SPN OS: Managing Network Services with Virtual Network Objects

Xi Wang, Cong Chen, Paparao Palacharla, Motoyoshi Sekiya
Fujitsu Labs of America
Richardson, Texas

Steffen Smolka, Nate Foster
Cornell University
Ithaca, New York

Abstract—This paper presents the design and implementation of a new controller platform, called the software-programmed networking (SPN) operating system (OS). SPN OS separates distributed control of virtual network services from the centralized management of network-wide resources. The key abstraction provided in SPN OS is a virtual network object (VNO), which encapsulates multiple layers of network stack as well as their associated resource constraints and service behaviors. VNOs are created by a VNO Arbiter, which handles provisioning of global network resources, but are managed independently by tenants through an intuitive object-oriented interface. This separation provides flexibility and scalable performance. We describe the design of SPN OS and VNOs, present a prototype implementation on top of the high-level language NetKAT, and discuss use cases.

Keywords—SDN; virtual network; OpenFlow; NetKAT

I. INTRODUCTION

The recent emergence of cloud-based services, software defined networking, and network function virtualization (SDN/NFV) is reshaping the entire networking ecosystem. An ever-increasing number of applications are moving to the cloud to be offered as networked services. These cloud-hosted applications reside as ‘virtual tenants’ in multitenant datacenters, each running on a ‘slice’ of the shared physical infrastructure. Virtualization plays a key role in these infrastructures, enabling flexible and efficient sharing of computing and network resources among tenants. However, although technologies such as Virtual Machines (VMs) and containers are now considered the de facto standard solutions for virtualizing computing resources, the right approach to network virtualization largely remains an open issue.

Software-Defined Networking (SDN) [1] is a powerful technology for implementing various forms of network virtualization, since it allows each network entity to be “software-defined” rather than manually configured. For example, using OpenFlow [2], a traffic flow entity can be “defined” using an arbitrary combination of layer identifiers (header space). Then, by installing packet-forwarding rules associated with the flow to the physical switches, an SDN controller can ensure that this particular traffic entity “flows” across the network accordingly. In a similar fashion, SDN controllers such as FlowVisor [3], OpenVirtEX [4], ONIX [5], Pyretic [6], VMware NVP [7], ONOS [8], and others can “define” a virtual network entity in terms of a virtual topology that is mapped down to the physical switches through a network hypervisor. Each virtual network entity is managed by a separate tenant controller, allowing the tenant to control a portion of network traffic and a subset of the physical network via the network hypervisor.

However, existing virtualization solutions do not provide sufficiently expressive abstractions for building virtual network services. For example, on many platforms, a virtual service cannot predefine its behavior under failure scenarios. Instead, the SDN controller must detect and respond to failures dynamically, leading to service interruptions at the virtual layer and unpredictable performance during restoration. Put another way, existing solutions define the properties of a virtual network service (e.g., its topology, virtual-to-physical mapping relationship, etc.), but its behavior (e.g., its packet-processing rules) must be controlled externally by a separate controller, not by the service itself. As an analogy, this is similar to creating a VM by only allocating computing resources and not allowing it to launch computing tasks on its own—i.e., the tasks for each VM must always be launched externally from the hypervisor.

Such an approach may limit the flexibility and functionality of individual virtual network entities and also raise scalability concerns for the centralized controller and associated hypervisor. These problems are likely to be further exacerbated when introducing control for circuit-switched networks (e.g., provisioning/scheduling, protection/restoration) and optical layer considerations (e.g., reachability, quality-of-transmission monitoring), etc. into the virtual network entities (e.g., virtual optical network provisioning [9]). A more efficient and scalable solution is needed to support numerous concurrent virtual network entities with a wide variety of abstractions, while allowing each virtual network to execute highly-flexible and efficient control logic on its own network slice.

This paper presents a new controller platform, called the software-programmed networking (SPN) operating system (OS), that is designed to enable flexible management of virtual network services. The key abstraction offered by the controller is a virtual network object (VNO) that encapsulates all network properties and service behavior associated with a virtual network service throughout its service lifecycle. To ensure that the needs of all customers are met, VNOs are provisioned by a logically-centralized Arbiter that manages network resources across multiple layers including optical circuits. The Arbiter can use advanced algorithms that exploit global knowledge to allocate and optimize network resources. However, to provide
flexibility to customers and ensure scalable performance. VNOs can be managed independently by tenant applications through a natural object-oriented interface.

