Northbound APIs for SDN: Design Principles, Challenges, and Opportunities

Anusha Reddy*, Nikhil Raj
* Department of Information Technology, SRM Institute of Science and Technology, Kattankulathur, India

ABSTRACT
Most traditional network architectures have their data plane and control plane put together, that is, they are vertically integrated. To combat the disadvantages of these kind of architectures, the concept of programmable networks was introduced, and has gained a lot of momentum since then. These are known as Software Defined Networks (SDN). The usual physical boundaries in a network like switches and routers are broken in SDN architecture using Application Programming Interfaces (API). The API that resides in between the controller and the application layer in SDN is known as the Northbound API. Northbound API presents a network abstraction interface to the applications and the management systems at the top of the SDN stack, and is hence considered to be the most important component of SDN Architecture. Due to its dynamic nature and the many developments occurring in it, the standardization of Northbound API has been a topic of discussion. This paper gives a brief overview of SDN, with a focus on the Northbound API. We survey the different kinds of Northbound API that currently exist, and then present the various opinions that exist on their standardization.

KEYWORDS: Network Management, SDN, Northbound API, Standardization

INTRODUCTION
Traditional network architectures are both hard to manage and complex in nature. This is because of their vertical integration (where the data and control plane are together) and also because the difficulty in configuring it and then trying to reconfigure it when faults occurred. A new paradigm was required that broke vertical integration and allowed an administrator to program the network. [1]
Along with this, the advent of a set of a network devices with purpose built application specific integrated circuits (ASICs) that have shortcomings such as flexibility and extensibility also necessitated the need of a new paradigm. The limited set of commands that are provided in a network based on an embedded operating system (OS) added to the need. [2] These reasons have prompted the creation of Software Defined Networks (SDN).
Software Defined Networks is an emerging network architecture that allows a centralized software program to control the behaviour of an entire network. This is done by separating the components that make up a network (such as the Data Plane, Control Plane, Infrastructure Plane) in a completely different way as opposed to its traditional counterpart. It divides the network's control logic from the underlying routers and switches.
It also allows network administrators to dictate the infrastructure of the network through this centralized software, which is known as the SDN Controller, instead of manually working with the physical switches or routers that make up the network. It also introduces the ability of programming the network, a relatively new concept in the area of Networking.
Software Defined Networks hence makes it easier to create new abstractions in networking, essentially simplifying network management and facilitating network evolution. The architecture of SDN is given below.
A Software Defined Network comprises of an Infrastructure Layer and a Control Layer that communicate with one another through a Southbound API. Above the Control Layer reside the Business Application that exercise their control over the network using a Northbound API. [3] The Infrastructure Layer comprises of the hardware found in the network in terms of routers and switches. The infrastructure found in an SDN is less expensive than ones found in a traditional network and they do not need to be constantly changed in order to upgrade the network.

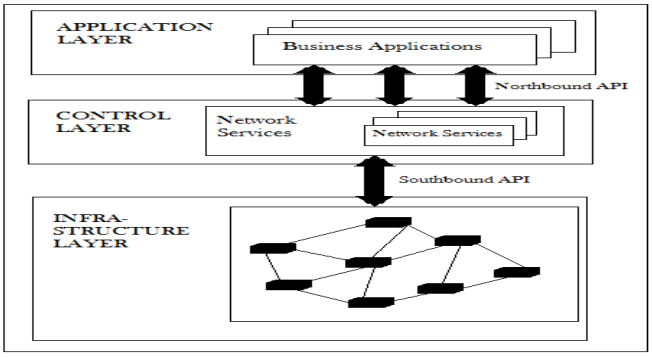

Fig. 1. SDN Architecture

This is because they are now programmable and their control lies with the SDN Controller. They are connected to the controller via the Southbound APIs.

- Southbound APIs form the interface between the SDN Controller and the network switches or routers. These have control over the forwarding operations, event notifications, statistical reporting and also advertise the capabilities of the network. Essentially, it allows a controller to define the behaviour of the hardware in the network. The standardized and most common Southbound API is OpenFlow.
- The Control Layer comprises the SDN Controller which is the centralized unit in charge of translating the network requirements from the SDN Applications down to the network switches. It also provides the Business Applications residing in the Application Layer with an abstract view of the network. This is done through the Northbound API.
- The Application Layer consists of programs that explicitly communicate their network requirements and expected network behaviour to the controller through the Northbound APIs.
- Northbound APIs present an abstraction of network functions with a programmable interface for applications to consume the network services and configure the network dynamically. They allow the applications to dictate the behaviour of the network.

According to Deigo Kreutz et al. [1], there are four pillars with which SDN can be defined as a network architecture. They are as follows:

- The control and data planes are decoupled. Control functionality is removed from network devices that will become simple (packet) forwarding elements. They do not contain any kind of network logic within them.
- Forwarding decisions are flow-based, instead of destination-based. A flow in the context of SDN, is a sequence of packets between a source and a destination. All packets of a flow receive identical service policies at the forwarding devices. The abstraction in this flow of traffic allows unifying the behaviour of different types of network devices, including routers, switches, firewalls, and so on. This enables unprecedented flexibility to the network
- Control logic is moved to an external entity, the known as the SDN controller. This is essentially a software platform that runs on commodity server technology and provides the essential resources and abstractions to facilitate the programming of forwarding devices based on a logically centralized, abstract network view. Its purpose is therefore similar to that of a traditional operating system.
- The network is programmable through software applications running on top of the Controller that interacts with the underlying data plane devices. This is a fundamental characteristic of SDN, considered as its main value proposition. And this is where the Northbound API is extensively used.

