Network Virtualization in Multi-tenant Datacenters

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, and Rajiv Ramanathan, *VMware;* Scott Shenker, *International Computer Science Institute and the University of California, Berkeley;* Alan Shieh, Jeremy Stribling, Pankaj Thakkar, Dan Wendlandt, Alexander Yip, and Ronghua Zhang, *VMware*

https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/koponen

This paper is included in the Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI '14).

April 2–4, 2014 • Seattle, WA, USA

ISBN 978-1-931971-09-6

Open access to the Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI '14) is sponsored by USENIX

Network Virtualization in Multi-tenant Datacenters

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*

*VMware, Inc. †UC Berkeley and ICSI

Operational Systems Track

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

Managing computational resources used to be a time-consuming task requiring the acquisition and configuration of physical machines. However, with server virtualization – that is, exposing the software abstraction of a server to users – provisioning can be done in the time it takes to load bytes from disk. In the past fifteen years server virtualization has become the dominant approach for managing computational infrastructures, with the number of virtual servers exceeding the number of physical servers globally [2, 18].

However, the promise of seamless management through server virtualization is only partially realized in practice. In most practical environments, deploying a new application or development environment requires an associated change in the network. This is for two reasons:

Topology: Different workloads require different network topologies and services. Traditional enterprise workloads using service discovery protocols often require flat L2, large analytics workloads require L3, and web services often require multiple tiers. Further, many applications depend on different L4-L7 services. Today, it is difficult for a single physical topology to support the configuration requirements of all of the workloads of an organization, and as a result, the organization must build multiple physical networks, each addressing a particular common topology.

Address space: Virtualized workloads today operate in the same address space as the physical network.[1] That is, the VMs get an IP from the subnet of the first L3 router to which they are attached. This creates a number of problems:

- Operators cannot move VMs to arbitrary locations.
- Operators cannot allow VMs to run their own IP Address Management (IPAM) schemes. This is a common requirement in datacenters.
- Operators cannot change the addressing type. For example, if the physical network is IPv4, they cannot run IPv6 to the VMs.

Ideally, the networking layer would support similar properties as the compute layer, in which arbitrary network topologies and addressing architectures could be overlayed onto the same physical network. Whether hosting applications, developer environments, or actual tenants, this desire is often referred to as shared multi-tenancy; throughout the rest of this paper we refer to this as a *multi-tenant datacenter* (MTD).

Unfortunately, constructing an MTD is difficult because while computation is virtualized, the network is not. This may seem strange, because networking has long had a number of virtualization primitives such as VLAN (virtualized L2 domain), VRFs (virtualized L3 FIB), NAT (virtualized IP address space), and MPLS (virtualized path). However, these are traditionally configured on a box-by-box basis, with no single unifying abstraction that can be invoked in a more global manner. As a result, making the network changes needed to support server virtualization requires operators to configure many boxes individually, and update these configurations in response to changes or failures in the network. The result is excessive operator overhead and the constant risk of misconfiguration and error, which has led to painstaking change log systems used as best practice in most environments. It is our experience in numerous customer environments that while compute provisioning is generally on the order of minutes, network provisioning can take months. Our experience is commonly echoed in analyst reports [7, 29].

[1] This is true even with VMware VDS and Cisco Nexus 1k.

Academia (as discussed in Section 7) and industry have responded by introducing the notion of *network virtualization*. While we are not aware of a formal definition, the general consensus appears to be that a network virtualization layer allows for the creation of virtual networks, each with independent service models, topologies, and addressing architectures, over the same physical network. Further, the creation, configuration and management of these virtual networks is done through global abstractions rather than pieced together through box-by-box configuration.

And while the idea of network virtualization is not new, little has been written about how these systems are implemented and deployed in practice, and their impact on operations.

In this paper we present NVP, a network virtualization platform that has been deployed in dozens of production environments over the last few years and has hosted tens of thousands of virtual networks and virtual machines. The target environment for NVP is enterprise datacenters, rather than mega-datacenters in which virtualization is often done at a higher level, such as the application.

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 in two ways: the tenant's VMs send packets and the tenant configures the network elements forwarding these packets. In configuring, tenants can access tenant- and element-specific control planes that take switch, routing, and security configurations similar to modern switches and routers, translating them into low-level packet forwarding instructions. A service provider's network consists of a physical forwarding infrastructure and the system that manages and extends this physical 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 tenant control planes, as depicted in Figure 1. Its purpose is to provide the proper abstractions both to tenant's control planes and endpoints; we describe these abstractions below:

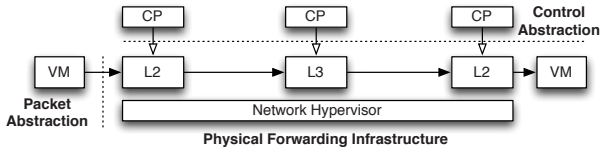

Figure 1: A network hypervisor sits on top of the service provider infrastructure and provides the tenant control planes with a control abstraction and VMs with a packet abstraction.

