



Network Virtualization in Multi-tenant Datacenters

TECHNICAL REPORT

TR-2013-001E

Teemu Koponen, Keith Amidon, Peter Balland, Martín Casado, Anupam Chanda, Bryan Fulton, Igor Ganichev, Jesse Gross, Natasha Gude, Paul Ingram, Ethan Jackson, Andrew Lambeth, Romain Lenglet, Shih-Hao Li, Amar Padmanabhan, Justin Pettit, Ben Pfaff, Rajiv Ramanathan, Scott Shenker*, Alan Shieh, Jeremy Stribling, Pankaj Thakkar, Dan Wendlandt, Alexander Yip, Ronghua Zhang

*International Computer Science Institute & UC Berkeley



Abstract

Multi-tenant datacenters represent an extremely challenging networking environment. Tenants want the ability to migrate unmodified workloads from their enterprise networks to service provider datacenters, retaining the same networking configurations of their home network. The service providers must meet these needs without operator intervention while preserving their own operational flexibility and efficiency. Traditional networking approaches have failed to meet these tenant and provider requirements. Responding to this need, we present the design and implementation of a network virtualization solution for multi-tenant datacenters.

1 Introduction

The work in this paper was necessitated by the merging of two technology trends. The first is the advent of large datacenters. These were initially developed to support large in-house Internet services, but now are also used to provide computing resources to others. These customers (or “tenants”, as we shall refer to them) can avail themselves of a large and professionally managed computational infrastructure on an as-needed basis without investing the significant capital expense or obtaining the extensive operational expertise needed to run their own. However, if this were the entire story, tenants would have to manually port their computations to the cloud to gain these advantages.

This barrier is largely removed by the second technology trend: virtualization. Modern hypervisors support virtual machines (VMs) that provide faithful resource abstractions. In this manner, one can decouple the logical requirements of the computation from the actual physical infrastructure that satisfies them. This decoupling is essential for allowing computations to run, without modification, either locally or in the cloud.

The combination of these two trends—datacenters and virtualization—has led to an increasing number of large multi-tenant datacenters (MTDs) where tenants can seamlessly deploy individual computations in the cloud.

However, to fully move internal operations to the cloud—that is, not just individual VMs but an interacting set of VMs connected by a nontrivial network—MTDs must provide faithful abstractions for all relevant resources in a modern enterprise environment. Computation and storage have long been successfully abstracted in modern hypervisors; however, networking services have not, since tenants are not typically given the ability to manage the cloud’s networking resources. Thus, networking is the last holdout in the virtualization revolution.

From the tenants’ perspective, network virtualization should allow them to configure the way their VMs are connected in the cloud much as they would configure the network in their own enterprise. From an MTD operator’s perspective, network virtualization should allow configuration of the physical network in a way that satisfies all of the tenant-driven requirements, while giving the operator flexibility to deploy a wide range of physical network technologies and topologies.

While a variety of techniques (e.g., VLANs) can isolate tenant traffic in a fairly static manner, there are no currently available techniques for providing the full generality of network virtualization we consider here. Thus, the goal of this paper is to provide the first in-depth discussion of how one supports network virtualization in an MTD. Our solution uses Software-Defined Networking (SDN) as a base technology, and our innovations include a network hypervisor built on top of an SDN controller cluster and various datapath enhancements. Our solution, which we will refer to as the Network Virtualization Platform (NVP), has been in production use for two years.

2 System Design

MTDs have a set of hosts connected by a physical network. Each host has multiple VMs supported by the host's hypervisor. Each host hypervisor has an internal software virtual switch that accepts packets from these local VMs and forwards them either to another local VM or over the physical network to another host hypervisor.

Just as the hypervisor on a host provides the right virtualization abstractions to VMs, we build our architecture around a network hypervisor that provides the right network virtualization abstractions. In this section we describe the network hypervisor and its abstractions.

2.1 Abstractions

A tenant interacts with a network (both its home network, and the network in the MTD) in two ways: the tenant's endpoints (e.g., hosts and/or VMs) send packets, and the tenant's control plane (e.g., routing protocols, manual configuration) configures network elements (e.g., switches and routers). A service provider's network consists of a physical forwarding infrastructure and the system that manages and extends this infrastructure, which is the focus of this paper.

The network hypervisor is a software layer interposed between the provider's physical forwarding infrastructure and the various tenant control planes. Its purpose is to provide the proper abstractions to a tenant's control plane and endpoints, and we describe these abstractions below:

Control abstraction. This abstraction must allow tenants to define a set of logical network elements (or, as we shall call them, datapaths) that they can configure (through their control planes) as they would physical network elements. Each logical datapath is defined by a packet forwarding pipeline interface that, similar to modern forwarding ASICs, contains a sequence of lookup tables, each capable of matching over packet headers and metadata established by earlier pipeline stages. At each stage, packet headers can be modified or the packet can be dropped altogether. The pipeline results in a forwarding decision, which is saved to the packet's metadata, and the packet is then sent out the appropriate port. Since our logical datapaths are implemented in software virtual switches, we have more flexibility than ASIC implementations; datapaths need not hardcode the type or number of lookup tables, and the lookup tables can match over arbitrary packet header fields.

Packet abstraction. This abstraction must enable packets sent by endpoints in the MTD to be given the same switching, routing, and filtering service as they would have in the tenant's home network. This can be accomplished within the packet forwarding pipeline model described above. For instance, to provide basic L2 forwarding semantics, the control plane could populate a single forwarding table with entries explicitly matching on destination MAC address and sending the packets to ports connected to the corresponding VMs. Alternatively, the control plane could install a special learning flow that forwards packets to ports where traffic from the destination MAC address was last received (which will time out in the absence of new traffic) and simply flood unknown packets. Similarly, broadcast destination addresses are handled with a flow entry that sends packets to all logical ports (excluding the port on which the packet was received) on the logical switch.

2.2 Virtualization Architecture

The network hypervisor supports these abstractions by implementing tenant-specific logical datapaths on top of the provider's physical forwarding infrastructure, and these logical datapaths provide the appropriate control and packet abstractions to each tenant.

In our NVP design, we implement the logical datapaths in the software virtual switches on each host, leveraging a set of tunnels between every pair of host-hypervisors (so the physical network sees nothing other than what appears to be ordinary IP traffic between the physical hosts). Almost the entire logical datapath is implemented on the virtual switch where the originating VM resides; after the logical datapath reaches a forwarding decision, the virtual switch tunnels it over the physical network to the receiving host hypervisor, which decapsulates the packet and sends it to the destination VM (see Figure 1). A centralized SDN controller cluster is responsible for configuring virtual switches with the appropriate logical forwarding.

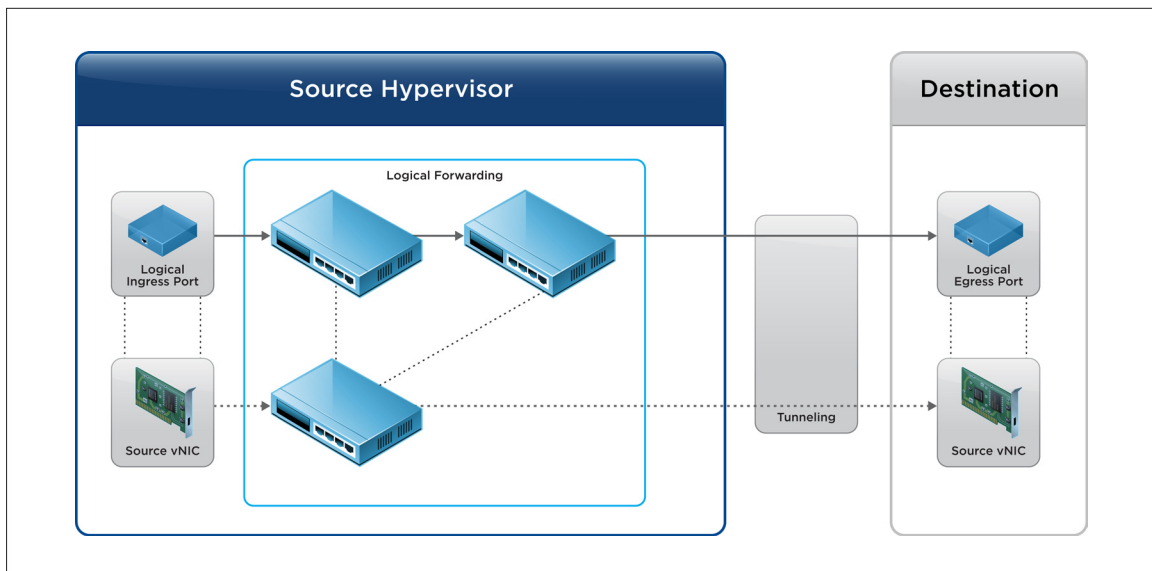


Figure 1. Logical forwarding is implemented by the virtual switch of the originating host hypervisor. After the logical forwarding decision is made, the packet is tunneled across the physical network to the receiving host hypervisor for delivery to the destination VM.

