SDN Management Layer: Design Requirements and Future Direction

Yuefeng Wang Ibrahim Matta
Computer Science Department, Boston University
Boston, MA 02215
{wyf, matta}@bu.edu

Technical Report BUCS-TR-2014-006

***Abstract*—Computer networks are becoming increasingly complex and difficult to manage. The research community has been expending a lot of efforts to come up with a general management paradigm that is able to hide the details of the physical infrastructure and enable flexible network management. Software Defined Networking (SDN) is such a paradigm that simplifies network management and enables network innovations.**

In this survey paper, by reviewing existing SDN management layers (platforms), we identify the general common management architecture for SDN networks, and further identify the design requirements of the management layer that is at the core of the architecture. We also point out open issues and weaknesses of existing SDN management layers. We conclude with a promising future direction for improving the SDN management layer.

I. Introduction

Traditional networks are managed through low-level and vendor-specific configurations of individual network components, which is a very complicated and error-prone process. And nowadays computer networks are becoming increasingly complex and difficult to manage. This increases the need for a general management paradigm that provides common management abstractions, hides the details of the physical infrastructure, and enables flexible network management. Making the network programmable (pioneered by earlier research in Active Networking [26]) leads to such a general paradigm, as programmability simplifies network management and enables network innovations.

Software Defined Networking (SDN) has been proposed to enable programmable networks. In SDN, the network is considered to have two components: (1) *control plane* which determines *how* to handle and forward data traffic, and (2) *data plane* which handles and forwards data traffic toward its destination. SDN separates the control plane and data plane, and focuses on programming the control plane through a network management layer[1]. Through a high-level interface provided by the network management layer, network managers can easily manage the network without dealing with the complexity of low-level network details.

In general, the data plane might not only be a forwarding plane that just stores and forwards packets (or discards them) through packet flow (forwarding) table manipulations, but it might also include more application-specific data processing capabilities [1][8]. This is similar to the focus of earlier research in Active Networking, where network devices (switches or routers) are expected to perform computation on and modification of packet contents [26]. In this paper we focus on the control plane only for the purpose of programming the forwarding of packet flows, $i.e.$ the network management layer for SDN networks.

The main contribution of this paper is to identify the general common management architecture for SDN networks, and further identify the design requirements of the network management layer that is at the core of the architecture. The rest of the paper is organized as follows. We present the common management architecture for SDN networks in Section II. Design requirements of the SDN management layer, along with open issues and weaknesses of existing management layers, are described in Section III. Section IV briefly describes our approach that, we believe, offers a promising future direction for improving the SDN management layer. Section V concludes the paper with future work.

II. Management Architecture for SDN networks

The core of a management architecture for SDN networks is the management layer as shown in Figure 1. A management layer should enable the monitoring and control of the network. The management layer itself does not manage the network but provides a programmatic interface to management (or user) applications, which in turn manage the network. Examples of management applications include access control, virtual-machine (VM) migration, traffic-aware path selection and path adaptation, and redirecting or dropping suspected attack traffic.

A. Management Architecture Overview

Figure 1 shows a general network management architecture for SDN networks. At the bottom are the network devices including switches or routers[2]. There is a process (*switch process*) running on each network device, and this process hides the internal details of the physical device but exposes a *Network Device Interface* (the so-called "Southbound API" [22]). The Network Device Interface provides a standardized way to access the switch processes which operate on the switches. The switch process is responsible for low-level operations on switches such as adding/removing packet flow entries, and configuration of ports and queues. The Management Layer consists of one or more *controller processes*, which may run on one or more physical servers. Controller processes

[1]We use the terms "management platform", "management layer" and "control platform" interchangeably.

[2]In this paper, switches and routers are considered to be the same, and both provide Layer 2 and Layer 3 operations.

collaborate to provide the network monitoring and control functionalities. The Management Layer exposes a *Network Management Interface* (the so-called "Northbound API" [22]) for management (or user) application processes to manage the network.

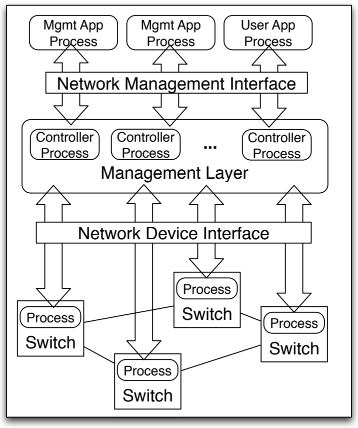

Fig. 1. A general network management architecture for SDN networks.