SDN directly addresses the fact that the current static architecture of networks is not suitable for dynamic computing and storage needs of current data centers and environments. Hence the features that this architecture has are [4]:

- The network is directly programmable as the control is decoupled from its forwarding functions.
- It is agile as this abstraction of control allows administrators to dynamically adjust the flow of traffic across a network according to changing needs.
- As it is centrally managed, a global view of the network can be maintained. This allows it to appear as a single, local switch to applications and policy engines.
- Network managers can write dynamic, automated SDN programs that configures and manages network resources, among other things. These can now be written as they do not depend on the proprietary software.
- Most of all, SDN is implemented through open standards which simplifies network design and operation as instructions are provided by SDN controllers instead of vendor specific devices and protocols. This standardization effort is a continuous effort with respect to SDN.

In SDN, the separation of the control and data planes are realised using a well-defined programming interface that exists between the switches, the SDN controller as well as the applications. They are typically connected by a closed control loop, where, firstly, the control plane receives network events from the data plane. Secondly, the control plane computes some network operations based on the events for the data plane and lastly, the data plane execute the operations that effectively change the state of the network. In order to complete the second function, the Northbound API has to provide a high-level API between the controller and the applications.

Having been originally coined to represent the work around OpenFlow at Stanford University in 2010 [5], SDN is now used to represent network architectures where the control and data plane have been decoupled and that have forwarding, distribution and specification abstractions. Being a relatively new paradigm, SDN and its related issues require standardization. The first and now industry standardized [6] Southbound API is called OpenFlow and was developed by the Open Networking Foundation (ONF). While some of them have been addressed, others are in the evolving process. One group that belongs to the latter category is the Northbound API of SDN which will be the focus of this paper.

The rest of the paper is organized as follows. Section II presents the Northbound API, its importance in Software Defined Networking and the need for its standardization. Section III deals with the different kinds of Northbound APIs that exist currently and while Section IV points out the advantages and disadvantages of its standardization. The paper is concluded in Section V.

MATERIALS AND METHODS

NORTHBOUND API

Software-Defined Networking (SDN) is a shift in network-based computing based on breaking existing physical boundaries on switches, routers, and controllers through well-defined APIs. An Application Programming Interface (API) is said to be used to define the software interaction among systems. [7] In SDN, these 'systems' refer to network applications and hardware such as routers, switches and so on. The 'programming' part of the API is what makes it necessary for SDN. This programming capability is what gives APIs the flexibility in contrast to other system exchanges. Along with this, their simplicity and efficiency is what makes it invaluable for SDNs. APIs therefore, essentially allow a 'system' to send a request or reply to queries.

As the most crucial component of SDN is the communication between network elements [8], APIs are used for it. There are two kinds of communications that occur in SDN; one is between a forwarding plane element and a controller and the next is between the controller the applications or business logic.

The former one is called a Southbound API. Southbound APIs facilitate efficient control over the network and enable the SDN Controller to dynamically make changes according to real time demands and needs. [9] It allows the controller to define the behavior of switches and routers (the hardware) at the bottom of the SDN Architecture. This is also the basic difference between southbound and northbound interfaces, that is, southbound APIs interact with hardware such as routers and switches of a network. OpenFlow, which was developed by the Open Networking Foundation (ONF), is the first and probably most well-known southbound interface. Due to the prevalence of Open SDN Architecture, other protocols other than OpenFlow can also be defined. This ensures that customers have maximum choice and flexibility while designing and deploying the SDN. [10]

The latter is known as the Northbound API which presents a network abstraction interface to the applications and the management systems at the top of the SDN stack. A Northbound API is one that puts applications in control of the network. Rather than tweaking and adjusting infrastructure repeatedly to get a service running correctly, a framework can be set up that allows the application to demand the infrastructure it needs.

Northbound APIs usually provide a list of vendor related base network functions which are then used to configure user specific infrastructure, and the controller interprets it into a language that each infrastructure node can understand. The primary reason for the existence of Northbound APIs is because external management systems or network applications may wish to extract information control the underlying network and parts of its behaviour. They also expose the universal network abstraction data model and functionality within the controller for use by network applications. They are used to facilitate innovation and have effective orchestration of the network. It is required for the alignment of the network to the needs of different applications and they enable maximum utility of the SDN Architecture.

According to SDX Central, Northbound APIs the most critical APIs in the SDN environment. [11] This is because the value of SDN is tied to the innovative applications it can potentially support and enable. Along with this, the controller should not become a monolithic application. It should interface with separate systems and provide updates about network performance as well as an interface for systems to provide network and security orchestration. These orchestration and management applications realize the true nature of an SDN network. [12] As they are so critical, Northbound APIs that support a wide variety of applications and different types are required. This is also why SDN Northbound APIs are currently the most variant component in a SDN environment. Since the northbound interface is primarily a software ecosystem, the implementation is done commonly through a driver which is at the forefront, while standards emerge later and are driven essentially by wide adoption. [13] It is crucial that the Northbound API be open and standard in order to promote portability and interoperability of applications. Kreutz et al. [1] compares it to a POSIX standard in operating systems, as it represents an abstraction that guarantees programming language and controller independence.

Northbound APIs can enable basic network functions like path computation, loop avoidance, routing and security. Network applications that could be optimized through the northbound interface include load balancers, software defined security services and orchestration applications across cloud resources. [14]

The implementation of Northbound APIs has been done in different ways. The various approaches used to implement it are:

- Define portable low-level application interfaces that make the southbound API look like device drivers.
- Create ad-hoc Northbound API that are specific to a controller. These have their own purpose and specific definitions.
- Translate application requirements into lower level service requests.
- Propose a general control platform based on Linux and abstractions such as the virtual file system.
- Allow network administrators to define module specific allocations and access control policies.