While tunnels can efficiently implement logical point-to-point communication, additional support is needed for logical broadcast or multicast services. For packet replication, NVP constructs a simple multicast overlay using additional physical forwarding elements (x86-based hosts running virtual switching software) called *service nodes*. Once a logical forwarding decision results in the need for packet replication, the packet is tunneled to a service node, which then replicates the packet to all host hypervisors that need a copy for their local VMs.

In addition, some tenants want to integrate their logical network with their existing physical one. This is done through *gateway* appliances (again, x86-based hosts running virtual switching software); all traffic from the physical network goes to the host hypervisor through this gateway appliance, and then can be controlled by NVP (and vice versa for the reverse direction). Gateway appliances can be either within the MTD or at the tenant's remote site. Figure 2 depicts the resulting arrangement of host hypervisors, service nodes, and gateways, which we collectively refer to as transport nodes.

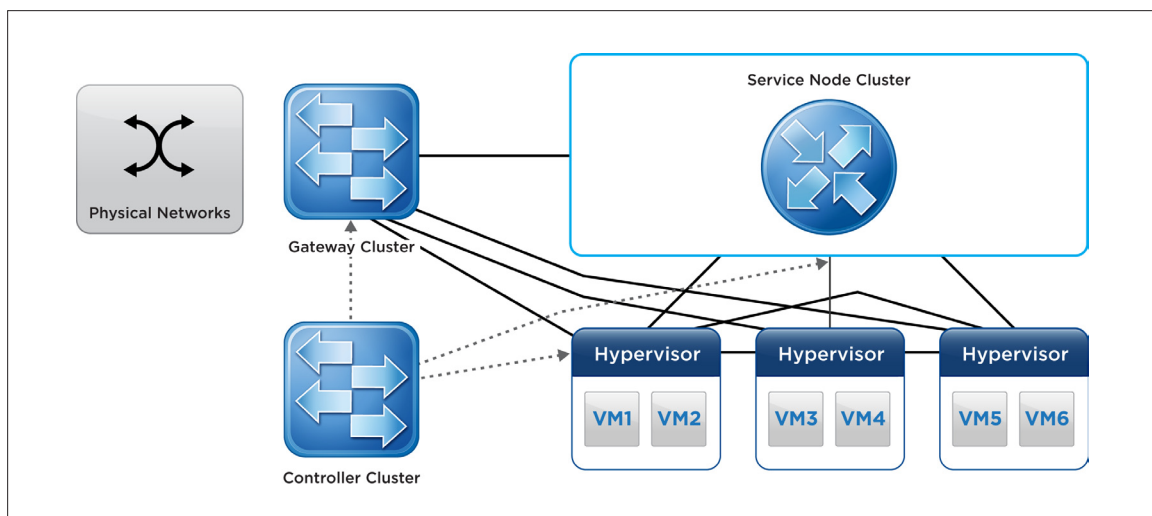


Figure 2. In NVP, a controller cluster manages the forwarding state at all transport nodes (hypervisors, gateways, service nodes), but is not involved in packet routing. Transport nodes are fully meshed over IP tunnels (solid lines). Gateways integrate the logical networks with physical networks, and service nodes provide replication for logical multicast/broadcast.

2.3 Design Challenges

This brief overview of NVP hides many design challenges, three of which we focus on in this paper.

Datapath design and acceleration. NVP relies on software switching. In Section 3 we describe the datapath and the substantial modifications needed to support high-speed x86 encapsulation.

Declarative programming. The controller cluster is responsible for computing all forwarding state and then disseminating it to the virtual switches. To minimize the cost of recomputation, ensure consistency in the face of varying event orders, and promptly handle network changes, we developed a declarative domain-specific language for the controller that we discuss in Section 4.

Scaling the cluster computation. In Section 5 we discuss the various issues associated with scaling the controller cluster.

After we discuss these design issues, we evaluate the performance of NVP in Section 6, discuss related work in Section 7, and then conclude in Section 8.

3 Virtualization Support at the Edge

The endpoints of the tunnels created and managed by NVP are in the virtual switches that run on host hypervisors, gateway nodes, and service nodes. We refer to this collection of virtual switches as the *network edge*. This section describes how NVP implements logical datapaths at the network edge, and how it achieves sufficient data plane performance on standard x86 hardware.

3.1 Implementing the Logical Datapath

NVP uses Open vSwitch (OVS) [23] in all transport nodes (host hypervisors, service nodes, and gateway nodes) to forward packets. OVS is remotely configurable by the NVP controller cluster by two protocols: one that can inspect and modify a set of flow tables (analogous to flow tables in physical switches),¹ and one that allows the controller to create and manage overlay tunnels to transport nodes and discover which VMs are hosted at a hypervisor [22].

The controller cluster uses these protocols to implement packet forwarding for logical datapaths. Each logical datapath consists of a series (*pipeline*) of logical flow tables, each with its own globally unique identifier. The tables consist of a set of *flow entries* that specify expressions to match against the header of a packet, and actions to take on the packet when a given expression is satisfied. Possible actions include modifying a packet, dropping a packet, sending it to a given egress port on the logical datapath, and writing in-memory metadata (analogous to registers on physical switches) associated with the packet and resubmitting it back to the datapath for further processing. A flow expression can match against this metadata, in addition to the packet's header. NVP writes the flow entries for each logical datapath to a single OVS flow table at each virtual switch that participates in the logical datapath.

Any packet entering OVS—either from a virtual network interface card (vNIC) attached to a VM, an overlay tunnel from a different transport node, or a physical network interface card (NIC)—must be sent through the logical pipeline corresponding to the logical datapath to which the packet belongs. For vNIC and NIC traffic, the service provider tells the controller cluster which ports on the transport node (vNICs or NICs) correspond to which logical datapath ports (see Section 5); for overlay traffic, the tunnel header of the incoming packet contains this information. Then, to connect each packet to its logical pipeline, NVP writes flows into the OVS flow table that match a packet based on its ingress port, and adds to the packet's metadata an identifier for the first logical flow table of the packet's logical datapath. As its action, this flow entry resubmits the packet back to the OVS flow table to begin its traversal of the logical pipeline.

¹ We use OpenFlow [21] for this protocol, though any flow management protocol with sufficient flexibility would work.

The control plane abstraction that NVP provides internally for programming the tables of the logical pipelines is largely the same as the interface to OVS's flow table, and NVP writes flow entries directly to OVS, with two important differences:

- **Matches.** Before each logical flow entry is written to OVS, NVP augments it to include a match over the packet's metadata for the logical table's identifier. This enforces isolation from other logical datapaths and places the lookup entry at the proper stage of the logical pipeline. In addition to this forced match, the control plane can program entries that match over arbitrary (logical) packet headers, and can use priorities to implement longest-prefix matching as well as complex ACL rules.
- **Actions.** NVP modifies each logical action sequence of a flow entry to write the identifier of the next logical flow table to the packet's metadata and to resubmit the packet back to the OVS flow table. This creates the logical pipeline, and also prevents the logical control plane from creating a flow entry that forwards a packet to a different logical datapath.

At the end of the packet's traversal of the logical pipeline, it is expected that a forwarding decision for that packet has been made: either drop the packet, or forward it to one or more logical egress ports. In the latter case, a special action is used to save this forwarding decision in the packet's metadata. (Dropping translates to simply not resubmitting a packet to the next logical table.) After the logical pipeline, the packet is then matched against egress flow entries written by the controller cluster according to their logical destination. For packets destined for logical endpoints hosted on other hypervisors (or for physical networks not controller by NVP), the action encapsulates the packet with a tunnel header that includes the logical forwarding decision, and outputs the packet to a tunnel port. This tunnel port leads to another hypervisor for unicast traffic to another VM, a service node in the case of broadcast and multicast traffic, or a gateway node for physical network destinations. If the endpoint happens to be hosted on the same hypervisor, it can be output directly to the logical endpoint's vNIC port on the virtual switch.²

At a receiving hypervisor, NVP has placed flow entries that match over both the physical ingress port for that end of the tunnel and the logical forwarding decision present in the tunnel header. The flow entry then outputs the packet to the corresponding local vNIC. A similar pattern applies to tunnel traffic received by service and gateway nodes.

The above discussion centers on a single L2 datapath, but generalizes to full logical topologies consisting of several L2 datapaths interconnected by L3 router datapaths. In this case, the OVS flow table would hold flow entries for all interconnected logical datapaths, and the packet would traverse each logical datapath by the same principles as it traverses the pipeline of a single logical datapath: instead of encapsulating the packet and sending it over a tunnel, the final action of a logical pipeline submits the packet to the first table of the next logical datapath.

Note that as a practical optimization, we constrain the logical topology such that logical L2 destinations can only be present at its edge.³ This restriction means that the OVS flow table of a sending hypervisor needs only to have flows for logical datapaths to which its local VMs are attached as well as those of the L3 routers of the logical topology.

