Public Review for

A Database Approach to SDN Control Plane Design

Bruce Davie, Teemu Koponen, Justin Pettit, Ben Pfaff, Martin Casado, Natasha Gude, Amar Padmanabhan, Tim Petty, Kenneth Duda and Anupam Chanda

This paper's reviewers had mixed feelings on the paper. On one hand, they were impressed and surprised that the authors opted for using a database technique to solve a networking problem instead of designing a new protocol with specific properties. This led to the design of the open-source `OVSDB` that already enabled several networking vendors to design interoperable products. On the other hand, the reviewers found that the paper contained too many engineering details and did not provide a quantitative evaluation of the proposed solution. Some reviewers suggested to summarise the paper in 6 pages, i.e. the normal page limit for technical papers published in CCR. In the end, we agreed to keep the engineering details in the paper because we expect that they could be of interest to the community as they show the details that matter in industry. Furthermore, the paper is tagged with the **Artifacts Available** badge given the availability of the `OVSDB` open-source software.

Public review written by
Olivier Bonaventure
Université catholique de Louvain

A Database Approach to SDN Control Plane Design

Bruce Davie
VMware
bdavie@vmware.com

Teemu Koponen
Styra
teemu.koponen@iki.fi

Ben Pfaff
VMware
blp@ovn.org

Justin Pettit
VMware
jpettit@ovn.org

Anupam Chanda
VMware
achanda@vmware.com

Martin Casado
VMware
mcasado@vmware.com

Kenneth Duda
Arista Networks
kduda@arista.com

Natasha Gude & Amar Padmanabhan
Facebook

Tim Petty
VMware
tpetty@vmware.com

ABSTRACT

Software-defined networking (SDN) is a well-known example of a research idea that has been reduced to practice in numerous settings. Network virtualization has been successfully developed commercially using SDN techniques. This paper describes our experience in developing production-ready, multi-vendor implementations of a complex network virtualization system. Having struggled with a traditional network protocol approach (based on OpenFlow) to achieving interoperability among vendors, we adopted a new approach. We focused first on defining the control information content and then used a generic database protocol to synchronize state between the elements. Within less than nine months of starting the design, we had achieved basic interoperability between our network virtualization controller and the hardware switches of six vendors. This was a qualitative improvement on our decidedly mixed experience using OpenFlow. We found a number of benefits to the database approach, such as speed of implementation, greater hardware diversity, the ability to abstract away implementation details of the hardware, clarified state consistency model, and extensibility of the overall system.

CCS Concepts

•**Networks** → **Network design principles; Network protocol design;** *Network manageability; Programmable networks;*

Keywords

Software-Defined Networking (SDN); protocols; protocol design; interoperability; databases

1 Introduction

One of the fundamental challenges in networking is the need for interoperability among systems and devices implemented by different teams working in different companies. For this reason, a great deal of effort is expended in the definition of protocols: precise definitions of the messages that are exchanged between systems and the procedures for sending, receiving, and processing those messages. Protocol definition is a complex and time-consuming task; it can take years to produce a specification (as many IETF attendees will attest). Extending such protocols to support new or unanticipated functionality can be an equally cumbersome process.

This paper describes our experiences in tackling such an interoperability problem in a commercial setting. Specifically, we had developed a network virtualization system [15] — a highly scalable, SDN-based system that supports the creation and management of virtual networks [13]. At the time, the "network virtualization edge" (NVE) was implemented only in software, using Open vSwitch [26] running on general purpose servers. We wanted to add hardware-based implementations of the NVE, primarily to provide higher performance.

Hardware switches that could potentially serve the NVE function are widely available, and data plane interoperability had been largely addressed by widespread support for the VXLAN tunnel encapsulation [18]. However, no existing control plane protocol clearly met our needs. Our prior experience with OpenFlow [22], which might have appeared to be the obvious choice, led us to consider alternative approaches.

Unlike many operators of large data centers, we are forced to accept a fairly heterogeneous environment. In particular, many of our customers have strong preferences about their choice of hardware switch vendor(s), so it is important for us to accommodate a wide diversity of switch implementations. Quite early in the process we found ourselves working with six different hardware vendors, with different switch hardware and software implementations. The forwarding models of their ASICs varied considerably. We needed a solution that could be implemented quickly by all these disparate vendors on their existing hardware.

One metric of success in this effort is the speed with which we were able to achieve interoperability. We obtained a basic level of interoperability with all six vendors within nine months of our first meeting to discuss the problem. Equally importantly, new capabilities that were either not fully defined or not anticipated in the initial design were readily added as the project progressed.

This "success", however, was built on the back of some decidedly less successful prior experience. Before settling on the database approach described in this paper, we had made two separate attempts to implement a similar feature set using OpenFlow to control a single vendor's switch hardware. These efforts (detailed in §4) demonstrated sufficient shortcomings with OpenFlow for this use case that we decided to take a hard look at a different approach.

Ultimately, we decided to apply database principles to the problem. That is, we didn't set out to design a protocol—we set out to model the information that needed to be shared among various components of the system, and then leveraged standard database techniques to distribute the appropriate state to the components that needed it. In this way, we abstracted away the details of the control wire protocol, and separated the state synchronization machinery from the networking functions. This is a sharp contrast to traditional network control protocol design, where state synchronization,

semantics of network function (*e.g.,* route computation), and wire representation are conflated into a single mechanism.[1]

This design choice had a number of benefits:

- We spent zero time specifying or arguing about the wire format.
- Any time we wanted to make a change to the interaction between the network virtualization controller and the NVE, we extended or modified the database schema. There was never any need to change the implementation of the wire protocol or the databases.
- We could simplify the implementations through use of a strong consistency model provided by the database. There was no need to deal with the complexities of eventual consistency, typical with control protocols.
- We could leverage a great deal of existing software (the database server, client libraries, and other software components).

This paper makes three main points:

- We present the experience of developing a commercial, production-quality system that has demonstrated interoperability among a logically centralized software controller and hardware switches from at least six independent vendors. The system is in production use at several customer sites.
- We show the value of a database approach to network product development: decoupling the responsibilities of control plane protocol functionality lets system design work focus on the high-level information model rather than the low-level wire format.
- We demonstrate the value of using a higher level of abstraction for the switch management in an SDN system than that provided by OpenFlow.

Along the way, we describe things that didn't go to plan and try to extract lessons from our mistakes.

We observe that this is an "experience" paper, as opposed to a more traditional "design, implement, evaluate" paper. The experience has been in building and testing multi-vendor systems to be deployed and operated by others, rather than in operating a network. We share the experience of developing a large commercial system under the constraints imposed by the need to deal with widely disparate implementations of both hardware and software. We evaluate our work more in terms of number of successfully interoperating implementations than in, say, performance. Acceptable performance of the control plane was something that we tested (see §8), but performance is not the main focus of this paper.[2] We describe experiences where things didn't go to plan, and attempt to extract lessons from our mistakes.

We have structured the paper as follows. In §2 and §3, we give a brief overview of related work and network virtualization, as well as the requirements that led to our work on hardware-based NVEs. §4 lays out the design approach that we followed and the earlier challenges that led to it. Then, in §5 and §6, we discuss the implementation and integration of the database technology with controller and hardware switches. We did not anticipate all the requirements early on (expectedly) and in §7 we detail how the system significantly evolved over the course of integration. Finally, we discuss the experience of testing the system in our own labs and those of our partners in §8, before summarizing our conclusions.