- Use SDN programming languages.

As each network applications have different requirements, a variety of Northbound API have been created to implement the applications. For example, the requirements for security applications are likely to be different to routing applications. This has also made the standardization of Northbound API difficult.

TYPES OF NORTHBOUND API

As mentioned in the previous section, current SDN controllers offer a variety of Northbound APIs. Currently, more than 20 different SDN controllers are available that feature different northbound APIs. [15] These can be broadly divided into three areas namely, RESTful APIs, specialized ad-hoc APIs and programming languages that are now used for northbound interfaces.

A. REST API

A Representational State Transfer (REST) API or an API that is RESTful (that adheres to the constraints of REST) is neither a protocol, language nor an established standard. It is basically constraints that an API must follow in order to be a RESTful API. These constraints were created by Roy Thomas Fielding in his dissertation [16]. They are paraphrased and mentioned here as:

- Starting with the Null Style
  There are two common perspectives on the process of architectural design, whether it be for buildings or for software. The first is that a designer starts with nothing and builds up an architecture from familiar components until it satisfies the needs of the intended system. The second is that a designer starts with the system needs as a whole without any constraints and then incrementally identifies and applies constraints to elements of the system in order to differentiate the design space and allow the forces that influence system behavior to flow naturally, in harmony with the system. The Null style is simply an empty set of constraints. From an architectural perspective, the null style describes a system in which there are no distinguished boundaries between components. It is the starting point for the description of REST.
- Client-Server
  The first constraints added to the hybrid style are those of the client-server architectural style. Separation of concerns is the principle behind the client-server constraints. By separating the user interface concerns from the data storage concerns, the portability of the user interface is improved across multiple platforms and improve scalability by simplifying the server components. Perhaps most significant to the Web, however, is that the separation allows the components to evolve independently, thus supporting the Internet-scale requirement of multiple organizational domains.
- Stateless
  Communication must be stateless in nature such that each request from client to server must contain all of the information necessary to understand the request, and cannot take advantage of any stored context on the server. Session state is therefore kept entirely on the client. This constraint induces the properties of visibility, reliability, and scalability. Visibility is improved because a monitoring system does not have to look beyond a single request in order to determine the full nature of the request. Reliability is improved because it eases the task of recovering from partial failures. Scalability is improved because not having to store state between requests allows the server component to quickly free resources, and further simplifies implementation because the server doesn't have to manage resource usage across requests. Like most architectural choices, the stateless constraint reflects a design trade off. The disadvantage is that it may decrease network performance by increasing the repetitive data sent in a series of requests, since that data cannot be left on the server in a shared context.
- Cache
  In order to improve network efficiency, cache constraints are added to form a client cache stateless server style. Cache constraints require that the data within a response to a request be implicitly or explicitly labeled as cacheable or non-cacheable. If a response is cacheable, then a client cache is given the right to reuse that response data for later, equivalent requests. The advantage of adding cache constraints is that they have the potential to partially or completely eliminate some interactions, improving efficiency, scalability, and user-perceived performance by reducing the average latency of a series of interactions. The trade off, however, is that a cache can decrease reliability if stale data within the cache differs significantly from the data that would have been obtained had the request been sent directly to the server.
- Uniform Interface
  The central feature that distinguishes the REST architectural style from other network-based styles is its emphasis on a uniform interface between components. By applying the software engineering principle of generality to the component interface, the overall system architecture is simplified and the visibility of interactions is improved. Implementations are decoupled from the services they provide, which encourages independent evolvability. The trade off, though, is that a uniform interface degrades efficiency, since information is transferred in a standardized form rather than one which is specific to an application's needs. The REST interface is designed to be efficient for large-grain hypermedia data transfer, optimizing for the

common case of the Web, but resulting in an interface that is not optimal for other forms of architectural interaction.

- Layered System
  In order to further improve behavior for Internet-scale requirements, we add layered system constraints. The layered system style allows an architecture to be composed of hierarchical layers by constraining component behavior such that each component cannot see beyond the immediate layer with which they are interacting. Layers can be used to encapsulate legacy services and to protect new services from legacy clients, simplifying components by moving infrequently used functionality to a shared intermediary. Intermediaries can also be used to improve system scalability by enabling load balancing of services across multiple networks and processors. The primary disadvantage of layered systems is that they add overhead and latency to the processing of data, reducing user-perceived performance. For a network-based system that supports cache constraints, this can be offset by the benefits of shared caching at intermediaries. Placing shared caches at the boundaries of an organizational domain can result in significant performance benefits. Such layers also allow security policies to be enforced on data crossing the organizational boundary, as is required by firewalls.
- Code on Demand
  The final constraint for REST comes from the code on demand style. REST allows client functionality to be extended by downloading and executing code in the form of applets or scripts. This simplifies clients by reducing the number of features required to be pre-implemented. Allowing features to be downloaded after deployment improves system extensibility. However, it also reduces visibility, and thus is only an optional constraint within REST. The notion of an optional constraint may seem like an oxymoron. However, it does have a purpose in the architectural design of a system that encompasses multiple organizational boundaries. It means that the architecture only gains the benefits of the optional constraints when they are known to be in effect for some realm of the overall system.

