Onix: A Distributed Control Platform for Large-scale Production Networks

Teemu Koponen*, Martin Casado*, Natasha Gude*, Jeremy Stribling*, Leon Poutievski†,
Min Zhu†, Rajiv Ramanathan†, Yuichiro Iwata‡, Hiroaki Inoue‡, Takayuki Hama‡, Scott Shenker§

Abstract

Computer networks lack a general control paradigm, as traditional networks do not provide any network-wide management abstractions. As a result, each new function (such as routing) must provide its own state distribution, element discovery, and failure recovery mechanisms. We believe this lack of a common control platform has significantly hindered the development of flexible, reliable and feature-rich network control planes.

To address this, we present Onix, a platform on top of which a network control plane can be implemented as a distributed system. Control planes written within Onix operate on a global view of the network, and use basic state distribution primitives provided by the platform. Thus Onix provides a general API for control plane implementations, while allowing them to make their own trade-offs among consistency, durability, and scalability.

1 Introduction

Network technology has improved dramatically over the years with line speeds, port densities, and performance/price ratios all increasing rapidly. However, network control plane mechanisms have advanced at a much slower pace; for example, it takes several years to fully design, and even longer to widely deploy, a new network control protocol.[1] In recent years, as new control requirements have arisen (*e.g.*, greater scale, increased security, migration of VMs), the inadequacies of our current network control mechanisms have become especially problematic. In response, there is a growing movement, driven by both industry and academia, towards a control paradigm in which the control plane is decoupled from the forwarding plane and built as a distributed system.[2]

In this model, a network-wide control platform, running on one or more servers in the network, oversees a set of simple switches. The control platform handles state distribution – collecting information from the switches and distributing the appropriate control state to them, as well as coordinating the state among the various platform servers – and provides a programmatic interface upon which developers can build a wide variety of management applications. (The term "management application" refers to the control logic needed to implement management features such as routing and access control.)[3] For the purposes of this paper, we refer to this paradigm for network control as *Software-Defined Networking* (SDN).

This is in contrast to the traditional network control model in which state distribution is limited to link and reachability information and the distribution model is fixed. Today a new network control function (*e.g.*, scalable routing of flat intra-domain addresses [21]) requires its own distributed protocol, which involves first solving a hard, low-level design problem and then later overcoming the difficulty of deploying this design on switches. As a result, networking gear today supports a baroque collection of control protocols with differing scalability and convergence properties. On the other hand, with SDN, a new control function requires writing control logic on top of the control platform's higher-level API; the difficulties of implementing the distribution mechanisms and deploying them on switches are taken care of by the platform. Thus, not only is the work to implement a new control function reduced, but the platform provides a unified framework for understanding the scaling and performance properties of the system.

Said another way, the essence of the SDN philosophy is that basic primitives for state distribution should be implemented once in the control platform rather than separately for individual control tasks, and should use well-known and general-purpose techniques from the distributed systems literature rather than the more specialized algorithms found in routing protocols and other network control mechanisms. The SDN paradigm allows network system implementors to use a single control platform to implement a range of control functions (*e.g.*, routing, traffic engineering, access control, VM migration) over a spectrum of control granularities (from individual flows to large traffic aggregates) in a variety of contexts (*e.g.*, enterprises, datacenters, WANs).

*Nicira Networks
†Google
‡NEC
§International Computer Science Institute (ICSI) & UC Berkeley

[1] See, for example, TRILL [32], a recent success story which has been in the design and specification phase for over 6 years.

[2] The industrial efforts in this area are typically being undertaken by entities that operate large networks, not by the incumbent networking equipment vendors themselves.

[3] Just to be clear, we only imagine a single "application" being used in any particular deployment; this application might address several issues, such as routing and access control, but the control platform is not designed to allow multiple applications to control the network simultaneously (unless the network is "physically sliced" [28]).

Because the control platform simplifies the duties of both switches (which are controlled by the platform) and the control logic (which is implemented on top of the platform) while allowing great generality of function, the control platform is the crucial enabler of the SDN paradigm. The most important challenges in building a production-quality control platform are:

- *Generality:* The control platform's API must allow management applications to deliver a wide range of functionality in a variety of contexts.
- *Scalability:* Because networks (particularly in the datacenter) are growing rapidly, any scaling limitations should be due to the inherent problems of state management, not the implementation of the control platform.
- *Reliability:* The control platform must handle equipment (and other) failures gracefully.
- *Simplicity:* The control platform should simplify the task of building management applications.
- *Control plane performance:* The control platform should not introduce significant additional control plane latencies or otherwise impede management applications (note that forwarding path latencies are unaffected by SDN). However, the requirement here is for adequate control-plane performance, not optimal performance. When faced with a tradeoff between generality and control plane performance, we try to optimize the former while satisficing the latter.[4]

While a number of systems following the basic paradigm of SDN have been proposed, to date there has been little published work on how to build a network control platform satisfying all of these requirements. To fill this void, in this paper we describe the design and implementation of such a control platform called Onix (Sections 2-5). While we do not yet have extensive deployment experience with Onix, we have implemented several management applications which are undergoing production beta trials for commercial deployment. We discuss these and other use cases in Section 6, and present some performance measures of the platform itself in Section 7.

Onix did not arise *de novo*, but instead derives from a long history of related work, most notably the line of research that started with the 4D project [15] and continued with RCP [3], SANE [6], Ethane [5] and NOX [16] (see [4,23] for other related work). While all of these were steps towards shielding protocol design from low-level details, only NOX could be considered a control platform offering a general-purpose API.[5] However, NOX did not adequately address reliability, nor did it give the application designer enough flexibility to achieve scalability.

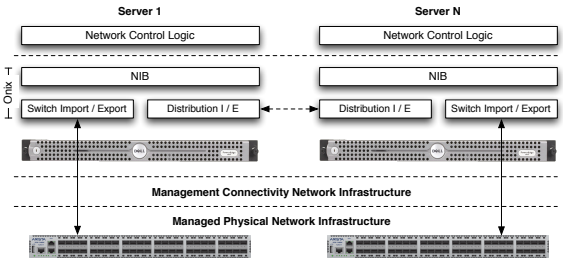

Figure 1: There are four components in an Onix controlled network: managed physical infrastructure, connectivity infrastructure, Onix, and the control logic implemented by the management application. This figure depicts two Onix instances coordinating and sharing (via the dashed arrow) their views of the underlying network state, and offering the control logic a read/write interface to that state. Section 2.2 describes the NIB.

The primary contributions of Onix over existing work are thus twofold. First, Onix exposes a far more general API than previous systems. As we describe in Section 6, projects being built on Onix are targeting environments as diverse as the WAN, the public cloud, and the enterprise data center. Second, Onix provides flexible distribution primitives (such as DHT storage and group membership) allowing application designers to implement control applications without re-inventing distribution mechanisms, and while retaining the flexibility to make performance/scalability trade-offs as dictated by the application requirements.

2 Design

Understanding how Onix realizes a production-quality control platform requires discussing two aspects of its design: the context in which it fits into the network, and the API it provides to application designers.

2.1 Components

There are four components in a network controlled by Onix, and they have very distinct roles (see Figure 1).

- *Physical infrastructure:* This includes network switches and routers, as well as any other network elements (such as load balancers) that support an interface allowing Onix to read and write the

[4]There might be settings where optimizing control plane performance is crucial. For example, if one cannot use backup paths for improved reliability, one can only rely on a fine-tuned routing protocol. In such settings one might not use a general-purpose control platform, but instead adopt a more specialized approach. We consider such settings increasingly uncommon.