A note on replicability. The work described in this paper depends heavily on the use of open source software components, the details of which are presented below. Much of the paper describes our design and implementation experience, which is qualitative and thus difficult to reproduce. However, a sufficient set of software artifacts are available for other researchers to create their own NVE and test its interoperability against an open source SDN controller. Further details are provided in §8.5.

2 Background and Related Work

Our initial work on network virtualization and SDN [13, 14] laid the foundation for this paper. Here we focus on the challenge of *multi-vendor* interoperability, which raises a host of practical problems.

The traditional approach to addressing multi-vendor interoperability is to design a protocol. We explain below how a database approach differs from traditional protocol design, and shows the benefits that accrued from taking that approach. In this context, both a "new" protocol like OpenFlow [22] and mainstream protocols such as BGP [28] are examples of traditional protocol design.

Several in the community have applied principles of databases to network problems. In particular, declarative routing [16, 17] considers modeling route computation as a database query problem. We are tackling a significantly different problem space: the *synchronization* of state among controllers and switches, not the computation of routing state.

Chen et al. [3, 4] have argued for the benefits of a declarative approach to the *configuration* and *management* of networks, but this work assumes standard networking protocols (BGP, IS-IS, etc.) still provide the control plane. Conversely, Reitblatt et al. explored the benefits of stronger consistency on SDN data planes [27]. Our focus is on consistency and interoperability in control plane protocols.

Statesman [32] exposes overall datacenter network state as a set of databases for management applications. It differs from our work both in that it focuses on maintaining network-wide safety and performance invariants rather than the control plane interactions between switches and controller, and in that it does not tackle the issues of multi-vendor interoperability.

It is also interesting to compare our work against that of the megascale data center operators such as Google [30] and Facebook [1]. Facebook reports that they use a conventional protocol—BGP4—as the control plane for the network. The megascale operators strive for homogeneity (and can achieve it) as an important simplification that enables them to operate efficiently at extremely large scale. Our core problem is that we are required by our customers to support heterogeneity, both in terms of hardware (lots of different switch ASICs with distinct forwarding capabilities) and software (implementations by many different teams). In essence, this paper is about how we deal with that heterogeneity.

An interesting similarity between our work and that of the megascale operators is the reliance on software tools to abstract away low-level communication details from application endpoints. For example, Facebook makes use of Thrift [31] to enable transport-independent RPC services for its applications, and Google uses protocol buffers (protobufs) [8] to provide a convenient way to encode structured application data for transmission over the wire. In abstracting away the details of the wire protocol, OVSDB gave us the same benefits of these tools, while also providing a suitable information exchange mechanism for our particular problem domain.

[1] It's worth emphasizing we did not modify the data plane protocols.

[2] Data plane performance, in particular, is determined by hardware implementation and is independent of our design choices.

3 Network Virtualization Overview

In this section, we provide relevant background to our efforts. We first give an overview of network virtualization in multi-tenant datacenters in general before detailing our implementation of network virtualization, NSX. We then set out to discuss the particular use cases and reasons driving towards the need to integrate hardware-based NVEs to NSX.

3.1 System

While there is some range of definitions of network virtualization, most recent descriptions include the creation of isolated and independently managed logical networks which come with their own addressing, topologies, and service models on top of a shared physical network [13, 15]. This decoupling of physical and logical network constructs is key to the power of network virtualization: in the typical deployment environment, a multi-tenant datacenter, it allows virtualized tenant workloads to remain unaware of changes in the physical network, all while the provider offloads the management of logical networks to its tenants.

Using the terminology of [15], it is the responsibility of Network Virtualization Edges (NVEs) to implement the network virtualization data plane. In NSX, NVEs have little intelligence and do not communicate with each other to compute forwarding state. Instead, it is a centralized controller cluster that calculates the state necessary to realize logical networks, and the controller programs the virtual switches acting as the NVEs accordingly. OVSDB is used to configure the virtual switches, and flow tables are programmed using OpenFlow. In this sense, the NSX system looks like a fairly standard SDN design: centralized control, distributed data plane, and simple switches.

In the data plane, NVEs use a network virtualization encapsulation protocol, such as VXLAN, to encapsulate the logical packets before sending them to the receiving VM. By using encapsulation and codifying logical network information into the encapsulation header, they provide isolation among logical networks, as well as between the physical and logical networks. The design of NSX and its NVE functionality is similar to the design discussed in [13].

To provide tenant VMs with a faithful duplication of the physical network service model, the NSX controller builds all logical packet forwarding operations on a *logical datapath* abstraction. For the tenant VMs, logical datapaths behave identically to physical ASIC packet forwarding pipelines: after a VM sends a packet, the packet traverses through a sequence of forwarding tables ultimately forming a forwarding decision, after which the packet is sent towards the next-hop logical datapath. The process repeats with the next logical datapath, until the final, receiving logical destination port has been reached. Using the current known location of the corresponding VM, the packet is then unicasted over a tunnel across the physical network to the receiving VM. There are two exceptions to this model which we discuss now.

Unlike a unicast packet tunneled from a source to a destination hypervisor, logical broadcast and multicast traffic requires packet replication. In many multi-tenant datacenters, the physical network provides only unicast L3 connectivity because IP multicast is considered unmanageable and unreliable at scale. NSX implements two approaches to avoid relying on physical multicast. First, the NVEs can unicast duplicate packets to all receiving NVEs. This may result in excessive load (for instance, due to limited uplink NIC capacity), and therefore NSX allows the NVEs to offload the replication to dedicated *service nodes*, which act as a simple multicast overlay for NVEs to use.

The second exception to the hypervisor virtual switch performing first-hop processing is the interfacing of logical networks with the physical world—that is, non-virtualized workloads and systems outside the virtualized data center. We discuss these in the following section.

3.2 Hardware Virtualization Edge

For traffic originating or terminating in hypervisors, a virtual switch is the natural "edge" of the network, being the first switch to touch packets originating in a VM. However, for traffic that enters or exits the logical network from non-virtualized workloads (*i.e.,* "bare-metal" servers without a hypervisor and associated virtual switch) or external to the data center ("north-south" traffic), there are more options to consider for the NVE. Specifically, quite a number of hardware switches and routers could potentially implement the NVE function. These devices typically support line-rate encapsulation and decapsulation of traffic with a network virtualization overlay encapsulation such as VXLAN.

To interconnect non-virtualized workloads and provide north-south connectivity, prior to the start of this project we implemented a gateway based on an x86 server "appliance" using a standalone instance of Open vSwitch (with no hypervisor). With recent advances in x86-based forwarding, *e.g.,* [5, 6, 11], the latest versions of Open vSwitch running on x86 provide fairly high performance (in the 10 Gbps range) and multiple gateways can be deployed to scale out the aggregate bandwidth. Nevertheless, hardware-based switches still substantially outperform x86-based forwarding. For example, top-of-rack (ToR) switches occupying 1 RU (one rack unit) are widely available with 32 ports of 40 Gbps, for a total throughput of 1.28 Tbps, well beyond that achievable by an x86 server. The high port density of a ToR switch, compared to a server with a handful of NICs, is also attractive in this environment, especially if there are many non-virtualized servers to be connected to the network virtualization overlay.

We often heard from customers that managing physical ports (those connected to bare-metal servers from the hardware NVE) should be as similar as possible to managing virtual ports (those that connect a VM to a virtual switch). This implies the ability to configure access control list (ACL) policies on physical ports just as we do on virtual ports.