Adopting REST for the SDN Northbound API has many benefits. [17] These benefits include:

- Decentralized management of dynamic resources
  REST does not use a centralized resource registry but relies on the connections between resources to discover and manage them as a whole. REST allows network elements, such as routers, switches, middle boxes to be dynamically deployed and changed in a distributed fashion.
- Heterogeneous clients
  REST separates resource representations, identification, and interaction, it can adjust resource representations and network protocols based on SDN client capabilities and network conditions to optimize API performance.
- Service composition
  The current trend in SDN is to use programming composition to achieve functional flexibility. For example, Click is a programming language that is used for data plane compositions. REST can provide service oriented compositions that are independent of programming languages and can run on different platforms.
- Localized migration
  The functions of SDN are fast evolving and northbound APIs are likely to change accordingly. REST API supports backward compatible service migration through localized migration by which a newly added resource only affects the resources that connect to it. It can also ease the tension between the new service deployments and backward compatibility.
- Scalability
  REST achieves server scalability by keeping the server stateless and improves server performance through layered caches. Due to this feature, as when an SDN controller needs to support a large number of concurrent host based applications and to use network resources in an efficient way.

However, to realize these benefits and advantages of REST, a set of REST constraints need to be maintained in designing a scalable and service oriented RESTful API. RESTful applications use HTTP requests to post, read and delete data. GET requests are used for read-only queries as they should not change the state of the server and its data. For example, in SDN, an application may use REST APIs to send an HTTP/HTTPS GET message through an SDN controller's IP address. For creating, updating and deleting data, POST requests are used. [18]

SDN Controllers that use RESTful API as their Northbound API include DISCO, Floodlight, HP VAN SDN, OpenContrail, ONOS and sMaRtLight. Of all the different kinds of APIs used to implement Northbound APIs, RESTful API is the one that is predominantly used of late.

B. Programming Languages

Another variety of northbound APIs offered by SDN Controllers are in the form of programming languages. [19] They abstract the inner details of the controller functions and data plane behavior from the application developers. Along with this, programming languages can provide a wide range of powerful abstractions. Most importantly, they provide a variety of basic building blocks to allow software module and application development to be done easily.

Just like computer languages shifted from hardware specific, assembly languages to high level programming languages, network programming languages have shifted from low level languages or instruction sets such as OpenFlow that basically act like forwarding devices to high level programming languages. This shift has also been spurred because traditional approaches for network configuration lead to brittle, low level configuration code that is both lengthy and grows stale as the network evolves. This kind of connectivity does not benefit the network. Moreover, reusability of software, modularity of code were compromised with these languages and their development process was more error prone than others.

Many of the challenges provided by these low level instruction sets are addressed through abstractions provided by the high level programming languages. An example of this is that rules generated for a task should not override the working of another task. SDN programming languages can be compared according to three main design criteria: the level of abstraction of the programming language, the class of language it belongs to, and the type of programmed policies. [20]

In terms of the level of abstraction, it can be compared with respect to low level and high level abstractions. Low level programming languages allow developers to deal with details related to OpenFlow, whereas high level programming languages translate information into a high level semantics. Hence, this allows programmers to focus on network management goals instead of details of low-level rules.

Most of the existing languages adopt the declarative programming paradigm, where what the program should accomplish is focused on while the how it is to be accomplished (the control) is delegated to the implementation. Within this itself, there are two different programming methodologies to express network policies. One is called logic programming where a program comprises a set of logical sentences. The other is called functional reactive programming which provides an effective approach to program reactive systems in a declarative manner. While logic programming is used mostly in the area of artificial intelligence, functional reactive programming is used in areas like robotics and multimedia.

A programming language can be used to develop either passive or active policies. The former can only observe the network state, while the latter can reactively affect the network wide state as a response to certain network events.

The features of programming languages in SDN are:

- Unlike low level languages such as OpenFlow that basically act like forwarding devices, high level programming languages can be used to solve problems at hand, rather than focus on low level details. This can be done by creating higher level abstractions, thereby simplifying the task of programming forwarding devices.
- Programming languages are also accelerating development in the field of SDN, as they provide more problem focused environments for network programmers.
- A big challenge in OpenFlow-based SDNs is the difficulty in ensuring multiple tasks of a single applications. These tasks include routing, access control, monitoring the network and so on. This can be better addressed by programming languages as the foundation of this problem is the generation of conflicting rules that are created by each application.
- Programming language abstractions also provide the ability of writing and creating programs for virtual network topologies. This concept is similar to object oriented programming where abstractions on data and functions can be made. From the point of view of SDN, an example of this would be to create simplified virtual network topologies instead of generating and installing rules in each forwarding device.
- Another feature is the ability to provide specialized abstractions that would cope with other management requirements, like monitoring. This is done by taking advantage of high level query instructions, whose simplicity and power make is easy to implement monitoring applications.
- The portability of programming languages is a feature that doesn't make it necessary for developers to re-implement their applications on different platforms. This is a very important feature with respect to Northbound API as a portable northbound interface will easily allow applications to run on different controllers without any modification.

There are many programming languages now in existence that are used to program SDNs. The most important ones, especially in correlation with the northbound interface are discussed here.

1. Frenetic

   Frenetic [21] is a domain-specific language for programming OpenFlow networks. It introduces a set of purely functional abstractions that enable modular program development, defines high-level, programmer centric, packet processing operators, and eliminates many of the difficulties of the two-tier programming model.

   It is embedded in Python and comprises of two levels of abstraction. They are:

   A limited, but high-level and declarative network query language. The query language provides means for reading the state of the network, merging different queries, and expressing high level predicates for classifying, filtering, transforming, and aggregating the packets' streams traversing the network.