[5]Only a brief sketch of NOX has been published; in some ways, this paper can be considered the first in-depth discussion of a NOX-like design, albeit in a second-generation form.

state controlling the element's behavior (such as forwarding table entries). These network elements need not run any software other than that required to support this interface and (as described in the following bullet) achieve basic connectivity.

- *Connectivity infrastructure:* The communication between the physical networking gear and Onix (the "control traffic") transits the connectivity infrastructure. This control channel may be implemented either in-band (in which the control traffic shares the same forwarding elements as the data traffic on the network), or out-of-band (in which a separate physical network is used to handle the control traffic). The connectivity infrastructure must support bidirectional communication between the Onix instances and the switches, and optionally supports convergence on link failure. Standard routing protocols (such as IS-IS or OSPF) are suitable for building and maintaining forwarding state in the connectivity infrastructure.

- *Onix:* Onix is a distributed system which runs on a cluster of one or more physical servers, each of which may run multiple Onix instances. As the control platform, Onix is responsible for giving the control logic programmatic access to the network (both reading and writing network state). In order to scale to very large networks (millions of ports) and to provide the requisite resilience for production deployments, an Onix instance is also responsible for disseminating network state to other instances within the cluster.

- *Control logic:* The network control logic is implemented on top of Onix's API. This control logic determines the desired network behavior; Onix merely provides the primitives needed to access the appropriate network state.

These are the four basic components of an SDN-based network. Before delving into the design of Onix, we should clarify our intended range of applicability. We assume that the physical infrastructure can forward packets much faster (typically by two or more orders of magnitude) than Onix (or any general control platform) can process them; thus, we do not envision using Onix to implement management functions that require the control logic to know about per-packet (or other rapid) changes in network state.

2.2 The Onix API

The principal contribution of Onix is defining a *useful* and *general* API for network control that allows for the development of scalable applications. Building on previous work [16], we designed Onix's API around a view of the physical network, allowing control applications to read and write state to any element in the network. Our API is therefore data-centric, providing methods for keeping state consistent between the in-network elements and the control application (running on multiple Onix instances).

More specifically, Onix's API consists of a data model that represents the network infrastructure, with each network element corresponding to one or more data objects. The control logic can: read the current state associated with that object; alter the network state by operating on these objects; and register for notifications of state changes to these objects. In addition, since Onix must support a wide range of control scenarios, the platform allows the control logic to customize (in a way we describe later) the data model and have control over the placement and consistency of each component of the network state.

The copy of the network state tracked by Onix is stored in a data structure we call the Network Information Base (NIB), which we view as roughly analogous to the Routing Information Base (RIB) used by IP routers. However, rather than just storing prefixes to destinations, the NIB is a graph of all network entities within a network topology. The NIB is both the heart of the Onix control model and the basis for Onix's distribution model. Network control applications are implemented by reading and writing to the NIB (for example modifying forwarding state or accessing port counters), and Onix provides scalability and resilience by replicating and distributing the NIB between multiple running instances (as configured by the application).

While Onix handles the replication and distribution of NIB data, it relies on application-specific logic to both detect and provide conflict resolution of network state as it is exchanged between Onix instances, as well as between an Onix instance and a network element. The control logic may also dictate the consistency guarantees for state disseminated between Onix instances using distributed locking and consensus algorithms.

In order to simplify the discussion, we assume that the NIB only contains physical entities in the network. However, in practice it can easily be extended to support logical elements (such as tunnels).

2.3 Network Information Base Details

At its most generic level, the NIB holds a collection of *network entities*, each of which holds a set of key-value pairs and is identified by a flat, 128-bit, global identifier. These network entities are the base structure from which all types are derived. Onix supports stronger typing through *typed entities*, representing different network elements (or their subparts). Typed entities then contain a predefined set of attributes (using the key-value pairs) and methods to perform operations over those attributes.

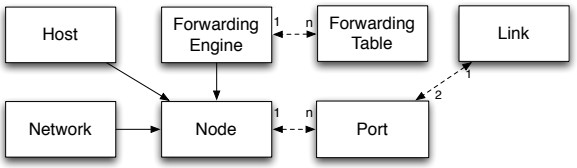

Figure 2: The default network entity classes provided by Onix's API. Solid lines represent inheritance, while dashed lines correspond to referential relation between entity instances. The numbers on the dashed lines show the quantitative mapping relationship (*e.g.*, one `Link` maps to two `Ports`, and two `Ports` can map to the same `Link`). Nodes, ports and links constitute the network topology. All entity classes inherit the same base class providing generic key-value pair access.

For example, there is a `Port` entity class that can belong to a list of ports in a `Node` entity. Figure 2 illustrates the default set of typed entities Onix provides – all typed entities have a common base class limited to generic key-value pair access. The type-set within Onix is not fixed and applications can subclass these basic classes to extend Onix's data model as needed.[6]

The NIB provides multiple methods for the control logic to gain access to network entities. It maintains an index of all of its entities based on the entity identifier, allowing for direct querying of a specific entity. It also supports registration for notifications on state changes or the addition/deletion of an entity. Applications can further extend the querying capabilities by listening for notifications of entity arrivals and maintaining their own indices.

The control logic for a typical application is therefore fairly straightforward. It will register to be notified on some state change (*e.g.*, the addition of new switches and ports), and once the notification fires, it will manipulate the network state by modifying the key-value pairs of the affected entities.

The NIB provides neither fine-grained nor distributed locking mechanisms, but rather a mechanism to request and release exclusive access to the NIB data structure of the local instance. While the application is given the guarantee that no other thread is updating the NIB within the same controller instance, it is not guaranteed the state (or related state) remains untouched by other Onix instances or network elements. For such coordination, it must use mechanisms implemented externally to the NIB. We describe this in more detail in Section 4; for now, we assume this coordination is mostly static and requires control logic involvement during failure conditions.

All NIB operations are asynchronous, meaning that updating a network entity only guarantees that the update message will eventually be sent to the corresponding network element and/or other Onix instances – no ordering or latency guarantees are given. While this has the potential to simplify the control logic and make multiple modifications more efficient, often it is useful to know when an update has successfully completed. For instance, to minimize disruption to network traffic, the application may require the updating of forwarding state on multiple switches to happen in a particular order (to minimize, for example, packet drops). For this purpose, the API provides a synchronization primitive: if called for an entity, the control logic will receive a callback once the state has been pushed. After receiving the callback, the control logic may then inspect the contents of the NIB and verify that the state is as expected before proceeding. We note that if the control logic implements distributed coordination, race-conditions in state updates will either not exist or will be transient in nature.

An application may also only rely on NIB notifications to react to failures in modifications as they would any other network state changes. Table 1 lists available NIB-manipulation methods.

Category	Purpose
Query	Find entities.
Create, destroy	Create and remove entities.
Access attributes	Inspect and modify entities.
Notifications	Receive updates about changes.
Synchronize	Wait for updates being exported to network elements and controllers.
Configuration	Configure how state is imported to and exported from the NIB.
Pull	Ask for entities to be imported on-demand.

Table 1: Functions provided by the Onix NIB API.

3 Scaling and Reliability

To be a viable alternative to the traditional network architecture, Onix must meet the scalability and reliability requirements of today's (and tomorrow's) production networks. Because the NIB is the focal point for the system state and events, its use largely dictates the scalability and reliability properties of the system. For example, as the number of elements in the network increases, a NIB that is not distributed could exhaust system memory. Or, the number of network events (generated by the NIB) or work required to manage them could grow to saturate the CPU of a single Onix instance.[7]

This and the following section describe the NIB distribution framework that enables Onix to scale to very

[6] Subclassing also enables control over how the key-value pairs are stored within the entity. Control logics may prefer different trade-offs between memory and CPU usage.

