SN Computer Science (2022) 3:502
https://doi.org/10.1007/s42979-022-01423-3

ORIGINAL RESEARCH

Designing a RESTful Northbound Interface for Incompatible Software Defined Network Controllers

Abdullah Alghamdi[1] · **David Paul**[1] · **Edmund Sadgrove**[1]

Received: 17 March 2022 / Accepted: 16 September 2022 / Published online: 1 October 2022

Abstract
In Software Defined Networking, applications often need to communicate with network controllers to query or modify the current state of the network. However, there is currently no standard northbound interface that allows this communication to occur. Instead, each Software Defined Networking implementation defines its own interface, meaning applications typically need to be modified to allow them to work with different controllers. In this paper we present a high-level design for a REST-like reactive northbound interface which would allow applications to be written once and then work with multiple otherwise-incompatible controllers.

Keywords Software defined networking · Northbound interface · RESTful

Introduction

Traditional methods for managing computer networks do not allow enough flexibility to support modern network requirements. With more devices connected to larger networks, configuration errors are widespread, but often difficult to resolve [1]. Further, security issues are more recognised and can require special attention, placing significant burden on network administrators [2].

Software Defined Networking (SDN) provides a solution by allowing computer networks to be designed and controlled programmatically. Administrators can centrally control networks with software. SDNs are relatively simple to implement, do not incur a great cost over traditional networks, and allow innovation through the design of new network applications [3]. Support from multiple vendors, including Cisco, Google, and HP, further cements SDN as the future for computer networks [4].

For applications on an SDN to be able to program the network, it is necessary for them to be able to access and potentially modify the current state of the network. Since network applications can have vastly different functionalities (e.g. from traffic management to improvements in security), the method to achieve this must be flexible. SDNs allow this by having forwarding devices interact with a (logically) centralised controller.

This centralised controller, which effectively separates control functions from the forwarding devices, sends instructions that specify how to process incoming packets to forwarding devices, via an API called the Southbound Interface (SBI). The Open Network Foundation (ONF) [5], which offers open standards to help popularise SDN technology, presents OpenFlow [6] as the standard SBI.

However, for SDN to reach its full potential, network applications require the ability to communicate directly with the SDN controller as well. They can do this via an SDN network's Northbound Interface (NBI). The NBI allows applications to both query the current state of the network, and to modify its behaviour by passing commands to be applied over the forwarding devices. SDN applications can either react solely to events that occur on the network (internal applications), or proactively interact with the network, potentially based on external events (external applications).

This article is part of the topical collection "Web Information Systems and Technologies 2021" guest edited by Joaquim Filipe, Francisco Domínguez Mayo and Massimo Marchior.

✉ David Paul
dpaul4@une.edu.au

Abdullah Alghamdi
aalgham9@myune.edu.au

Edmund Sadgrove
esadgro2@une.edu.au

1 School of Science and Technology, University of New England, Armidale, NSW 2351, Australia

Whilst the NBI is vital to obtain all the advantages SDN offers, there is no standard NBI that can be used by different SDN implementations [7]. Instead, different SDN controllers implement their own interface, meaning applications written for one SDN controller typically require redevelopment to use on a different controller. This porting of applications to different controllers can be expensive in both time and resources.

In our previous work [8], we have described some requirements necessary to design and implement an open RESTful NBI for SDN implementations to allow network applications to be written once and used with multiple controllers. This paper extends our previous work by describing a high-level design for a system that meets these requirements.

The motivation for our paper is to contribute to efforts to standardise the NBI of SDNs to improve both portability of applications and interoperability of controllers. The remainder of this paper is structured as follows. Section "Background Information" describes SDN, and especially the NBI, in more detail, culminating with the requirements we determined necessary for the development of an open RESTful interface between network applications and SDN controllers. Section "Design of a Reactive/Proactive NBI" then details how we plan to develop such a system, and the design is examined in section "Discussion". Section "Conclusion" then concludes the paper and presents ideas for future work.

Background Information

Software Defined Networking

SDN is a programmable control network approach designed to overcome limitations of traditional networks, such as complex network management and lack of scalability [9]. In SDN, network control functions are moved from the data plane to software called a controller. This splits control tasks from forwarding devices to ease network management, offer flexibility in packet routing, and centralise configuration of the data layer. The controller manages the higher control plane to determine how to route packets and sends appropriate instructions to the forwarding devices. The forwarding devices in the data plane then simply forward packets based on rules specified by the network controller. Network nodes can communicate with other nodes via the control plane, using protocols, such as BGP [10], OSPF [11], or MPLS [12].

Figure 1 provides a high-level overview of a typical SDN structure. The Data Layer and Control Layer communicate via the SBI, typically using OpenFlow [13]. The NBI is used to allow Applications and the Control Layer to communicate, though the exact protocol used depends on the SDN implementation [14].