A general-purpose, functional and reactive network policy management library. This library allows reasoning about a unified architecture based on the "see every packet" abstraction of Frenetic and describes network programs without the burden of low-level details. To govern packet forwarding, the functional and reactive based policy management library offers high level packet processing operators that manipulate packets as discrete streams only.
Frenetic introduces three important datatypes for representing, transforming, and consuming streams of values. They are events (that represent discrete, time-varying streams of values), event functions (that can transform events of one type into events of a possibly different type) and a listener (that consumes an events stream and produces a side effect on the controller). Frenetic has been used to implement many services such as load balancing, network topology discovery, fault tolerance routing and it is designed to cooperate with the controller NOX.

2. Procera
Procera [22] is a high level network control language that allows network operators to express reactive network control policies, without having to resort to general purpose programming of the network controller. By applying the principles of functional reactive programming, it offers a declarative, expressive, extensible, and compositional framework. This allows network operators to express realistic network policies that react to dynamic changes in network conditions. These dynamic changes can be originated from SDN's switches or external events such as user authentication, measurements of bandwidth, server load, and so on. This is why Procera has been designed to be reactive. Procera is both expressive and extensible, so users can easily extend the language by adding new constructs.

3. Nettle
By adapting facets of functional reactive programming and the design principles of domain specific languages, Nettle [23] was created to configure networks. It is embedded in the strongly typed language, Haskell. It can be understood as a language that exhibits electrical circuits where it transforms messages issued from switches into commands generated by the controller, by defining signal functions for them. Along with having built in signal functions, Nettle also allows programmers to define their own functions. Using the mechanism of representing message streams as continuous signals that are defined at discrete points of time, it elegantly unifies them. Moreover, it lets programmers to manipulate these continuous quantities to showcase abstract properties of any network. It is considered more appropriate for programming controllers because it is a relatively low level language compared to other languages.

4. NetCore
NetCore [24] is a high level programming language that is a successor to Frenetic, another programming language of SDN. As a significantly more powerful language, it manages the controller-switch interactions by compiling rich policies that are executed in a network. Along with this, it uses wildcard rules to generate sets of packet forwarding rules, and supports the processing of packets using arbitrary functions. The new compilation algorithms in this language, along with a new run time system ensures that packets are processed in an efficient manner on switches, as opposed to the traditional processing on the controller. The NetCore compiler and run time system find a challenge in the switches due its limited computation power. As it is of utmost importance for the switches to process the packets, NetCore divides programs dynamically (after analysing them) into two parts, where one part runs on the switches while another runs on the controller.

5. Pyretic
Another programming language from the creators of Frentic is Pyretic [25]. While Frenetic provided high level abstractions for programming SDNs and managing networks, Pyretic is a Python based platform that allows programmers to design refined applications in SDN. This is because it embodies the many concepts of network management such as dynamic updation of policies to respond to events that occur in a network, specifying packet forwarding policies and so on. It is an open source software with a BSDstyle license. The main goal of Pyretic was to overcome OpenFlow's shortcomings, specifically in terms of its programming nature and its role as a programming interface to switches in the network. It does this by combining multiple policies using one of its policy composition operators instead of compelling manual merging of multiple pieces of application logic. Its policies also support modular programming and it provides facilities to create dynamic policies as well.

6. NetKAT
NetKAT [26] is a network language for SDN programming that has a solid mathematical, semantic foundation. It is similar to NetCore and Pyretic although neither language can provide potentially infinite behaviours. NetKAT applies its complete equational theory and provides syntactic techniques for the same.

Hence, its applications in a network include checking reachability, isolating traffic between programs through non-interference properties that it provides, as well as compilation algorithms correctness. Its design includes a mathematical structure called Kleene algebra with tests (KAT) that provides a solid semantic foundation for NetKAT.

C. Other API

Many existing SDN Controllers propose define their own Northbound API such that they are customized to their specific needs. Onix, MobileFlow, PANE use NVP NBAPI, SDMN API, PANE API as their Northbound APIs respectively. [1]

Two interfaces that were created earlier and require special mention are NOSIX [27] and SFNet [28]. NOSIX is a low level application interface that attempted to represent an abstraction that provides programming language and controller independence. It allows development of low level applications while at the same time acting as a stepping stone for higher level frameworks. As it is designed to be a minimalistic model, it only targets one switch at a time and provides applications without resource constraints. SFNet on the other hand is a high level API that was designed to help with the exploration of software friendly network designs. Through a high level API, it allows applications to interact with lower level service requests and the network itself. It supports network status requests, resource reservation and other high level primitives. In spite of this SFNet has a limited scope.

RESULTS AND DISCUSSION

The southbound API already has a widely accepted proposal called OpenFlow but a common northbound API is a matter of open discussion. While the common consensus is that the Northbound API is very important, but creating a single standard for it is still too early. Hence it is likely that quite a few different northbound APIs will exist before consolidation occurs, not unlike the early days of the mobile operating system (OS) wars.

Unlike the southbound interface, there is no currently accepted standard for northbound interface and they are more likely to be implemented on an ad hoc basis for particular applications. As seen above, while there are several controllers that exist, their application interfaces are still in the early stages and independent. Until a clear northbound interface standard emerges, SDN applications will continue to be developed in an ad hoc fashion and the concept of flexible and portable network applications may have to wait for some time.

Currently the Open Networking Foundation (ONF), non-profit, user-driven organization dedicated to accelerating the adoption of open SDN [29], has been making progress in the standardization efforts of the Northbound API. ONF began its work with the Northbound API in 2012 with the Architecture and Framework Working Group [30]. This group studied the existing northbound API use cases and the controller Northbound APIs. They conceded that although valuable market education is received from every approach taken, there is a confusion of the programmatic interfaces that the vendors must write to in order to serve diverse SDN use cases.