B. OpenFlow-based SDN networks

In an SDN network, the Network Device Interface can be supported by any mechanism (protocol) that provides communication between the control plane (management layer) and data plane (switch processes). OpenFlow [20] is such a mechanism (protocol) that gives the management layer access to switches and routers. OpenFlow is the first standardized open protocol that allows network administrators or experimenters to adapt the configuration of switches and routers from different vendors in a uniform way so as to add and remove packet flow state (forwarding) entries.

As OpenFlow can be easily deployed on existing hardware, OpenFlow soon became popular in the research community and industry. OpenFlow enables programming of the hardware without needing vendors to expose the internal details of their devices. OpenFlow is now supported by major vendors, and OpenFlow-enabled switches are commercially available.

OpenFlow is now the most commonly deployed SDN technology and is seen as an enabler of SDN. However, OpenFlow is not the only mechanism to enable SDN and support the Network Device Interface, and any mechanism that could provide communication between the control plane and data plane can be used. Forwarding and Control Element Separation (ForCES) [37] protocol is an example, however it is not adopted by major switch/router vendors. In this paper, we focus on OpenFlow-based SDN networks, which have recently attracted a lot of attention in the network management area due to the growing popularity of OpenFlow.

C. Administrator-level Interface and User-level Interface

There are two types of interface that can be provided by the network management layer: administrator-level interface and user-level interface. An administrator-level interface is provided to the network administrator, who uses this interface to write management applications to monitor and control the network as a whole. This interface is provided by default by all management layers.

On the other hand, a user-level interface is provided to network end-users. End-users write general applications (such as a video conference application or Hadoop-based application) using this interface to affect the management of their traffic, and as a result, achieve better performance, security or predictable behavior for their applications [14]. To achieve the same goal in an SDN network in the absence of a user-level interface, end-users may either (1) have to out-of-band request service from the network administrator, which is inconvenient and increases the workload on the network administrator, or (2) use a dedicated per-application management controller that runs as the administrator, which makes it hard to combine different application management controllers on the same physical network since decisions from different management controllers may conflict with each other.

D. Policy-based Network Management and Scope

By *policy-based network management* we mean that network management can be expressed in terms of high-level policies instead of network device configurations, which are low-level and vendor-specific. The network management layer is responsible for translating these high-level policies into low-level and vendor-specific configurations of network devices (switches or routers). Policies are in the form of a set of rules that define a set of network conditions, responses to these network conditions, and network components that perform these responses [18]. Advantages of policy-based network management include: simplifying device, network and service management, enabling the provision of different services to different users, managing the increasing complexity of programming devices, and supporting business-driven network configurations [25].

Contribution: One of our contributions in this paper is introducing and defining the concept of *scope* and *scoping* in network management as follows. A network management layer manages a network over a certain *scope* that includes network's physical components, *i.e.* devices, and logical components, *i.e.* processes. For a distributed management layer that consists of multiple management controllers, each management controller is a process that has its management *subscope*, which consists of a subset of network components (devices and processes). Also each policy has its own *subscope* where the policy may only affect a subset of network components. A policy is enforced on the network through one or multiple management controllers. *Scoping* (or support for *scope*) means that a management layer explicitly defines the subscope induced by a given policy, and dynamically creates new management subscopes and associated controller processes to activate such a policy. Scoping enables fine-grained control over the network and better support for policy-based management.

III. Design Requirements of Management Layer

In this section we describe the design requirements of the management layer for OpenFlow-based SDN networks. We focus on the requirements that should be supported by the management layer, and the architectural components needed to meet these requirements. Implementation details and tradeoffs are outside the scope of this paper.

A. A Global Network View and General API

A basic requirement of the management layer is to provide a global network view and offer a general API, which simplifies the programming of management applications. NOX [16], as shown in Figure 2, is the first OpenFlow management platform that met such a requirement. It is a follow-up work to previous control platforms (SANE [6] / Ethane [5]) that only focused on security features (access control). The NOX management layer contains only one controller.

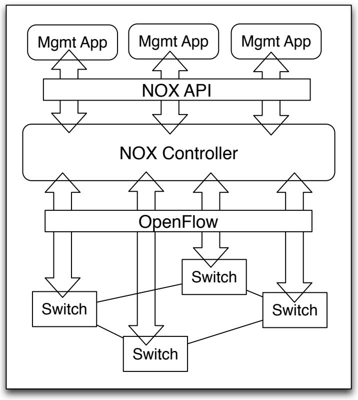

Fig. 2. NOX is a centralized management layer.