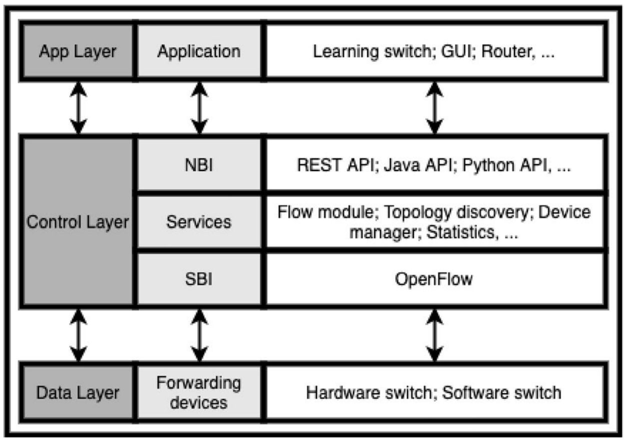
App Layer
Application
Learning switch; GUI; Router, ...
Control Layer
NBI
REST API; Java API; Python API, ...
Services
Flow module; Topology discovery; Device manager; Statistics, ...
SBI
OpenFlow
Data Layer
Forwarding devices
Hardware switch; Software switch

Fig. 1 SDN Architecture (from [8])

When a packet arrives at a forwarding device, the device consults its rules (called flows), along with information from the packet (e.g. the packet's header) to determine how the packet should be handled. If the device does not know how to handle a packet, it communicates with the control layer to determine what it should do. These forwarding devices can be traditional hardware switches, but are often software switches, such as Open vSwitch [15].

In charge of the Control Layer is a centralised network operating system known as the controller. The controller uses the SBI (typically implemented using OpenFlow) to provide forwarding devices with instructions on whether they should modify, drop, or forward packets. Forwarding devices drop packets by default, unless there is an instruction that specifies otherwise. The SBI is also used to provide statistical information, and to notify of packet arrivals or any change in status (e.g. if a new forwarding device is attached to the network).

The core services of a controller typically include:

- Topology service: maintains a network topology graph, using forwarding devices to track changes in the network structure
- Inventory service: Records basic information about SDN resources connected to the network
- Host tracking service: Discovers IP and MAC addresses of hosts connected to the network
- Statistics service: Uses counter information in switches to provide network statistics

To allow Applications to access these controller services, an NBI is used. Unfortunately, there is no standard protocol for the NBI, with each SDN implementation using its own incompatible interface [16]. For example, some controllers only provide an interface in their implementation language (e.g. Java). In any case, the interface typically only works with one particular controller [17]. Table 1 lists incompatible SDN controllers with links to their northbound interfaces, none of which are compatible.

Regardless of the implementation, the NBI provides an abstracted view of the network to any applications. The NBI often represents the entire network as if it were implemented with a single large switch, rather than matching the actual physical layout of the network, though this abstracted view can be probed to discover details of all existing switches, ports, links, hosts, flows, actions, and data models used in the network.

The NBI is used by the controller to notify applications of events in the network. This could be a major event, such as a change in the network's topology, or something more minor, such as the controller receiving a packet. Applications can also call functions on the controller to modify details of the network, such as whether packets should be accepted or rejected.

SDN Applications

Applications in an SDN network can be classed as either internal (or reactive), or external (or proactive). Internal applications respond when a packet is received by a forwarding device and the forwarding device passes it to the controller because it does not know what to do with it. The internal application can then notify the controller how such packets should be handled in the future, and the controller enforces this new policy over its forwarding devices. External applications are not required to wait for an event (such as the arrival of a packet) before contacting the controller to modify the network's policy and can do so at any time.

One main difference between internal and external applications is that internal applications often create new network resources that can be utilised by other applications. This means that internal applications become part of the programmable network, whereas external applications can only be controlled from the outside. For example, an internal load balancer may allow other applications to query or modify its behaviour through the NBI.

The NBI can be broken up into two main components: the Response API and the Listener API. The Response API allows applications to use the controller to modify the network. The Listener API allows applications to register event listeners that get notified when a relevant event occurs (e.g. a relevant packet arrives at the controller). Figure 2 shows the generic design of an internal application which processes packets it receives through the Listener API and then uses the Response API to modify the network based on the packet contents. External applications use the Response API based on external triggering, rather than on the contents of packets received through the Listener API.

The Response API is often implemented as a RESTful [8] interface because REST offers the following advantages:

- Simplicity: Simple HTTP methods can be used by most programming languages.
- Flexibility: All resources are represented as URIs and accessed in the same way.
- Extensibility: New resources can be added by registering an appropriate URI.
- Security: HTTPS can be used to ensure communications are secure.