3.2 Forwarding Performance

OVS, as a virtual switch, must classify each incoming packet against its entire flow table in software. However, the flow entries written by NVP can contain wildcards for any irrelevant parts of a packet header, and constant time flow lookups would benefit from TCAMs not available on standard x86 hardware.⁴

To achieve efficient flow lookups on standard x86 hardware, OVS exploits *traffic locality*: the fact that all packets belonging to a single flow of traffic (e.g., one of a VM's TCP connections) will traverse exactly the same set of flow entries. OVS consists of a kernel module and a userspace program. The kernel module sends the first packet of

² For brevity, we don't discuss logical MAC learning or stateful matching operations; but, in short, the logical control plane can provide actions that create new lookup entries in the logical tables, based on incoming packets. These primitives allow the control plane to implement L2 learning and stateful ACLs, in a manner similar to advanced physical forwarding ASICs.

³ We have found little practical value in supporting logical routers that are interconnected through logical switches without any tenant VMs.

⁴ There are many papers on the problem of packet classification without TCAMs. See for instance [10, 30].

each new flow into userspace, where it is matched against the full flow table, including wildcards, as many times as the logical datapath traversal requires. Then, the userspace program installs *exact-match flows* into a flow table in the kernel, which contain a match for every part of the flow (L2–L4 headers). Future packets in this same flow can then be matched entirely by the kernel. Existing work considers flow caching in more detail [3, 17].

While exact-match kernel flows alleviate the challenges of flow classification on x86, NVP’s encapsulation of all traffic can introduce significant overhead. This overhead does not tend to be due to tunnel header insertion, but to the operating system’s inability to enable standard NIC hardware offloading mechanisms for encapsulated traffic.

There are two standard offload mechanisms relevant to this discussion. TCP Segmentation Offload (TSO) allows the operating system to send TCP packets larger than the physical MTU to a NIC, which then splits them into MSS-sized packets and computes the TCP checksums for each packet on behalf of the operating system. Large Receive Offload (LRO) does the opposite and collects multiple incoming packets into a single large TCP packet and, after verifying the checksum, hands it to the operating system. The combination of these mechanisms provides a significant reduction in CPU usage for high-volume TCP transfers. Similar mechanisms exist for UDP traffic; the generalization of TSO is called Generic Segmentation Offload (GSO).

Current Ethernet NICs do not support offloading in the presence of any IP encapsulation in the packet. That is, even if a VM’s operating system would have enabled TSO (or GSO) and handed over a large frame to the virtual NIC, the virtual switch of the underlying hypervisor would have to break up the packets into standard MTU-sized packets and compute their checksums before encapsulating them and passing them to the NIC. Today’s NICs are simply not capable of seeing into the encapsulated packet.

To overcome this limitation and re-enable hardware offloading for encapsulated traffic with existing NICs, NVP uses an implementation of a recently introduced encapsulation method called STT as its default tunnel type [5].⁵ STT encapsulation places a standard, but fake, TCP header format after the physical IP header. After this, there is a shim header including contextual information that specifies, among other things, the logical destination of the packet. The actual logical packet (starting with its Ethernet header) follows. As a NIC processes an STT packet, it will first encounter this fake TCP header, and consider everything after that to be part of the TCP payload. Thus, the NIC can employ its standard offloading mechanisms.

Although on the wire the STT encapsulation header looks like standard TCP header, the STT protocol is stateless and requires no TCP handshake procedure between the tunnel endpoints. VMs obviously can (and will) run TCP over the logical packets exchanged over the encapsulation.

Placing contextual information into the encapsulation header, at the start of the fake TCP payload, allows for a second optimization: this information is not transferred in every physical packet, but only once for each large packet sent to the NIC. Therefore, the cost of this context information is amortized over all segments produced out of the original packet, and additional information (e.g., context useful in debugging a live NVP system) can be included as well.

Using hardware offloading in this way comes with a significant downside: gaining access to the logical traffic and contextual information requires reassembling the segments, unlike with traditional encapsulation protocols in which every datagram seen on wire has all headers in place. This limitation makes it difficult, if not impossible, for the high-speed forwarding ASICs used in hardware switch appliances to inspect encapsulated logical traffic. However, we have found such appliances to be rare in NVP production deployments. Another complication is that STT uses its own TCP transport port in the fake TCP header, which may require administrators to punch firewall holes in middleboxes in the physical network. For such environments where compliance is more important than efficiency, NVP supports other more standard IP encapsulation protocols.

⁵ NVP also supports other tunnel types, such as GRE [6] and VXLAN [31] for reasons discussed shortly.

3.3 Fast Failovers

Providing highly available dataplane connectivity is a priority for NVP. Logical traffic between VMs flowing over a direct hypervisor-to-hypervisor tunnel clearly cannot survive the failure of either hypervisor, and must rely on path redundancy provided by the physical network to survive the failure of any physical network elements. However, the failure of any of the new networking appliances that NVP introduces—service and gateway nodes—must cause only minimal, if any, dataplane outage.

For this reason, NVP deployments typically have multiple service nodes, to ensure that any one service node failure does not disrupt logical broadcast and multicast traffic. The NVP controller cluster instructs hypervisors to load balance their traffic requiring packet replication across a bundle of all service node tunnels by using flow hashing algorithms similar to ECMP [11]. The hypervisor monitors these tunnels using a simple heartbeat protocol loosely based on 802.1ag CFM [14]. If the hypervisor fails to receive heartbeats from a particular service node for a configurable period of time, it removes (without involving the controller cluster) the failed service node from the load-balancing tunnel bundle and continues to use the remaining service nodes.

As discussed in Section 2, gateway nodes bridge logical networks and physical networks. For the reasons listed above, NVP deployments typically involve multiple gateway nodes for each bridged physical network. Hypervisors monitor their gateway tunnels and fail over to backups, in the same way they do for service tunnels. However, having multiple points of contact with a particular physical network presents a problem: NVP must ensure that no loops between the logical and physical networks are possible. If a gateway blindly forwarded logical traffic to the physical network, and vice versa, any traffic sent by a hypervisor over a gateway tunnel could wind up coming back into the logical network via another gateway attached to the same network, due to MAC learning algorithms running in the physical network.

NVP solves this by having each cluster of gateway nodes (those bridging the same physical network) elect a leader among themselves. Any gateway node that is not currently the leader will disable its hypervisor tunnels and will not bridge traffic between the two networks, eliminating the possibility of a loop. Gateways bridging a physical L2 network use a lightweight leader election protocol, whereby each gateway broadcasts CFM packets onto that L2 network and listens for broadcasts from all other known gateways. Each gateway runs a deterministic algorithm to pick the leader, and if it fails to hear broadcasts from that node for a configurable period of time, it picks a new leader.⁶ Broadcasts from an unexpected gateway cause all gateways to disable their tunnels to prevent possible loops.

One final wrinkle is that, because a gateway node is acting as a switch to hosts on its attached logical network, its attached physical network will learn that the path to the MAC address for each VM on that logical network should go through the acting leader. If a new gateway becomes leader, the physical network's learning tables must be updated. Depending on network configuration, this can happen one of two ways. Typically, the new lead gateway sends a reverse ARP for each MAC address that could be affected. Alternatively, the gateway nodes may be configured to participate in the physical network's Spanning Tree using 802.1d [13]. On failover, the new lead gateway will send a topology change notification, flushing the network's learning tables.

⁶ Leader election for gateways bridging an L3 physical network works on the same principles, though the details differ slightly.

4 Forwarding State Computation

In this section, we describe how NVP computes the forwarding state for the virtual switches. We focus on a single controller and defer discussion about distributing the computation over a cluster to the following section.

4.1 Computational Structure of Controller

The controller inputs and outputs are structured as depicted in Figure 3. First, hypervisors and gateways provide the controller with location information for vNICs over the configuration protocol (1), updating this information as virtual machines migrate. Hypervisors also provide the MAC address for each vNIC. Second, service providers configure the system through the NVP API (see the following section) (2). This configuration state changes as new tenants enter the system, as logical network configuration for these tenants change, and as the physical configuration of the overall system (e.g., the set of managed transport nodes) changes.