The global view of NOX includes the switch-level topology, and the location of users, hosts and services. NOX constructs the network view and bindings between (user, host and service) names and addresses through packet-flow initiations and built-in base applications that use DNS, DHCP and LLDP. The view does not include the current state of network traffic, but applications can query the status of switches through OpenFlow messages.

register_handler $< event\_name, handler >$
send_openflow$< dpid, of_message >$
install_datapath_flow $< dpid, flowAttr, actions >$
delete_datapath_flow $< dpid, flowAttr >$

TABLE I
Selected NOX API.

A selected subset of NOX API is shown in Table I. The NOX API enables applications to send OpenFlow messages to a switch (identified by *dpid*) or to install/remove flow rules on it. The NOX API also allows applications to register handlers for particular events. These events include connection creation and deletion, user registration and unregistration, link going up and down, switch join and leave, packet received, switch statistics received, and other application-specified events.

NOX controls network traffic by sending instructions to switches through OpenFlow messages, which install flow state (forwarding) entries in switches. A flow entry in OpenFlow switches contains a set of header matching fields, packet counters and corresponding actions. When a packet arrives at a switch, if the packet matches a flow entry in the switch, the switch updates the counter and applies corresponding actions. If the packet does not match any flow entry, the packet is forwarded to the management layer (NOX controller), and the controller determines what to do by checking registered event handlers.

As illustrative examples, we describe next how NOX performs network discovery, and access control and routing: (1) For network discovery, each switch sends out LLDP messages through its ports to its neighbors. When LLDP messages are received by neighbor switches, as these messages do not match any flow entry, they are forwarded to the NOX controller. Through monitoring the sending and receiving of these LLDP messages by switches, NOX figures out the network topology. (2) For access control, the first packet to the destination from the sender is forwarded to the NOX controller by the first-hop switch as it does not have a corresponding flow entry. When the NOX controller receives this packet, the built-in access control application (handler) decides whether the flow is allowed or not. If so, the built-in routing application computes the Layer-2 route in a centralized way (similar to the Routing Control Platform in [3]) based on the network topology, and translates the route to a set of flow entries installed in switches along the path to the destination; otherwise the packet is simply discarded.

Writing complicated programs with NOX is difficult since (1) management applications have to configure each switch separately, as well as the behavior of the NOX controller itself when no matching rule is found when a switch receives a packet, and (2) different flow rules are not easy to compose as NOX does not support rule operations such as negation and union. Many management platforms with high-level language support have been proposed to simplify management programming, wherein they translate the programs written in a high-level language into low-level switch configurations. These platforms include Flow-based Management Language (FML) [17], Procera [31], Frentic [15], Pyretic [21], and Maple [32]. Also, the NOX controller is a single-threaded process and not optimized for performance, and many multi-threaded management controllers have been proposed, including NOX-MT [30] and Beacon [10].

Open Issues: Even though many high-level languages have been developed, programming management applications still has to deal with a lot of low-level details of the network, such as per-link or per-switch configurations. Also there is no standard SDN management API — many different management APIs have been proposed, but they are not extended from existing ones and there is not much evolution of these APIs.

B. Distributed Controllers

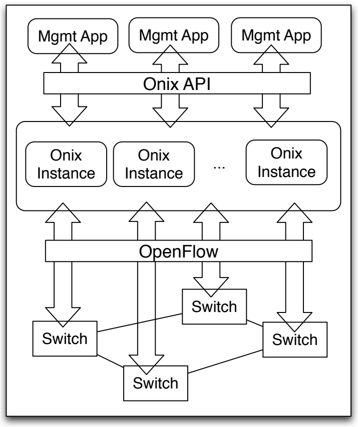

Fig. 3. Onix consists of distributed controllers (Onix instances).

NOX is a centralized management layer that has a single controller. However, for a large-scale network, a centralized management layer is not enough with respect to scalability and reliability, so it is necessary to distribute the management layer to run on distributed controllers. Onix [19], shown in Figure 3, is a distributed (proprietary) management layer that consists of multiple Onix instances (controllers). Each Onix instance connects to and manages a subset of the network devices. Onix enables flexible network state distribution primitives between Onix instances and between Onix instances and switches, so management applications do not have to re-implement the distribution mechanism. Onix helps address the scalability issue through multiple Onix instances, and also by enabling the partitioning and aggregation of the management subscope of each Onix instance.