The head of the Architecture and Framework Working Group, Robert Sherwood also believed that standardization of the Northbound API is not necessary, and that a lot of development and experimentation has to be done before reaching that stage. [31] Roy Chua echoes this sentiment in his article [32] by saying that instead of trying to form committees to create a standard API, research teams ought to be encouraged to create APIs that suit the applications and let competing versions compete in the marketplace. He believes that the initial interface needn't be perfect or right on time, and that development and ensuing competition, the API will evolve into a robust and useful Northbound API.

Many other discussions agree with this perspective that standardization at this point is not required. Isabelle GUIS in an article [15] says that implementation of a protocol that is linked with hardware components is a cumbersome task and that standards were created for these beforehand so that there is an agreement that the protocol was stable enough to implement in invest in. Since northbound interfaces are essentially software based, it is the norm that implementation take place first and then standards emerge later. Different companies offer different APIs and in the end more than one could result in a de-facto standard.

Brent Salisbury [33] on the other hand believes that Northbound API requires some coherent coalescence so that applications and what seems like a million dollar industry can begin. Due to the disparity that occurs in Research & Education community, he is not optimistic that a well-defined, open and flexible solution that is not overly complex can be created if left to them. He also believes that vendors essentially want choice to transcend networks and meet their varying application requirements and not answers while being confident that natural selection will occur as it has done in the past and that eventually we will start with many Northbound API and end up with only a few. [34]

Greg Ferro [35] in his article that gives an overview of SDN and concludes by saying that Northbound APIs are unlikely to be standardised and was not aware of any initiative in the area. He feels that Northbound APIs remain completely closed and software in this space is iterating rapidly and new ideas seem to float up every day and this makes it hard to envision if an open standard can be developed. This belief is also reinforced in [36] and [37]

while the former mentions that there is a stark disagreement on whether market forces or vendor committees should decide on the Northbound API.

While this ongoing debate ensues, various competing vendors such as Cisco and Juniper are adapting the concepts of software defined networking into its world of service providers and enterprise buyers. [38] Stacey Higginbotham writes that Cisco plans on deal with the commodification of networking gear by finding its solution in the northbound traffic. By putting in as much software value as it can into its hardware, it allows developers to build applications on top. While ONF views SDN as the decoupling of the switches and the intelligence required to direct packets around a network, she says that Cisco's vision is that if customers can access and program this intelligence, this decoupling is unnecessary. With disparate views such as these, it is not surprising that much headway has not been made in the standardization effort.

Ivan Pepelnjak gives many reasons for the current state of affairs [39] primary of which is that developers usually use standard libraries and APIs that have been created by others and the tight confines of a standard API could affect that. Other reasons for why the standardization effort has not made too much progress may to be allow maximum possible vendor lock in without being restricted by a standard API. Along with this, ONF has concentrated its standardization efforts mainly on OpenFlow and left the Northbound API to individual companies creatively until the market matures and the best API implementations can be selected as the foundation for the standardization effort.

In October 2013, however, the ONF created the Northbound Interface Working Group. This new Working Group will develop information models for northbound interfaces, prototyping and gaining end user feedback on selected examples with real code. The goal is to reduce end-user confusion on the Northbound Interface and to help the application developers actively seeking an open API to develop code against. [40]

They are motivated by the fact that SDN's ultimate promise is realized only when the customers, their applications and orchestration systems are able to make full and effective use of the decoupled network control and data planes without being tied to a single vendor's controller API.

Many of their motivations are mentioned in their Charter Application [41], a few of which are mentioned below:

- An agreed upon northbound interface standard is essential for establishing a vibrant SDN application ecosystem
- Developer investment should be to improve and differentiate their applications, rather than porting between fragmented proprietary APIs
- The northbound interface is an evolving software artefact. Hence, it is imperative that the output of this group be closely aligned with code producing entities that will develop an implementation of APIs before they are standardized.

The working group hence intends to participate in both the definition, and the implementation of the northbound interface in collaboration with open source communities where appropriate.

- SDN use cases such as cloud orchestration, Unified Communications and Collaboration, network virtualization, amongst others, are being deployed today and standardization of resulting APIs can occur in parallel with, and inform the work done for domain agnostic APIs. Hence, is also the right time to demonstrate leadership in defining domain specific APIs.

It refutes the agreement that more development and evolution is required before standardization can be thought of by saying that:

- While SDN is a relatively new paradigm, the argument that it is 'too early' to standardize APIs contradicts the fact that network operators are clearly concerned about the lack of such standards track activity as an obstacle to deployment. - Unless there is an overall architecture for these APIs defined, they will organically grow fragmented by individual interests and domain specific implementations

ONF however concedes that the reason for its hesitance to christen a northbound standard is because there a variety of possible interfaces. [42] In an SDN controller instance, API's are required at different latitudes (or levels) and that access to one or more of these levels may be a requirement for a given application. It also has functional grouping that is, the variety of vertical functions that an interface can serve as shown in the figure below.