VNOs are similar in spirit to the virtual topologies supported in many existing SDN controllers—indeed, VNOs are based on a natural graph abstraction that can be manipulated using standard tools. However, unlike virtual topologies, VNOs are designed to provide fine-grained control over features of the underlying physical network such as bandwidth and backup paths. For example, the Arbiter might provide several ways to map the logical topology to circuits at the physical layer, and a VNO executing a traffic engineering application might control the mapping used to achieve good performance. To the best of our knowledge, this feature of allowing each VNO to execute its own service control logic is not provided in any current SDN controller.

To evaluate our design for a new controller platform based on VNOs, we have built a prototype implementation on top of a high-level network programming language NetKAT [10] and OpenFlowJ [11]. In our implementation, a VNO specifies its logical service attributes (e.g., the topology and forwarding behavior) in NetKAT and uses the compiler to translate logical programs into forwarding rules that can be installed on switches using OpenFlowJ. We illustrate the use of SPN OS on a network example with virtual “big switch” and ring topologies, and present use cases that identify applications of VNOs to common virtual network management tasks.

Overall, the main contributions of this paper are as follows: (i) we motivate the need for new SDN control abstractions (Section II); (ii) we present the design of SPN OS, VNOs, and the Arbiter (Section III); (iii) we describe a prototype implementation of SPN OS based on NetKAT and OpenFlowJ and demonstrate running examples (Sections IV and V), and we discuss future use cases enabled by SPN OS (Section VI).

II. SPN OS OVERVIEW

A major challenge in the design of SDN controllers is developing abstractions that allow applications to exert control over their own traffic while empowering service providers to manage network-wide resources implemented over their circuits (e.g., OTN/optical circuits in packet-optical networks). This challenge is not met in most existing controllers, which provide abstractions based on virtual topologies but do not handle resources such as bandwidth and backup paths, or allow tenants to participate in management of virtual-physical mappings. SPN OS is designed to enforce a clear separation between control decisions made by tenants through VNOs and those made by operators through the Arbiter. The Arbiter allocates resources (bandwidth, optical lightpaths, etc.) when VNOs are created, and tenants are free to decide how to utilize the resources encapsulated in a VNO, often without having to consult the Arbiter.

Fig. 1 depicts the architecture of SPN OS, instantiated with a pool of VNOs, each potentially managed by a different tenant. We illustrate the main features of the architecture through a running example in which a tenant wishes to create a new VNO that connects several datacenters around the world for the purposes of deploying a new cloud-hosted application.

First, the tenant requests a new VNO from the Arbiter. The request specifies the VNO’s resource requirements, and is represented concretely as a topology decorated with annotations about bandwidth, fault tolerance, and other service attributes. SPN OS provides several ways of creating requests including pre-defined VNO templates, a programmatic interface based on NetKAT, and a visual editor. Given a config that captures the VNO’s requirements and preferences, the tenant could issue the following commands,

```java
myCloudVNO = new BigSwitchVNO();
myCloudVNO.init(config);
```

to instantiate a pre-defined template for a “big switch” VNO.

Second, the Arbiter creates one or more virtual-to-physical Mapping Patterns (MPs), either using hints supplied in the config or by computing them automatically. Because not all MPs are feasible in all network conditions, the arbiter identifies serviceable MPs that meet the preferences and constraints specified by the tenant. The Arbiter writes the serviceable MPs into the myCloudVNO object, together with information about constraints such as time periods, resiliency, cost, etc., and returns it to the tenant.

Third, the tenant selects an initial MP from the VNO based on current preferences and network conditions (congestion, availability, etc.), and it writes the selected MP into the VNO,

```java
myCloudVNO.mapTo(MP1);
```

which has the effect of using the physical paths specified in the MP to implement the virtual topology.

Likewise, if the tenant wishes to protect the VNO against failures by using the paths in MP2 as a backup, or set a restoration policy for recovering after a failure, it includes the following commands in the VNO as its built-in functions:

```java
monitorPhysicalNetwork();
remapTo(MP2);
restoreFrom(MP3);
```

The resources needed to implement these behaviors will be reserved for myCloudVNO, but not activated. Instead, these commands create triggers that will fire when network

---

1 These mappings could be nested, leading to virtual-to-virtual mapping patterns, but in this paper we focus on just one level of virtualization.
conditions change. They also notify the Arbiter, which takes such policies into account when performing admission control, resource reservation, and bookkeeping for other VNOs.