North-south traffic is another use case. Many data centers use one or more routers with specific WAN (wide area network) features to handle such traffic. Given such a router with built-in support for VXLAN encapsulation, it is appealing to use such as device as the NVE, rather than having to hop through an x86 gateway en route to the WAN router.

Finally, there may be an argument in favor of using hardware-based NVEs to minimize latency. It's not yet clear if this is a compelling use case, in part because there are so many other sources of latency aside from the NVE.

Hardware switches do come with their own constraints. An ASIC datapath is relatively inflexible compared to one implemented on a general-purpose CPU, and significant changes of functionality have to wait for a new ASIC development cycle (typically years) as well as a replacement of physical hardware, with cost and availability implications. By contrast, a software-based edge can be upgraded relatively frequently. For these reasons, we see many production deployments of network virtualization systems using the software-based gateway successfully, and we expect that to continue, at least till next-generation programmable hardware data planes [2] become widely available in production environments.

A few additional factors impact the design and implementation of a hardware NVE. First, the control plane CPU of a commercial switch is often relatively underpowered. This is not fundamental, but still represents the current state of the industry. Second, a hardware

switch generally must support a range of configuration interfaces, such as command line and API-driven configuration. These varied models needed to be accommodated in our design.

4 Design Approach

In tackling the problem of adding a hardware NVE to our system, our first question was: how should the central controller communicate with the NVE? We decomposed this question into two parts: determining the appropriate abstractions for the information to be exchanged and establishing a suitable mechanism for synchronizing that information.

As described in more detail shortly, the first step led us to realize that we didn't need to operate on low-level forwarding state (as OpenFlow does, for example). The second step led us to the conclusion that we could treat switch management as a generic database synchronization problem instead of as a specific task of forwarding state management.

4.1 Finding Appropriate Abstractions

In a canonical SDN system [9, 14, 22], a logically centralized controller (or a cluster of controllers) runs a control plane and provides switches with forwarding state to use in packet forwarding. The switches then apply these instructions, with little change, to their data planes. This is in stark contrast to the traditional approach of switch design in which control and data plane intercommunication is not exposed outside a chassis but local control planes drive the data planes, and control plane instances form a distributed control plane.

In our integration of hardware NVEs to NSX, we leveraged the concept of logical centralization but chose not to compute forwarding state directly in the central controller. Instead, we elevated the level of abstraction of the switch–controller communication interface: a controller provides a switch with *control plane configuration* and expects each switch to locally compute the detailed forwarding state for that switch's data plane. Figure 1 contrasts these differences between traditional, SDN, and NVE approaches.

Our motivation for this higher level of abstraction was twofold: first, many hardware platforms that would otherwise make for suitable NVEs lack OpenFlow forwarding semantics. A useful analogy is that OpenFlow is like assembly language, and limits the system to only devices that have the OpenFlow "instruction set"; by moving to a higher level of abstraction, we gain the benefits of high level languages which can execute on many different instruction set architectures.

Second, our prior experience led us to consider a higher level of abstraction. In earlier projects, we had uncovered a number of problems with the canonical SDN approach using OpenFlow. Those earlier projects were less ambitious than our multi-vendor NVE effort; each focused on a single hardware switch vendor. However, even then the specifics of the switching chipsets created a major challenge: while the *syntax* (OpenFlow) is universal and standardized across types of switches, the forwarding model *semantics* are not universal across OpenFlow capable hardware switches.

We had observed that each chipset had a specific, fixed arrangement of flow tables, each with extremely limited matching functionality and mechanisms for conveying metadata across stages. The flow tables are often implemented by leveraging the ACL tables of the switch ASICs. These ACL tables are often backed by TCAMs and typically have just a few thousand rules shared among dozens of ports, severely limiting the number of flows that can be pushed by the controller. This is orders of magnitude less than Open vSwitch and insufficient for many deployments. At the same time, the ASICs have many other useful forwarding functions, but because they are not general enough to meet the OpenFlow match requirements, those features are unavailable to a controller that insists on speaking to the switch using OpenFlow. In effect, the switch ASIC has limited resources for "assembly language" programming, and lots of resources to directly implement higher level abstract operations (*e.g.,* L2 or IP forwarding). If we insist on controlling it with OpenFlow, we lose access to those resources capable of these higher level operations. Ultimately we developed chipset-specific, non-standard OpenFlow extensions so that the controller could access more features from the ASICs.

It is easy to see how this situation became a burden. For each chipset, developers had to understand both the nuances of the individual packet processing pipelines and the systemic interactions among those pipelines. In effect, they had to compile the desired high level behavior to match the specific capabilities of the ASIC target. This was painful enough with two different hardware targets that we were reluctant to repeat it across six or more distinct hardware architectures.

Thus, the data model that we developed was independent of the specific hardware details. We allow a controller to specify *what* a virtual network should consist of (which physical ports are logically connected to which virtual ports, etc.) rather than *how* the switch should forward packets to implement that network. The task of computing the latter from the former is left to each switch vendor.

Of course, it's easier said than done to come up with a data model that is so generic that any switch architecture can implement the mapping from "what" to "how". We collaborated with several switch vendors (including some who used multiple different ASIC architectures) to verify that the data model was sufficiently general.

It is worth mentioning that our preference towards abstracting the switch–controller communication was not greatly influenced by scaling concerns. While moving away from centrally computing and disseminating the low-level flow entries should reduce the controller load, it was really the concerns of establishing interoperability across types of switches and teams that forced us to raise the level of abstraction.

We return to the configuration data model below in §5. We now turn to the second step in tackling the design problem: deciding how to exchange this more abstract configuration information among the switches and controllers.

4.2 Generic State Synchronization

Network control plane protocols are domain-specific state synchronization protocols: whether it is a routing protocol or OpenFlow, the protocols exchange state updates between endpoints to effectively replicate the state across systems. They are domain-specific in the sense that they define the exact content and its semantics. That is, while the protocols typically leave the door open for adding new types of attributes to be synchronized, the extensions are typically to augment the information model and semantics, not to replace them.

For this reason, our desire to be able to freely evolve the data model argued against using existing network control plane protocols. To understand why we considered the control protocols non-ideal and database protocols as a better starting point, we can separate a typical control plane endpoint implementation into three parts as follows:

- At the lowest level, a wire protocol serializes and transmits content messages across the wire.
- Internally, an endpoint implements a data model and expresses all synchronized content with it.

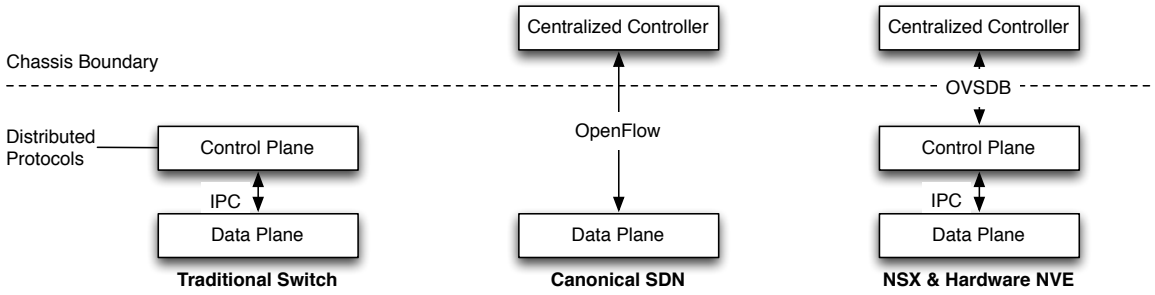

Figure 1: A comparison of traditional switch design (left), canonical SDN design (center) and NVE design (right).