Control abstraction. This abstraction must allow tenants to define a set of logical network elements (or, as we will call them, logical datapaths) that they can configure (through their control planes) as they would physical network elements. While conceptually each tenant has its own control planes, the network hypervisor provides the control plane *implementations* for the defined logical network elements.[2] 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 they would have in the tenant's home network. This can be accomplished within the packet forwarding pipeline model described above. For instance, the control plane might want to provide basic L2 forwarding semantics in the form of a *logical switch*, which connects some set of tenant VMs (each of which has its own MAC address and is represented by a *logical port* on the switch). To achieve this, the control plane could populate a single logical forwarding table with entries explicitly matching on destination MAC addresses and sending the matching 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, it could broadcast destination addresses 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] In other words, the network hypervisor does not run third-party control plane binaries but the functionality is part of the hypervisor itself. While running a third-party control plane stack would be feasible, we have had no use case for it yet.

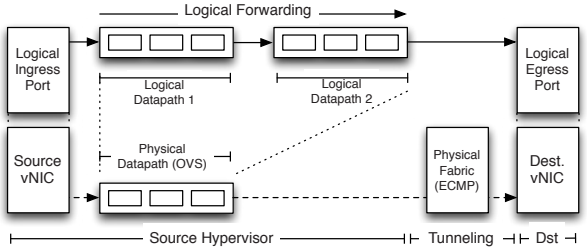

Figure 2: The virtual switch of the originating host hypervisor implements logical forwarding. After the packet has traversed the logical datapaths and their tables, the host tunnels it across the physical network to the receiving host hypervisor for delivery to the destination VM.

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). The logical datapath is almost entirely 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 2). A centralized SDN controller cluster is responsible for configuring virtual switches with the appropriate logical forwarding rules as tenants show up in the network.[3]

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 host tunnels the packet to a service node, which then replicates the packet to all host hypervisors that need to deliver a copy to their local VMs. For deployments not concerned about the broadcast traffic volume, NVP supports configurations without service nodes: the sending host-hypervisor sends a copy of the packet directly to each host hypervisor needing one.

In addition, some tenants want to interconnect their logical network with their existing physical one. This is done via *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 3 depicts the resulting arrangement of host hypervisors, service nodes and gateways, which we collectively refer to as *transport nodes*.

[3]NVP does not control physical switches, and thus does not control how traffic between hypervisors is routed. Instead, it is assumed the physical network provides uniform capacity across the servers, building on ECMP-based load-balancing.

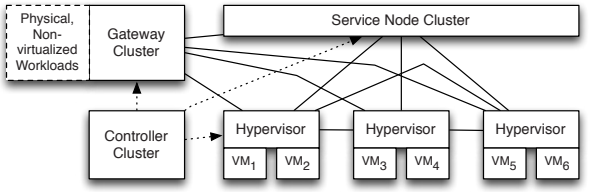

Figure 3: In NVP, controllers manage the forwarding state at all transport nodes (hypervisors, gateways, service nodes). Transport nodes are fully meshed over IP tunnels (solid lines). Gateways connect the logical networks with workloads on non-virtualized servers, and service nodes provide replication 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 computation. In Section 5 we discuss the 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 Sections 8 and 9.

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, gateways 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) [32] in all transport nodes (host hypervisors, service nodes, and gateway nodes) to forward packets. OVS is remotely configurable by the

NVP controller cluster via two protocols: one that can inspect and modify a set of flow tables (analogous to flow tables in physical switches),[4] and one that allows the controller to create and manage overlay tunnels and to discover which VMs are hosted at a hypervisor [31].

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 it, sending it to a given egress port on the logical datapath, and modifying 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. We emphasize that this model of a logical table pipeline (as opposed to a single table) is the key to allowing tenants to use existing forwarding policies with little or no change: with a table pipeline available to the control plane, tenants can be exposed to features and configuration models similar to ASIC-based switches and routers, and therefore the tenants can continue to use a familiar pipeline-based mental model.

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 (see Section 5); for overlay traffic, the tunnel header of the incoming packet contains this information. Then, the virtual switch connects each packet to its logical pipeline by pre-computed flows that NVP writes into the OVS flow table, which match a packet based on its ingress port and add 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.

The control plane abstraction 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 logical 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, NVP uses a special action 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 controlled 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.[5]

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 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

[4] We use OpenFlow [27] for this protocol, though any flow management protocol with sufficient flexibility would work.

[5] 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.

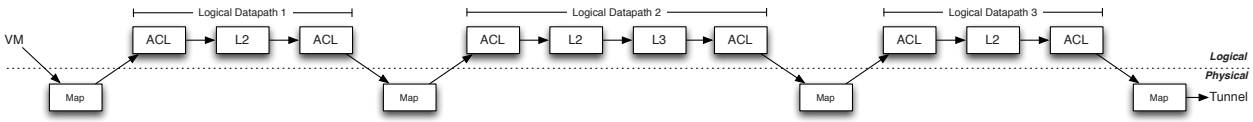

Figure 4: Processing steps of a packet traversing through two logical switches interconnected by a logical router (in the middle). Physical flows prepare for the logical traversal by loading metadata registers: first, the tunnel header or source VM identity is *mapped* to the first logical datapath. After each logical datapath, the logical forwarding decision is mapped to the next logical hop. The last logical decision is mapped to tunnel headers.

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. Figure 4 depicts how a packet originating at a source VM first traverses through a logical switch (with ACLs) to a logical router before being forwarded by a logical switch attached to the destination VM (on the other side of the tunnel). This is a simplified example: we omit the steps required for failover, multicast/broadcast, ARP, and QoS, for instance.