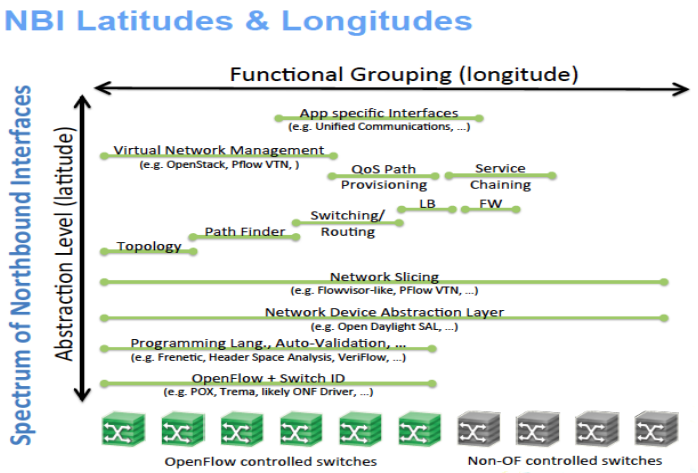

Fig. 2. Northbound Interfaces Latitudes and Longitudes [41]

This diagram, as given in the Charter is based on the studies of the Architecture and Framework Working Group Northbound API Study and is an example of the concept of multiple API latitudes. Hence, the working group will define the specifics of the actual APIs standardized and where it should be placed in the two dimensional API stack. It will also define and develop APIs for domain specific as well as general interaction.

ONF Member Companies such as HP, Huawei, Intel, NEC, Plexxi and Radware have welcomed the Northbound Interface Working Group [40] and since August 2014, the Working Group has opened submissions for northbound interfaces. [43] Even with these developments though, the SDN committee remains apprehensive about standardising the Northbound API. Craig Matsumoto, who attended the OpenDaylight Summit [44] reports that while it is known that applications sit at a variety of "latitudes" north, it is still early to count the number of northbound API types that would be needed. While the path to be taken towards standardization is still unclear, he says that there are three paths that are strongly disavowed. These are:

- Letting northbound entities delve into and program the details of the network.
- Giving business applications unfettered control and power over the network
- Making the business application people specify what the northbound interfaces should be.
- Creating a neutral northbound sandwich layer that all network elements can communicate with using one interface.

Hence, although standardisation efforts have begun, there is still a long way to go before it reaches fruition.

CONCLUSION

In this paper, we have briefly discussed the need for SDN and its architecture and comprehensively explained the Northbound API and the reason for its existence. We also look at the many kinds of Northbound API, focusing on REST API and Programming Languages in SDN. The other Northbound API that are currently in existence are customized to the needs of specific controllers, and are hence many in number.

We then went on to explain the standardization effort made with respect to Northbound API, and whether in fact there is any need for it. Although popular opinion is that it is too early for standardization to occur for the Northbound API, the ONF has created a group in order to do so. However, this group seems to tackle the standardization effort in a different manner, and we have concluded the paper by presenting that manner.

The Northbound API is a field of SDN that has a lot of potential in research and we believe that it will decisively shape the way applications interact with the network.

ACKNOWLEDGEMENTS

This section should be typed in character size 10pt Times New Roman, Justified.

REFERENCES

[1] Diego Kreutz, Fernando M. V. Ramos, Paulo Verissimo, Christian Esteve Rothenberg, Siamak Azodolmolky, Steve Uhlig, "Software Defined Networking: A Comprehensive Survey", October 2014

[2] H. Farhady et al., "Software-Defined Networking: A survey", Comput. Netw. (2015), http://dx.doi.org/10.1016/j.comnet.2015.02.014

[3] Open Networking Foundation, "SDN Architecture Overview Version 1.0", https://www.opennetworking.org/images/stories/downloads/sdn-resources/technical-reports/SDN-architecture-overview-1.0.pdf

[4] Open Networking Foundation, "Software-Defined Networks (SDN) Definition", https://www.opennetworking.org/sdn-resources/sdn-definition

[5] K. Greene, "MIT Tech Review 10 Breakthrough Technologies: Software-defined Networking", http://www2.technologyreview.com/article/412194/tr10-software-defined-networking/, 2009.