Onix maintains a Network Information Base (NIB), which contains all *network entities*, including nodes, links, ports, forwarding tables, and so on. The NIB is replicated and distributed over Onix instances, and Onix makes sure that the NIB state is consistent among them. Each network entity is stored as an object in the NIB. Onix provides a more general API than NOX: it enables management applications to access (*i.e.* create, destroy, inspect, and modify) network entities through operations on the NIB, and it also supports notification callbacks on some network state changes. The operations on the NIB are automatically translated to flow operations on switches. This is different from NOX as NOX applications have to specify operations on each switch.

Open Issues: Scoping (Section II-D) is not well supported in Onix and other SDN distributed management layers such as HyperFlow [29]. Onix supports the creation of new Onix instances with new scopes through aggregation or partitioning, but the new scope is restricted to devices that are physically close to each other. Scope in Onix is thus flat, *i.e.* it spans only one level of processes and a higher-level scope that spans distant processes is not supported. Furthermore, it is not easy to define the subscope induced by a given policy.

C. Network Virtualization

Network virtualization provides support for multiple isolated virtual networks to be built on top of the same physical network. It is an important aspect of the management layer since (1) it can improve resource utilization of the physical network by enabling network consolidation, and (2) it can be used to build (virtual) testbeds that provide a safe and realistic environment for developing and testing new network features (protocols and applications) in isolation before running them on the real network. FlowVisor [24] is a centralized management layer that provides network virtualization, thus it enables building and controlling multiple user-defined virtual networks on the same physical network. FlowVisor can be seen as a network hypervisor as shown in Figure 4.

FlowVisor acts as a transparent proxy between user-defined guest controllers and switches. It enables multiple NOX controllers (or other controllers such as Beacon [10]) to share the same switches. Each guest controller has full control over its subscope, or so-called network *slice* (an instance of virtual network), where a *slice* is a subscope (subset) of the scope managed by FlowVisor. FlowVisor provides transparency and isolation between slices by inspecting, rewriting and policing OpenFlow messages that it receives from guest controllers.

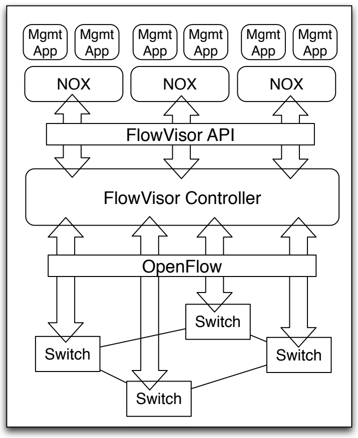

Fig. 4. FlowVisor acts as a network hypervisor to provide network virtualization.

In FlowVisor, a *flowspace* assigned to a slice is defined by a collection of packet header fields including: src/dst MAC address, VLAN id, Ethernet protocol type, IP protocol, src/dst IP address, ToS/DSCP and src/dst port number. FlowVisor isolates slices from each other by making sure slices' flowspaces do not overlap.

FlowVisor has several drawbacks including: (1) the virtual topologies are restricted by the physical topology. If two physical switches, to which two virtual switches map, are not directly connected in the physical network, then these two virtual switches cannot be directly connected in the virtual network; and (2) virtual networks do not have a separate virtual flowspace. Flowspaces of the physical network are assigned to different virtual networks, and the same flowspace cannot be controlled by different slices. To overcome the above drawbacks, several management layers have been proposed such as ADVisor (Advanced FlowVisor) [23] and FlowN [9]. Both ADVisor and FlowN enable the creation of virtual topologies that are completely decoupled from the underlying physical network, and guest controllers have completely separate virtual flowspaces.

Open Issues: The management layers mentioned above can only provide network virtualization over networks that are under a single administrative domain. Network virtualization across multiple administrative domains has recently attracted more attention, $e.g.$ [11], [27], as this is important in environments such as a cloud computing marketplace where multiple cloud providers are present.

D. User-level Interface Support

As we have mentioned in Section II-C, it is important for the management layer to support a user-level interface that enables better user application performance. FlowVisor enables users to place control over the network through network virtualization, but each user has to program a separate controller which introduces more overhead. PANE [14], as shown in Figure 5, is a centralized management layer, which directly delegates read and write authority from the network administrator to end-users by providing a user-level interface.

PANE is developed based on the concept of *participatory networks*. PANE enables multiple user applications to place controls over the network (including reserving resources, providing hints about future traffic, and querying network state). PANE uses a Network Information Base (NIB) to store network elements and their states (including hosts, switches, ports, queues, links, and their capacity such as rate-limiter or per-port output queues in a switch).