- Finally, the endpoint implements logic consuming and acting on the data synchronized.

Consider OpenFlow, which conflates all three concerns into a single mechanism. First, to change the data model, it is likely one has to change the wire protocol. While the use of an Interface Definition Language (IDL), as once proposed for OpenFlow, could ease this task, the conflation is more fundamental. In particular, OpenFlow couples its consistency model to its data model: for example, the protocol specification codifies the exact functional behavior of switches in case of a flow referring to a non-existing port. On the other hand, while extremely specific for parts of the data model, the semantics are frustratingly underspecified in other parts: for instance, the exact resulting behavior in write conflicts (due to multiple controllers writing) may be hard to derive without knowing the specifics of the underlying switch implementation. This was a particular concern given the clustered nature of the NSX controller.

Similar conflation has always existed in traditional network control protocols, such as routing protocols, SNMP [10] and NETCONF [7]. However, unlike OpenFlow, these protocols have had a particularly simple logic (*e.g.,* route computation process or clients accessing individual SNMP counters), so reasoning about the overall protocol behavior in terms of consistency has posed little or no challenges. Indeed, this approach has served the vendors relatively well, although recent routing protocol features (*e.g.,* nonstop forwarding capabilities for routing protocols [19, 29]) have questioned the particular eventual consistency model of routing protocols. While NETCONF, the proposed replacement of SNMP, explicitly decouples the data model from its wire protocol operations, it still does not explicitly consider consistency.

To summarize the above, existing network control protocols have either complicated further evolving their data models through coupling their data to the consistency model, or they have chosen a particular weak consistency model with less limitations for evolving the data but which limits the practical complexity of the logic operating over such data. This is the exact opposite of database replication protocols, which do not embed a data model into their low-level wire protocol, are semantic-free regarding the logic built to use the data, *and* do not couple the consistency model and integrity properties to the data synchronized. This freedom to evolve the data model independently from the wire protocol and consistency semantics persuaded us not to use any existing control protocol, but instead consider the switch–controller synchronization as a generic database synchronization problem.

4.3 Architectural Limitations

Considering switch management as a database synchronization problem elevates the focus from the low-level protocol interoperability details to the control plane semantics but many fundamental challenges of designing for interoperability remain and, in fact, are orthogonal to this change.

In particular, as with traditional protocols, operational requirements necessitate support for multiple, co-existing control plane software versions to allow for gradual, rolling upgrades. While the versioning complications of low-level protocol details are abstracted away through use of a general database protocol, the overall strategy for schema versioning is no different from (nor more complex than) traditional protocol versioning: the implementations have to gradually phase changes in (and out), always considering semantics across a supported version matrix.

Moreover, it is still for the system designers to determine the exact level of the abstraction for the information exchanged, and thus, determine the preferred trade-offs between controller and switch complexity, as well as between generality and specificity of switch functionality. We now turn to this problem of information modeling.

5 Database Implementation

In this section we discuss the database implementation and database protocol used in NVE. We then describe the database schema that was developed for the initial NVE functionality.

5.1 Overview

State synchronization between NVE and the controller cluster sets no particularly special requirements for the database technology. Indeed, our decision to use the Open vSwitch configuration database management protocol (OVSDB) [25] was largely a non-technical decision: teams involved had prior experience with it, and the liberal open source license of its primary implementation (packaged as a part of Open vSwitch) allowed easy access across company boundaries.

The OVSDB protocol is a typical database access protocol: it connects database clients to a database server over a network. In NSX, each NVE hardware switch runs an independent OVSDB database server, and controllers in the controller cluster connect as clients to each server. NSX uses the protocol to implement table replication. For each column in a database table, either the switch or controller is a designated master (sole owner for its content), and the opposite peer holds a replica synchronized over the protocol.

Note that the OVSDB server resides on the NVE and the controller is a client of that server. This means that each server is responsible for quite a small amount of data: only the data relevant to a given NVE is stored on that device. This simplifies the implementation and minimizes resource requirements.

The protocol is lightweight and favors simplicity over high performance in implementations. Its implementation needs no multithreading, query language, or query planning capabilities. These simplifications together reduce concerns about switch control CPU resource consumption, and yet pose no practical limitations to our use case given the simplicity of the NVE data access and low volume of the data held in the database. While we had some initial concerns

that certain network events might lead to excessive transaction rates on the database, optimizations have not proven necessary.

The network virtualization controller initiates the bidirectional state synchronization process. For this purpose, the controllers in the controller cluster shard managed NVE switches among themselves; that is, for each NVE there is one master controller and a second controller ready to take over in case of master failure.

While the controller is the database client, it is actually the NVE switch that opens a TCP connection (optionally with TLS) to a member of the controller cluster.[3] If the controller is not the current master, the switch is instructed (via an entry in the database) to contact another controller. Once the controller receives the connection, it issues database queries to retrieve the entire contents of all the tables to be synchronized. After the initial snapshot retrieval, the controller enters a delta update mode. For each column of which the switch is the master, the controller installs a trigger to receive updates about future changes to those columns, thus being able to maintain its local replica. For a controller-mastered column, the controller translates any updates applied to the local master database to updates sent to the switch. This state synchronization process continues in both directions until the connection is closed.

Instead of being exposed to the wire protocol format and semantics, both controller and switch software developers now interface with database tables or their replicas, synchronized for them. This leaves the developer with the responsibility to define tables and their columns using a wide range of data types, indices, and any referential integrity requirements, and then to either consume or populate table contents. Unlike traditional network control protocols, the developer is shielded from the low-level protocol details, and the wire representation is largely irrelevant.

In addition to the explicit and generic data model definition tools, OVSDB primitives can implement many of the consistency and isolation models used in general-purpose database management systems. So far, instead of the eventual consistency model commonly seen in network control protocols, to simplify reasoning about consistency, in our work with OVSDB we have adopted sequential consistency, batching database operations (across tables) into atomically applied transactions. Contrary to routing protocols and OpenFlow, state consistency can be guaranteed over all updates, whether they span multiple attribute types, rows or tables. Some of the benefits of this consistency model became more apparent as the project progressed, and are described in §7.

5.2 Hardware NVE Schema

Our development started with a modest goal of establishing logical L2 connectivity between hypervisors (with virtual workloads) and hardware NVE switches (connected to physical workloads) over VXLAN encapsulation protocol. That is, using the terms introduced in §3, we set the goal of realizing a *logical datapath abstraction*. Using the abstraction we can provide tenants with logical switches which in turn implement a L2 broadcast domain for interconnecting the tenant physical and virtual workloads.

One of the guiding principles that we used in our design was to use a consistent model of a *logical port* that could apply to either true virtual ports (which are typically virtual NICs on VMs) or ports on the NVE. We settled on the ⟨port, VLAN⟩ pair as the best choice for an equivalent port concept on physical NVEs. Our logical datapath allows logical ports from either physical or virtual worlds to be mapped to a logical L2 broadcast domain.

For this purpose, we designed a database schema for NVE. Its tables and their relations are depicted in Figure 2. A complete description of the schema is available as part of Open vSwitch [33]. The schema includes three types of information:

Physical information. All managed NVE switches have a record in a `PhysicalSwitch` table. Each record holds management and tunnel IP addresses of a particular NVE switch. The switch populates this information based on its local configuration determined out-of-band. The `PhysicalPort` table provides the inventory of ports on a physical switch. The `VLANBindings` column of this table provides the means by which the NSX controller can map a ⟨port, VLAN⟩ pair to a logical switch. The `PhysicalLocator` table contains information about overlay tunnel endpoints and encapsulations.

