to long-lived flows and number of active flows. The optimization function is to find the appropriate value for the initial
congestion window size \( w_0 \) as long as the maximum link utilization is kept below 70\%. If the link utilization goes above 70\% for any of the links, the Oracle will stop sending CUEs and the CCAs will fall back to the default values of the initial congestion window size and the RTO.

**OpenTCP\(_2\) – Minimize FCTs, limit drops.** While OpenTCP\(_1\) aims at improving FCTs by adapting \texttt{init\_cwnd} and ignoring packet drops, we define a second variant called OpenTCP\(_2\) which improves upon OpenTCP\(_1\) by keeping the packet drop rate below 0.1\%. This means that when the CCA observes a packet drop rate above 0.1\% it reverts the initial congestion window size and the RTO to their default values. This ensures that when the CCA observes a packet drop rate above 0.1\% it returns the initial congestion window size and the RTO to their default values.

**Impact on flow completion times.** Figure 5.7 depicts the CDF of flow completion times for OpenTCP\(_1\), OpenTCP\(_2\), and unmodified TCP. Here, the tail of the FCT curve for OpenTCP\(_1\) is almost 64\% shorter than for the unmodified TCP. As mentioned above, this leads to faster job completion in our experiments. Additionally, more than 45\% of the flows finish in under 260 seconds in OpenTCP\(_1\), indicating a 62\% improvement over unmodified TCP. Similarly, OpenTCP\(_2\) helps 80\% of the flows finish in a fraction of a second, a significant improvement over OpenTCP\(_1\). This is because of OpenTCP\(_2\)'s fast reaction to drop rate at the end-hosts. In OpenTCP\(_2\), the tail of the FCT curve is 324 seconds, a 59\% improvement over unmodified
TCP. The FCT tail of OpenTCP₂ is slightly (8%) longer than OpenTCP₁ since it incorporates conditional CUEs and does not aggressively increase the initial congestion window. Moreover, note that the big gap between FCT of TCP and OpenTCP variants is due to nature of all-to-all benchmark that is stressing the SciNet network. In this benchmark, every process is sending and receiving to/from all other processes. This creates a TCP map-behaviour similar to TCP incast throughput collapse [119].

**Impact on congestion window size.** The improvements in FCTs are a direct result of increasing $init_{cwnd}$ in OpenTCP₁. Figure 5.8 presents a histogram of congestion window sizes for OpenTCP₁ and unmodified TCP. In the case of unmodified TCP, more than 90% of packets are sent when the source has a congestion window size of three, four, or five segments. OpenTCP₁, however, is able to operate at a larger range for congestion window sizes: more than 50% of packets are sent while the congestion window size is greater than five segments.

**Impact on packet drops.** The FCT improvements in OpenTCP₁ and OpenTCP₂ come at a price. Clearly, operating at larger congestion window sizes will result in a higher probability of congestion in the network, and thus, may lead to packet drops. As Figure 5.9 shows, the drop rate distribution for unmodified TCP has a shorter tail compared to OpenTCP₁ and OpenTCP₂. As expected, OpenTCP₁ introduces a considerable amount of drops in the network since the CUEs are not conditional and thus the CCAs do not react to packet drops. OpenTCP₂ has no drops for more than 81% of the flows. This is a significant improvement over unmodified TCP. However, the highest 18% of drops in OpenTCP₂ are worse than those in unmodified TCP, as OpenTCP₂ needs to observe packet drops before it can react. Some flows will take the hit in this case, and might end up with relatively large drop rates.

**The Flash Benchmark.** All the experiments described so far use the “all-to-all” IMB benchmark. The Flash benchmark is a selection of user jobs that we use to evaluate OpenTCP under realistic workloads. Flash solves the Sedov point-blast wave problem [109], a self-similar description of the evolution of a blast wave arising from a powerful explosion. The Flash benchmark represents a common implementation and traffic patterns in SciNet; it generates
Figure 5.9: Comparing the CDF of FCT (a) and drop rate (b) of OpenTCP₁, OpenTCP₂, and regular TCP while running the “Flash” benchmark.

about 12,000 flows. Figure 5.9 compares the CDF of flow completion times and drop rates for OpenTCP₁, OpenTCP₂, and unmodified TCP. Like the results presented above, both OpenTCP₁ and OpenTCP₂ improve the tail of the FCT curve by at least 40%. But, OpenTCP₂ is more successful at reducing the drop rate as it has an explicit rule for controlling the drops.