As an optimization, we constrain the logical topology such that logical L2 destinations can only be present at its edge.[6] 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; the receiving hypervisor is determined by the logical IP destination address, leaving the last logical L2 hop to be executed at the receiving hypervisor. Thus, in Figure 4, if the sending hypervisor does not host any VMs attached to the third logical datapath, then the third logical datapath runs at the receiving hypervisor and there is a tunnel between the second and third logical datapaths instead.

3.2 Forwarding Performance

OVS, as a virtual switch, must classify each incoming packet against its entire flow table in software. However, flow entries written by NVP can contain wildcards for any irrelevant parts of a packet header. Traditional physical switches generally classify packets against wildcard flows using TCAMs, which are not available on the standard x86 hardware where OVS runs, and so OVS must use a different technique to classify packets quickly.[7]

To achieve efficient flow lookups on x86, OVS exploits *traffic locality*: the fact that all of the 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 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 [5, 22].

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 OS. 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 OS. 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 encapsulation method called STT [8].[8] STT places a standard, but fake, TCP header after the physical IP header. After this, there is the actual encapsulation 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,

[6] We have found little value in supporting logical routers interconnected through logical switches without tenant VMs.

[7] There is much previous work on the problem of packet classification without TCAMs. See for instance [15, 37].

[8] NVP also supports other tunnel types, such as GRE [9] and VXLAN [26] for reasons discussed shortly.

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 packet looks like standard TCP packet, the STT protocol is stateless and requires no TCP handshake procedure between the tunnel endpoints. VMs can 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 the segments produced out of the original packet and additional information (*e.g.,* for debugging) 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 may confuse middleboxes on the path. STT uses its own TCP transport port in the fake TCP header, however, and to date administrators have been successful in punching any necessary holes in middleboxes in the physical network. For environments where compliance is more important than efficiency, NVP supports other, more standard IP encapsulation protocols.

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 appliances that NVP introduces – service and gateway nodes – must cause only minimal, if any, dataplane outage.

For this reason, NVP deployments have multiple service nodes, to ensure that any one service node failure does not disrupt logical broadcast and multicast traffic. The controller cluster instructs hypervisors to load-balance their packet replication traffic across a bundle of service node tunnels by using flow hashing algorithms similar to ECMP [16]. The hypervisor monitors these tunnels using BFD [21]. If the hypervisor fails to receive heartbeats from a 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.

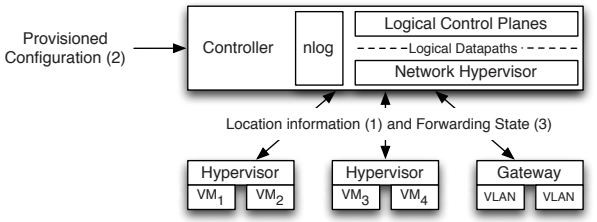

Figure 5: Inputs and outputs to the forwarding state computation process which uses nlog, as discussed in §4.3.

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.[9] 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: each gateway broadcasts CFM packets [19] 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.[10] Broadcasts from an unexpected gateway cause all gateways to disable their tunnels to prevent possible loops.

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 5. First, hypervisors and gateways

[9] Gateway and service nodes do not monitor hypervisors, and thus, they have little per tunnel state to maintain.

[10] L3 gateways can use ECMP for active-active scale-out instead.

provide the controller with location information for vNICs over the OVS configuration protocol [31] (1), updating this information as virtual machines migrate. Hypervisors also provide the MAC address for each vNIC.[11] 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 when the physical configuration of the overall system (*e.g.,* the set of managed transport nodes) changes.

Based on these inputs, the logical control plane computes the logical lookup tables, which the network hypervisor augments and transforms into physical forwarding state (realized as logical datapaths with given logical lookup entries, as discussed in the previous section). The forwarding state is then pushed to transport nodes via OpenFlow and the OVS configuration protocol (3). OpenFlow flow entries model the full logical packet forwarding pipeline, whereas OVS configuration database entries are responsible for the tunnels connecting hypervisors, gateways and service nodes, as well as any local queues and scheduling policies.[12]

The above implies the computational model is entirely proactive: the controllers push *all* the necessary forwarding state down and do not process any packets. The rationale behind this design is twofold. First, it simplifies the scaling of the controller cluster because infrequently pushing updates to forwarding instructions to the switch, instead of continuously punting packets to controllers, is a more effective use of resources. Second, and more importantly, failure isolation is critical in that the managed transport nodes and their data planes must remain operational even if connectivity to the controller cluster is transiently lost.

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. A single input type corresponds to a single configured logical feature or physical property; for instance, a particular type of logical ACL may be a single logical input type, whereas the location of a vNIC may be a single physical input information type. Similarly, each output type corresponds to a single type of attribute being configured over OpenFlow or the OVS configuration protocol; for example, a tunnel parameter and particular type of ACL flow entries are both examples of individual output types.

The total amount of input state is also large, being proportional to the size of the MTD, and the state changes frequently as VMs migrate and tenants join, leave, and reconfigure their logical networks. The controller needs to react quickly to the 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 compute and update the forwarding state incrementally 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 especially test.