Logical information. For logical switch creation, the schema defines a table `LogicalSwitch`. The table specifies a globally unique on-wire tunnel key to use with the encapsulation protocols to identify the logical switch to which a packet corresponds.[4] The controller is responsible for populating the table while the NVE switch consumes the table to drive its configuration and slicing of ASICs to logical switches.

Logical–physical bindings. To compute the exact forwarding entries for the ASICs, NVE needs to bind the logical MAC addresses to their reachability information which may involve tunneling details. NVE schema defines a number of MAC address tables for this purpose.

Since the controller does not know all MAC addresses of the connected physical workloads in advance but has to dynamically learn them, the NVE populates a table `UcastMacsLocal` with MAC addresses it has learned (by traditional L2 learning) from incoming physical workload traffic. The controller replicates this table from each NVE switch, gathers entries from all managed switches (including hypervisors), and disseminates the resulting collection using a table `UcastMacsRemote` to every NVE switch, referring in it to tunneling reachability information held in `PhysicalLocator` table. By joining these tables with the `LogicalSwitch` table, the NVE switch can compute its local L2 unicast forwarding entries, with the necessary encapsulation information, for its forwarding ASICs.

Similarly, to implement multicast and broadcast support the NVE switch populates a table `McastMacsLocal` for the controller, which can merge content from all switches, and then disseminate the results back to NVE switches through a table `McastMacsRemote`. This table is also used by the controller to send instructions for flooding of packets unknown MAC addresses, whether to unicast copies of the packet to all recipients or whether to offload packet replication to a dedicated service node.

6 Integration Experience

In this section, we describe our experiences in integrating hardware NVEs into the NSX system. This entailed both extending NSX to deal with the new class of devices and integrating the OVSDB into the switch software.

6.1 Extending the Controller

The NSX controller is essentially a system for managing state. Desired logical network state is created by northbound APIs; physical state is obtained from hypervisors and gateways; the controller

[3] This allows a single point in the system to decide which controller should have an active connection established with a switch.

[4] In VXLAN, this key is called Virtual Network Identifier (VNI). All network virtualization encapsulation protocols have a similar id.

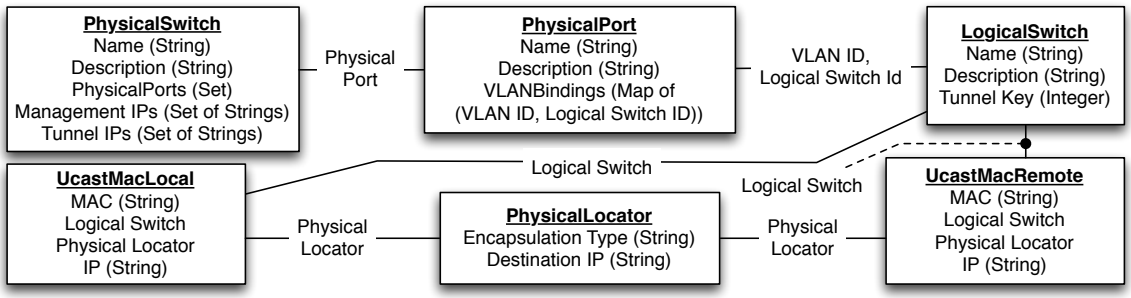

Figure 2: A simplified relational model of the NVE schema. Multicast MAC bindings and logical statistics are excluded for brevity.

then calculates what new state needs to be installed in the physical devices to realize the desired logical network.

The state that NSX manages consists of two layers: the logical and the physical. Clearly, nothing in the logical layer should be affected by the details of how a particular physical device is realized. The abstractions that the controller deals with remain the same, as does the bulk of the machinery for managing the state.

That said, the controller clearly needs to be able to communicate state changes to a physical device in terms that it can understand. A hardware NVE needs to be told using the OVSDB protocol and the NVE schema what its new state should be at any point in time. By contrast, a software NVE gateway would be told using OpenFlow what its forwarding rules should be.[5]

The changes we made to NSX are roughly analogous to the task of adding a new device to a conventional operating system: the high level operations within the OS remain consistent as you add new devices, but ultimately a device driver needs to map system calls to low-level I/O register changes that the device understands. Similarly, large parts of NSX are independent of the type of physical device being controlled, but we needed new code to map from the required abstract state to the OVSDB commands to communicate that state.

Consider, for example, the case where a logical switch has been created to interconnect two logical ports. One port is attached to the virtual NIC on a VM, the other port is attached to a ⟨port, VLAN⟩ pair on a physical switch. Suppose the VM migrates from one hypervisor to another. The hypervisors communicate this change in physical location of a VM to the NSX controller. Given this new piece of information, NSX recomputes the desired state for all the physical devices involved in this logical switch, and determines that the hardware NVE needs to have its forwarding table updated. Nothing to this point is specific to the type of device. Now NSX computes a set of OVSDB commands that it will send to the NVE to update the `UcastMacsRemote` table so that the NVE will correctly forward packets destined to the newly migrated VM.

As noted in §4 there is a further benefit of using OVSDB to communicate with hardware NVEs compared to the use of OpenFlow. Because the concepts communicated in OVSDB are at a higher level of abstraction (*e.g.,* "connect this logical port to this logical switch", rather than "install the following flow rules") there is less work for the centralized controller. The controller deals in logical network abstractions, and then compiles those down to OpenFlow rules when programming an Open vSwitch instance. When communicating with a hardware NVE, much of the communication is in terms of logical network abstractions, so there is no compilation step. At the time of writing, we have also started to implement a higher level of abstraction for the communication path between the controller and hypervisor virtual switches so that this benefit will be realized for software NVEs as well.

[5]Some configuration of Open vSwitch instances is performed by OVSDB, but forwarding state is largely programmed by OpenFlow [13]. As noted in §4, the most serious drawbacks of OpenFlow relates to its interaction with hardware forwarding.

6.2 Switch Software

The details of the database integration to switch software architecture are vendor-specific but typically involve the following steps. First, the vendor decides on the database implementation. In most cases the OVSDB server is a port of the open source OVSDB implementation to the switch OS. Two vendors (to date) have written their own OVSDB servers suited to their OS. The vendor then develops an OVSDB client that runs locally on the switch (typically leveraging libraries from the Open vSwitch distribution). That client provides the interface between the database and the rest of the switch OS. For example, the OVSDB client would gather the names of the ports on the switch from the internal configuration database of the switch, and populate that information in the OVSDB server, thus making it available to the network virtualization controller. When the NSX controller needs to tell the NVE how to reach some VMs, it provides their MAC addresses in the `UcastMacsRemote` table and the IP addresses of the appropriate hypervisors in the `Physical-Locator` table. The OVSDB client on the switch is notified of these changes, and then passes the information to the appropriate code to program the forwarding ASICs accordingly.

This approach led to a net reduction in complexity for the overall implementation. Switch vendors map OVSDB operations to their specific hardware (much as in a conventional protocol) but no vendor-specific behavior is exposed to the NSX controller, in sharp contrast to our earlier efforts.

7 Platform Evolution

One of the greatest benefits of the database approach was the ability to add new functionality to the system with no protocol modification. Every time we needed to add some new capability, or modify an existing capability, we were able to do so by modifying the NVE schema. In some cases we didn't even need to do that—we simply defined the new parameter in the schema documentation, leaving the schema itself unchanged. Of course, functional changes still require new code both in the controller and the switch to expose the new capability, but the developers can focus solely on that.