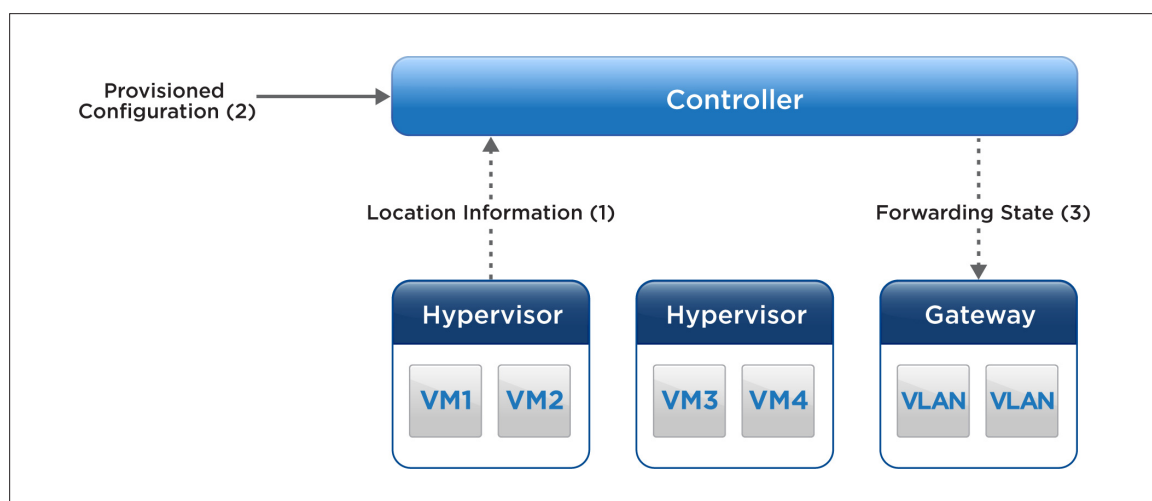


Figure 3. Inputs and outputs to the forwarding state computation process.

Based on these inputs the controller computes forwarding state (3), which is pushed to transport nodes via OpenFlow and the OVS configuration protocol. OpenFlow flow entries model the full logical packet forwarding pipeline, as discussed in the previous section. The controller also computes OVS configuration database entries consisting of the tunnels connecting hypervisors, gateways, and service nodes, and any local queues and scheduling policies.

4.2 Computational Challenge

The input and output domains of the controller logic are complex: in total, the controller uses 123 types of input to generate 81 types of output. The input state is large, being proportional to the size of the MTD, and changes frequently as VMs migrate, tenants enter and exit, and tenants reconfigure their logical networks.

The controller needs to react quickly to these input changes. Given the large total input size and frequent, localized input changes, a naïve implementation that reruns the full input-to-output translation on every change would be computationally inefficient. Incremental computation allows us to recompute only the affected state and push the delta down to the network edge. We first used a hand-written state machine to incrementally compute and update the forwarding state in response to input change events. However, we found this approach to be impractical due to the number of event types that need to be handled as well as their arbitrary interleavings. Event handling logic must account for dependencies on previous or subsequent events, deferring work or rewriting previously generated outputs as needed. In many languages, such code degenerates to a reactive, asynchronous style that is difficult to write, comprehend, and test. Ensuring correct handling of all events and their orderings quickly becomes an intractable problem as the size of the codebase grows.

4.3 Automatically Generating an Incremental State Machine

To overcome this problem, we implemented a domain-specific, declarative language called nlog for computing the network forwarding state. It allows us to separate logic specification from the state machine that implements the logic. The logic is written in a declarative manner that specifies a *function* mapping the controller input to output, without worrying about state transitions and input event ordering. The state transitions are handled by a compiler that generates the event processing code and by a runtime that is responsible for consuming the input change events and recomputing all affected outputs.

Declarations in nlog are Datalog queries: a single declaration is a join over a number of tables that produces immutable tuples for a *head table*. Any change in the joined tables results in (incremental) re-evaluation of the join and possibly in adding tuples to, or removing tuples from, this head table. Joined tables may be either *input tables* representing external changes or *internal tables* holding only results computed by declarations. Head tables may be internal tables or *output tables* which cause changes external to the nlog runtime based on tuples added to, or removed from, the table. The nlog language does not currently support recursive declarations or negation. Table 1 summarizes NVP's use of nlog.

| | |
|--|-------|
| System input types | 123 |
| System output types | 81 |
| All tables (internal, input, and output) | 889 |
| Declarative rules | 1224 |
| Lines of declarative code | 27000 |

Table 1. nlog stats.

The following code is an example of a simplified nlog declaration for creating tunnel endpoints. The tunnels depend on the API-provided information, such as the logical datapath configuration and the tunnel encapsulation, as well as hypervisor-provided vNIC locations.

```
tunnel(phys_switch_id, encap_type, dst_ipaddr) :-
    log_port_present(log_port_id, phys_switch_id),
    log_port_present(log_port_id_2,
                    other_phys_switch_id),
    phys_switch_connector(other_phys_switch_id,
                        encap_type, dst_ipaddr),
    log_port(log_port_id, log_datapath_id),
    log_port(log_port_id_2, log_datapath_id),
    log_datapath_encap(log_datapath_id, encap_type);
```

The above declaration updates the head table tunnel for each logical port in the logical datapath identified by *log_datapath_id*. The head table is an output table consisting of rows each with three data columns; changes to this table create or destroy tunnels to a remote hypervisor (identified by *dst_ipaddr*) on a hypervisor identified by *phys_switch_id* for a specific tunnel encapsulation type (*encap_type*) provided by the API. As the VMs migrate, the *log_port_present* input table is updated for the corresponding vNICs, and changes to the *tunnel* output table cause tunnels to be created or removed as necessary on the corresponding hypervisors.

Decoupling event ordering from control logic definition improves development and testing. Developers can focus on how the forwarding state is derived; the resulting program code is high-level and compact. Validation work is substantially reduced: such testing need only focus on the final output and does not have to explore every sequence of state transitions.

Even though the incremental update model allows quick convergence after changes, it is not intended for reacting to dataplane failures at dataplane time scales. For this reason, NVP precomputes any state necessary for dataplane failure recovery. For instance, the forwarding state computed for tunnels includes any necessary backup paths to allow the virtual switch running on a transport node to react instantly and independently to network failures (see Section 3.3).

Language Extensions. The joins mentioned earlier can only rearrange existing column data. Because most nontrivial programs also transform column data, nlog provides extension mechanisms for specifying transformations in C++.

First, a developer can implement and install a *function table*, which can be considered to be a virtual table where certain columns of a row are a stateless function of others. For example, a function table could compute the sum of two integer columns and place it in a third column, or merge two OpenFlow flow matches (each matching over a different protocol header) into one flow matching over both headers. Function tables are restricted to operate on a single tuple. The base language provides various functions for primitive column types (e.g., ints, strings, UUIDs). NVP extends these with functions operating over flow and action types, which are used to construct the complex match expressions and action sequences that constitute the flow entries that model logical datapaths.

Second, if more complicated transformations are required, developers can hook together a pair of an output and an input table through arbitrary C++ code. Declarations produce tuples into the output table, which are fed to the output table implementation, where tuples are transformed in C++ and fed back to nlog through the input table. While this complicates both the C++ code and the nlog declarations, this extension imposes few restrictions on language extensions. For instance, we use this technique to implement a hysteresis table that dampens external events (such as a network link flapping) by queueing tuples for a small amount of time in C++.

Compiler and Runtime. The compiler is fairly simple. Given a single declaration (a join over nlog tables), the compiler generates a query plan for every type of event (i.e., tuples being added to or removed from the table) that can trigger incremental re-evaluation of the declaration. We currently use static query planning: using representative table snapshots saved when the system is in steady state, a heuristic algorithm searches for the best possible plan by sampling the tables in the snapshot. The compiler emits a query plan that is loaded at runtime.

The nlog runtime is written in C++ and is single threaded. The input tuples are fed into a queue from which the runtime system takes one tuple at a time and runs through all possible queries that can be affected by this tuple. This in turn produces more internal tuples which may trigger further queries. This evaluation loop is repeated until we reach a fixed point at which no more tuples are produced. At this point, the nlog runtime moves onto the next input event. The nlog runtime has a primitive scheduler that can be used to prioritize computation across various logical datapaths. When processing events that require recomputation of multiple logical datapaths, this scheduling reduces the average recomputation time across all affected logical datapaths.

For validation and debugging, the runtime system can snapshot the runtime state at arbitrary points. Offline inspection of such state has proven to be helpful in quickly pinpointing problems in forwarding logic. The offline state is also used by various automated tests of the controller cluster.

5 Controller Cluster

In this section we discuss the design of the controller cluster: the distribution of physical forwarding state computation to provide the logical datapaths, the auxiliary distributed services the distribution of the computation requires, and finally the implementation of the API provided for the service provider.

5.1 Scaling and Availability of Computation

Scaling. The forwarding state computation is easily parallelizable, and NVP divides computation into a loosely coupled two-layer hierarchy, with each layer consisting of a cluster of processes running on multiple controllers. All computation is implemented in nlog as discussed in the previous section.

Figure 4 illustrates the two-layer distributed controller cluster. The top layer consists of *logical controllers*. The cluster is sharded over logical datapaths. By isolating the computation for each logical datapath we can execute their computation on different controllers.