[7] In one of our upcoming deployments, if a single-instance application took one second to analyze the statistics of a single `Port` and compute a result (*e.g.*, for billing purposes), that application would take two months to process all `Ports` in the NIB.

large networks, and to handle network and controller failure.

3.1 Scalability

Onix supports three strategies that can used to improve scaling. First, it allows control applications to *partition* the workload so that adding instances reduces work without merely replicating it. Second, Onix allows for *aggregation* in which the network managed by a cluster of Onix nodes appears as a single node in a separate cluster's NIB. This allows for federated and hierarchical structuring of Onix clusters, thus reducing the total amount of information required within a single Onix cluster. Finally, Onix provides applications with control over the *consistency* and *durability* of the network state. In more detail:

- *Partitioning.* The network control logic may configure Onix so that a particular controller instance keeps only a subset of the NIB in memory and up-to-date. Further, one Onix instance may have connections to a subset of the network elements, and subsequently, can have fewer events originating from the elements to process.

- *Aggregation.* In a multi-Onix setup, one instance of Onix can expose a subset of the elements in its NIB as an aggregate element to another Onix instance. This is typically used to expose less fidelity to upper tiers in a hierarchy of Onix controllers. For example, in a large campus network, each building might be managed by an Onix controller (or controller cluster) which exposes all of the network elements in that building as a single aggregate node to a global Onix instance which performs campus-wide traffic engineering. This is similar in spirit to global control management paradigms in ATM networks [27].

- *Consistency and durability.* The control logic dictates the consistency requirements for the network state it manages. This is done by implementing any of the required distributed locking and consistency algorithms for state requiring strong consistency, and providing conflict detection and resolution for state not guaranteed to be consistent by use of these algorithms. By default, Onix provides two data stores that an application can use for state with differing preferences for durability and consistency. For state applications that favor durability and stronger consistency, Onix offers a replicated transactional database and, for volatile state that is more tolerant of inconsistencies, a memory-based one-hop DHT. We return to these data stores in Section 4.

The above scalability mechanisms can be used to manage networks too large to be controlled by a single Onix instance. To demonstrate this, we will use a running example: an application that can establish paths between switches in a managed topology, with the goal of establishing complete routes through the network.

Partition. We assume a network with a modest number of switches that can be easily handled by a single Onix instance. However, the number and size of all forwarding state entries on the network exceeds the memory resources of a single physical server.

To handle such a scenario, the control logic can replicate all switch state, but it must partition the forwarding state and assign each partition to a unique Onix instance responsible for managing that state. The method of partitioning is unimportant as long as it creates relatively consistent chunks.

The control logic can record the switch and link inventory in the fully-replicated, durable state shared by all Onix instances, and it can coordinate updates using mechanisms provided by the platform. However, information that is more volatile, such as link utilization levels, can be stored in the DHT. Each controller can use the NIB's representation of the complete physical topology (from the replicated database), coupled with link utilization data (from the DHT), to configure the forwarding state necessary to ensure paths meeting the deployment's requirements throughout the network.

The resulting distribution strategy closely resembles the use of head-end routers in MPLS [24] to manage tunnels. However, instead of a DHT, MPLS uses intra-domain routing protocols to disseminate the link utilization information.

Aggregate. As our example network grows, partitioning the path management no longer suffices. We assume that the Onix instances are still capable of holding the full NIB, but the control logic cannot keep up with the number of network events and thus saturates the CPU. This scenario follows from our experience in which CPU is commonly the limiting factor for control applications.

To shield remote instances from high-rates of updates, the application can aggregate a topology as a single logical node, and use that as the unit of event dissemination between instances. For example, the topology can be divided into logical areas, each managed by a distinct Onix instance. Onix instances external to an area would know the exact physical topology within the area, but would retrieve only topologically-aggregated link-utilization information from the DHT (originally generated by instances within that area).

This use of topological aggregation is similar to ATM PNNI [27], in which the internals of network areas are aggregated into single logical nodes when exposed to neighboring routers. The difference is that the Onix instances and switches still have full connectivity between

them and it is assumed that the latency between any element (between the switches and Onix instances or between Onix instances) is not a problem.

Partition further. At some point, the number of elements within a control domain will overwhelm the capacity of a single Onix instance. However, due to relatively slow change rates of the physical network, it is still possible to maintain a distributed view of the network graph (the NIB).

Applications can still rely on aggregating link utilization information, but in a partitioned NIB scheme, they would use the inter-Onix state distribution mechanisms to mediate requests to switches in remote areas; this can be done by using NIB attributes as a remote communication channel. The "request" and "response" are relayed between the areas using the DHT. Because this transfer might happen via a third Onix instance, any application that needs faster response times may configure DHT key ranges for areas and use DHT keys such that for the modified entity its attributes are stored within the proper area.

It is possible for this approach to scale to wide-area deployment scenarious. For example, each partition could represent a large network area, and each network is exposed as an aggregate node to a cluster of Onix instances that make global routing decisions over the aggregate nodes. Thus, each partition makes local routing decisions, and the cluster makes routing decisions between these partitions (abstracting each as a single logical node). The state distribution requirements for this approach would be almost identical to hierarchical MPLS.

Inter-domain aggregation. Once the controlled network spans two separate ASes, sharing full topology information among the Onix instances becomes infeasible due to privacy reasons and the control logic designer needs to adapt the design again to changed requirements.

The platform does not dictate how the ASes would peer, but at a high-level they would have two requirements to fulfill: *a)* sharing their topologies at some level of detail (while preserving privacy) with their peers, and *b)* establishing paths for each other proactively (according to a peering contract) or on-demand, and exchanging their ingress information. For both requirements, there are proposals in academia [13] and industry deployments [12] that applications could implement to arrange peering between Onix instances in adjacent ASes.

3.2 Reliability

Control applications on Onix must handle four types of network failures: forwarding element failures, link failures, Onix instance failures, and failures in connectivity between network elements and Onix instances (and between the Onix instances themselves). This section discusses each in turn.

Network element and link failures. Modern control planes already handle network element and link failures, and control logic built on Onix can use the same mechanisms. If a network element or link fails, the control logic has to steer traffic around the failures. The dissemination times of the failures through the network together with the re-computation of the forwarding tables define the minimum time for reacting to the failures. Given increasingly stringent requirements convergence times, it may be preferrable that convergence be handled partially by backup paths with fast failover mechanisms in the network element.

Onix failures. To handle an Onix instance failure, the control logic has two options: running instances can detect a failed node and take over the responsibilities of the failed instance quickly, or more than one instance can simultaneously manage each network element.

Onix provides coordination facilities (discussed in Section 4) for detecting and reacting to instance failures. For the simultaneous management of a network element by more than one Onix instance, the control logic has to handle lost update race conditions when writing to network state. To help, Onix provides hooks that applications can use to determine whether conflicting changes made by other instances to the network element can be overridden. Provided the control logic computes the same network element state in a deterministic fashion at each Onix instance, *i.e.*, every Onix instance implements the same algorithm, the state can remain inconsistent only transiently. At the high-level, this approach is similar to the reliability mechanisms of RCP [3], in which multiple centralized controllers push updates over iBGP to edge routers.