4.3 Incremental State Computation with nlog

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. Note that nlog is not used by NVP's users, only internally by its developers; users interact with NVP via the API (see §5.3).

nlog declarations 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 (input types) or *internal tables* holding only results computed by declarations. Head tables may be internal tables or *output tables* (output types), which cause changes external to the nlog runtime engine when tuples are added to or removed from the table. nlog does not currently support recursive declarations or negation.[13] In total, NVP has about 1200 declarations and 900 tables (of all three types).

[11] The service provider's cloud management system can provision this information directly, if available.

[12] One can argue for a single flow protocol to program the entire switch but in our experience trying to fold everything into a single flow protocol only complicates the design.

[13] The lack of negation has had little impact on development but the inability to recurse complicates computations where the number of iterations is unknown at compile time. For example, traversing a graph can only be done up to maximum diameter.

```
# 1. Determine tunnel from a source hypervisor
#    to a remote, destination logical port.
tunnel(dst_lport_id, src_hv_id, encap, dst_ip) :-
   # Pick logical ports & chosen encap of a datapath.
   log_port(src_lport_id, log_datapath_id),
   log_port(dst_lport_id, log_datapath_id),
   log_datapath_encap(log_datapath_id, encap),

   # Determine current port locations (hypervisors).
   log_port_presence(src_lport_id, src_hv_id),
   log_port_presence(dst_lport_id, dst_hv_id),

   # Map dst hypervisor to IP and omit local tunnels.
   hypervisor_locator(dst_hv_id, dst_ip),
   not_equal(src_hv_id, dst_hv_id);

# 2. Establish tunnel via OVS db. Assigned port # will
#    be in input table ovsdb_tport. Ignore first column.
ovsdb_tunnel(src_hv_id, encap, dst_ip) :-
   tunnel(_, src_hv_id, encap, dst_ip);

# 3. Construct the flow entry feeding traffic to tunnel.
#    Before resubmitting packet to this stage, reg1 is
#    loaded with 'stage id' corresponding to log port.
ovs_flow(src_hv_id, of_expr, of_actions) :-
   tunnel(dst_lport_id, src_hv_id, encap, dst_ip),
   lport_stage_id(dst_lport_id, processing_stage_id),
   flow_expr_match_reg1(processing_stage_id, of_expr),
   # OF output action needs the assigned tunnel port #.
   ovsdb_tport(src_hv_id, encap, dst_ip, port_no),
   flow_output_action(port_no, of_actions);
```

Figure 6: Steps to establish a tunnel: 1) determining the tunnels, 2) creating OVS db entries, and 3) creating OF flows to output packets into tunnels.

The code snippet in Figure 6 has simplified nlog declarations for creating OVS configuration database tunnel entries as well as OpenFlow flow entries feeding packets to tunnels. The tunnels depend on API-provided information, such as the logical datapath configuration and the tunnel encapsulation type, as well as the location of vNICs. The computed flow entries are a part of the overall packet processing pipeline, and thus, they use a controller-assigned stage identifier to match with the packets sent to this stage by the previous processing stage.

The above declaration updates the head table *tunnel* for all pairs of logical ports in the logical datapath identified by *log_datapath_id*. The head table is an internal table consisting of rows each with four data columns; a single row corresponds to a tunnel to a logical port *dst_lport_id* on a remote hypervisor *dst_hv_id* (reachable at *dst_ip*) on a hypervisor identified by *src_hv_id* for a specific encapsulation type (*encap*) configured for the logical datapath. We use a function *not_equal* to exclude tunnels between logical ports on a single hypervisor. We will return to functions shortly.

In the next two declarations, the internal *tunnel* table is used to derive both the OVS database entries and OpenFlow flows to output tables *ovsdb_tunnel* and *ovs_flow*. The declaration computing the flows uses functions *flow_expr_reg1* and *flow_output_action* to compute the corresponding OpenFlow expression (matching over register 1) and actions (sending to a port assigned for the tunnel). As VMs migrate, the *log_port_presence* input table is updated to reflect the new locations of each *log_port_id*, which in turn causes corresponding changes to *tunnel*. This will result in re-evaluation of the second and third declaration, which will result in OVS configuration database changes that create or remove tunnels on the corresponding hypervisors, as well as OpenFlow entries being inserted or removed. Similarly, as tunnel or logical datapath configuration changes, the declarations will be incrementally re-evaluated.

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 independently to network failures (see §3.3).

Language extensions. Datalog joins can only rearrange existing column data. Because most non-trivial programs must also transform column data, nlog provides extension mechanisms for specifying transformations in C++.

First, a developer can implement a *function table*, which is 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 create OpenFlow match expressions or actions (like in the example above). The base language provides various functions for primitive column types (*e.g.,* integers, 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 logical datapath flow entries. Finally, the developer is provided with *not_equal* to express inequality between two columns.

Second, if developers require more complicated transformations, they can hook an output and an input table together through arbitrary C++ code. Declarations produce tuples into the output table, which transforms them into C++ and feeds them to the output table C++ implementation. After processing, the C++ code transforms them back into tuples and passes them to nlog through the input table. For instance, we use this technique to implement hysteresis that dampens external events such as a network port status flapping.