Share :	$S \in \{P\} \times \{F\} \times \{Priv\}$
Principal :	P = (user, host, app)
Flow :	$F =<$ srcIP, dstIP, proto, srcPort, dstPort>
Privilege :	$Priv$ = **CanDeny** t secs $\mid$ **CanAllow** t secs
	$\mid$ **CanReserve** n Mbps $\mid$ **CanRatelimit** n Mbps
	$\mid$ **CanWayPoint** {IP} $\mid$ **CanAvoid** {IP}
Request:	Req = **Deny** t secs $\mid$ **Allow** t secs
	$\mid$ **Reserve** n Mbps $\mid$ **Ratelimit** n Mbps
	$\mid$ **WayPoint** {IP} $\mid$ **Avoid** {IP}

TABLE II
SOME OF THE CONCEPTS IN THE PANE API.

Some of the concepts in the PANE API are shown in Table II. In PANE, a *principal* is a triple consisting of an application running on a host by a user. A *flow* is identified by source/destination IP address, port number and transport protocol type. A *share* determines privileges that principals have on certain flows. Such *privileges* include allowing/denying traffic for t seconds, reserving/rate-limiting bandwidth up to n Mbps, and directing traffic through or around particular IP addresses. PANE maintains a *share tree* that stores the privileges of principals, and shares in the share tree are added (or removed) by the network administrator. Principals manage flows over a certain time period by sending *Requests* to PANE. PANE in turn decides whether such requests can be realized in the network based on the share tree.

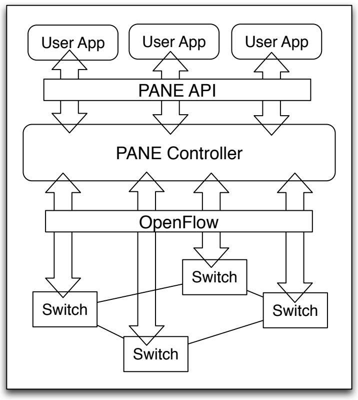

Fig. 5. PANE provides user-level API to end-users.

Open Issues: PANE allows users to reserve bandwidth, however other aspects of QoS support (including loss rate and delay guarantees) are not supported. A management layer should provide users with an API that offers predictable network connections as this is crucial for user application performance. However, since existing SDN systems are tied to the TCP/IP architecture, the rudimentary "best-effort" delivery service of TCP/IP makes it hard for the SDN management layer to support QoS requirements. Other requirements, such as mobility and security, are also made challenging due to limitations inherent in the TCP/IP architecture [7], [2].

E. Network Orchestration

The management layer may receive requests from different management (or user) applications. These requests may conflict with each other (for example, one request may deny all traffic to port 80, and another one may allow such traffic), which affects the normal operation of the network. Many management layers (such as NOX and Onix) expect applications themselves to avoid or resolve conflicts, but this is difficult to achieve especially when applications belong to different users. So it is important for the management layer to provide a network orchestration mechanism: the capability of resolving conflicts between different applications.

PANE [14] resolves conflicts between different user-level applications through Hierarchical Flow Tables (HFTs) [13]. HFTs constitute a policy tree where each node in the tree stores one or more *policy atoms* (requests that are installed on the network). A policy atom is a pair of flow matching rule and corresponding action. In PANE, a conflict happens when policy atoms overlap with each other, $i.e.$, there is a flow that matches more than one policy atom with contradictory actions.

To resolve conflict, when a packet arrives at PANE, PANE first finds all matching policy atoms in the policy tree, and applies the *conflict-resolution operator* based on the positions of policy atoms in the policy tree, and eventually returns a single resolved action. A conflict-resolution operator takes two policy atoms as an input, and returns a resolved action based on their relation in the policy tree (in-node, parent-child, or sibling-sibling). Namely, PANE first resolves the conflict between policy atoms in the same node (in-node), then in siblings under the same parent node, and lastly resolves the conflict with the parent node. The semantics of the conflict-resolution operators need to be predefined by the PANE administrator and can be extended.

Open Issues: PANE and other work such as Maestro [4] focus on resolving conflicts between requests sent to the management layer. However, an important aspect that is not yet well studied is how to compose different policies (which may or may not conflict with each other) over different scopes (or the same scope) in order to achieve better performance in terms of resource utilization, routing convergence and overhead, *etc*.