[6] Open Networking Foundation, "OpenFlow", https://www.opennetworking.org/sdn-resources/openflow
[7] Fishnet Security, "SDN APIs: A New Vocabulary for Network Engineers", https://www.fishnetsecurity.com/6labs/blog/sdn-apis-new-vocabulary-network-engineers
[8] Matt Oswalt, "SDN and Programming", http://keepingitclassless.net/2013/09/sdn-and-programming/
[9] SDX Central, "What is SDN Southbound API?", https://www.sdxcentral.com/resources/sdn/south-bound-interfaces-api/
[10] Big Switch Networks, "Open SDN Architecture", http://www.bigswitch.com/sites/default/files/sdn_overview.pdf
[11] SDX Central, "What is SDN Northbound API?", https://www.sdxcentral.com/resources/sdn/north-bound-interfaces-api/
[12] Jody Brazil, "The Northbound API is key to OpenFlow's success", https://www.sdxcentral.com/articles/contributed/the-northbound-api-is-the-key-to-openflows-success/2012/11/
[13] Isabelle GUIS, "The SDN Gold Rush To The Northbound API," November 2012, http://www.sdncentral.com/technology/thesdn-gold-rush-to-the-northbound-api/2012/11/
[14] Azodolmolky, Siamak (2013-10-25), "Software Defined Networking with OpenFlow", Packt Publishing.
[15] Sally Johnson, "A primer on northbound APIs: Their role in a software defined network", http://searchsdn.techtarget.com/feature/A-primer-on-northbound-APIs-Their-role-in-a-software-defined-network
[16] Roy Thomas Fielding, "Chapter 5. Representational State Transfer (REST)", 2000, https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm
[17] Wei Zhou, Li Li, Min Luo, Wu Chou, "REST API Design Patterns for SDN Northbound API", 2014 28th International Conference on Advanced Information Networking and Applications Workshops, 2014
[18] Darien Hirotsu, "REST APIs in SDN: An introduction for network engineers", January 2015, http://searchsdn.techtarget.com/tip/REST-APIs-in-SDN-An-introduction-for-network-engineers
[19] Sakir Sezer, Sandra Scott-Hayward, Pushpinder Kaur Chouhan, Barbara Fraser David Lake, Jim Finnegan, Niel Viljoen, Netronome Marc Miller, Navneet Rao, "Are we Ready for SDN? Implementation Challenges for Software Defined Networks", July 2013, IEEE Communications Magazine
[20] Y. Jarraya, T. Madi, and M. Debbabi, "A survey and a layered taxonomy of software-defined networking," Communications Surveys Tutorials, IEEE, vol. PP, no. 99, pp. 1–1, 2014.
[21] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker, "Frenetic: a network programming language," SIGPLAN Not., 2011.
[22] A. Voellmy, H. Kim,and N. Feamster, "Procera: a language for highlevel reactive network control," in Proceedings of the first workshop on Hot topics in software defined networks, ser. HotSDN '12. New York, NY, USA: ACM, 2012, pp. 43–48.
[23] A. Voellmy and P. Hudak, "Nettle: taking the sting out of programming network routers," in Proceedings of the 13th international conference on Practical aspects of declarative languages, ser. PADL'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 235–249.
[24] C. Monsanto, N. Foster, R. Harrison, and D. Walker, "A compiler and run-time system for network programming languages," SIGPLAN Not., vol. 47, no. 1, pp. 217–230, Jan. 2012.
[25] J. Reich, C. Monsanto, N. Foster, J. Rexford, and D. Walker, "Modular SDN Programming with Pyretic," USENIX ;login, vol. 38, no. 5, October 2013.
[26] C. J. Anderson, N. Foster, A. Guha, J.-B. Jeannin, D. Kozen, C. Schlesinger, and D. Walker, "NetKAT: Semantic foundations for networks," SIGPLAN Not., vol. 49, no. 1, pp. 113–126, Jan. 2014.
[27] M. Yu, A. Wundsam, and M. Raju, "NOSIX: A lightweight portability layer for the SDN OS,"SIGCOMM Comput. Commun. Rev., vol. 44, no. 2, pp. 28–35, Apr. 2014.
[28] K.-K. Yap, T.-Y. Huang, B. Dodson, M. S. Lam, and N. McKeown, "Towards software-friendly networks," in Proceedings of the first ACM asia-pacific workshop on Workshop on systems, ser. APSys '10. New York, NY, USA: ACM, 2010, pp. 49–54.
[29] Open Networking Foundation, "What is ONF?", https://www.opennetworking.org/images/stories/downloads/about/onf-what-why.pdf
[30] Open Networking Foundation, "Charter: Architecture and Framework Working Group", https://www.opennetworking.org/images/stories/downloads/working-groups/charter-architecture-framework.pdf
[31] John Dix, "Clarifying the role of software defined enwtroking northbound APIs", May 2013, http://www.networkworld.com/article/2165901/lan-wan/clarifying-the-role-of-software-defined-networking-northbound-apis.html
[32] Roy Chua, "OpenFlow Northbound API-A New Olympic Sport", July 2012, https://www.sdxcentral.com/articles/editorial/openflow-northbound-api-olympics/2012/07/
[33] Brent Salisbury, "The Northbound API-A Big Little Problem", June 2012, http://networkstatic.net/the-northbound-api-2/
[34] Brent Salibsury, "SDN APIs-I Think I Have Seen This Movie Before", July 2012, http://networkstatic.net/sdn-apis-this-all-feels-pretty-familair/
[35] Greg Ferro, "Northbound API, Southbound API, East/North-LAN Navigation in an OpenFlow World and an SDN compass", August 2012, http://etherealmind.com/northbound-api-southbound-api-eastnorth-lan-navigation-in-an-openflow-world-and-an-sdn-compass/

[36] B. Casemore, "Northbound API: The standardization debate," September 2012, http://nerdtwilight.wordpress.com/2012/09/18/northbound-api-the-standardization-debate/
[37] David Lenrow, "A change is blowing in from the Northbound API", April 2012, https://www.sdxcentral.com/articles/contributed/a-change-is-blowing-in-from-the-north-bound-api/2012/04/
[38] Stacey Higginbotham, "For Cisco's SDN strategy look North", June 2012, https://gigaom.com/2012/06/13/for-ciscos-sdn-strategy-look-north/
[39] Ivan Pepelnjak, "SDN Controller Northbound API is the Crucial Missing Piece", September 2012, http://blog.ipspace.net/2012/09/sdn-controller-northbound-api-is.html
[40] Open Networking Foundation, "Press Releases: Open Networking Foundation announces Northbound Interfaces Working Group", https://www.opennetworking.org/news-and-events/press-releases/1182-open-networking-foundation-introduces-northbound-interface-working-group
[41] Open Networking Foundation, "Northbound Interfaces Working Group", https://www.opennetworking.org/images/stories/downloads/working-groups/charter-nbi.pdf
[42] Craig Matsumoto, "ONF will tackle SDN's Northbound Interface", October 2013, https://www.sdxcentral.com/articles/news/onf-decides-tackle-sdns-northbound-interface/2013/10/
[43] Open Networking Foundation, "NBI Submissions", August 2014, http://www.onfsdninterfaces.org/index.php/onf-nbi-leadership-roundtable-materials/7-nbi-submissions
[44] Craig Matsumoto, "Stalking SDN' Elusive Northbound Interface", February 2014, https://www.sdxcentral.com/articles/news/stalking-sdns-elusive-northbound-interface/2014/02/