5 Controller Cluster

In this section we discuss the design of the controller cluster: the distribution of physical forwarding state computation to implement the logical datapaths, the auxiliary distributed services that 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-

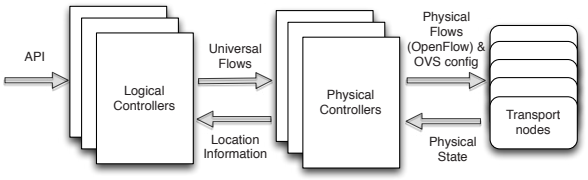

Figure 7: NVP controllers arrange themselves into two layers.

coupled two-layer hierarchy, with each layer consisting of a cluster of processes running on multiple controllers. We implement all of this computation in nlog, as discussed in the previous section.

Figure 7 illustrates NVP's two-layer distributed controller cluster. The top layer consists of *logical controllers*. NVP assigns the computation for each logical datapath to a particular live controller using its identifier as a sharding key, parallelizing the computation workload.

Logical controllers compute the flows and tunnels needed to implement logical datapaths, as discussed in Section 3. They encode all computed flow entries, including the logical datapath lookup tables provided by the logical control planes and instructions to create tunnels and queues for the logical datapath, as *universal flows*, an intermediate representation similar to OpenFlow but which abstracts out all transport-node-specific details such as ingress, egress or tunnel port numbers, replacing them with abstract identifiers. The universal flows are published over RPC to the bottom layer consisting of *physical controllers*.

Physical controllers are responsible for communicating with hypervisors, gateways and service nodes. They translate the location-independent portions of universal flows using node- and location-specific state, such as IP addresses and physical interface port numbers (which they learn from attached transport nodes), as well as create the necessary configuration protocol instructions to establish tunnels and queue configuration. The controllers then push the resulting *physical flows* (which are now valid OpenFlow instructions) and configuration protocol updates down to the transport nodes. Because the universal-to-physical translation can be executed independently for every transport node, NVP shards this responsibility for the managed transport nodes among the physical controllers.

This arrangement reduces the computational complexity of the forwarding state computation. By avoiding the location-specific details, the logical controller layer can compute one "image" for a single ideal transport node participating in a given logical datapath (having $O(N)$ tunnels to remote transport nodes), without considering the tunnel mesh between all transport nodes in its full $O(N^2)$ complexity. Each physical controller can then translate that image into something specific for each of the transport nodes under its responsibility.

Availability. To provide failover within the cluster, NVP provisions hot standbys at both the logical and physical controller layers by exploiting the sharding mechanism. One controller, acting as a sharding coordinator, ensures that every shard is assigned one master controller and one or more other controllers acting as hot standbys. On detecting the failure of the master of a shard, the sharding coordinator promotes the standby for the shard to master, 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. The coordinator itself is a highly-available service that can run on any controller and will migrate as needed when the current coordinator fails.

Because of their large population, transport nodes do not participate in the cluster coordination. Instead, OVS instances are configured by the physical controllers to connect to both the master and the standby physical controllers for their shard, though their master controller will be the only one sending them updates. Upon master failure, the newly-assigned master will begin sending updates via the already-established connection.

5.2 Distributed Services

NVP is built on the Onix controller platform [23] and thus has access to the elementary distributed services Onix provides. To this end, NVP uses the Onix replicated transactional database to persist the configuration state provided through API, but it also implements two additional distributed services.

Leader election. Each controller must know which shard it manages, and must also know when to take over responsibility of slices managed by a controller that has disconnected. Consistent hashing [20] is one possible approach, but it tends to be most useful in very large clusters; with only tens of controllers, NVP simply elects a sharding coordinator using Zookeeper [17]. 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.

Label allocation. A network packet encapsulated in a tunnel must carry a label that denotes the logical egress port to which the packet is destined, so the receiving hypervisor 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 datapaths. Because encapsulation rules for different logical datapaths 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 compact to minimize packet overhead. We use Zookeeper to implement a label allocator that ensures labels will not

be reused until NVP deletes the corresponding datapath. The logical controllers use this label allocation service to assign logical egress port labels at the time of logical datapath creation, and then disseminate the labels to the physical controllers via universal flows.

5.3 API for Service Providers

To support integrating with a service provider's existing cloud management system, NVP exposes an HTTP-based REST 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. Logical controllers react to changes to these logical elements, enabling or disabling features on the corresponding logical control plane accordingly. The cloud management system uses 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/or their customers.

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

6 Evaluation

In this section, we present measurements both for the controller cluster and the edge datapath implementation.

6.1 Controller Cluster

Setup. The configuration in the following tests has 3,000 simulated hypervisors, each with 21 vNICs for a total of 63,000 logical ports. In total, there are 7000 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 configures the logical control planes to use port ACLs on 49,188 of the logical ports and generic ACLs for 1,553 of the logical switches.[14]

The test control cluster has three nodes. Each controller is a bare-metal Intel Xeon 2.4GHz server with 12 cores, 96GB of memory, and 400GB hard disk. The logical and physical computation load is distributed evenly among the controllers, with one master and one standby per shard. The physical network is a dedicated switched 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 XenServer 5.6 physical hypervisors, and are connected via Xen bridges to the physical network. We test four types of conditions the cluster may face.

[14]This serves as our base validation test; other tests stress the system further both in scale and in complexity of configurations.