Connectivity infrastructure failures. Onix state distribution mechanisms decouple themselves from the underlying topology. Therefore, they require connectivity to recover from failures, both between network elements and Onix instances as well as between Onix instances. There are a number of methods for establishing this connectivity. We describe some of the more common deployment scenarios below.

It is not unusual for an operational network to have a dedicated physical network or VLAN for management. This is common, for example, in large datacenter buildouts or hosting environments. In such environments, Onix can use the management network for control traffic, isolating it from forwarding plane disruptions. Under this deployment model, the control network uses standard networking gear and thus any disruption to the control network is handled with traditional protocols (*e.g.*, OSPF or spanning tree).

Even if the environment does not provide a separate control network, the physical network topology is typically known to Onix. Therefore, it is possible for the control logic to populate network elements with static forwarding state to establish connecitivty between Onix and the switches. To guarantee connectivity in presence of failures, source routing can be combined with multi-pathing (also implemented below Onix): source routing packets over multiple paths can guarantee extremely reliable connectivity to the managed network elements, as well as between Onix instances.

4 Distributing the NIB

This section describes how Onix distributes its Network Information Base and the consistency semantics an application can expect from it.

4.1 Overview

Onix's support for state distribution mechanisms was guided by two observations on network management applications. First, applications have differing requirements on scalability, frequency of updates on shared space, and durability. For example network policy declarations change slowly and have stringent durability requirements. Conversely, logic using link load information relies on rapidly-changing network state that is more transient in nature (and thus does not have the same durability requirements).

Second, distinct applications often have different requirements for the consistency of the network state they manage. Link state information and network policy configurations are extreme examples: transiently-inconsistent status flags of adjacent links are easier for an application to resolve than an inconsistency in network-wide policy declaration. In the latter case, a human may be needed to perform the resolution correctly.

Onix supports an application's ability to choose between update speeds and durability by providing two separate mechanisms for distributing network state updates between Onix instances: one designed for high update rates with guaranteed availability, and one designed with durability and consistency in mind. Following the example of many distributed storage systems that allow applications to make performance/scalability tradeoffs [2, 8, 29, 31], Onix makes application designers responsible for explicitly determining their preferred mechanism for any given state in the NIB – they can also opt to use the NIB solely as storage for local state. Furthermore, Onix can support arbitrary storage systems if applications write their own *import* and *export modules*, which transfer data into the NIB from storage systems and out of the NIB to storage systems respectively.

In solving the applications' preference for differing consistency requirements, Onix relies on their help: it expects the applications to use the provided coordination facilities [19] to implement distributed locking or consensus protocols as needed. The platform also expects the applications to provide the implementation for handling any inconsistencies arising between updates, if they are not using strict data consistency. While applications are given the responsibility to implement the inconsistency handling, Onix provides a programmatic framework to assist the applications in doing so.

Thus, application designers are free to determine the trade-off between potentially simplified application architectures (promoting consistency and durability) and more efficient operations (with the cost of increased complexity). We now discuss the state distribution between Onix instances in more detail, as well as how Onix integrates network elements and their state into these distribution mechanisms.

4.2 State Distribution Between Onix Instances

Onix uses different mechanisms to keep state consistent between Onix instances and between Onix and the network forwarding elements. The reasons for this are twofold. First, switches generally have low-powered management CPUs and limited RAM. Therefore, the protocol should be lightweight and primarily for consistency of forwarding state. Conversely, Onix instances can run on high powered general compute platforms and don't have such limitations. Secondly, the requirements for managing switch state are much narrower and better defined than that needed by any given application.

Onix implements a transactional persistent database backed by a replicated state machine for disseminating all state updates requiring durability and simplified consistency management. The replicated database comes with severe performance limitations, and therefore it is intended to serve only as a reliable dissemination mechanism for slowly changing network state. The transactional database provides a flexible SQL-based querying API together with triggers and rich data models for applications to use directly, as necessary.

To integrate the replicated database with the NIB, Onix includes import/export modules that interact with the database. These components load and store entity declarations and their attributes from/to the transactional database. Applications can easily group NIB modifications together into a single transaction to be exported to the database. When the import module receives a trigger invocation from the database about changed database contents, it applies the changes to the NIB.

For network state needing high update rates and availability, Onix provides a one-hop, eventually-consistent, memory-only DHT (similar to Dynamo [9]), relaxing the consistency and durability guarantees provided by the replicated database. In addition to the common

get/put API, the DHT provides soft-state triggers: the application can register to receive a callback when a particular value gets updated, after which the trigger must be reinstalled. False positives are allowed to simplify the implementation of the DHT replication mechanism. The DHT implementation manages its membership state and assigns key-range responsibilities using the same coordination mechanisms provided to applications.

Updates to the DHT by multiple Onix instances can lead to state inconsistencies. For instance, while using triggers, the application must be carefully prepared for any race conditions that could occur due to multiple writers and callback delays. Also, the introduction of a second storage system may result in inconsistencies in the NIB. In such cases, the Onix DHT returns multiple values for a given key, and it is up to the applications to provide conflict resolution, or avoid these conditions by using distributed coordination mechanisms.

4.3 Network Element State Management

The Onix design does not dictate a particular protocol for managing network element forwarding state. Rather, the primary interface to the application is the NIB, and any suitable protocol supported by the elements in the network can be used under the covers to keep the NIB entities in sync with the actual network state. In this section we describe the network element state management protocols currently supported by Onix.

OpenFlow [23] provides a performance-optimized channel to the switches for managing forwarding tables and quickly learning port status changes (which may have an impact on reachability within the network). Onix turns OpenFlow events and operations into state that it stores in the NIB entities. For instance, when an application adds a flow entry to a `ForwardingTable` entity in the NIB, the OpenFlow export component will translate that into an OpenFlow operation that adds the entry to the switch TCAM. Similarly, the TCAM entries are accessible to the application in the contents of the `ForwardingTable` entity.

For managing and accessing general switch configuration and status information, an Onix instance can opt to connect to a switch over a configuration database protocol (such as the one supported by Open vSwitch [26]). Typically this database interface exposes the switch internals that OpenFlow does not. For Onix, the protocol provides a mechanism to receive a stream of switch state updates, as well as to push changes to the switch state. The low-level semantics of the protocol closely resembles the transactional database (used between controllers) discussed above, but instead of requiring full SQL support from the switches, the database interface has a more restricted query language that does not provide joins.

Similarly to the integration with OpenFlow, Onix provides convenient, data-oriented access to the switch configuration state by mapping the switch database contents to NIB entities that can be read and modified by the applications. For example, by creating and attaching `Port` entities with proper attributes to a `ForwardingEngine` entity (which corresponds to a single switch datapath), applications can configure new tunnel endpoints without knowing that this translates to an update transaction sent to the corresponding switch.

4.4 Consistency and Coordination

The NIB is the central integration point for multiple data sources (other Onix instances as well as connected network elements); that is, the state distribution mechanisms do not interact directly with each other, but rather they import and export state into the NIB. To support multiple applications with possibly very different scalability and reliability requirements, Onix requires the applications to declare what data should be imported to and exported from a particular source. Applications do this through the configuration of import and export modules.

The NIB integrates the data sources without requiring strong consistency, and as a result, the state updates to be imported into NIB may be inconsistent either due to the inconsistency of state within an individual data source (DHT) or due to inconsistencies between data sources. To this end, Onix expects the applications to register inconsistency resolution logic with the platform. Applications have two means to do so. First, in Onix, entities are C++ classes that the application may extend, and thus, applications are expected simply to use inheritance to embed *referential inconsistency detection logic* into entities so that applications are not exposed to inconsistent state.[8] Second, the plugins the applications pass to the import/export components implement *conflict resolution logic*, allowing the import modules to know how to resolve situations where both the local NIB and the data source have changes for the same state.