In the following sections, we describe capabilities that have been added to the system as it has evolved after the initial data model design laid out in §5.

Bidirectional Forwarding Detection. Bidirectional Forwarding Detection (BFD) [12] is a general-purpose mechanism for detecting the liveness of network links or paths. It was introduced to our network virtualization system as a means to determine the health of tunnels in the overlay network. This enables detection of dataplane issues that could lead to traffic loss. For example, NSX depends on a set of service nodes to perform replication of broadcast and

flooded unknown traffic. To ensure that service nodes are available and reachable, we run BFD over the tunnels between hardware NVEs and service nodes. An NVE distributes traffic requiring replication among the set of currently reachable service nodes, using ECMP (equal cost multipath) hashing. If a service node becomes unreachable according to the BFD session status, that service node is taken out of the set of valid destinations for the multipath bundle. BFD was added to Open vSwitch during the course of this project, and a table to enable BFD to be configured on a per-tunnel basis was added to the schema.

Hierarchical Switch Control. The original system design called for the network virtualization controller to directly control each hardware edge device. In other words, there would be a one-to-one mapping of OVSDB server instances to hardware NVEs. However, we encountered scenarios in which we needed to insert an intermediate control point between the network virtualization controller and one or more switches. We refer to such an intermediate control point as a Hardware Switch Controller (HSC). A single HSC runs the OVSDB server for one or more switches and then disseminates database updates further to switches (over a channel that may be proprietary).

Modifying the data model to include the HSC concept was fairly straightforward. We included a physical switch table, in which each line represents one physical switch under the control of the HSC. We also added a tunnel table, so that the overlay tunnels terminating on each hardware switch could be configured independently.

Layer 2 High Availability. Layer 2 switches often provide some sort of link high-availability mechanism, referred to as "multi-chassis link aggregation" (MLAG) or a similar term. MLAG provides redundant connectivity to a single host equipped with two (or more) NICs. One NIC is cabled to one switch, the second NIC is cabled to the second switch, and both links are monitored for liveness. As long as both are operating correctly, traffic is spread across the links; if a link, NIC, or switch fails, all traffic moves onto the remaining healthy link.

We leveraged this capability to provide a highly available hardware NVE for the overlay network, building on the HSC model. Consider two switches A and B operating as an MLAG pair. We run an OVSDB server on switch A, and let that OVSDB instance be the HSC for both switches. We also run an OVSDB server on switch B, and consider it the backup for the instance on switch A.

When everything is working normally, NSX communicates only with the server on switch A. This instance provides control for both switches. MLAG ports that straddle the two switches appear just like normal ports in the OVSDB database. If switch B fails, the software on switch A updates the database accordingly (*e.g.,* showing that switch B ports are no longer available).

In the event that switch A fails, the OVSDB server on switch B needs to take over. Depending on the vendor's MLAG implementation, switch B may or may not have a complete copy of the database from switch A. When it connects to the controller, the controller sees a connection "flap." As a result, it will check that the database matches the desired state that it had previously pushed to switch A, and make updates as necessary.

This highlights the value of the database approach. From the controller point of view, adding the support for link high-availability required essentially no work. This is because we did not have to hide the protocol disconnection (due to failover) from the logic consuming the state from the database (replicas): as the database synchronization maintains the consistency of the replicas (through atomically applied batches of changes), the disconnection remains invisible to higher layers. If our protocol had not provided such strong guarantees of consistency, our implementation would have resembled highly available BGP implementations which share TCP state across route processor (or chassis) boundaries.

Logical Layer 3. Our initial implementation treated the hardware NVE as an L2 device. That is, the service model that it provides is forwarding of Ethernet frames from a physical port to a logical switch and *vice versa*. We realized early on that we would also want to provide an L3 forwarding model, in which we could create logical routers and associate physical ports with those routers.

Our approach to adding L3 functionality was to add a `LogicalRouter` table that is similar to the `LogicalSwitch` table. Rather than interconnecting logical ports, a logical router interconnects logical switches, which in turn connect to ports. This provides fairly rich L3 capabilities. Further details of logical L3 are omitted for space reasons but can be found in the open source schema [33].

8 Testing and Deployment Experience

Interoperability testing between NSX and the NVE implementations has been performed by the NSX test team, hardware partners, and customers. In this section we describe the testing approach and some of the results. One notable observation is that the database approach enabled us to focus more effort on issues of scale and performance than on functional correctness of the protocol.

8.1 Software Tools

To provide troubleshooting support for the hardware NVE extensions we developed the `vtep-ctl` tool. (VTEP is another name for NVE.) This tool is analogous to `ovs-vsctl`, which provides the means to configure an Open vSwitch instance by interacting with the instance's OVSDB server. Similarly, `vtep-ctl` interacts with the OVSDB server of a hardware NVE. It implements an OVSDB client that can read and write a database conforming to the NVE schema. For example, the command sequence:

```
% vtep-ctl add-ps nve100
% vtep-ctl list-ps
```

first creates a record for a physical switch called `nve100`, and then displays all the physical switches in the database.

This tool provides a convenient and generic way to introspect the relevant system state, without having to deal with any of the specifics and complexity of the NSX controller. The source code for this utility also provides a working example of an OVSDB client implementation that switch vendors can use in developing their own OVSDB client code.

We also developed software to emulate a hardware NVE. The emulator was used to validate the correctness of the NSX implementation. We also made it available to hardware vendors to help in their testing. When a problem is encountered, one can compare the behavior of the NVE emulator to that of the hardware NVE, including examination of the logs from the respective OVSDB servers. This was an effective way to localize problems to the appropriate component. Further information on this and other tools is provided in §8.5.

8.2 Virtualized Test Environment

For several years, the developers of NSX have used a private cloud as an environment for testing their code. This environment comprises a set of hypervisors, a cloud management platform based on OpenStack [23], and a working version of NSX that builds virtual networks interconnecting VMs and gateways. We call this instance of NSX the "infrastructure" instance. When a developer wants to

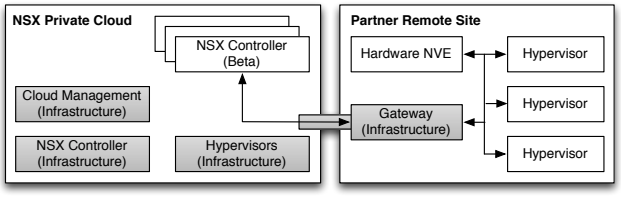

Figure 3: A virtualized test environment.

test a new version of NSX, he can load his test images into a set of VMs, and interconnect them into a suitable topology using the infrastructure instance of NSX. This is nested network virtualization: the test implementation of NSX is running inside a virtual network provided by the infrastructure version of NSX. We typically use a recent (pre-production) version of NSX for the infrastructure layer, as a form of "dogfooding"—engineers depend on their own product working to perform their jobs.

We extended this test cloud to enable hardware vendors to leverage the same environment. We deployed an instance of the NSX gateway on a server located at each partner site. This enables us to extend virtual networks from our private cloud to their test lab by tunneling across the public Internet. Inside our private cloud, for each partner, we run a dedicated instance of the NSX controller (a set of VMs) with the latest code supporting hardware NVEs. These VMs were connected to a logical network (managed by the infrastructure instance of NSX) and that logical network was mapped to a VLAN on a physical port of the infrastructure gateway located at the partner site. The partner can now see a running instance of the NSX controller as reachable on their local network. The partner deployed hypervisors with the appropriate version of Open vSwitch to be controlled by NSX, and their hardware NVE. By launching API requests to the NSX controller, or by interacting with the web-based GUI on the VM providing that service, they were able to place hypervisors under the control of NSX and establish OVSDB connections from the hardware switches to the controller.