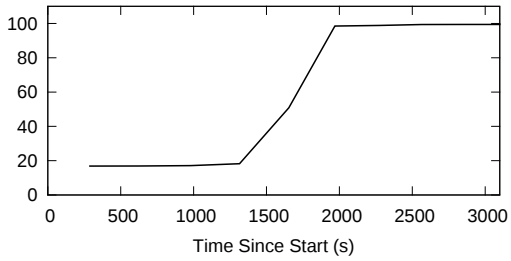

Figure 8: Cold start connectivity as a percentage of all pairs connected.

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 failover 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 perform 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.

Finally, we note that we do not consider routing convergence of any kind in the tests. Physical routing protocols handle any failures in the connectivity between the nodes, and thus, aside from tunnel failovers, the network hypervisor can remain unaware of such events.

Results. Figure 8 shows the percentage of correct pings over time for the cold start test, beginning at time 0. It

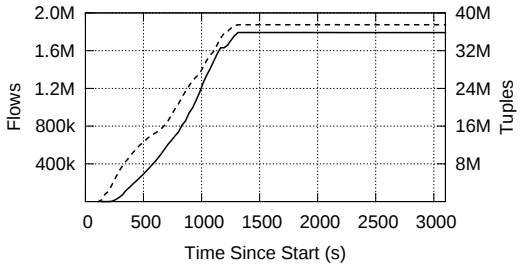

Figure 9: Total physical flows (solid line) and nlog tuples (dashed line) in one controller after a cold start.

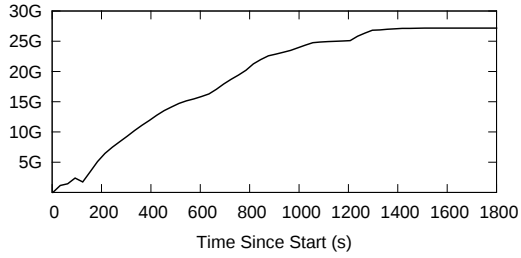

Figure 10: Memory used by a controller after a cold start.

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 flow changes after each configuration change. Thus, cold start 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 fully 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. While the cold-start time is long, it is relevant only in catastrophic outage conditions and thus considered reasonable: after all, if hypervisors remain powered on, the data plane will also remain functional even though the controllers have to go through cold-start (as in the restore test below).

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; this is due to an implementation artifact stemming from arbitrary evaluation order in nlog. Second, for a single ping to start working, the correct flows need to be set up on all the transport nodes on the path of the ping (and ARP request/response, if any).

We do not include a graph for connectivity correctness during the restore or failover cases, but merely note that connectivity correctness remains at 100% during these tests. The connectivity is equally well-maintained in the case of adding or removing controllers to the cluster, but again we do not include a graph here for brevity.

Figure 9 shows the total number of tuples, as well as the total number of 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 of the possible factors that can affect a flow. After converging, the measured controller uses approximately 27G of memory, as shown in Figure 10.

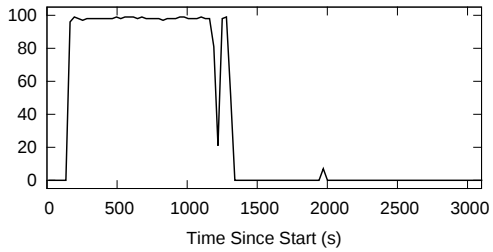

Figure 11: nlog load during cold start.

Since our test cluster has 3 controllers, 1.8M flows is 2/3 of all the flows in the system, because this one controller is the master for 1/3 of the flows and standby for 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 11 shows nlog load during the cold start test. nlog is almost 100% busy for 20 minutes. This shows that controller can read its database and connect to the switches (thereby populating 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 sends the computed state to each hypervisor.

A similar load graph for the steady state test is not included but we merely report the numeric results, highlighting nlog's ability to process incremental changes to inputs: the addition of 10 logical ports (to the 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

Tunnel performance. Table 1 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 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 the results only take into account 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.

We see that the throughput of GRE is much lower

	No encap	STT	GRE
TX CPU load	49%	49%	85%
RX CPU load	72%	119%	183%
Throughput	9.3Gbps	9.3Gbps	2.4Gbps

Table 1: Non-tunneled, STT, and GRE performance.

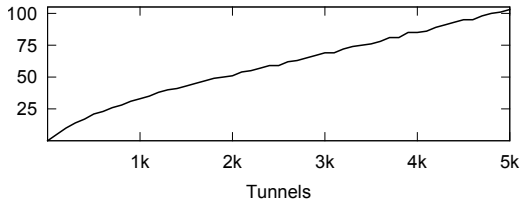

Figure 12: Tunnel management CPU load as a % of a single core.

and requires more CPU than either of the other methods due to its 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 constituting a single STT frame before passing the result of coalescing down to OS. After all, for NIC the TCP payload is a byte stream and not a single jumbo frame spanning multiple datagrams on the wire; therefore, if there is enough time between two wire datagrams, the NIC may decide to pass the current result of the coalescing to the OS, just to avoid introducing excessive extra latency. STT requires the full set of segments before it can remove the encapsulation header within the TCP payload and deliver the original logical packet, and so on these occasions it must perform the remaining coalescing in software.