For example, consider a new `Node` N, imported into the NIB from the replicated database. If N contains a reference in its list of ports to `Port` P that has not yet been imported (because they are retrieved from the network elements, not from the replicated database), the application might prefer that N not expose a reference to P to the control logic until P has been imported. Furthermore, if the application is using the DHT to store statistics about the number of packets forwarded by N, it is possible for the import module of an Onix instance to retrieve two different values for this number from the DHT (*e.g.*, due to rebalancing of controllers' responsibilities within a cluster, resulting in two controllers transiently updating the same value). The

[8] Any inconsistent changes remain pending within the NIB until they can be applied or applications deem it invalid for good.

application's conflict resolution logic must reconcile these values, storing only one into the NIB and back out to the DHT.

This leaves the application with a consistent topology data model. However, the application still needs to react to Onix instance failures and use the provided coordination mechanisms to determine which instances are responsible for different portions of the NIB. As these responsibilities shift within the cluster, the application must instruct the corresponding import and export modules to adjust their behaviors.

For coordination, Onix embeds Zookeeper [19] and provides applications with an object-oriented API to its filesystem-like hierarchical namespace, convenient for realizing distributed algorithms for consensus, group membership, and failure detection. While some applications may prefer to use Zookeeper's services directly to store persistent configuration state instead of the transactional database, for most the object size limitations of Zookeeper and convenience of accessing the configuration state directly through the NIB are a reason to favor the transactional database.

5 Implementation

Onix consists of roughly 150,000 lines of C++ and integrates a number of third party libraries. At its simplest, Onix is a harness which contains logic for communicating with the network elements, aggregating that information into the NIB, and providing a framework in which application programmers can write a management application.

A single Onix instance can run across multiple processes, each implemented using a different programming language, if necessary. Processes are interconnected using the same RPC system that Onix instances can use among themselves, but instead of running over TCP/IP it runs over local IPC connections. In this model, supporting a new programming language becomes a matter of writing a few thousand lines of integration code, typically in the new language itself. Onix currently supports C++, Python, and Java.

Independent of the programming language, all software modules in Onix are written as loosely-coupled components, which can be replaced with others without recompiling Onix as long as the component's binary interface remains the same. Components can be loaded and unloaded dynamically and designers can express dependencies between components to ensure they are loaded and unloaded in the proper order.

6 Applications

In this section, we discuss some applications currently being built on top of Onix. In keeping with the focus of the paper, we limit the applications discussed to those that are being developed for production environments. We believe the range of functionality they cover demonstrates the generality of the platform. Table 2 lists the ways in which these applications stress the various Onix features.

Ethane. For enterprise networks, we have built a network management application similar to Ethane [5] to enforce network security policies. Using the Flow-based Management Language (FML) [18] network administrators can declare security policies in a centralized fashion using high-level names instead of network-level addresses and identifiers. The application processes the first packet of every flow obtained from the first hop switch: it tracks hosts' current locations, applies the security policies, and if the flow is approved, sets up the forwarding state for the flow through the network to the destination host. The link state of the network is discovered through LLDP messages sent by Onix instances as each switch connects.

Since the aggregate flow traffic of a large network can easily exceed the capacity of a single server, large-scale deployment of our implementation, it requires multiple Onix instances to partition the flow processing. Further, having Onix on the flow-setup path makes failover between multiple instances particularly important.

Partitioning the flow-processing state requires that all controllers be able to set up paths in the network, end to end. Therefore, each Onix instance needs to know the location of all end-points as well as the link state of the network. However, it is not particularly important that this information be strongly consistent between controllers. At worst, a flow is routed to an old location of the host over a failed link, which is impossible to avoid during network element failures. It is also unnecessary for the link state to be persistent, since this information is obtained dynamically. Therefore, the controllers can use the DHT for storing link-state, which allows tens of thousands of updates per second (see Section 7).

Distributed Virtual Switch (DVS). In virtualized enterprise network environments, the network edge consists of virtual, software-based L2 switch appliances within hypervisors instead of physical network switches [26]. It is not uncommon for virtual deployments (especially in cloud-hosting providers) to consist of tens of VMs per server, and to have hundreds, thousands or tens of thousands of VMs in total. These environments can also be highly dynamic, such that VMs are added, deleted and migrated on the fly.

To cope with such environments, the concept of a distributed virtual switch (DVS) has arisen [33]. A DVS roughly operates as follows. It provides a logical switch abstraction over which policies (*e.g.*, policing, QoS, ACLs) are declared over the logical switch ports. These ports are bound to virtual machines through integration with the hypervisor. As the machines come and go and move around the network, the DVS ensures that the

Control Logic	Flow Setup	Distribution	Availability	Integration
Ethane	!		!	
Distributed virtual switch				!
Multi-tenant virtualized datacenter		!		!
Scale-out carrier-grade IP router			!	

Table 2: Aspects of Onix especially stressed by deployed control logic applications.

policies follow the VMs and therefore do not have to be reconfigured manually; to this end, the DVS integrates to the host virtualization platform.

Thus, when operating as part of a DVS application, Onix is not involved in forwarding plane flow setup, but only invoked when VMs are created, destroyed, or migrated. Hypervisors are organized as pools consisting of a reasonably small number of hypervisors and VMs typically do not migrate across pools; and therefore, the control logic can easily partition itself according to these pools. A single Onix instance then handles all the hypervisors of a single pool. All the switch configuration state is persisted to the transactional database, whereas all VM locations are not shared between Onix instances.

If an Onix instance goes down, the network can still operate. However, VM dynamics will no longer be allowed. Therefore, high availability in such an environment is less critical than in the Ethane environment described previously, in which an Onix crash would render the network inoperable to new flows. In our DVS application, for simplicity reasons reliability is achieved through a cold standby prepared to boot in a failure condition.

Multi-tenant virtualized data centers. Multi-tenant environments exacerbate the problems described in the context of the previous application. The problem statement is similar, however: in addition to handling end-host dynamics, the network must also enforce both addressing and resource isolation between tenant networks. Tenant networks may have, for example, overlapping MAC or IP addresses, and may run over the same physical infrastructure.

We have developed an application on top of Onix which allows the creation of tenant-specific L2 networks. These networks provide a standard Ethernet service model and can be configured independently of each other and can span physical network subnets.

The control logic isolates tenant networks by encapsulating tenants' packets at the edge, before they enter the physical network, and decapsulating them when they either enter another hypervisor or are released to the Internet. For each tenant virtual network, the control logic establishes tunnels pair-wise between all the hypervisors running VMs attached to the tenant virtual network. As a result, the number of required tunnels is $O(N^2)$, and thus, with potentially tens of thousands of VMs per tenant network, the state for just tunnels may grow beyond the capacity of a single Onix instance, not to mention that the switch connections can be equally numerous.[9]

Therefore, the control logic partitions the tenant network so that multiple Onix instances share responsibility for the network. A single Onix instance manages only a subset of hypervisors, but publishes the tunnel end-point information over the DHT so any other instances needing to set up a tunnel involving one of those hypervisors can configure the DHT import module to load the relevant information into the NIB. The tunnels themselves are stateless, and thus, multiple hypervisors can send traffic to a single receiving tunnel end-point.