**OpenTCP has low overhead.** Table 5.1 divides OpenTCP’s overhead into four categories: (i) the CPU overhead associated with data aggregation and CUE generation at the Oracle, (ii) the CPU overhead associated with data collection and CUE enforcement in the end-hosts, (iii) the required bandwidth to transfer statistics from the end-hosts to the Oracle, and (iv) the required bandwidth to disseminate CUEs to the end-hosts. We summarize OpenTCP’s overhead for refresh periods of 1, 5, and 10 minutes in Table 5.1. The table shows that OpenTCP has a negligible processing and bandwidth overhead, making it easy to scale OpenTCP to large clusters.
Table 5.1: The approximate OpenTCP overhead for a network with ~4000 nodes and ~100 switches for different Oracle refresh cycles.

<table>
<thead>
<tr>
<th></th>
<th>1 min</th>
<th>5 min</th>
<th>10 min</th>
</tr>
</thead>
<tbody>
<tr>
<td>Oracle CPU Overhead (%)</td>
<td>0.9</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>CCA CPU Overhead (%)</td>
<td>0.5</td>
<td>0.1</td>
<td>0.0</td>
</tr>
<tr>
<td>Data Collection (Kbps)</td>
<td>453</td>
<td>189</td>
<td>95</td>
</tr>
<tr>
<td>CUE Dissemination (Kbps)</td>
<td>530</td>
<td>252</td>
<td>75</td>
</tr>
</tbody>
</table>

of hundreds of thousands of nodes. Clearly, OpenTCP’s refresh cycle plays a critical role in its overhead on networking elements.

**Stability.** To realize OpenTCP in practical settings, we assume that the network operator has expertise in defining the congestion control policies appropriate for the network. Moreover, to achieve pragmatic stability, there should be a monitoring system in place which alerts the operator of churns and instabilities in the network. This monitoring component should alert the controller whenever there is an oscillation between states. For example, if Oracle is making adjustments to TCP flows in time $t_1$ and immediately in time $t_1 + T$ those changes are reverted back, it is a good indication of a potential unstable condition. In this case, the operator should be notified to adjust either the congestion control policies, the stability constraints, or the overall resources in the network. One simple stability metric is number of times a rule is applied and reverted. The monitoring system can measure such stability metrics and alert the operator.

On top of adapting TCP parameters based on network and traffic conditions, OpenTCP can go beyond to include application semantics into its decision making process. For example, the target video streaming rate for video servers can be fed to OpenTCP’s Oracle. Another application semantic could be file sizes in a transfer that OpenTCP can use to derive hints for TCP sessions.
5.2 Beehive SDN Controller

Beehive is a generic platform for distributed programming. As such, Beehive provides the building blocks for an SDN controller, not the specific functionalities of a controller. That is, in our design, the SDN controller is an application (and perhaps the most important use-case) of Beehive. In this section, we discuss the SDN controller we have built on top of Beehive to showcase the effectiveness of our proposal for SDN. It is important to note that the controller we present here is only one possible design, and Beehive can be used as a framework to develop other SDN control plane abstractions.

**SDN Controller Suite.** We have designed and implemented an SDN abstraction on top of Beehive. Our implementation is open-source. 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 5.10.

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): 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.

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 a scalable replacement for polling 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. 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 among them). 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 configurations 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 a distributed application 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 basically a poller that removes 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
Figure 5.11: 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.
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 channel overheads.

**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 persistence, fault-tolerance, runtime instrumentation, and optimized placement, readily available in Beehive. We admit that the lines of code, although commonly used, is not a perfect representative of application complexity. Finding a proper complexity metric for control applications is beyond the scope of this thesis and we leave it to future works.

**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, the controller sends one ping each 100ms). If a driver does not respond to a few (3 by default) consecutive node controller's 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 a hive that hosts the node controller fails, the platform will automatically delegate the node to a follower bee on another hive. 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 `PacketOuts`. 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.

Before presenting the results, let us first clarify what happens for the control applications upon a failure. As shown in Figure 5.11a, 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 5.11b new masters may be both collocated on the same hive, or (ii) as shown in Figure 5.11c, new masters will be on different hives.

As portrayed in Figure 5.11d, when the new masters are elected from the same hive the latency after the failover 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 5.11e). This is where the automatic placement optimization can be of measurable help. As shown in Figure 5.11f, 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 4.11, 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 5.12a). 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 will perform consistently. This is an important

---

\(^3\)We cannot prefer one bee over another bee in a colony in election time since it will obviate the provable convergence of Raft.
property of Beehive. As shown in Figure 5.12b, 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. We stress each hive by one cbench process. In these experiments, we use 5 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.

![Figure 5.12](image.png)

**Figure 5.12:** 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.

As shown in Figure 5.13, 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. Moreover, Beehive's throughput is significantly higher than ONOS 1.2 [17] for the same forwarding functionality. This experiment demonstrates that, although Beehive is easy-to-use, it can scale considerably well. In essence, hiding the boilerplates of distributed programming enables us to systematically optimize the control plane for all applications. In these experiments, ONOS replicates the forwarding application on all nodes. As a result, unlike Beehive, we cannot use a replication factor smaller than the cluster size. For example, when we use a 5-node cluster, ONOS uses all the five nodes and we cannot run an experiment with a replication factor of 3.