Connection set up. OVS connection setup performance has been explored in the literature (see *e.g.,* [32–34]) and we have no new results to report here, though we return to the topic shortly in Section 8.

Tunnel scale. Figure 12 shows the keepalive message processing cost as the number of tunnels increases. This test is 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 500ms, and the results indicate a single CPU core can process and respond to them in a timely manner for up to 5000 tunnels.

7 Related Work

NVP borrows from recent advances in datacenter network design (*e.g.,* [1, 12, 30]), 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 topics where we feel it useful to distinguish our work from previous efforts. While NVP relies on SDN [3, 4, 13, 14, 23, 27] in the form of an OpenFlow forwarding model and a control plane managed by a controller, NVP requires significant extensions.

Virtualization of the network forwarding plane was first described in [6]; 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 [35], and FlowVisor introduced the concept of slicing to OpenFlow and SDN [36]. 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 [38] 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 [24, 25] is the most closely related work, but it focuses on concise, intuitive modeling of distributed algorithms. Since the early versions of NVP, our focus has been on structuring the computation within a single node to allow efficient incremental computation. Frenetic [10, 11] and Pyretic [28] have argued for reactive functional programming to simplify the implementation of packet forwarding decisions, but they focused on reactive packet processing rather than the proactive computations considered here. Similarly to NVP (and [6] before it), Pyretic [28] identifies the value of an abstract topology and uses it to support composing modular control logic.

8 Discussion

After having presented the basic design and its performance, we now return to discuss which aspects of the design were most critical to NVP's success.

8.1 Seeds of NVP's Success

Basing NVP on a familiar abstraction. While one could debate which 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) was to make logical networks look *exactly* like current network configurations. 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 correctness of the resulting computations was hard to ensure because of their dependency on event orderings. By moving to nlog, we not only ensured correctness independent of ordering, but also reduced development time significantly.

Leveraging the flexibility of software switching. Innovation in networking has traditionally moved at a glacial pace, with ASIC development times competing with the IETF standardization process for 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 a diverse developer community. Moreover, because it is a software switch, we could add new functionality without concerns about artificial limits on packet matches or actions.

8.2 Lessons Learned

Growth. With network virtualization, spinning up a new environment for a workload takes a matter of minutes instead of weeks or months. While deployments often start cautiously with only a few hundred hypervisors, once the tenants have digested the new operational model and its capabilities their deployments typically witness rapid growth resulting in a few thousand hypervisors.

The story is similar for logical networks. Initial workloads require only a single logical switch connecting a few tens of VMs, but as the deployments mature, tenants migrate more complicated workloads. At that point, logical networks with hundreds of VMs attached to a small number of logical switches interconnected by one or two logical routers, with ACLs, become more typical. The overall trends are clear: in our customers' deployments, both the number of hypervisors as well as the complexity and size of logical networks tend to grow steadily.

Scalability. In hindsight, the use of OpenFlow has been a major source of complications, and here we mention two issues in particular. First, the overhead OpenFlow introduces within the physical controller layer became the limiting factor in scaling the system; unlike the logical controller which has computational complexity of $O(N)$, the need to tailor flows for each hypervisor (as required by OpenFlow) requires $O(N^2)$ operations. Second, as the deployments grow and clusters operate closer to their memory limits, handling transient conditions such as controller failovers requires careful coordination.

Earlier in the product lifecycle, customers were not willing to offload much computation into the hypervisors. While still a concern, the available CPU and memory resources have grown enough over the years that in the coming versions of the product, we can finally run the physical controllers within the hypervisors without concern. This has little impact to the overall system design but moving the physical controllers down to the hypervisors reduces the cluster requirements by an order of magnitude. Interestingly, this also makes OpenFlow a local protocol within the hypervisor, which limits its impact on the rest of the system.

Failure isolation. While the controller cluster provides high-availability, the non-transactional nature of OpenFlow results in situations where switches operate over inconsistent and possibly incomplete forwarding state due to a controller crash or connectivity failure between the cluster and hypervisor. While a transient condition, customers expect better consistency between the switches and controllers. To this end, the next versions of NVP make all declarative computation and communication channels "transactional": given a set of changes in the configuration, all related incremental updates are computed and pushed to the hypervisors as a batch which is then applied atomically at the switch.

Forwarding performance. Exact match flow caching works well for typical workloads where the bulk of the traffic is due to long-lived connections; however, there are workloads where short-lived connections dominate. In these environments, exact match caching turned out to be insufficient: even if the packet forwarding rates were sufficiently high, the extra CPU load introduced was deemed unacceptable by our customers.

As a remedy, OVS replaced the exact match flow cache with *megaflows*. In short, unlike exact match flow cache, megaflows caches *wildcarded* forwarding decisions matching over larger traffic aggregates than a single transport connection. The next step is to re-introduce the exact match flow cache and as a result there will be three layers of packet processing: exact match cache handling packets after the first packets of transport connections (one hash lookup), megaflows that handle most of the first packets of transport connections (a single flow classification) and a slow path finally handling the rest (a sequence of flow classifications).

9 Conclusion

Network virtualization has seen a lot of discussion and popularity in academia and industry, although little has been written about practical network virtualization systems, or how they are implemented and deployed. In this paper, we described the design and implementation of NVP, a network virtualization platform, that has been deployed in production environments for last few years.