Scale-out carrier-grade IP router. We are currently considering a design to create a scale-out BGP router using commodity switching components as the forwarding plane. This project is still in the design phase, but we include it here to demonstrate how Onix can be used with traditional network control logic.

In our design, Onix provides the "glue" between the physical hardware (a collection of commodity switches) and the control plane (an open source BGP stack). Onix is therefore responsible for aggregating the disparate hardware devices and presenting them to the control logic as a single forwarding plane, consisting of an L2/L3 table, and a set of ports. Onix is also responsible for translating the RIB, as calculated by the BGP stack, into flow entries across the cluster of commodity switches.

In essence, Onix will provide the logic to build a scale-out chassis from the switches. The backplane of the chassis is realized through the use of multiple connections and multi-pathing between the switches, and individual switches act as line-cards. If a single switch fails, Onix will alert the routing stack that the associated ports on the full chassis have gone offline. However, this should not affect the other switches within the cluster.

The control traffic from the network (*e.g.*, BGP or IGP traffic) is forwarded from the switches to Onix, which annotates it with the correct logical switch port and forwards it to the routing stack. Because only a handful of switches are used, the memory and processing demands

[9]The VMs of a single tenant are not likely to share physical servers to avoid fate-sharing in hardware failure conditions.

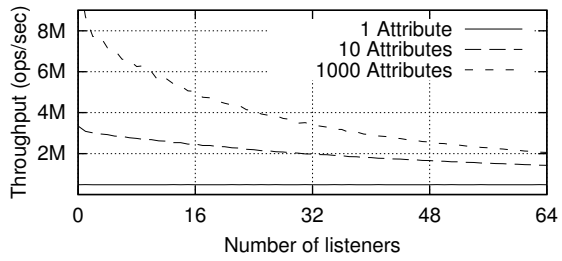

Figure 3: Attribute modification throughput as the number of listeners attached to the NIB increases.

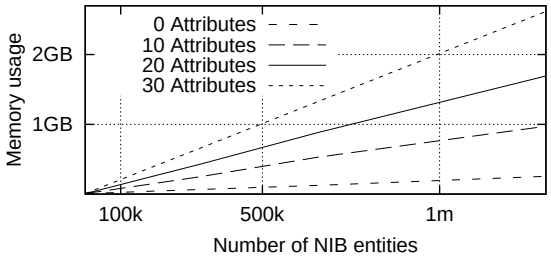

Figure 4: Memory usage as the number of NIB entities increases.

of this applications are relatively modest. A single Onix instance with an active failover (on which the hardware configuration state is persistent) is sufficient for even very large deployments. This application is discussed in more detail in [7].

7 Evaluation

In this section, we evaluate Onix in two ways: with micro-benchmarks, designed to test Onix's performance as a general platform, and with end-to-end performance measurements of an in-development Onix application in a test environment.

7.1 Scalability Micro-Benchmarks

Single-node performance. We first benchmark three key scalability-related aspects of a single Onix instance: throughput of the NIB, memory usage of the NIB, and bandwidth in the presence of many connections.

The NIB is the focal point of the API, and the performance of an application will depend on the capacity the NIB has for processing updates and notifying listeners. To measure this throughput, we ran a micro-benchmark where an application repeatedly acquired exclusive access to the NIB (by its cooperative thread acquiring the CPU), modified integer attributes of an entity (which triggers immediate notification of any registered listeners), and then released NIB access. In this test, none of the listeners acted on the notifications of NIB changes they received. Figure 3 contains the results. With only a single attribute modification, this micro-benchmark essentially becomes a benchmark for our threading library, as acquiring

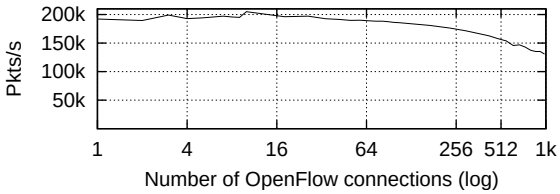

Figure 5: Number of 64-byte packets forwarded per second by a single Onix node, as the # of switch connections increases.

exclusive access to the NIB translates to a context switch. As the number of modified attributes between context switches increases, the effective throughput increases because the modifications involve only a short, fine-tuned code path through the NIB to the listeners.

Onix NIB entities provide convenient state access for the application as well as for import and export modules. The NIB must thus be able to handle a large number of entries without excessive memory usage. Figure 4 displays the results of measuring the total memory consumption of the C++ process holding the NIB while varying both network topology size and the number of attributes per entity. Each attribute in this test is 16 bytes (on average), with an 8-byte attribute identifier (plus C++ string overhead); in addition, Onix uses a map to store attributes (for indexing purposes) that reserves memory in discrete chunks. A zero-attribute entity, including the overhead of storing and indexing it in the NIB, consumes 191 bytes. The results in Figure 4 suggest a single Onix instance (on a server-grade machine) can easily handle networks of millions of entities. As entities include more attributes, their sizes increase proportionally.

Each Onix instance has to connect to the switches it manages. To stress this interface, we connected a (software) switch cloud to a single Onix instance and ran an application that, after receiving a 64-byte packet from a random switch, made a forwarding decision without updating the switch's forwarding tables. That is, the application sent the packet back to the switch with forwarding directions for that packet alone. Because of the application's simplicity, the test effectively benchmarks the performance of our OpenFlow stack, which has the same code path for both packets and network events (such as port events). Figure 5 shows the stack can perform well (forwarding over one hundred thousand packets per second), with up to roughly one thousand concurrent connections. We have not yet optimized our implementation in this regard, and the results highlight a known limitation of our threading library, which forces the OpenFlow protocol stack to do more threading context switches as the number of connections increases. Bumps in the graph are due to the operating system scheduling the controller process over multiple CPU cores.

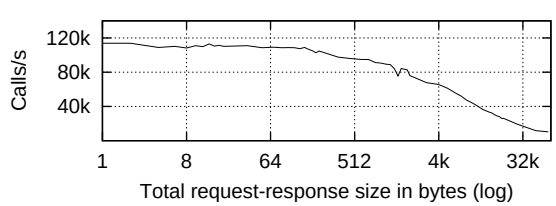

Figure 6: RPC calls per second processed by a single Onix node, as the size of the RPC request-response pair increases.

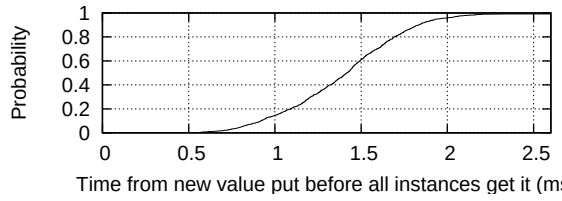

Figure 7: A CDF showing the latency of updating a DHT value at one node, and for that update to be fetched by another node in a 5-node network.

Multi-node performance. Onix instances use three mechanisms to cooperate: two state update dissemination mechanisms (the DHT and the replicated, transactional database) and the Zookeeper coordination mechanism. Zookeeper's performance has been studied elsewhere [19], so we focus on the DHT and replicated database.