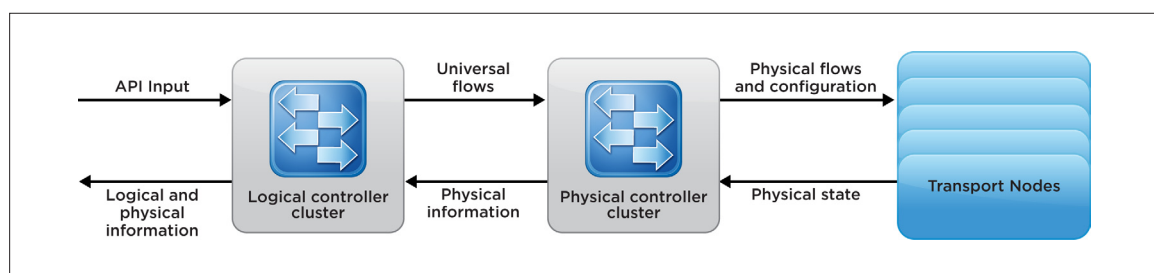


Figure 4. NVP controller cluster.

Logical controllers compute the flows and tunnels needed to implement logical datapaths as discussed in Section 3. All computed flow entries—including the logical datapath lookup tables provided by the logical control planes, instructions to create tunnels, and queues for the logical datapath—are encoded as *universal flows*. Universal flows are an intermediate representation that is similar to OpenFlow but which abstracts out all network edge specific details such as ingress, egress, or tunnel port numbers. Instead of specific port numbers, the universal flows contain abstract identifiers for ingress, egress, or tunnel ports. The universal flows are published over RPC to the bottom layer consisting of *physical controllers*.

Physical controllers are responsible for the communication to hypervisors, gateways, and service nodes. They translate the location-independent portions of universal flows with node and location-specific state, such as IP addresses and physical interface port numbers, as well as create the necessary config protocol instructions to establish tunnels and queue configuration. The resulting *physical flows*, which are now valid OpenFlow instructions, and configuration protocol updates are then pushed down to the edge nodes. Because the universal-to-physical translation can be executed independently for every transport node, the cluster is sharded over the managed transport nodes.

This arrangement is convenient from the sharding point of view, but it also reduces the computational complexity of the forwarding state computation. The avoidance of location-specific details allows the logical controller layer to compute a single “image” for a single ideal transport node (having $O(N)$ tunnels to remote transport nodes), without considering the tunnel mesh between all transport nodes in its full $O(N^2)$ complexity.

Logical controllers also run the logical control planes to compute the logical lookup table content based on the high-level configuration provided through the API. Control planes are co-located in the logical controller doing the computation for the corresponding logical datapath.

Availability. To provide failover within the cluster, NVP provisions hot standbys at both logical and physical controller layers by exploiting the sharding mechanism. The sharding coordinator ensures that every shard is assigned with one controller acting as the master and the other controller(s) acting as the hot standby. On detecting the failure of the master of a shard, the sharding coordinator promotes the standby for the shard to be the master of the shard, and assigns a new controller instance as the standby for the shard. On detecting the failure of the standby for a shard, the sharding coordinator assigns a new standby for the shard.

Because of their large population, transport nodes do not participate in the cluster coordination. Instead, OVS instances are configured to connect to both the master and the standby physical controllers for their shard; upon detecting master failure, they switch to the standby.

5.2 Distributed Services

NVP implements a number of distributed services to support the distribution of the computation over the controller cluster.

Leader election. Each controller must know which slices it manages, and must also know when to take over responsibility of slices managed by a controller that has disconnected. Consistent hashing [16] is one possible approach, but it tends to be most useful in very large clusters. With only tens of controllers, we find it advantageous to elect a leader using Zookeeper [12], and have that leader assign responsibilities directly to each controller. This approach makes it easier to implement sophisticated assignment algorithms that can ensure, for instance, that each controller has equal load and that assignment churn is minimized as the cluster membership changes. If the leader fails or becomes disconnected from the majority of nodes, a new leader will be elected in its place and will maintain as many of the previous assignments as possible. Any logical or physical controller participates in this leader election process with all other logical or physical controllers respectively.

Label allocation. A network packet encapsulated in a tunnel must carry a label that denotes the logical network to which the packet belongs, so the destination can properly process it. This identifier must be globally unique at any point in time in the network, to ensure data isolation between different logical networks. Because encapsulation rules for different logical networks may be calculated by different NVP controllers, the controllers need a mechanism to pick unique labels and ensure they will stay unique in the face of controller failures. Furthermore, the identifiers must be relatively small to minimize packet overhead. We again use Zookeeper to implement a consistent label allocator that ensures labels will not be reused until the corresponding logical datapath is deleted. Each controller can claim a section of the label space and allocate labels to logical networks as needed to avoid accessing Zookeeper for each assignment. This label allocation service is used by the logical controllers to assign the logical network labels, which are then disseminated to the physical controllers via the universal flows.

Configuration storage. NVP implements a replicated state machine [26] on top of Zookeeper to power a persistent datastore for NVP's configuration state (data written through the API exposed by NVP, discussed shortly below). The state machine serializes updates to this state and, if requested by the API user, blocks a controller from executing a new API request until that controller has received all previous updates. It also provides a mechanism for saving a consistent snapshot of the configuration state for archival and rollback purposes.

An NVP controller cluster will typically be installed and maintained by datacenter administrators, some of whom may not have much experience running distributed infrastructure. This is in contrast to most distributed infrastructure, where expert administrators maintain a cluster entirely in-house. To make NVP as friendly as possible to inexperienced users, we have built an administration layer on top of Zookeeper that automates many of the normally manual aspects of configuring and troubleshooting a distributed cluster.

5.3 API for Service Providers

To support integration of the service provider cloud management system, NVP exposes an HTTP API in which network elements, physical or logical, are presented as objects. Examples of physical network elements include transport nodes, while logical switches, ports, and routers are logical network elements. Changes to logical elements are listened for by the tenant logical control plane, causing them to enable/disable features accordingly. The cloud management system would use these APIs to provision tenant workloads, and a command-line or a graphical shell implementation could map these APIs to a human-friendly interface for service provider administrators and customers.

A single API operation can include state from multiple transport nodes, or include both logical and physical information. Thus, API operations generally merge information from multiple logical or physical controllers. Depending on the operation, NVP may retrieve information on-demand from multiple controllers in response to a specific API request, or proactively by continuously collecting the necessary state from multiple sources to a single aggregation point.

NVP also provides interfaces for monitoring network traffic. Providers can configure taps to the logical traffic, and NVP instructs the virtual switches to pass the matching packets over standard, lightweight protocols [27] already used by physical routers and switches.

6 Evaluation

In this section, we present measurements both for the NVP controller cluster and the edge datapath implementation. For these tests, we use NVP's product testing harness, through which we validate each NVP release before making it available to customers.

6.1 Controller Cluster

Setup. The configuration in the following tests has 3,000 simulated hypervisors, each with 21 virtual interfaces for a total of 63,000 logical ports. In total, there are 7,000 logical datapaths, each coupled with a logical control plane modeling a logical switch. The average size of a logical datapath is 9 ports, but the size of each logical datapath varies from 2 to 64. The test also configures the logical control planes to use port-specific ACLs on 49,188 of the logical ports and generic isolation ACLs for 1,553 of the logical switches.

The test control cluster has three nodes. Each controller is a bare-metal Intel Xeon E5645 2.4GHz machine with 12 cores, 96GB of memory, and 400GB hard disk. The physical network is a dedicated switched test network.

Each simulated hypervisor is a Linux VM that contains an OVS instance with a TUN device simulating each virtual interface on the hypervisor. The simulated hypervisors run within physical hypervisors running XenServer 5.6, and are connected via Xen bridges to the physical network. We ran four types of tests to simulate conditions the cluster may face.

Cold Start. The cold start test simulates bringing the entire system back online after a major datacenter disaster in which all servers crash and all volatile memory is lost. In particular, the test starts with a fully configured system in a steady state, shuts down all controllers, clears the flows on all OVS instances, and restarts everything.

Restore. The restore test simulates a milder scenario where the whole control cluster crashes and loses all volatile state but the dataplane remains intact.

Failover. The failure test simulates a failure of a single controller within a cluster.

Steady State. In the steady state test, we start with a converged idle system. We then add 10 logical ports to existing switches through API calls, wait for connectivity correctness on these new ports, and then delete them. This simulates a typical usage of NVP, as the service provider provisions logical network changes to the controller as they arrive from the tenant.

In each of the tests, we send a set of pings between logical endpoints and check that each ping either succeeds if the ping is supposed to succeed, or fails if the ping is supposed to fail (e.g., when a security policy configuration exists to reject that ping). The pings are grouped into rounds, where each round measures a sampling of logical port pairs. We continue to send ping rounds until all pings have the desired outcome and the controllers finish processing their pending work. The time between the rounds of pings is 5–6 minutes in our tests.

While the tests are running, we monitor the sizes of all the nlog tables. From this, we can deduce the number of flows computed by nlog, since these are stored in a single table. Because nlog is running in a dedicated thread, we measure the time this thread was running and sleeping to get the load for nlog computation.