IV. Proposed Approach: RINA Architecture

In the previous section, we pointed out open issues and weaknesses of existing SDN management layers, including weak QoS support and manageability. In summary, existing SDN management layers are tied to the Internet architecture, which is known to be flawed in many respects such as security, mobility and QoS support. Tied to TCP/IP, this inevitably introduces these problems into the management layer and costs more just to work around these problems. The research community has been trying to improve the SDN management layer by resorting to ad-hoc patches that resolve issues with TCP/IP. Taking QoS support as an example, earlier versions of the OpenFlow protocol only provide operations on forwarding entries, and do not allow operations on switch queues and scheduling policies, which are important aspects to support QoS. Many SDN management layers (such as PANE [14]), in their attempt to provide QoS support, have to rely on mechanisms such as reservations and prioritized queue management. Also most existing SDN management layers are limited to networks within a single administrative domain. And it is not easy to define new scopes (or subscopes) of management, and so far there are no common SDN mechanisms to facilitate collaboration across different administrative domains.

We believe that a better approach is to build a management architecture on top of a new network architecture that avoids the shortcomings of the TCP/IP architecture. Our solution is to adopt the Recursive InterNetwork Architecture (RINA) [7], [2], which inherently solves such shortcomings by addressing the communication problem in a fundamental and structured way.

A. RINA Overview

RINA is based on the fundamental principle that *networking is Inter-Process Communication (IPC) and only IPC*. RINA has two main design principles: (1) divide and conquer (recursion), and (2) separation of mechanisms and policies.

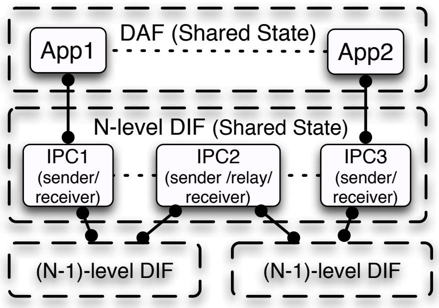

Fig. 6. RINA overview.

As shown in Figure 6, a collection of distributed application processes with shared state is called a *Distributed Application Facility (DAF)*, and each DAF performs a certain function such as weather forecast service or communication service. A *Distributed IPC Facility* (DIF) is a specialization of a DAF whose job is only to provide communication service. In RINA, application processes talk to each other using the communication service provided by a certain underlying DIF. A DIF defines and operates over a certain management scope (*i.e.* range of operation in terms of scale and performance characteristics). Recursively, a DIF that provides a larger communication scope, uses the communication services provided by multiple lower-level and smaller-scope DIFs.

RINA simplifies the network system by using only two policy-configurable protocols: (1) the Common Distributed Application Protocol (CDAP), which is the only application protocol required and is used for programming network management and user applications; and (2) the Error and Flow Control Protocol (EFCP), which is used for data transfer. More details about the RINA architecture can be found in [7], [2].

B. RINA Policy-based Management Architecture

RINA provides better manageability support with scoping—it enables recursive dynamic layer (DIF) instantiation [12], where a DIF (virtual network) with a new management scope can be dynamically and recursively formed over existing management scopes supported by underlying DIFs. The new scope can be a subscope of an existing scope, and more importantly, it can be a larger scope that spans multiple existing scopes (over multiple management administrative domains), *i.e.*, RINA supports nested scopes. DIFs over different scopes can be easily configured with different policies, but they use the same recursive RINA mechanisms [7].

What's more, RINA inherently and explicitly supports QoS through the RINA API [36] to connect application processes. Due to the explicitness of the QoS request during the connection allocation phase, RINA can achieve better resource utilization, and more importantly, help end-users improve application performance. The provisioning of QoS can be easily supported by RINA's recursive mechanisms, such as flow allocation and error control, and the management policies can be recursively composed over different management scopes.

Under the RINA management architecture, the distinction in SDNs between Southbound API and Northbound API is eliminated, and these two APIs are merely replaced by the unified and recursive RINA API, which provides a high-level (both administrative and user) interface (cf. Section II-C). Management (and user) applications can be programmed recursively using the RINA API [33] over different scopes at different levels. Inspired by RINA, the SPlit ARChitecture (SPARC) [28] advocates recursive SDN control but its realization exposes recursively an OpenFlow-based API, hence SPARC suffers from limitations such as being tied to TCP/IP and not recursing an IPC building block (DIF) that exposes a high-level abstraction for connecting applications. In other words, a RINA-based SDN would recursively expose an API that is programmable through high-level policies, rather than rules that use low-level information on switches and transport flows. For example, RINA programmers can simply allocate a flow with a certain flow policy (such as QoS) using application (or service) names as follows:

```
Flow flow = new Flow(srcAppName, destAppName, policy);
allocate (flow);
```

On the other hand, NOX programmers have to deal with flow rules on each switch (cf. Table I), and PANE programmers have to specify transport flow details such as IP address, port number and protocol type (cf. Table II).