Figure 5.13: Throughput of Beehive's learning switch application and ONOS's forwarding application in different cluster sizes. The number of nodes (i.e., the x-axis) in this graph represents the nodes on which we launch cbench. For Beehive, we always use a 5-node cluster. For ONOS, the cluster size is equal to the replication factor since all nodes in the cluster will participate in running the application.

5.3 Simple Routing in Beehive

Routing mechanisms can be more challenging to implement on existing distributed control platforms compared to a centralized controller, especially if we want to realize a scalable routing algorithm that requires no manual configuration and is not topology-specific. In this section, we demonstrate how simple it is to design and implement a generic k-shortest path algorithm using Beehive.

Distributed Shortest Path. Suppose we already have a topology discovery algorithm that
automatically emits a Discovery message whenever an edge is found in the graph. To calculate the shortest paths in a distributed fashion, we employ an algorithm that is very similar to traditional distance vector routing protocols. As such, we will emit routing advertisement messages to signal that we have found a new path towards a given destination. The routing advertisements are in the form of \( R = (N_1, N_2, D) \), which means \( N_1 \) has a route to \( N_2 \) with a length of \( D \). Whenever, a node \( N \) receives an advertisement, it emits further advertisements by adding its neighbors to the route iff the received advertisement is a shortest path. For example, suppose \( N_2, N_3 \) and \( N_4 \) are the neighbors of \( N_1 \). When \( N_1 \) receives a shortest route \( R = (N_1, A, D) \) to \( A \), it emits \((N_2, A, D + 1)\), \((N_3, A, D + 1)\), and \((N_4, A, D + 1)\). This process would continue until the network converges.

In such a setting, the state of our application is accessed on a per-node basis. That is, as shown in Figure 5.14, we have a dictionary that stores the neighbors of each node, and a dictionary that stores all the routes initiated from each node. Such a simple application can be easily distributed on several hives to scale.

Convergence Evaluation. We have implemented this routing algorithm using Beehive in a few hundred lines of code and evaluated our implementation using a simulated network. In these simulations, we assume a k-ary fat-tree network topology with homogenous switches of 4, 8,
16, 32, 64, and 128 ports (respectively with 8, 16, 128, 512, 2k, 8k edge switches connecting 16, 128, 1k, 8k, 65k, 524k end-hosts). For the purpose of simulation, we use two auxiliary control applications: one randomly emits discovery events based on the simulated network topology, and the other receives and calculates the statistics for rule installation commands from the routing module. We define the convergence time as the difference between the first discovery message and the last rule installation. That is the time it takes to provide full multi-path connectivity between every single two edge switches in the network.

Using a cluster of six controllers, each allocated four cores and 10GB of memory, we measured the convergence time (i) when the platform does a cold start (i.e., has no information about the network topology), (ii) when a new POD is added to the network and (iii) when a new link is added to the network. In all experiments, each controller controls an equal number of core switches and PODs.

Note that, we have \(O(k^2)\) alternative paths between two edge switches in a k-ary fat-tree network and, hence, \(O(k^6)\) alternative paths in total. When we use high-density switches with 64 or 128 ports in a fat-tree network, the number of Route messages exchanged and the memory to store the routes is well beyond the capacity of a single commodity VM. This is where Beehive’s control plane demonstrates its scalability.

As shown in Figure 5.15a, considering the size of the network, this algorithm convergences in a reasonable time from a cold state. Moreover, when we add a new POD, the system converges in less than a second for switches of 32-ports or less (Figure 5.15b). More interestingly, as shown in Figure 5.15c, when we randomly add a new link, the system converges in less than a second, even for larger networks. By allocating more resources for Beehive, one can seamlessly improve these results with no modifications in the routing application. To demonstrate this, we conducted another set of experiments for a fat-tree network built from 64-port switches. As shown in Figure 5.15d, Beehive’s control plane seamlessly scales as we add more controllers and the convergence time continually diminishes. We note that, if additional computing resources are available, adding a new controller requires no additional configuration and is practically
In this section, we presented real-world use-cases of our proposals. OpenTCP, the Beehive SDN controller, and the simple routing algorithm demonstrate that Kandoo and Beehive are quite scalable, despite their simplicity. It is important to note that our proposals are not limited to these specific usecases and can be applied to a wide range of problems beyond SDN applications. For instance, Beehive can be utilized to implement a key-value store and a message queue that are scalable, distributed, and automatically optimized.