Results. The left graph of Figure 5 shows the percentage of correct pings over time for the cold start test, beginning at time 0. It starts at 17% because 17% of the pings are expected to fail, which they do in the absence of any flows pushed to the datapath. Note that unlike typical OpenFlow systems, NVP does not send packets for unclassified flows to the controller cluster; instead, NVP precomputes all necessary flows on each configuration change. Thus, the cold-start case represents a worst-case scenario for NVP: the controller cluster must compute all state and send it to the transport nodes before connectivity can be 100% established. Although it takes NVP nearly an hour to achieve full connectivity in this extreme case, the precomputed flows greatly improve dataplane performance at steady state.

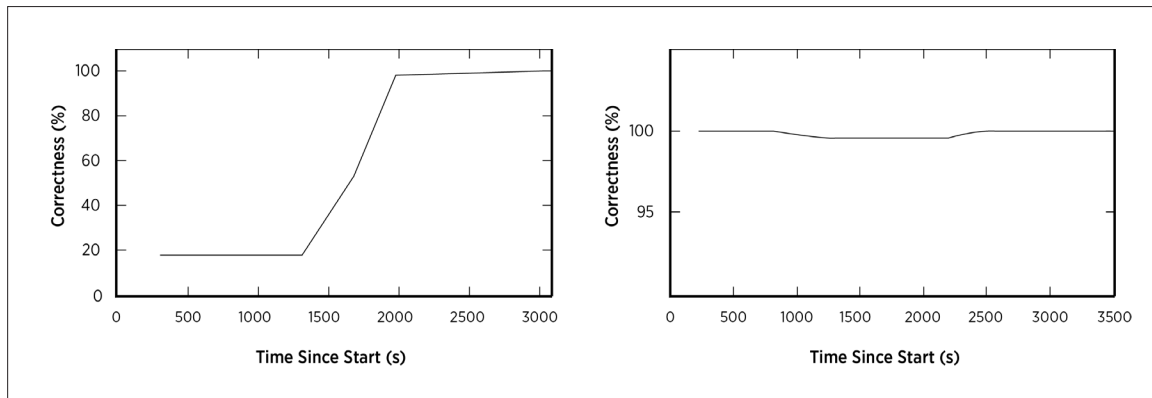


Figure 5. Cold start (left) and restore (right) connectivity correctness.

The connectivity correctness is not linear for two reasons. First, NVP does not compute flows for one logical datapath at a time, but does so in parallel for all of them. Second, for a single ping to start working, the correct flows need to be setup on all the transport nodes on the path of the ping (and ARP request/response, if any). The first reason is an implementation artifact stemming from the arbitrary execution order in nlog. We are working on adding a better scheduler to nlog runtime.

The right graph of Figure 5 shows the percentage of correct pings over time for a restore test, beginning at time 0. Since the datapath state is left in place by this test, the connectivity correctness remains almost at 100% throughout the test with a short dip of about 1%. The dip is due to an implementation detail of our label allocation algorithm that causes inconsistent labels for given logical datapaths across a restart; we are actively addressing this.

Figure 6 shows the percentage of correct pings over time during a controller failure. The connectivity correctness remains at 100% during the test. We note the connectivity is equally well maintained in the case of adding or removing controllers to the cluster, but we don't include a corresponding graph here.

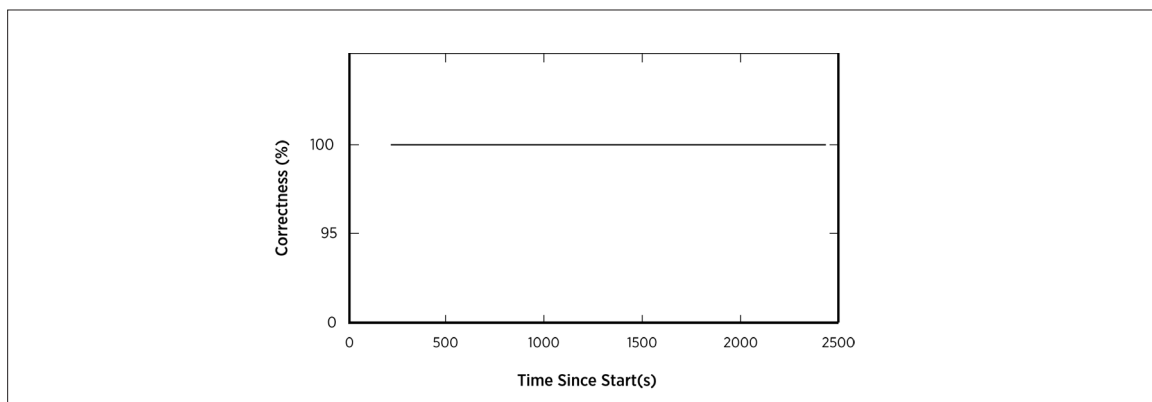


Figure 6. Ping correctness during a controller failure.

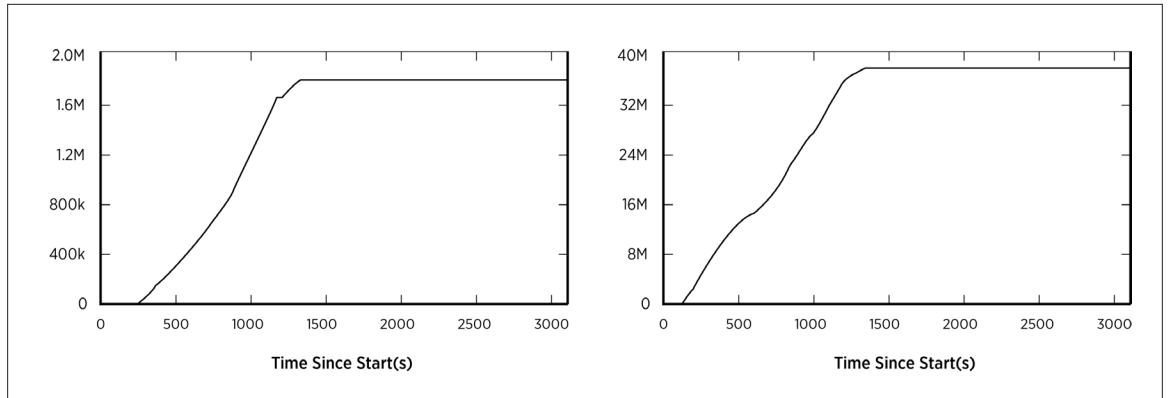


Figure 7. Total physical flows (left) and nlog tuples (right) in one controller after a cold start.

Figure 7 shows the total tuples as well as total flows produced by nlog on a single controller over time during the cold start test. The graphs show that nlog is able to compute about 1.8M flows in about 20 minutes, involving about 37M tuples in total across all nlog tables. This means that to produce 1 final flow, we have an average of 20 intermediary tuples, which points to the complexity of incorporating all possible factors that can affect a flow. After converging, the measured controller uses approximately 27G of memory, as shown in Figure 8.

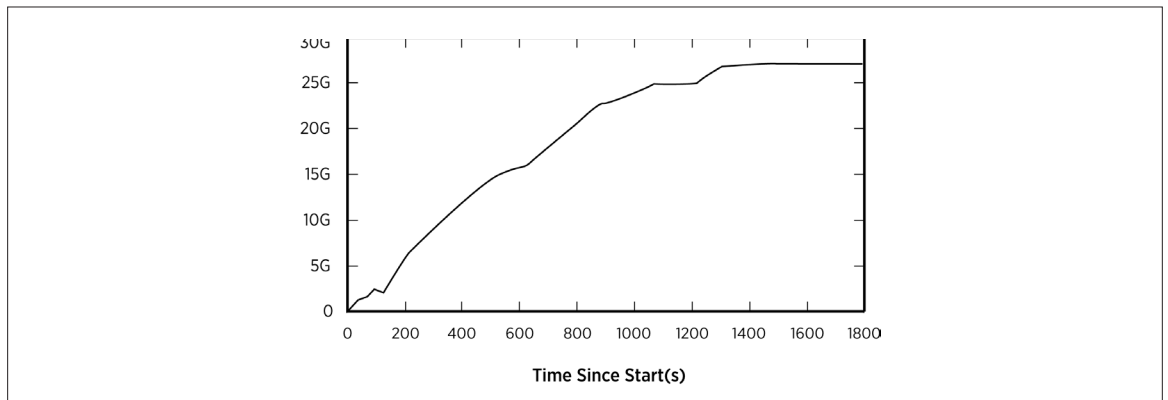


Figure 8. Memory used by a controller after a cold start.

Since our test cluster has 3 controllers, 1.8M flows is $\frac{2}{3}$ of all the flows in the system, because this one controller is the master for $\frac{1}{3}$ of the flows and standby for $\frac{1}{3}$ of the flows. Additionally in this test, nlog produces about 1.9M tuples per minute on average. At peak performance, it produces up to 10M tuples per minute.