Our RINA management architecture does not build upon existing SDN (or traditional) network architectures. To deploy RINA, physical nodes should support recursive process management, as RINA views networking as only inter-process communication. RINA could also be deployed in the presence of legacy physical devices through a shim layer (overlay) that virtually connects RINA-capable nodes and enables their communication. This overlay thus forms the base case for RINA's recursive communication service. On top of that, RINA uses its own recursive mechanisms (*e.g.*, resource allocation, flow and error control) to provide communication service of different SLAs (service-level agreements), and manages the network over different DIFs that can be easily configured with different policies (*e.g.* for addressing, routing, and authentication within each DIF).

Our preliminary work [33], [36], [34], [35] indicates that RINA's policy-based architecture enables programming of networks to improve their management, thus it offers a promising direction for SDN.

V. Conclusion

In this survey paper, through reviewing existing SDN management layers, we identified the common management architecture for SDN networks, as well as the design requirements of the management layer that is at the core of the architecture. We also pointed out open issues and weaknesses of existing management layers, including weak QoS support and manageability. We proposed and briefly presented a recursive, policy-configurable approach toward a superior management of networks. We continue to investigate the benefits of our approach through analysis and experimentation.

Acknowledgment

We would like to thank the National Science Foundation (NSF grant CNS-0963974) and the GENI Project Office.

References

[1] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz. Forwarding Metamorphosis: Fast Programmable Match-action Processing in Hardware for SDN. *SIGCOMM Comput. Commun. Rev.*, 43(4):99–110, Aug. 2013.

[2] Boston University RINA Lab. http://csr.bu.edu/rina/.

[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 *Proceedings of the 2Nd Conference on Symposium on Networked Systems Design & Implementation - Volume 2*, NSDI'05, pages 15–28, Berkeley, CA, USA, 2005. USENIX Association.

[4] Z. Cai, F. Dinu, J. Zheng, A. L. Cox, and T. E. Ng. The preliminary design and implementation of the maestro network control platform. Technical report, Rice University Technical Report TR08-13, 2008.

[5] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. McKeown, and S. Shenker. Ethane: Taking Control of the Enterprise. *SIGCOMM Comput. Commun. Rev.*, 37(4):1–12, Aug. 2007.

[6] M. Casado, T. Garfinkel, A. Akella, M. J. Freedman, D. Boneh, N. McKeown, and S. Shenker. SANE: A Protection Architecture for Enterprise Networks. In *Proceedings of the 15th Conference on USENIX Security Symposium - Volume 15*, USENIX-SS'06, Berkeley, CA, USA, 2006. USENIX Association.

[7] J. Day, I. Matta, and K. Mattar. Networking is IPC: A Guiding Principle to a Better Internet. In *Proceedings of ReArch'08 - Re-Architecting the Internet (co-located with CoNEXT)*, New York, NY, USA, 2008.

[8] M. Dobrescu, N. Egi, K. Argyraki, B.-G. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy. RouteBricks: Exploiting Parallelism to Scale Software Routers. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 15–28, New York, NY, USA, 2009. ACM.

[9] D. Drutskoy, E. Keller, and J. Rexford. Scalable Network Virtualization in Software-Defined Networks. *Internet Computing, IEEE*, 17(2):20–27, March 2013.

[10] D. Erickson. The Beacon Openflow Controller. In *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, HotSDN '13, pages 13–18, New York, NY, USA, 2013. ACM.

[11] F. Esposito, D. Di Paola, and I. Matta. A General Distributed Approach to Slice Embedding with Guarantees. In *IFIP Networking Conference*, Brooklyn, New York, USA, May 2013.

[12] F. Esposito, Y. Wang, I. Matta, and J. Day. Dynamic Layer Instantiation as a Service. In *Demo at USENIX Symposium on Networked Systems Design and Implementation (NSDI 2013)*, Lombard, IL, USA, April, 2013.

[13] A. D. Ferguson, A. Guha, C. Liang, R. Fonseca, and S. Krishnamurthi. Hierarchical Policies for Software Defined Networks. In *Proceedings of the First Workshop on Hot Topics in Software Defined Networks*, HotSDN '12, pages 37–42, New York, NY, USA, 2012. ACM.

[14] A. D. Ferguson, A. Guha, C. Liang, R. Fonseca, and S. Krishnamurthi. Participatory Networking: An API for Application Control of SDNs. *SIGCOMM Comput. Commun. Rev.*, 43(4):327–338, Aug. 2013.

[15] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker. Frenetic: A Network Programming Language. *SIGPLAN Not.*, 46(9):279–291, Sept. 2011.