This approach made it easy for the NSX development team to update the controller software without having access to the partner lab. We were also able to troubleshoot issues by examining logs on the controllers in our cloud. At the same time, partners had access to their own hardware and could troubleshoot it locally, and update software and firmware. This was instrumental in making rapid progress with a large number of physically distributed partners.

8.3 Physical Test Lab

The cloud-based test environment is effective for functional tests of interoperability between NSX and the hardware NVEs, but it is not well provisioned for scale and performance testing. To test at large scale and to evaluate performance, we built a physical testbed with:

- hardware NVEs under test (up to six at a time);
- approximately 20 physical servers, each a dual-socket, 8-core Intel server with 256G main memory;
- traffic generation tools; and
- switches to interconnect hypervisors, NVEs and traffic generators, all connected with 10Gbps links.

20 servers is not a large number, but our test environment makes use of nested virtualization to emulate a much larger deployment. By nesting hypervisors on 16 of the servers we were able to deploy 4,096 VMs in the testbed. Each VM was configured with four virtual NICs to increase the total number of MAC addresses in the system to over 16,000 for scale testing purposes. A data center with over 4,000 virtual machines and 16,000 MACs is hardly megascale, but it represents a larger-than-average enterprise deployment. Furthermore, recall that a single NVE only keeps state relevant to the virtual networks of which it is a member, so in a real deployment it would be exceptional if a single NVE had to track 16,000 MACs. In this sense the test setup is representative of much larger data centers than those of most customers.

8.4 Test Results and Discussion

We summarize here some notable observations gathered from the NSX testing team, test teams from the hardware partners, and customers who have tested the system.

We divide testing issues broadly into those affecting functional correctness, and those affecting scale or performance. We encountered few functional bugs across the six hardware systems tested. Most common among these were issues related to connecting to multiple NSX controller nodes, and handling failure of controller nodes. Notably, these issues are quite particular to an SDN system that uses a scale-out controller architecture. Because the open source OVSDB implementation has been designed explicitly to operate in such a multi-master environment, these issues were typically easy to address. In most cases it was as simple as starting the OVSDB server on the NVE with the correct parameters.

The most notable failure in testing was in a customer test lab. Whereas our test lab had the NVE building VXLAN tunnels to hypervisors on different IP subnets, our customer's lab had all VXLAN tunnel endpoints (VTEPs) on the same subnet. Surprisingly, the NVE was unable to correctly encapsulate packets in this case. Within a few days the problem was localized to a shortcoming of the specific forwarding ASIC. It may be too much to claim that the database approach helped directly in the quick diagnosis, but having an identical control plane across all hardware (unlike our prior approach with OpenFlow) helped identify the problem as vendor-specific. The switch vendor was then able to localize it further to the forwarding hardware. The quick solution was to modify the customer test topology to place the VTEPs in separate subnets (typical in production settings).

Another set of testing issues arose because two hardware partners developed their own OVSDB server code based on [25] rather than using the open source server. Hence we encountered an additional class of bugs—OVSDB protocol level issues—when testing against these partners. These partners still achieved most of the benefits from the database approach, such as decoupling state synchronization from forwarding plane behavior, but were unable to benefit from software reuse to the same extent as other partners.

BFD testing also exposed some interoperability challenges. Whereas the Open vSwitch implementation of BFD is extremely flexible, and can use any addresses in the BFD packet headers, we uncovered a range of different constraints among the hardware partners. For example, some hardware switches can only receive a BFD packet if the destination MAC and IP address of the BFD packet match addresses assigned to the physical switch. As we uncovered these constraints, which were not the same for all vendors, we evolved the schema so that a NVE could tell the controller what address constraints needed to be satisfied.

Scale testing includes testing how long it takes the system to stabilize after a cold start or a disruption when large amounts of state might need to be resynchronized. For example, a NVE is reset and needs to receive information about 16,000 VMs' MAC addresses from the controller, program its MAC tables accordingly, and start forwarding traffic from the physical interfaces towards the VMs over VXLAN tunnels. Because this time depends on both hardware (e.g.,

time to update a hardware forwarding table) and software (the path between the OVSDB server and the hardware-specific drivers), we observed considerable variability in convergence times. However, after some performance tuning we were about to bring the time below 10 seconds for all hardware.

"Performance," in this context, is all about the convergence of the control plane, and only becomes an issue in large-scale failure scenarios. Data plane performance is unaffected by the control plane design details: after receiving OVSDB updates from a controller, the switch local control plane proactively programs all necessary forwarding rule updates to the data plane so it can operate at line speed without consulting the switch control plane. Also, unlike some SDN designs, data packets are never sent to the central controllers.

To summarize the large amount of testing, we observe that the database approach allowed us to focus primarily on scale testing, since functional correctness was largely assured once the OVSDB server was implemented. For the majority of partners who re-used the open source OVSDB server, we were saved from having to debug OVSDB itself.

8.5 Replication of Results

The point of this paper is to illustrate the design approach that we took, and how it evolved over time with experience in a commercial setting. Hence, we're not hoping to see quantitative results replicated. In one sense, the fact that we were able to achieve interoperability across so many vendors was itself an exercise in replicable research. That said, we produced a number of software artifacts which may be used to conduct experiments such as those described in this paper. We describe those artifacts in the following paragraphs.

All the open-source code used in this paper is part of the Open vSwitch repository [20]. For the purposes of this paper we refer to the OVS version 2.6 software branch [21]. To install the VTEP emulator described in this paper, refer to `README.ovs-vtep.md` in the `vtep` directory. The VTEP emulator runs exactly the same OVSDB server code and schema as used in this paper, but does not require a hardware switch. Instead, it uses a software switch (an instance of OVS) that is controlled by the OVSDB server.

While many readers of this paper will not have access to the NSX commercial software product, we have also developed an open-source network virtualization controller, OVN (Open Virtual Network) [24], as part of the Open vSwitch project. A full installation of Open vSwitch version 2.6 includes OVN, and a tutorial on running OVN can be found in the file `OVN-tutorial.md` in the `tutorial` directory. With OVN and the VTEP emulator, the reader is able to construct in software a complete system including a controller and NVE (i.e., VTEP). This will enable the reader to test functional correctness, experiment with schema changes, and perform performance testing using only commodity hardware and open-source software.

Finally, we note that the existence of the OVN controller is another proof point demonstrating the power of the OVSDB approach in enabling interoperability among SDN system components. While this paper focused on enabling many different NVE implementations to interoperate with NSX, we now also have at least two different network virtualization controllers that can interoperate with the NVEs.

9 Concluding Remarks

In this paper we described our experiences in integrating hardware NVEs to NSX using two design principles: first, elevating the interoperability to the level of a data model and second, using standard database techniques to synchronize state between control plane elements. These combined with our reliance on open source allowed us to achieve interoperability between a network virtualization controller and six switch vendors in nine months.

What did we learn from this experience? Many of us had designed and/or implemented protocols before, and this experience was definitely better than the traditional approach in certain key respects. We were able to achieve interoperability quickly, and across a large number of hardware and software implementations. We avoided some shortcomings of traditional approaches, and of prior efforts to control hardware switches using OpenFlow. By using a higher level of abstraction to communicate between controller and switches, we avoided exposing developers to the low-level specifics of the hardware, and allowed a much broader range of hardware architectures to be supported. By using a general database approach to synchronizing state, we greatly simplified the process of evolving the control plane.