The throughput of our memory-based DHT is effectively limited by the Onix RPC stack. Figure 6 shows the call throughput between an Onix instance acting as an RPC client, and another acting as an RPC server, with the client pipelining requests to compensate for network latency. The DHT performance can then be seen as the RPC performance divided by the replication factor. While a single value update may result in both a notification call and subsequent get calls from each Onix instance having an interest in the value, the high RPC throughput still shows our DHT to be capable of handling very dynamic network state. For example, if you assume that an application fully replicates the NIB to 5 Onix instances, then each NIB update will result in 22 RPC request-response pairs (2 to store two copies of the data in the DHT, $2*5$ to notify all instances of the update, and $2*5$ for all instances to fetch the new value from both replicas and reinstall their triggers). Given the results in Figure 6, this implies that the application, in aggregate, can handle 24,000 small DHT value updates per second. In a real deployment this might translate, for example, to updating a load attribute on 24,000 link entities every second – a fairly ambitious scale for any physical network that is controlled by just five Onix instances. Applications can use aggregation and NIB partitioning to scale further.

Our replicated transactional database is not optimized for throughput. However, its performance has not yet become a bottleneck due to the relatively static nature of the data it stores. Table 3 shows the throughput for different query batching sizes (1/3 of queries are INSERTs, and 2/3 are SELECTs) in a 5-node replicated database. If the application stores its port inventory in the replicated database, for example, without any batching it can process 17 port additions and removals per second, along with about 6.5 queries per second from each node about the existence of ports ($17 + 6.5 * 5 \sim 49.7$).

Queries/trans	**1**	**10**	**20**	**50**	**100**
Queries/s	49.7	331.9	520.1	541.7	494.4

Table 3: The throughput of Onix's replicated database.

7.2 Reliability Micro-Benchmarks

A primary consideration for production deployments is reliability in the face of failures. We now consider the three failure modes a control application needs to handle: link failures, switch failures, and Onix instance failures. Finally, we consider the perceived network communication failure time with an Onix application.

Link and switch failures. Onix instances monitor their connections to switches using aggressive keepalives. Similarly, switches monitor their links (and tunnels) using hardware-based probing (such as 802.1ag CFM [1]). Both of these can be fine-tuned to meet application requirements.

Once a link or switch failure is reported to the control application, the latencies involved in disseminating the failure-related state updates throughout the Onix cluster become essential; they define the absolute minimum time the control application will take to react to the failure *throughout* the network.

Figure 7 shows the latencies of DHT value propagation in a 5-node, LAN-connected network. However, once the controllers are more distant from each other in the network, the DHT's pull-based approach begins to introduce additional latencies compared to the ideal push-based methods common in distributed network protocols today. Also, the new value being put to the DHT may be placed on an Onix instance not on the physical path between the instance updating the value and the one interested in the new value. Thus, in the worst case, a state update may take four times as long as it takes to push the value (one hop to put the new value, one to notify an interested Onix instance, and two to get the new value).

In practice, however, this overhead tends not to impact network performance, because practical availability requirements for production traffic require the control application to prepare for switch and link failures proactively by using backup paths.

Onix instance failures. The application has to detect failed Onix instances and then reconfigure responsibilities within the Onix cluster. For this, applications rely on the

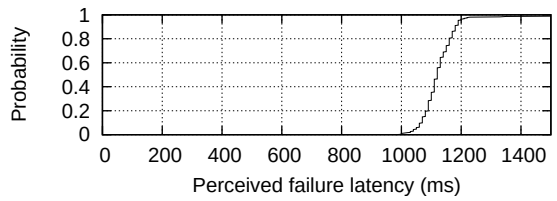

Figure 8: A CDF of the perceived communication disruption time between two hosts when an intermediate switch fails. These measurements include the one-second (application-configurable) keepalive timeout used by Onix. The hosts measure the disruption time by sending a ping every 10 ms and counting the number of missed replies.

Zookeeper coordination facilities provided by Onix. As with its throughput, we refer the reader to a previous study [19] for details.

Application test. Onix is currently being used by a number of organizations as the platform for building commercial applications. While scaling work and testing is ongoing, applications have managed networks of up to 64 switches with a single Onix instance, and Onix has been tested in clusters of up to 5 instances.

We now measure the end-to-end failure reaction time of the multi-tenant virtualized data center application (Section 6). The core of the application is a set of tunnels creating an L2 overlay. If a switch hosting a tunnel fails, the application must patch up the network quickly to ensure continued connectivity withing the overlay.

Figure 8 shows how quickly the application can create new tunnels to reestablish the connectivity between hosts when a switch hosting a tunnel fails. The measured time includes the time for Onix to detect the switch failure, and for the application to decide on a new switch to host the tunnel, create the new tunnel endpoints, and update the switch forwarding tables. The figure shows the median disruption for the host-to-host communication is 1120 ms. Given the configured one-second switch failure detection time, this suggests it takes Onix 120 ms to repair the tunnel once the failure has been detected. Although this application is unoptimized, we believe these results hold promise that a complete application on Onix can achieve reactive properties on par with traditional routing implementations.

8 Related Work

As mentioned in Section 1, Onix descends from a long line of work in which the control plane is separated from the dataplane [3–6, 15, 16, 23], but Onix's focus on being a production-quality control platform for large-scale networks led us to focus more on reliability, scalability, and generality than previous systems. Ours is not the first system to consider network control as a distributed systems problem [10, 20], although we do not anticipate the need to run our platform on end-hosts, due to the flexibility of merchant silicon and other efforts to generalize the forwarding plane [23], and the rapid increase in power of commodity servers.

An orthogonal line of research focuses on offering network developers an extensible forwarding plane (*e.g.*, RouteBricks [11], Click [22] and XORP [17]); Onix is complementary to these systems in offering an extensible control plane. Similarly, Onix can be the platform for flexible data center network architectures such as SEATTLE [21], VL2 [14] and Portland [25] to manage large data centers. This was explored somewhat in [30].

Other recent work [34] reduces the load of a centralized controller by distributing network state amongst switches. Onix focuses on the problem of providing generic distributed state management APIs at the controller, instead of focusing on a particular approach to scale. We view this work as distinct but compatible, as this technique could be implemented within Onix.

Onix also follows the path of many earlier distributed systems that rely on applications' help to relax consistency requirements in order to improve the efficiency of state replication. Bayou [31], PRACTI [2], WheelFS [29] and PNUTS [8] are examples of such systems.

9 Conclusion

The SDN paradigm uses the control platform to simplify network control implementations. Rather than forcing developers to deal directly with the details of the physical infrastructure, the control platform handles the lower-level issues and allows developers to program their control logic on a high-level API. In so doing, Onix essentially turns networking problems into distributed systems problem, resolvable by concepts and paradigms familiar for distributed systems developers.

However, this paper is not about the *ideology* of SDN, but about its *implementation*. The crucial enabler of this approach is the control platform, and in this paper we present Onix as an existence proof that such control platforms are feasible. In fact, Onix required no novel mechanisms, but instead involves only the judicious use of standard distributed system design practices.

What we should make clear, however, is that Onix does not, by itself, solve all the problems of network management. The designers of management applications still have to understand the scalability implications of their design. Onix provides general tools for managing state, but it does not magically make problems of scale and consistency disappear. We are still learning how to build control logic on the Onix API, but in the examples we have encountered so far management applications are far easier to build with Onix than without it.

Acknowledgments

We thank the OSDI reviewers, and in particular our shepherd Dave Andersen, for their helpful comments. We also thank the various team members at Google, NEC, and Nicira who provided their feedback on the design and implementation of Onix. We gratefully acknowledge Satoshi Hieda at NEC, who ran measurements that appear in this paper.

References

[1] 802.1ag - Connectivity Fault Management Standard. `http://www.ieee802.org/1/pages/802.1ag.html`.

[2] BELARAMANI, N., DAHLIN, M., GAO, L., NAYATE, A., VENKATARAMANI, A., YALAGANDULA, P., AND ZHENG, J. PRACTI Replication. In *Proc. NSDI* (May 2006).