The Listener API, on the other hand, is typically provided as a native API on the controller [18]. One of the main reasons for this is that the Listener API is required to support asynchronous notifications, whilst RESTful APIs only support request-response interactions [17]. This means that the Listener API for different controllers often use different languages and technologies, making them incompatible.

Further, even the Response API is not standardised [17]. Thus, applications written for one controller cannot typically be used by another controller without modification, even for external applications. In some cases, SDN programming languages can be used to allow applications to be used by a few different controllers [19], but this approach is not scalable. Instead, we argue that the time is now right for a standard NBI to be developed [8].

Standardising a NBI is difficult because different applications may have very different requirements. Because of this, many different NBIs have been proposed [7], though they

Table 1 Incompatible NBIs for different SDN controllers

SDN Controller	Northbound interface link(s)
ONOS	https://api.onosproject.org/ https://wiki.onosproject.org/display/ONOS/Appendix+B%3A+REST-APIs
OpenDaylight	https://docs.opendaylight.org/en/stable-sulfur/javadoc.html https://docs.opendaylight.org/projects/netvirt/en/latest/specs/fluorine/ovs_based_na_responder_for_gw.html?highlight=REST%20API#rest-api
Floodlight	https://floodlight.atlassian.net/wiki/spaces/floodlightcontroller/pages/1343539/Floodlight+REST+API#FloodlightRESTAPI-FloodlightRESTAPI
RYA	https://ryu.readthedocs.io/en/latest/app/ofctl_rest.html

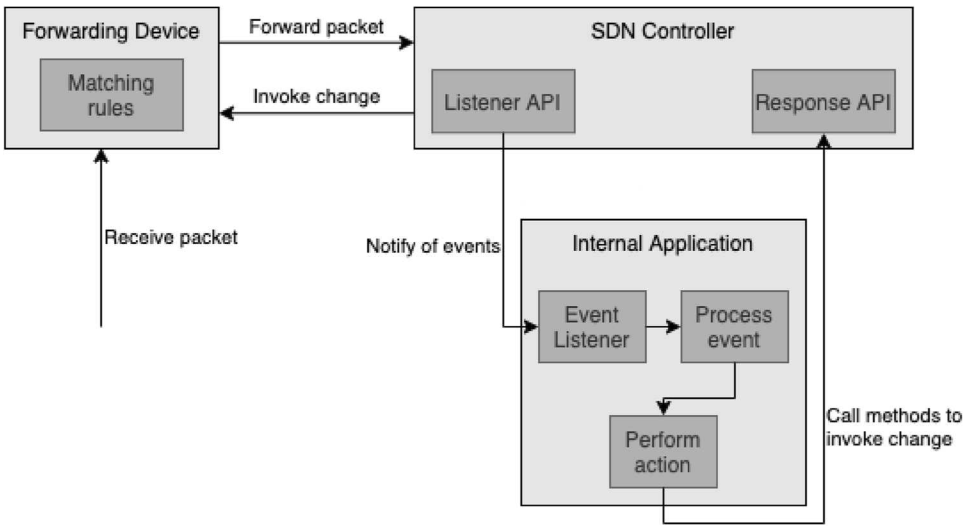

Fig. 2 Generic design of an internal SDN application (from [8], extended from [15])

often only cover a limited set of operations or technologies. However, studying many of the available NBIs has allowed us to determine a set of requirements for both the Response and Listener APIs that make up an NBI.

NBI API Requirements

To develop an open, flexible, and independent NBI API, we believe a REST-like interface is ideal. This will ensure interoperability and portability of applications using a lightweight standard interface that could be used in practically any implementation. The design must follow the ONF guidelines [20] and cover the wide range of use cases necessary for an NBI. This section summarises the requirements of such an API from our previous work [8].

The Response API component should be implemented as a RESTful service on the controller. This will allow the development of a stable, extensible API that can be implemented by multiple controllers. The Response API must support the entire SDN application lifecycle, meaning its available actions should include at least the following:

- Reading actions:
  - o Read topology
  - o Read statistics
  - o Read flows
  - o Read controller information
  - o Read incoming packet
- Writing actions:
  - o Insert flow
  - o Modify flow
  - o Delete flow
  - o Forward packet
  - o Set priority

The Listener API component of the NBI, which allows notification of events, should be flexible and extendable, with support for at least the following events:

- Packet arrived
- Flow added
- Flow removed
- New device added
- Device removed

Since the Listener API is required to support asynchronous notification of events, it cannot be a RESTful service (since RESTful services only support request/response). However, we propose that a REST-like service be implemented to allow listeners to register with a controller. That is, a typical REST-like request should be sent by an application to register a listener, and, rather than being stateless, the controller should remember the listeners that are registered to it for certain events. Then, when a relevant event occurs, the controller should call a RESTful service implemented by the application, which can use a typical request/response pattern to reply to the controller.