One might ask if we simply "cheated" by avoiding a traditional standards process. In fact, what we did is common enough in the history of protocol standardization. Protocol design exercises often start within a small team, entirely outside the standardization bodies, and are picked up later by a standards body. Our process was certainly consensus-based, among a reasonably large group of vendors; it simply happened outside the confines of a recognized standards body. Indeed, we can claim that we conformed to the IETF mantra of "rough consensus and running code."

Significantly, our approach has since been adopted by other projects. Not only have new hardware vendors implemented the NVE functionality and tested it against NSX, but other SDN controllers have chosen to use an OVSDB database approach to communicate with NVEs. OVN (Open Virtual Network) [24], for example, uses OVSDB not only for communicating with hardware NVEs but also as the control plane for software virtual switches. NSX is also developing a more abstract interface than OpenFlow for switch management.

With the benefit of hindsight over both this project and its less successful predecessors, we believe that we have uncovered shortcomings with OpenFlow as a protocol for control of current generation hardware, and more generally of the conflating of concerns that afflicts many traditional control protocols. Moreover, in using database techniques and an effective set of high-level abstractions, we have found a better way to achieve interoperability among the diverse implementations of equipment from many vendors. We hope that this work may influence the future design of network protocols.

10 Acknowledgments

Many anonymous reviewers have commented on this paper since it was first submitted, and we thank them for helping to shape it. We also wish to thank our colleagues across many networking companies who contributed to developing and testing the technology described in this paper.

11 References

[1] A. Andreyev. Introducing data center fabric, the next-generation facebook data center network. `https://code.facebook.com/posts/360346274145943/introducing-data-center-fabric-the-next-generation-facebook-data-center-network/`, Nov. 2014.

[2] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker. P4: Programming

Protocol-independent Packet Processors. In *Proc. of SIGCOMM*, 2014.
[3] X. Chen, Y. Mao, Z. M. Mao, and J. E. van der Merwe. Declarative configuration management for complex and dynamic networks. In *Proc. CoNEXT*, 2010.
[4] X. Chen, Y. Mao, Z. M. Mao, and J. E. van der Merwe. DECOR: declarative network management and operation. *Computer Communication Review*, 40(1):61–66, 2010.
[5] M. Dobrescu, K. J. Argyraki, and S. Ratnasamy. Toward Predictable Performance in Software Packet-Processing Platforms. In *Proc. of NSDI*, 2012.
[6] M. Dobrescu, N. Egi, K. Argyraki, B.-G. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy. RouteBricks: Exploiting Parallelism to Scale Software Routers. In *Proc. of SOSP*, 2009.
[7] R. Enns, M. Bjorklund, J. Schoenwaelder, and A. Bierman. NETCONF Configuration Protocol. RFC 6241, IETF, June 2011.
[8] Google. Protocol Buffers. `https://developers.google.com/protocol-buffers/`.
[9] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker. NOX: Towards an Operating System for Networks. *SIGCOMM CCR*, 38, 2008.
[10] D. Harrington, R. Preshun, and B. Wijnen. An Architecture for Describing Simple Network Management Protocol (SNMP) Management Frameworks. RFC 3411, IETF, Dec. 2002.
[11] Intel. *Intel Data Plane Development Kit (Intel DPDK): Programmer's Guide*, October 2013.
[12] D. Katz and D. Ward. Bidirectional Forwarding Detection (BFD). RFC 5880, IETF, June 2010.
[13] T. Koponen, K. Amidon, P. Balland, M. Casado, A. Chanda, B. Fulton, I. Ganichev, J. Gross, P. Ingram, E. Jackson, A. Lambeth, R. Lenglet, S.-H. Li, A. Padmanabhan, J. Pettit, B. Pfaff, R. Ramanathan, S. Shenker, A. Shieh, J. Stribling, P. Thakkar, D. Wendlandt, A. Yip, and R. Zhang. Network Virtualization in Multi-tenant Datacenters. In *Proc. of NSDI*, Seattle, WA, Apr. 2014.
[14] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama, and S. Shenker. Onix: A Distributed Control Platform for Large-scale Production Networks. In *Proc. of OSDI*, Oct. 2010.
[15] M. Lasserre, F. Balus, T. Morin, N. Bitar, and Y. Rekhter. Framework for Data Center (DC) Network Virtualization. RFC 7365, IETF, Oct. 2014.
[16] B. T. Loo, T. Condie, M. N. Garofalakis, D. E. Gay, J. M. Hellerstein, P. Maniatis, R. Ramakrishnan, T. Roscoe, and I. Stoica. Declarative networking. *Commun. ACM*, 52(11):87–95, 2009.
[17] B. T. Loo, J. M. Hellerstein, I. Stoica, and R. Ramakrishnan. Declarative Routing: Extensible Routing with Declarative Queries. In *Proc. of SIGCOMM*, 2005.
[18] M. Mahalingam, D. Dutt, K. Duda, P. Agarwal, L. Kreeger, T. Sridhar, M. Bursell, and C. Wright. Virtual eXtensible Local Area Network (VXLAN): A Framework for Overlaying Virtualized Layer 2 Networks over Layer 3 Networks. RFC 7348, IETF, Aug. 2014.
[19] J. Moy, P. Pillay-Esnault, and A. Lindem. Graceful OSPF Restart. RFC 3623, IETF, November 2003.
[20] Open vSwitch – An Open Virtual Switch. `http://www.openvswitch.org`, September 2016.
[21] Open vSwitch version 2.6. `https://github.com/openvswitch/ovs/tree/branch-2.6`, November 2016.
[22] OpenFlow. `http://www.opennetworking.org/sdn-resources/onf-specifications/openflow`, January 2014.
[23] OpenStack – Open Source Software for Creating Private and Public Clouds. `http://www.openstack.org`, September 2016.
[24] ovn-architecture—Open Virtual Network architecture. `http://openvswitch.org/support/dist-docs/ovn-architecture.7.html`.
[25] B. Pfaff and B. Davie. The Open vSwitch Database Management Protocol. RFC 7047, IETF, Dec. 2013.
[26] B. Pfaff, J. Pettit, T. Koponen, E. J. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, K. Amidon, and M. Casado. The Design and Implementation of Open vSwitch. In *Proc. of NSDI*, Oakland, CA, May 2015.
[27] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker. Abstractions for Network Update. In *Proc. of SIGCOMM*, 2012.
[28] Y. Rekhter, T. Li, and S. Hares. A Border Gateway Protocol 4 (BGP-4). RFC 4271, IETF, Jan. 2006.
[29] S. Sangli, E. Chen, R. Fernando, J. Scudder, and Y. Rekhter. Graceful Restart Mechanism for BGP. RFC 4724, IETF, January 2007.
[30] A. Singh et al. Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google's Datacenter Network. In *Proc. of SIGCOMM*, 2015.
[31] M. Slee, A. Agarwal, and M. Kwiatkowsk. Thrift: Scalable Cross-Language Services Implementation. `http://thrift.apache.org/static/files/thrift-20070401.pdf`.
[32] P. Sun, R. Mahajan, J. Rexford, L. Yuan, M. Zhang, and A. Arefin. A Network-state Management Service. In *Proc. of SIGCOMM*, 2014.
[33] Hardware VTEP Database Schema. `http://openvswitch.org/support/dist-docs/vtep.5.pdf`.