To activate the VNO, the tenant issues the command,

myCloudVNO.activate(now);

which configures the underlying physical network according to the MP and brings the VNO up so that the new cloud application can begin executing. Later, if the tenant wishes to temporarily deactivate the VNO service, it can issue the command,

myCloudVNO.deactivate(now);

which releases the network resources. The VNO service can be reactivated later:

myCloudVNO.reactivate(now);

Upon reactivation the allocation of resources may be different from the original allocation. For example, a different wavelength may be assigned to a certain lightpath. The VNO presents a consistent view, making such reallocation of resources transparent to the tenant. The flexibility brought by virtualization empowers the tenant to optimize resources while retaining provisioning capability for the operator.

III. SPN OS PLATFORM

The SPN OS platform is implemented using Object-Oriented Network Virtualization (OONV) [12]. Unlike controllers based on virtual topologies, a VNO provides a complete representation of a virtual network service throughout its lifecycle. In addition to unique identity and properties, a VNO also defines its behavior as a set of methods for executing its own custom functions. Whereas SDN programs define a virtual network topology as a collection of logical and physical properties and operates it externally, SPN programs define self-contained VNOs that execute autonomously.

The SPN OS may reside in logically-centralized dedicated servers or within one or more network elements. A physical (or virtualized) network operated and managed by SPN OS is called a SPN-enabled network. An operator that manages the SPN OS and SPN-enabled network to provide SPN-based virtual network services is called a SPN service provider. A user with virtual networks residing in the SPN service provider’s network is called a SPN tenant. Each service offered to a tenant is represented as a VNO in the SPN OS.

The main components in SPN OS are the VNO Pool and Arbiter, as shown in Fig. 1. The Pool holds all VNOs, using efficient storage primitives. The Arbiter interacts with and manages VNOs in the Pool, SPN tenants, and the SPN-enabled network, performing admission control, bookkeeping, scheduling, verification, policy enforcement, security, authentication, authorization, accounting, etc. It may also implement functions such as oversubscription through statistical multiplexing and sharing. Note that the Arbiter is only involved in tasks related to physical resource allocation for VNOs. During normal operation, most interactions between the VNO, its owner, and the SPN-enabled network are performed directly without consulting the Arbiter. In addition, the Arbiter would typically be replicated for scalability and fault tolerance. Overall, SPN enables centralized resource management at the Arbiter, as well as distributed control at individual VNOs. This approach is a departure from traditional SDN, where control and management functions are implemented using logically-centralized controllers.

A VNO is a complete representation of a virtual network service throughout its lifecycle. It provides topology virtualization and address space virtualization and also has built-in functions for traffic engineering, resiliency, in-service reconfiguration, upgrade, re-optimization, live migration, etc. The VNO’s properties include virtual node list, virtual node interface list, virtual link list, virtual address space, routing table, scheduler, etc. Depending on the level-of-transparency policy set for the VNO, it may also hold properties such as physical network topology (“as seen” from this VNO), virtual to physical topology and address space mappings, etc. The VNO’s behavior includes functions executed during initialization, during operation, and decommissioning.

A SPN tenant has both full visibility and control over the VNOs it owns through the SPN OS. VNO functions can be either executed automatically (e.g., scheduled reconfiguration, protection, and restoration) or invoked by the owner (such as live migration). A VNO can be designed either from scratch or generated from pre-defined VNO templates, such as Big Switch, Fat Tree, Protected Ring, etc. A VNO can interact with owners through northbound APIs, the Arbiter through Pool management APIs, the SPN-enabled network through southbound APIs, and with other VNOs through inter-VNO APIs, which are used for multi-VNO coordinated operations such as intra- and inter-domain networking. This provides another advantage over current controllers, which do not generally support inter-virtual network interfaces.