Figure 9 shows nlog load during the cold start (left) and steady state (right) tests. In the cold start test, nlog is almost 100% busy for 20 minutes. This shows that a controller can read its database and connect to the switches (and thereby populate nlog input tables) faster than nlog can process it. Thus, nlog is the bottleneck during this part of the test. During the remaining time, NVP is sending the computed state to each hypervisor. During the remaining time, NVP is sending the computed state to each hypervisor.

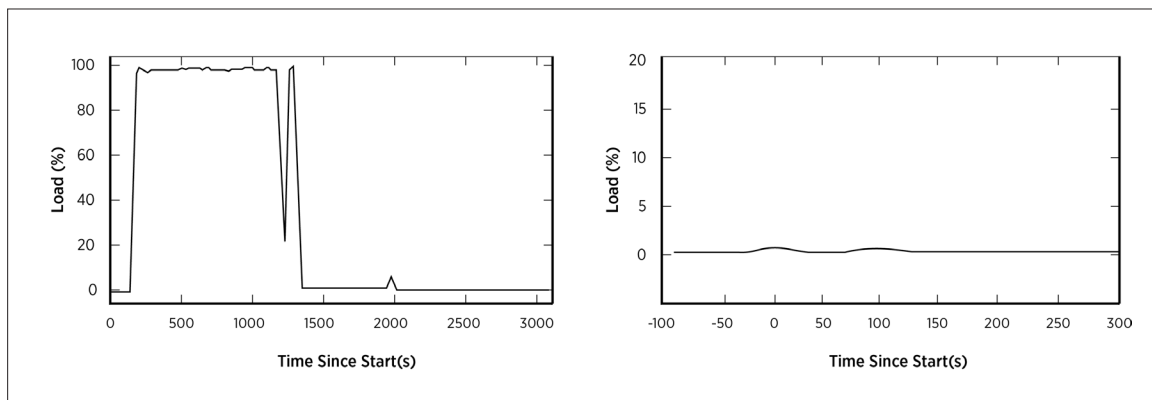


Figure 9. Left: nlog load during coldstart. Right: nlog load while adding (at 0 sec) and removing 10 ports (at 90s).

The steady state test shows nlog’s ability to process incremental changes to inputs. Addition of 10 logical ports (to existing 63,000) results in less than 0.5% load for a few seconds. Deleting these ports results in similar load. This test represents the usual state of a real deployment—constantly changing configuration at a modest rate.

6.2 Transport Nodes

We now look at the performance of the network edge.

| | NO ENCAP | STT | GRE |
|-------------|----------|---------|---------|
| TX CPU load | 49% | 49% | 85% |
| RX CPU load | 72% | 119% | 183% |
| Throughput | 9.3Gbps | 9.3Gbps | 2.4Gbps |

Table 2. Non-tunneled, GRE, and STT performance benchmark.

Tunnel Performance. Table 2 shows the throughput and CPU overhead of using non-tunneled, STT, and GRE to connect two hypervisors. We measured throughput using Netperf’s TCP_STREAM test. Tests ran on two Intel Xeon E5-2650 2.0GHz servers with 8 cores, 32GB of memory, and Intel 10Gb NICs, running Ubuntu 12.04 and KVM. The CPU load represents the percentage of a single CPU core used, which is why the result may be higher than 100%. All results take into account only the CPU used to switch traffic in the hypervisor, and not the CPU used by the VMs. The test sends a single flow between two VMs on the different hypervisors.

In the test we see that the throughput of GRE is much lower and requires more CPU than either of the other methods due to the inability to use hardware offloading. However, STT’s use of the NIC’s TCP Segmentation Offload (TSO) engine makes its throughput performance comparable to non-tunneled traffic between the VMs. STT uses more CPU on the receiving side of the tunnel because, although it is able to use LRO to coalesce incoming segments, LRO does not always wait for all packet segments before delivering a coalesced packet to the operating system. STT requires the full set of segments before it can remove the encapsulation headers and deliver the original logical packet, and so on these occasions it must perform the remaining coalescing in software.

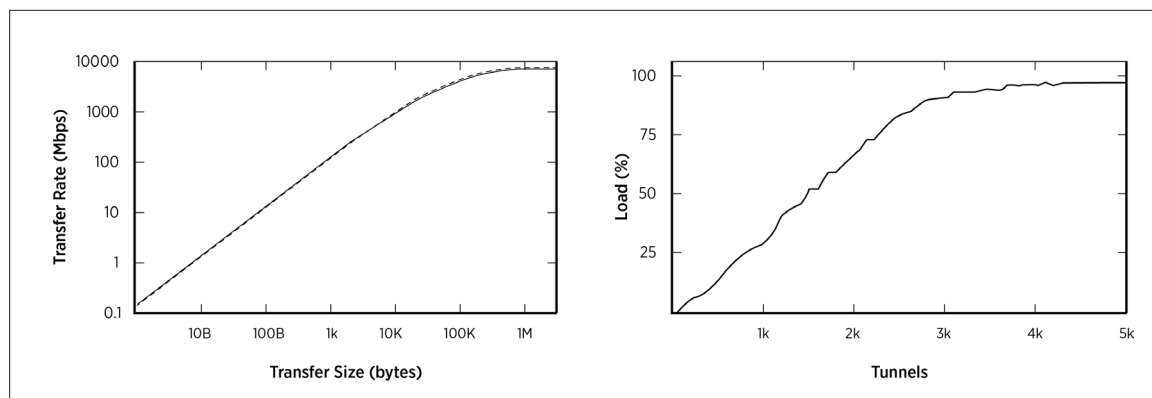


Figure 10. Left: VM experienced transfer throughput with OVS (solid line) and the Linux bridge (dashed line) on varying transfer sizes. Right: tunnel management CPU load as a function of number of tunnels managed.

Connection Set Up. The left graph of Figure 10 shows the throughput for varying lengths of a NetPIPE transfer between two VMs on the same hypervisor, from one of the tenant VM's point of view. This transfer goes directly from one VM to another through the hypervisor's virtual switch without using any encapsulation, and the test compares the performance of these transfers using the native Linux network bridge to that of OVS. The results show that OVS performance is essentially identical to that of the Linux bridge, in terms of virtual switching.

Tunnel Scale. The right graph of Figure 10 shows the tunnel keepalive message processing cost as the number of tunnels increases. This test is especially relevant for our gateways and service nodes, which have tunnels to potentially large numbers of hypervisors and must respond to keepalives on all of these tunnels. The test sends heartbeats at intervals of 300ms, and the results indicate the tested node can process and respond to them in a timely manner for up to 3000 tunnels. Future work is needed to reduce the required CPU resources further, and to parallelize heartbeat processing.

7 Related Work

NVP borrows from recent advances in datacenter network design, software forwarding, programming languages, and software-defined networking, and thus the scope of related work is vast. Due to limited space, we only touch on a few topics where we feel it useful to distinguish our work from previous efforts. First, while NVP relies on SDN (see [1, 2, 8, 9, 18, 21]), in the form of an OpenFlow forwarding model and a control plane managed by a controller, NVP requires significant extensions to this basic SDN paradigm to provide network virtualization.

Virtualization of the network forwarding plane was first proposed in [4]; NVP develops this concept further and provides a detailed design of an edge-based implementation. However, network virtualization as a general concept has existed since the invention of VLANs that slice Ethernet networks. Slicing as a mechanism to share resources is available at various layers: IP routers are capable of running multiple control planes over one physical forwarding plane [25], and Flowvisor introduced the concept of slicing to OpenFlow and SDN [29]. However, while slicing provides isolation, it does not provide either the packet or control abstractions that enable tenants to live within a faithful logical network. VMs were proposed as a way to virtualize routers, [32] but this is not a scalable solution for MTDs.

NVP uses a domain-specific declarative language for efficient, incremental computation of all forwarding state. Expressing distributed (routing) algorithms in datalog [19, 20] is the most closely related work, but it focuses on concise, intuitive modeling of distributed algorithms. Our focus is on structuring the computation within a single node to allow efficient incremental computation. Frenetic [7] and its subsequent refinements use reactive functional programming to simplify the implementation of packet forwarding decisions, but they are focused on reactive packet processing rather than the proactive computations considered here.

8 Discussion

After having presented the basic design and its performance, we now return to discuss two points: which aspects of the design were most critical to NVP's success, and what are the implications of NVP for SDN in general.

8.1 Seeds of NVP's Success

Basing NVP on a familiar abstraction. While one could debate what abstraction best facilitates the management of tenant networks, the key design decision (which looks far more inevitable now than four years ago when we began this design exercise) was to make logical networks look exactly like current network configuration. Even though current network control planes have many flaws, they represent a large installed base. NVP enables tenants to use their current network policies *without modification* in the cloud, which greatly facilitates adoption of both NVP and MTDs themselves.