Acknowledgments. We would like to thank our shepherd, Ratul Mahajan, and the reviewers for their valuable comments.

10 References

[1] M. Al-Fares, A. Loukissas, and A. Vahdat. A Scalable, Commodity Data Center Network Architecture. In *Proc. of SIGCOMM*, August 2008.

[2] T. J. Bittman, G. J. Weiss, M. A. Margevicius, and P. Dawson. Magic Quadrant for x86 Server Virtualization Infrastructure. Gartner, June 2013.

[3] 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.

[4] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. McKeown, and S. Shenker. Ethane: Taking Control of the Enterprise. In *Proc. of SIGCOMM*, August 2007.

[5] M. Casado, T. Koponen, D. Moon, and S. Shenker. Rethinking Packet Forwarding Hardware. In *Proc. of HotNets*, October 2008.

[6] M. Casado, T. Koponen, R. Ramanathan, and S. Shenker. Virtualizing the Network Forwarding Plane. In *Proc. of PRESTO*, November 2010.

[7] D. W. Cearley, D. Scott, J. Skorupa, and T. J. Bittman. Top 10 Technology Trends, 2013: Cloud Computing and Hybrid IT Drive Future IT Models. Gartner, February 2013.

[8] B. Davie and J. Gross. A Stateless Transport Tunneling Protocol for Network Virtualization (STT). Internet draft. draft-davie-stt-04.txt, IETF, September 2013.

[9] D. Farinacci, T. Li, S. Hanks, D. Meyer, and P. Traina. Generic Routing Encapsulation (GRE). RFC 2784, IETF, March 2000.

[10] 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*, September 2011.

[11] N. Foster, R. Harrison, M. L. Meola, M. J. Freedman, J. Rexford, and D. Walker. Frenetic: A High-Level Language for OpenFlow Networks. In *Proc. of PRESTO*, November 2010.

[12] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. VL2: A Scalable and Flexible Data Center Network. In *Proc. of SIGCOMM*, August 2009.

[13] 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.

[14] 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.

[15] P. Gupta and N. McKeown. Packet Classification on Multiple Fields. In *Proc. of SIGCOMM*, August 1999.

[16] C. Hopps. Analysis of an Equal-Cost Multi-Path Algorithm. RFC 2992, IETF, November 2000.

[17] 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.

[18] Server Virtualization Multiclient Study. IDC, January 2012.

[19] IEEE. 802.1ag - Virtual Bridged Local Area Networks Amendment 5: Connectivity Fault Management. Standard, IEEE, December 2007.

[20] 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 STOC*, May 1997.

[21] D. Katz and D. Ward. Bidirectional Forwarding Detection (BFD). RFC 5880, IETF, June 2010.

[22] C. Kim, M. Caesar, A. Gerber, and J. Rexford. Revisiting Route Caching: The World Should Be Flat. In *Proc. of PAM*, April 2009.

[23] 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*, October 2010.

[24] B. T. Loo, T. Condie, J. M. Hellerstein, P. Maniatis, T. Roscoe, and I. Stoica. Implementing Declarative Overlays. In *Proc. of SOSP*, October 2005.

[25] B. T. Loo, J. M. Hellerstein, I. Stoica, and R. Ramakrishnan. Declarative Routing: Extensible Routing with Declarative Queries. In *Proc. of SIGCOMM*, August 2005.

[26] M. Mahalingam et al. VXLAN: A Framework for Overlaying Virtualized Layer 2 Networks over Layer 3 Networks. Internet draft. draft-mahalingam-dutt-dcops-vxlan-08.txt, IETF, February 2014.

[27] 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.

[28] C. Monsanto, J. Reich, N. Foster, J. Rexford, and D. Walker. Composing Software Defined Networks. In *Proc. of NSDI*, April 2013.

[29] B. Munch. IT Market Clock for Enterprise Networking Infrastructure, 2013. Gartner, September 2013.

[30] R. Niranjan Mysore, A. Pamboris, N. Farrington, N. Huang, P. Miri, S. Radhakrishnan, V. Subramanya, and A. Vahdat. PortLand: A Scalable Fault-tolerant Layer 2 Data Center Network Fabric. In *Proc. of SIGCOMM*, August 2009.

[31] B. Pfaff and B. Davie. The Open vSwitch Database Management Protocol. RFC 7047, IETF, December 2013.

[32] B. Pfaff, J. Pettit, T. Koponen, K. Amidon, M. Casado, and S. Shenker. Extending Networking into the Virtualization Layer. In *Proc. of HotNets*, October 2009.

[33] L. Rizzo. Netmap: a Novel Framework for Fast Packet I/O. In *Proc. of USENIX ATC*, June 2012.

[34] L. Rizzo, M. Carbone, and G. Catalli. Transparent Acceleration of Software Packet Forwarding Using Netmap. In *Proc. of INFOCOM*, March 2012.

[35] E. Rosen and Y. Rekhter. BGP/MPLS IP Virtual Private Networks. RFC 4364, IETF, February 2006.

[36] 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*, October 2010.

[37] S. Singh, F. Baboescu, G. Varghese, and J. Wang. Packet Classification Using Multidimensional Cutting. In *Proc. of SIGCOMM*, August 2003.

[38] Y. Wang, E. Keller, B. Biskeborn, 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.