IV. THE NETKAT LANGUAGE

A distinctive feature of SPN OS is its use of the high-level language NetKAT for VNO programming [10]. Generally speaking, the introduction of a high-level language to a networking platform enables the decoupling of generalized service representation from the localized service instantiation. In the context of SPN OS, a VNO written using a high-level language abstraction in NetKAT instead of device-level programming provides portability of the virtual network service. In other words, the same VNO can be saved, exported, or live-migrated to any other SPN OS-enabled network, or be compiled to another instance of the same service, with guaranteed service quality and behavior.

To represent services in NetKAT, a VNO defines three programs: (i) the virtual network topology, (ii) the virtual network forwarding behavior, and (iii) the virtual-to-physical mapping. SPN OS leverages NetKAT’s unique capability to model the behavior of the entire network, including its topology. At creation, a VNO is programmed with a policy $p$, a topology $t$, and a virtual-to-physical relation $R$. At the time of VNO service instantiation, all three programs are provided to the NetKAT compiler, which produces forwarding rules that are translated to switch-level instructions in OpenFlow.
The NetKAT language provides primitives for matching and modifying packet headers, as well combinators such as union and sequential composition that merge smaller programs into larger ones. NetKAT enables programmers to think in terms of mathematical functions on packets histories, where a packet \( pk \) is a record of fields and a history \( h \) is a list of packets representing a forwarding path through the network. This is quite different from hardware-level APIs such as OpenFlow, which require thinking about details such as forwarding table rules, matches, priorities, actions, timeouts, etc. NetKAT fields \( f \) include standard packet headers such as Ethernet source and destination addresses, VLAN tags, etc., as well as special fields to indicate the port \( pt \) and switch \( sw \) where the packet is located in the network as well as the payload \( \text{payload} \).

NetKAT’s syntax is defined by the grammar in Fig. 2. Predicates \( a, b ::= \text{true} \mid \text{false} \mid f = n \mid f \neq n \mid a + b \mid a \cdot b \mid \neg a \) describe logical predicates on packets and include primitive tests \( f = n \), which check whether field \( f \) matches \( n \), as well as the standard collection of boolean operators. Policies \( p, q ::= a \mid f \leftarrow n \mid p + q \mid p \cdot q \mid p^* \mid [sw; pt] \to [sw'; pt'] \) describe functions on packets and include filters \( a \), which drop all packets that do not satisfy \( a \); modifications \( f \leftarrow n \), which set the \( f \) field to \( n \); union \( p + q \), which copies the input packet and processes one copy using \( p \) and the other copy using \( q \), and then takes the union of the results; sequence \( p \cdot q \) which processes the input packet using \( p \) and then feeds the outputs of \( p \) into \( q \); Kleene star \( p^* \), which iterates \( p \) zero or more times; and \( [sw; pt] \to [sw'; pt'] \), which represents a link between switch-port pairs.

Readers familiar with Frenetic [13], Pyretic [6], or NetCore [14], will be familiar with this model. However, note that NetKAT can model the behavior of the entire network, including the topology. It can also be used to implement virtual networks [6, 15] using a simple encoding.

V. IMPLEMENTATION AND RUNNING EXAMPLES

We have built a prototype implementation of SPN OS in Java. Fig. 3 depicts our prototype’s graphical user interface. Our current system models the mechanisms related to the lifecycle of virtual network services including request, activation, and deactivation using the VNO abstraction that can be executed on SDN switches emulated in Mininet.

The VNOs and Arbiter use GraphDB for their internal topology and mapping representation, as well as a Java-wrapped NetKAT compiler that translates VNOs to equivalent OpenFlow forwarding rules. Each VNO contains its own OpenFlow 1.0 control logic (using OpenFlowJ library) and controls its own slice of the virtual network through the SPN OS. The SPN OS currently serves as a proxy controller between multiple VNOs and the network, due to the limitation of OpenFlow specification that only allows switches to connect to one controller at a time. On the initiation of a VNO, the Arbiter obtains the VNO’s NetKAT program through its init() method, compiles the program, and returns the resulting forwarding rules to the Arbiter, which maintains the flow tables for all of the switches relevant to the VNO. The Arbiter can perform verification on such configurations to check validity (e.g., no logic flaws or unsupported hardware features) and serviceability (e.g., sufficient physical resources). The verified configurations are then written into the VNO as a valid candidate service pattern. Meanwhile, the topology, mapping and resource allocation information in both the VNO and the Arbiter are also updated accordingly. The VNO can use its one or more valid service patterns to activate, reconfigure, or deactivate itself on demand. For example, a VNO running on an OpenFlow network may send forwarding rules contained in one service pattern to control packet flows in one direction, and later switch to another service pattern to redirect flows in another direction. When decommission() is invoked, the VNO is deleted from the VNO Pool, and its associated resource allocation is released from the Arbiter’s global map.