[3] CAESAR, M., CALDWELL, D., FEAMSTER, N., REXFORD, J., SHAIKH, A., AND VAN DER MERWE, K. Design and Implementation of a Routing Control Platform. In *Proc. NSDI* (April 2005).

[4] CAI, Z., DINU, F., ZHENG, J., COX, A. L., AND NG, T. S. E. The Preliminary Design and Implementation of the Maestro Network Control Platform. Tech. rep., Rice University, Department of Computer Science, October 2008.

[5] CASADO, M., FREEDMAN, M. J., PETTIT, J., LUO, J., MCKEOWN, N., AND SHENKER, S. Ethane: Taking Control of the Enterprise. In *Proc. SIGCOMM* (August 2007).

[6] CASADO, M., GARFINKEL, T., AKELLA, A., FREEDMAN, M. J., BONEH, D., MCKEOWN, N., AND SHENKER, S. SANE: A Protection Architecture for Enterprise Networks. In *Proc. Usenix Security* (August 2006).

[7] CASADO, M., KOPONEN, T., RAMANATHAN, R., AND SHENKER, S. Virtualizing the Network Forwarding Plane. In *Proc. PRESTO* (November 2010).

[8] COOPER, B. F., RAMAKRISHNAN, R., SRIVASTAVA, U., SILBERSTEIN, A., BOHANNON, P., JACOBSEN, H.-A., PUZ, N., WEAVER, D., AND YERNENI, R. PNUTS: Yahoo!'s Hosted Data Serving Platform. In *Proc. VLDB* (August 2008).

[9] DECANDIA, G., HASTORUN, D., JAMPANI, M., KAKULAPATI, G., LAKSHMAN, A., PILCHIN, A., SIVASUBRAMANIAN, S., VOSSHALL, P., AND VOGELS, W. Dynamo: Amazon's Highly Available Key-value Store. In *Proc. SOSP* (October 2007).

[10] DIXON, C., KRISHNAMURTHY, A., AND ANDERSON, T. An End to the Middle. In *Proc. HotOS* (May 2009).

[11] DOBRESCU, M., EGI, N., ARGYRAKI, K., CHUN, B.-G., FALL, K., IANNACCONE, G., KNIES, A., MANESH, M., AND RATNASAMY, S. RouteBricks: Exploiting Parallelism To Scale Software Routers. In *Proc. SOSP* (October 2009).

[12] FARREL, A., VASSEUR, J.-P., AND ASH, J. A Path Computation Element (PCE)-Based Architecture, August 2006. RFC 4655.

[13] GODFREY, P. B., GANICHEV, I., SHENKER, S., AND STOICA, I. Pathlet Routing. In *Proc. SIGCOMM* (August 2009).

[14] GREENBERG, A., HAMILTON, J. R., JAIN, N., KANDULA, S., KIM, C., LAHIRI, P., MALTZ, D. A., PATEL, P., AND SENGUPTA, S. VL2: A Scalable and Flexible Data Center Network. In *Proc. SIGCOMM* (August 2009).

[15] GREENBERG, A., HJALMTYSSON, G., MALTZ, D. A., MYERS, A., REXFORD, J., XIE, G., YAN, H., ZHAN, J., AND ZHANG, H. A Clean Slate 4D Approach to Network Control and Management. *SIGCOMM CCR 35*, 5 (2005), 41–54.

[16] GUDE, N., KOPONEN, T., PETTIT, J., PFAFF, B., CASADO, M., MCKEOWN, N., AND SHENKER, S. NOX: Towards an Operating System for Networks. In *SIGCOMM CCR* (July 2008).

[17] HANDLEY, M., KOHLER, E., GHOSH, A., HODSON, O., AND RADOSLAVOV, P. Designing Extensible IP Router Software. In *Proc. NSDI* (May 2005).

[18] HINRICHS, T. L., GUDE, N. S., CASADO, M., MITCHELL, J. C., AND SHENKER, S. Practical Declarative Network Management. In *Proc. of SIGCOMM WREN* (August 2009).

[19] HUNT, P., KONAR, M., JUNQUEIRA, F. P., AND REED, B. ZooKeeper: Wait-free Coordination for Internet-Scale Systems. In *Proc. Usenix Annual Technical Conference* (June 2010).

[20] JOHN, J. P., KATZ-BASSETT, E., KRISHNAMURTHY, A., ANDERSON, T., AND VENKATARAMANI, A. Consensus Routing: The Internet as a Distributed System. In *Proc. NSDI* (April 2008).

[21] KIM, C., CAESAR, M., AND REXFORD, J. Floodless in SEATTLE: A Scalable Ethernet Architecture for Large Enterprises. In *Proc. SIGCOMM* (August 2008).

[22] KOHLER, E., MORRIS, R., CHEN, B., JANNOTTI, J., AND KAASHOEK, M. F. The Click Modular Router. *ACM Trans. on Computer Systems 18*, 3 (August 2000), 263–297.

[23] MCKEOWN, N., ANDERSON, T., BALAKRISHNAN, H., PARULKAR, G., PETERSON, L., REXFORD, J., SHENKER, S., AND TURNER, J. OpenFlow: Enabling Innovation in Campus Networks. *SIGCOMM CCR 38*, 2 (2008), 69–74.

[24] Multiprotocol Label Switching Working Group. `http://datatracker.ietf.org/wg/mpls/`.

[25] MYSORE, R. N., PAMBORIS, A., FARRINGTON, N., HUANG, N., MIRI, P., RADHAKRISHNAN, S., SUBRAM, V., AND VADHAT, A. PortLand: A Scalable Fault-Tolerant Layer 2 Data Center Network Fabric. In *Proc. SIGCOMM* (August 2009).

[26] PFAFF, B., PETTIT, J., KOPONEN, T., AMIDON, K., CASADO, M., AND SHENKER, S. Extending Networking into the Virtualization Layer. In *Proc. HotNets* (October 2009).

[27] Private Network-Network Interface Specification Version 1.1 (PNNI 1.1), April 2002. ATM Forum.

[28] SHERWOOD, R., GIBB, G., YAP, K.-K., APPENZELLER, G., CASADO, M., MCKEOWN, N., AND PARULKAR, G. Can the Production Network Be the Testbed? In *Proc. OSDI* (October 2010).

[29] STRIBLING, J., SOVRAN, Y., ZHANG, I., PRETZER, X., LI, J., KAASHOEK, M. F., AND MORRIS, R. Flexible, Wide-Area Storage for Distributed Systems with WheelFS. In *Proc. NSDI* (April 2009).

[30] TAVAKOLI, A., CASADO, M., KOPONEN, T., AND SHENKER, S. Applying NOX to the Datacenter. In *Proc. HotNets* (October 2009).

[31] TERRY, D. B., THEIMER, M. M., PETERSEN, K., DEMERS, A. J., SPREITZER, M. J., AND HAUSER, C. H. Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. In *Proc. SOSP* (December 1995).

[32] TOUCH, J., AND PERLMAN, R. Transparent Interconnection of Lots of Links (TRILL): Problem and Applicability Statement. RFC 5556, IETF, May 2009.

[33] VMware vNetwork Distributed Switch, Simplify Virtual Machine Networking. `http://vmware.com/products/vnetwork-distributed-switch`.

[34] YU, M., REXFORD, J., FREEDMAN, M. J., AND WANG, J. Scalable Flow-Based Networking with DIFANE. In *Proc. SIGCOMM* (August 2010).