5.4 Remarks

Figure 5.15: Convergence time of the flat routing algorithm in a k-ary fat-tree topology.
Chapter 6

Conclusion and Future Directions

In this thesis, we have deconstructed several scalability concerns of SDN and proposed two distributed programming models, Kandoo and Beehive, that can scale well while preserving simplicity. We showed that some of the presumed limitations of SDN are not inherent and only stem from the historical evolution of OpenFlow. Using the real-world use cases of OpenTCP and the Beehive SDN controller, we demonstrated that such limitations can be easily overcome using an aptly designed software platform that is able accommodate various control functions. In essence, with a well-designed distributed network programming paradigm, we are able to preserve the promised simplicity of SDN at scale.

The two-layer hierarchy of controllers in Kandoo (Chapter 3) gives network programmers the option of choosing between local and network-wide state to implement different control functionalities. Local controllers, without the need for a centralized network view, have high parallelism and can be placed close to forwarding elements to shield the control channels from the load of frequent events. Non-local applications, on the other hand, are deployed in the root controller and can access the network-wide state. Employing Kandoo, network programmers merely provide a flag that whether an application is local or non-local, and the rest is seamlessly handled by Kandoo. As such, Kandoo is very simple to use, yet scales an order of magnitude better than centralized controllers.
Kandoo enables network programmers to implement applications such as elephant flow rerouting and OpenTCP (Chapter 5) in a simple way with low overheads. For example, OpenTCP can collect the traffic matrix in a large-scale HPC datacenter without hindering the performance of MPI jobs, and can adapt TCP to optimize the network. As we demonstrated, OpenTCP can considerably improve flow completion times, which interestingly improves the perceived end-to-end performance of the MPI jobs.

The generic programming model proposed in Beehive (Chapter 4) has a wider application domain than Kandoo and can support different types of functions in addition to local and centralized applications. Using Beehive's programming model, applications are written as centralized message handlers that store their state in dictionaries. Applications written in that programming model are automatically transformed (either at compile-time or at runtime) into their distributed counterparts while preserving their intended behavior. Beehive hides many boilerplates of distributed programming such as concurrency, synchronization, and replication. With runtime instrumentation and placement optimization, Beehive is capable of automatically improving the performance of the control plane.

Beehive is expressive and can be used to implement other distributed controllers including Kandoo. Using several SDN and non-SDN use cases, such as a key-value store and an SDN controller (Chapter 5), we demonstrated that real-world, distributed applications can be implemented in Beehive with an effort comparable to centralized controllers. Despite their simplicity, the throughput and the latency of the implemented applications are on par with existing distributed controllers. More importantly, Beehive instruments applications at runtime and automatically optimizes the placement of control applications. Using Beehive SDN controller, we have demonstrated that this automatic optimized placement results in significant improvements in the perceived latency of the controller without any intervention by network programmers.

Future Direction. Over the course of this research, we have encountered interesting open problems that have not been addressed in this thesis. We believe addressing the following open
problems can result in valuable and impactful research:

1. **Alternative Optimized Placement Methods**: In Beehive, we presented a greedy algorithm that tries to optimize the control plane’s latency by reducing the volume of inter-hive traffic. Although this is very effective for forwarding SDN applications (as shown in Chapter 5), there are optimization approaches that can be exploited for other types of applications. For example, another potentially useful placement method is to minimize the tail latency of control applications. That objective can indeed increase the inter-hive traffic, but may result in better performance for some specific applications. Moreover, extending Beehive’s runtime instrumentation, one can provide richer information to the placement and the optimization methods.

2. **Acceleration**: Porting Kandoo and Beehive runtimes to switches and co-processors in forwarding hardware enables us to push the control functions as close as possible to the data path. This can result in considerable performance improvements and can widen the application range of our proposals. It is important to note that this does not result in coupling the control plane back with the data plane. Instead, we still keep the control plane decoupled, yet provide an option of running code closest to the data path. That is, the control plane is still decoupled from the data plane, but is placed directly on the forwarding element. We do not aim at packet processing in the control plane, but with this acceleration we can process more frequent events at scale.

Another approach to accelerate network programs is to compile local control application (either Kandoo local applications or Beehive application with local mapped cells) onto programmable hardware (e.g., NetFPGA [94]). This cross compilation will probably need a limited programming abstraction, but can result in a system that is capable of providing scalable packet processing. More importantly, with such a cross compilation, Beehive can automatically choose the placement of these functions and can seamlessly optimize the network.
3. **Verification:** Given that Beehive and Kandoo both have well-defined structures for control application, it is feasible to develop static analyzers for control applications that are able to detect bugs or potential performance bottlenecks. We developed such an analyzer to automatically generate Map functions based on message handlers. This static analyzer can be extended to detect inter-dependencies of message handlers and to provide the cases in which an application will become centralized. Such a static analyzer can also shed light on potential performance problems and bugs (e.g., deadlocks) without the need to test the application at runtime.
Bibliography


Bibliography