To enhance modularity and flexibility, we implemented a version of the NetKAT compiler that executes an embedded web server to provide “compilation-as-a-service.” The server
offers RESTful APIs that (i) accept requests to compile a NetKAT program and (ii) obtain the flow tables for each switch. The messages in both directions are represented in JSON to ensure portability. In the future, we may integrate the compiler into SPN OS, and add support for northbound interfaces such as OpenStack Neutron, OpenDaylight’s group-based policies, etc. as well as southbound interfaces such as NETCONF/YANG, GMPLS, TL1, etc.

A. Examples: Big Switch and Ring VNOs

We present running examples of Big Switch and Ring VNOs to demonstrate our SPN OS implementation. Suppose that the physical packet network topology is as shown in the lower side of Fig. 4, and we wish to create two VNOs that implement “big switch” and “ring” topologies as shown in the upper side of Fig. 4. The service for the Big Switch VNO provides end hosts $h_1$, $h_2$, and $h_3$ with all-pairs connectivity. Fig. 5 shows a NetKAT program that implements this functionality. One may notice that the NetKAT program is capable of representing the virtual network service in a very simplified form. That is, for all traffic destined for IP 10.0.0.1, forward to port 1, for IP 10.0.0.2, to port 2, and so on. In this case, the virtual-to-physical mapping (more specifically, the virtual port connection-to-physical path mapping) is not specified. Accordingly, such mapping together with physical path routing will be automatically decided by the virtual and NetKAT compiler during compilation.

Due to the Java implementation of SPN OS, instead of the native NetKAT program mentioned above, a Java-wrapped version (as shown in Fig. 6) of the same program is written in the Big Switch VNO’s Service Representation field. Upon VNO initiation, the Arbiter sends the native NetKAT program (after removing the Java wrapper) to the compiler, which translates the program into OpenFlow forwarding rules. The Arbiter verifies these rules and then writes them in the VNO’s Service Pattern field. The VNO can then launch its virtual network service, which configures the relevant OpenFlow switches. The VNO enters service and connectivity between hosts is established. The service for the second VNO is similar: it provides connectivity to hosts $h_1$, $h_4$, $h_5$, and $h_6$ by forwarding around the ring. The corresponding NetKAT program (elided here) is slightly more involved due to more complex topology and forwarding behavior.

To illustrate the behavior of these VNOs, Fig. 7 show the results of a pingall test in Mininet. Although simple, this example shows that the VNOs correctly implement the desired services while ensuring isolation. In particular, $h_2$ in the first VNO cannot communicate with $h_4$ in the second VNO. However, since $h_1$ belongs to both VNOs, it can communicate with all hosts. Our system achieves this by first using NetKAT’s virtual compiler to translate each VNO separately, and then using its global compiler to translate the union of the devirtualized programs, which adds the necessary tags [15].

VI. USE CASES ENABLED BY SPN OS

Controllers today are usually based on logically-centralized architectures—a design that makes it easy to mix management and control functions, without a clear distinction between the two. In contrast, SPN OS separates distributed control of VNO
services from centralized management of network-wide resources. Each VNO operates in a self-administrative, autonomous fashion in response to user commands, network conditions, inter-VNO communications, etc. Meanwhile, VNOs are provisioned by the Arbiter and are subject to its policy-based supervision. We believe this separation of functionality and responsibility provides better (e.g., more flexible, scalable, robust, secure) support for a number of advanced virtual network services. The following potential use cases highlight the features and advantages of our design.