[16] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker. NOX: Towards an Operating System for Networks. *SIGCOMM Comput. Commun. Rev.*, 38(3):105–110, July 2008.

[17] T. L. Hinrichs, N. S. Gude, M. Casado, J. C. Mitchell, and S. Shenker. Practical Declarative Network Management. In *Proceedings of the 1st ACM Workshop on Research on Enterprise Networking*, WREN '09, pages 1–10, New York, NY, USA, 2009. ACM.

[18] M. Jude. Policy-based Management: Beyond The Hype. *Business Communication Review*, pages 52–56, 2001.

[19] 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 *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–6, Berkeley, CA, USA, 2010. USENIX Association.

[20] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. OpenFlow: Enabling Innovation in Campus Networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74, Mar. 2008.

[21] C. Monsanto, J. Reich, N. Foster, J. Rexford, and D. Walker. Composing Software-defined Networks. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, nsdi'13, pages 1–14, Berkeley, CA, USA, 2013. USENIX Association.

[22] Open Networking Foundation. https://www.opennetworking.org/.

[23] E. Salvadori, R. Doriguzzi Corin, A. Broglio, and M. Gerola. Generalizing Virtual Network Topologies in OpenFlow-Based Networks. In *Global Telecommunications Conference (GLOBECOM 2011), 2011 IEEE*, pages 1–6, Dec 2011.

[24] R. Sherwood, G. Gibb, K.-K. Yap, G. Appenzeller, M. Casado, N. McKeown, and G. Parulkar. FlowVisor: A Network Virtualization Layer. In *Technical Report, OpenFlow-TR-2009-1, OpenFlow Consortium*, 2009.

[25] J. Strassner. *Policy-based Network Management: Solutions for the Next Generation*. Morgan Kaufmann, 2003.

[26] D. L. Tennenhouse, J. M. Smith, W. D. Sincoskie, D. J. Wetherall, and G. J. Minden. A Survey of Active Network Research. *Comm. Mag.*, 35(1):80–86, Jan. 1997.

[27] The OpenDaylight Project. http://www.opendaylight.org/.

[28] The SPARC Project: Split Architecture for Carrier Grade Networks. http://www.fp7-sparc.eu/.

[29] A. Tootoonchian and Y. Ganjali. HyperFlow: A Distributed Control Plane for OpenFlow. In *Proceedings of the 2010 Internet Network Management Conference on Research on Enterprise Networking*, INM/WREN'10, pages 3–3, Berkeley, CA, USA, 2010. USENIX Association.

[30] A. Tootoonchian, S. Gorbunov, Y. Ganjali, M. Casado, and R. Sherwood. On Controller Performance in Software-defined Networks. In *Proceedings of the 2nd USENIX Conference on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services*, Hot-ICE'12, pages 10–10, Berkeley, CA, USA, 2012. USENIX Association.

[31] A. Voellmy, H. Kim, and N. Feamster. Procera: A Language for High-level Reactive Network Control. In *Proceedings of the First Workshop on Hot Topics in Software Defined Networks*, HotSDN '12, pages 43–48, New York, NY, USA, 2012. ACM.

[32] A. Voellmy, J. Wang, Y. R. Yang, B. Ford, and P. Hudak. Maple: Simplifying SDN Programming Using Algorithmic Policies. *SIGCOMM Comput. Commun. Rev.*, 43(4):87–98, Aug. 2013.

[33] Y. Wang, N. Akhtar, and I. Matta. Programming Routing Policies for Video Traffic. In *International Workshop on Computer and Networking Experimental Research using Testbeds (CNERT 2014), co-located with ICNP 2014*, Raleigh, NC, USA, October 2014.

[34] Y. Wang, F. Esposito, I. Matta, and J. Day. Recursive InterNetworking Architecture (RINA) Boston University Prototype Programming Manual. In *Technical Report BUCS-TR-2013-013, Boston University*, 2013.

[35] Y. Wang, F. Esposito, I. Matta, and J. Day. RINA: An Architecture for Policy-Based Dynamic Service Management. In *Technical Report BUCS-TR-2013-014, Boston University*, 2013.

[36] Y. Wang, I. Matta, F. Esposito, and J. Day. Introducing ProtoRINA: A Prototype for Programming Recursive-Networking Policies. *ACM SIGCOMM Computer Communication Review (CCR)*, July 2014.

[37] L. Yang, R. Dantu, T. Anderson, and R. Gopal. Forwarding and control element separation (ForCES) framework. *RFC 3746, April*, 2004.