Declarative state computation. Early versions of NVP used manually designed state machines to compute forwarding state. These rapidly became unwieldy as additional features were added, and the basic correctness of the resulting computations was hard to ensure because of their dependency on event orderings. By moving to the declarative language nlog, we not only ensured correctness independent of ordering, but also reduced development and debugging time significantly.

Leveraging the flexibility of open-source software. Innovation in networking has traditionally moved at a glacial pace, with ASIC development times competing with the IETF standardization process which is slower. On the forwarding plane, NVP is built around Open vSwitch (OVS). OVS went from a crazy idea to a widely used component in SDN designs in a few short years, with no haggling over standards, low barriers to deployment (since it is merely a software upgrade), and an increasingly diverse developer community. Moreover, because it is a purely software switch, we could add new functionality as needed without concerns about various artificial limits on packet matchings or actions. In addition, we see a rapid move towards deploying middlebox services (load balancers, deep packet inspecting firewalls, etc.) in tenant networks, and this can also be done through software processing at the edge.

While some have expressed concern about the viability of software packet processing, partial ports of OVS to Netmap [24] and Intel's DPDK [15] show promise of significant performance improvements by directly copying packets into userspace and bypassing the kernel cache. In addition, next-generation NICs will provide acceleration for a number of tunneling formats, and subsequent NICs may include the ability to directly copy packets from the NIC to a VM—bypassing the hypervisor—but still under the direction of the virtual switch in the host hypervisor. Thus, we see a long-term future for accomplishing most network functionality through software packet processing at the edge, and count it as one of the design decisions that allowed NVP to succeed.

8.2 Implications for SDN

SDN is widely heralded as “the future of networking” [28], promising both easier and more advanced management of network control planes. However, our experience with SDN in building NVP suggests a more complicated story.

“Canonical” SDN comprises three basic building blocks: the OpenFlow forwarding model implemented in switches, a network operating system running on a controller (or cluster of controllers) managing those switches via OpenFlow, and one or more network management applications running on top of the network operating system. The key benefit of this canonical version of SDN is greatly improved modularity: SDN can run over any switch that supports OpenFlow, and network management applications need not concern themselves with the communication to/from switches and can merely act on the state gathered by the network operating system. This greatly simplifies the *implementation* of network management, as developers need only focus on the management application.

However, in this canonical picture of SDN, developers must still deal with the complexity of the entire physical network; that is, they are responsible for mapping the various control requirements into the forwarding configuration for every physical (and software) switch. For instance, if one tries to manage an MTD via SDN, there would be less code to write than if one started without SDN, but one would still need to grapple with the complications of 1000s of tenants and their individual control plane requirements, and map these into configurations for every physical switch.

Network virtualization simplifies this by allowing each tenant to specify a logical network and then configure that network. Thus, the tenant's control plane then operates *on a network of their own choosing*, rather than having to cope with the increasingly complex underlying physical infrastructure.

In addition, by focusing all functionality at the edge, NVP only needs to configure the forwarding at the source host hypervisor; the host-to-host forwarding provided by the physical infrastructure is an orthogonal issue with a wide variety of acceptable solutions.

Thus, network virtualization in general, and NVP in particular, decouple network management from the physical infrastructure in two profound ways: first, by decoupling tenant control planes from the physical infrastructure, and then decoupling the implementation of NVP from the physical infrastructure by using software switching at the edge. This dual decoupling from the physical infrastructure may turn out to be of more lasting importance than SDN itself.

9 References

1. M. Caesar, D. Caldwell, N. Feamster, J. Rexford, A. Shaikh, and J. van der Merwe. Design and Implementation of a Routing Control Platform. In Proc. NSDI, April 2005.
2. M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. McKeown, and S. Shenker. Ethane: Taking Control of the Enterprise. In Proc. of SIGCOMM, 2007.
3. M. Casado, T. Koponen, D. Moon, and S. Shenker. Rethinking Packet Forwarding Hardware. In Proc. Of HotNets, October 2008.
4. M. Casado, T. Koponen, R. Ramanathan, and S. Shenker. Virtualizing the Network Forwarding Plane. In Proc. Of PRESTO, 2010.
5. B. Davie and J. Gross. A Stateless Transport Tunneling Protocol for Network Virtualization (STT). Internet draft. draft-davie-stt-03.txt, IETF, March 2013.
6. D. Farinacci, T. Li, S. Hanks, D. Meyer, and P. Traina. Generic Routing Encapsulation (GRE). RFC 2784, IETF, March 2000.
7. N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker. Frenetic: a Network Programming Language. In Proc. of SIGPLAN ICFP, 2011.
8. A. Greenberg, G. Hjalmtysson, D. A. Maltz, A. Myers, J. Rexford, G. Xie, H. Yan, J. Zhan, and H. Zhang. A Clean Slate 4D Approach to Network Control and Management. SIGCOMM CCR, 35(5), 2005.
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. P. Gupta and N. McKeown. Packet Classification on Multiple Fields. In Proc. of SIGCOMM, 1999.
11. C. Hopps. Analysis of an Equal-Cost Multi-Path Algorithm. RFC 2992, IETF, November 2000.
12. P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. Zookeeper: Wait-free coordination for Internet-scale systems. In Proc. of USENIX ATC, June 2010.
13. IEEE. 802.1d - Media Access Control (MAC) Bridges. Standard, IEEE, June 2004.
14. IEEE. 802.1ag - Virtual Bridged Local Area Networks Amendment 5: Connectivity Fault Management. Standard, IEEE, December 2007.
15. Packet Processing on Intel Architecture, 2013. <http://www.intel.com/content/www/us/en/intelligent-systems/intel-technology/packet-processing-is-enhanced-with-software-from-intel-dpdk.html>.
16. D. Karger, E. Lehman, F. Leighton, M. Levine, D. Lewin, and R. Panigrahy. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web. In Proc. of the 29th Symposium on Theory of Computing, May 1997.

17. C. Kim, M. Caesar, A. Gerber, and J. Rexford. Revisiting Route Caching: The World Should Be Flat. In Proc. Of PAM, 2009.
18. 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, 2010.
19. B. T. Loo, T. Condie, J. M. Hellerstein, P. Maniatis, T. Roscoe, and I. Stoica. Implementing Declarative Overlays. In Proc. of SOSP, 2005.
20. B. T. Loo, J. M. Hellerstein, I. Stoica, and R. Ramakrishnan. Declarative Routing: Extensible Routing with Declarative Queries. In Proc. of SIGCOMM, Philadelphia, PA, 2005.
21. N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. OpenFlow: Enabling Innovation in Campus Networks. SIGCOMM CCR, 38(2):69–74, 2008.
22. B. Pfaff and B. Davie. The Open vSwitch Database Management Protocol. Internet draft. draft-pfaff-ovsdb-proto-01.txt, IETF, February 2013.
23. B. Pfaff, J. Pettit, T. Koponen, M. Casado, and S. Shenker. Extending Networking into the Virtualization Layer. In Proc. of HotNets, 2009.
24. L. Rizzo. Netmap: a Novel Framework for Fast Packet I/O. In Proc. of USENIX ATC, 2012.
25. E. Rosen and Y. Rekhter. BGP/MPLS IP Virtual Private Networks. RFC 4364, IETF, February 2006.
26. F. B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. ACM Comput. Surv., 22(4):299–319, December 1990.
27. sFlow Version 5. http://www.sflow.org/sflow_version_5.txt.
28. S. Shenker. The Future of Networking, the Past of Protocols. <http://www.youtube.com/watch?v=YHeyuD89n1Y>.
29. R. Sherwood, G. Gibb, K.-K. Yap, G. Appenzeller, M. Casado, N. McKeown, and G. Parulkar. Can the Production Network Be the Testbed? In Proc. of OSDI, 2010.
30. S. Singh, F. Baboescu, G. Varghese, and J. Wang. Packet Classification Using Multidimensional Cutting. In Proc. of SIGCOMM, 2003.
31. VXLAN: A Framework for Overlaying Virtualized Layer 2 Networks over Layer 3 Networks. <http://tools.ietf.org/html/draft-mahalingam-dutt-dcops-vxlan-00>.
32. Y. Wang, E. Keller, B. Biseborn, J. van der Merwe, and J. Rexford. Virtual Routers on the Move: Live Router Migration as a Network-management Primitive. In Proc. of SIGCOMM, August 2008.



VMware, Inc. 3401 Hillview Avenue Palo Alto CA 94304 USA Tel 877-486-9273 Fax 650-427-5001 www.vmware.com

Copyright © 2013 VMware, Inc. All rights reserved. This product is protected by U.S. and international copyright and intellectual property laws. VMware products are covered by one or more patents listed at <http://www.vmware.com/go/patents>. VMware is a registered trademark or trademark of VMware, Inc. in the United States and/or other jurisdictions. All other marks and names mentioned herein may be trademarks of their respective companies. Item No: VMW-TWP-NETWORK-VIRTUALIZATION-IN-MULTITENANT-DATA-CENTERS-USLET-104