1) Fine-grained protection and restoration. Optical networks today guarantee limited forms of protection—e.g., under a single link or node failure. When optical protection fails, mechanisms at the packet layer (e.g., MPLS fast reroute/OSPF routing table update) are triggered to reroute the traffic, often without coordination with the underlying optical layer. This siloed approach makes it difficult to provide sufficient agility and resiliency for applications, which can require highly coordinated action among a group of lightpaths and across packet and optical layers. In contrast, a VNO can provide finer-granular protection and more resilient restoration by encapsulating multi-layer, multi-faceted protection and restoration strategies. The Arbiter can calculate and constantly update Mapping Patterns (MPs) for in-service VNOs based on current network conditions. A VNO may hold multiple protection MPs in parallel for fine-granular protection, such as a protection MP for surviving single link-cut, a second MP for surviving double link-cut, and so on. Under failure events, the VNO adaptively switches to the suitable protection MP in response to failure conditions. The VNO also holds backup MPs for best-effort restoration under massive failures (e.g., natural disasters) that are beyond anticipated scenarios.

Each VNO can execute a run-time monitor to track network conditions and, on detection of a failure, automatically switch to a protection or restoration MP. A switch to a protection MP can be executed instantaneously, without Arbiter intervention. In case of restoration, a VNO first asks the Arbiter to verify its restoration MPs and then switches to a valid MP. In addition to virtual-to-physical lightpath remapping, the VNO may also contain MPs that support dynamic virtual node-to-physical node remapping for enhanced resiliency. As the capacity of optical circuits grows to multi-100 Gbps, a single circuit failure may impact a large number of services with wide-ranging complexities and performance attributes. Computing massive number of complex restoration solutions upon failure may lead to severe delay and unsatisfactory restoration performance. Since the restoration of a VNO only requires identifying a valid restoration MP from pre-calculated and up-to-date candidates, it may offer more timely and resilient strategies.

2) Live migration. As virtual networks (VN) become more widely deployed, the demand for live migration will also grow. As an example, a cloud operator may wish to migrate a VN and its VMs together. VN live migration within the same physical network can be supported by SPN OS in a straightforward way, as it can be implemented as a type of pre-planned “protection” remapping for the VNO. Prior to live migration, the VNO would ask the Arbiter to calculate and write a destination MP that reflects the target locations of each node. The VNO would then perform live migration by simply switching from the current MP to the destination MP using remapTo() method. Going a step further, for live migration to a separate physical network managed by another SPN OS, the whole network status could be captured as a VNO image (like a VM image) and transferred to the destination SPN OS where it would be loaded, recompiled, and instantiated as a new VN service.

3) Provisioning with oversubscription. Virtualization offers a way to implement well-engineered, oversubscribed network services. Network oversubscription can be supported at the packet layer through statistical multiplexing, as well as supported at the optical layer through statistical sharing [16]. In SPN OS, oversubscription can be specified by the Arbiter when it performs admission control and resource allocation for VNOs. Dedicated resources can be allocated to VNOs that do not allow oversubscription, while shared resources can be allocated to VNOs that do (possibly at lower cost). The sharing policy (e.g., first-come-first-serve) and schedule (e.g., every other minute) can be written in the VNO’s service pattern. Upon activation, each VNO will utilize resources based on its service pattern, and freely share them with other VNOs. Depending on service types, VNOs sharing common resources may interact with each other to “negotiate” and schedule resource usage through inter-VNO communication in an autonomous manner. The Arbiter is only involved when a VNO needs to make a change to its service pattern—e.g., requires more dedicated resources instead of shared ones. The Arbiter also performs network-wide policy enforcement to ensure that all VNOs are using resources properly and the network resources are not oversubscribed beyond service level agreements, etc.

VII. CONCLUSION

This paper presented the design and implementation of SPN OS, a new controller framework that enables each tenant to flexibly manage its own virtual network service using the abstraction of a VNO. The specification of virtual network service in a VNO using NetKAT language can form the basis of future intent framework that provides universal portability across controller platforms. In the future, we plan to extend the backend of our prototype implementation to target optical networks, possibly leveraging recently proposed extensions to OpenFlow and Mininet. We also plan to explore issues such as security and performance, along with new use cases for SPN OS such as virtual network inter-networking in which multiple VNOs are composed vertically or horizontally to provide rich collaborative (or competitive) network services.

ACKNOWLEDGMENTS

The authors wish to thank Shrutarshi Basu for his numerous contributions to the SPN OS prototype. Our work is supported in part by the National Science Foundation under grants CNS-1111698, CNS-1413972, CCF-1422046, and CCF-1253165; and the Office of Naval Research under grants N00014-12-1-0757 and N00014-15-1-2177.

REFERENCES