Whilst the overall NBI design should be extensible, consideration of existing implementations suggests the following resources should be supported as a minimum:

- Hosts
- Switches
- Applications
- Messages
- Network topology
- Statistics

- Events

Other resources may be required in particular implementations, which is why extensibility of the NBI API is important. Further, the exact details of each resource might differ between implementations, meaning that flexibility is imperative.

One of the advantages of this approach is that is that a shim implementation of the proposed NBI API could be implemented for any controller that supports the minimum functionality required. Applications could then interact with the controller via this shim implementation, which converts requests from the REST-like service defined here to the internal implementation of the controller. This would allow backwards compatibility, which will be especially useful when evaluating the new NBI API we are proposing. Further, it eases the process of extending support for the new API to new controllers or applications.

Design of a Reactive NBI

Whilst there is currently no standard NBI, OpenFlow is considered the standard SBI [13]. To maximise compatibility, we believe that a standard NBI should be developed using OpenFlow as its base, mainly due to its standardisation as the SBI: whilst some changes will be required for the new functionality, utilising similar RESTful interfaces to the SBI should ensure familiarity for SDN practitioners. We have specified that an NBI should consist of both a Response API and a Listener API. In this work, we will concentrate on the Listener API.

We recommend that a REST-like Listener API be implemented in each internal SDN application. Each Listener (i.e. internal application) receives a notification from the SDN controller when a particular event occurs. This will require controllers to expose a simple registration interface and to remember registered listeners so the controller can use their RESTful interfaces to notify whenever a relevant event occurs.

Our design of the Listener API component of the NBI consists of three modules: the Listener, the Packet Processor, and the Flow Manager.

Listener

The Listener module listens for particular events on the network. For this purpose, it implements the following submodules:

- Message listener: A listener that is notified whenever a packet meeting certain criteria arrives at the controller, or if a flow is modified or expires.
- Switch listener: A listener that is notified whenever a forwarding device joins or leaves a network, a port configuration of a switch has changed, or a new inter-link between forwarding devices is discovered.
- Host listener: A listener that is notified whenever a host joins or leaves a network, or changes its information.

Packet Processor

The Packet Processor module is required to determine how packets that arrive at the internal application should be handled. The packet processor uses the header portion of an incoming packet to determine whether a packet should be dropped, forwarded, or passed to the Flow Manager to programme a new flow. At the very least, all unmatched packets that the application does not wish to handle should be returned to the controller.

Flow Manager

The Flow Manager interacts with the controller to create a new flow entry, which the controller can then pass on to forwarding devices. When passed a packet from the Packet Processor, it determines the new flow that is required and utilises the controller's RESTful interface to distribute it through to forwarding devices.

Discussion

The most common event that causes an application to request the controller take action is the arrival of a packet, which typically leads to one of the following actions [15]:

• Packet-specific actions: instructing forwarding devices to delete, flood, or forward the packet to a certain port.

• Flow-specific actions: programming a new flow entry and installing it in a forwarding device to allow the device to handle such packets locally without contacting the controller.

To improve compatibility with OpenFlow, we propose that the Message listener submodule of the Listener module be implemented using the OpenFlow Message Listener [21], which is used to forward incoming packets to other modules. The controller could then forward necessary packets to registered applications to determine whether the packet should be forwarded, dropped, or cause a new flow to be programmed into the network. The Switch Listener and Host Listener could be based on a similar interface, borrowing from OpenFlow whenever possible, to maintain familiarity over the API.

The Listener module is the component of an internal SDN application that receives notifications from the controller. The Packet Processor and Flow Manager, on the other hand, respond to these notifications through the controller's

Response API. We again suggest that OpenFlow be used wherever possible in these components, since it is already the standard for SBIs. For example, since OpenFlow is used to communicate Flows from the controller to forwarding devices, a similar (though slightly more abstract) representation could be used between the Flow Manager of the application and the controller.

Conclusion

SDN separates the control and data layers of networks, allowing greater flexibility and control by making a network programmable. However, whilst SDN controllers use OpenFlow as the standard southbound interface, there is currently no standard northbound interface (NBI).

Since the NBI is used to allow applications to communicate with controllers, this means that applications are typically not compatible between different controller implementations. Whilst partial solutions, such as SDN programming languages, can help here, a well-defined open and extendable NBI is required. Our goal is to provide such an NBI. This interface will increase portability of applications between SDN implementations and help solve current incompatibility issues.

Using lessons learned from existing NBI implementations, and building on OpenFlow, we believe the correct solution is a RESTful Response API implemented on SDN controllers to allow reading and writing actions over the network, and a REST-like Listener API that allows applications to register with controllers and be notified of certain network events.

Each application that requires notification of events could then implement a RESTful interface that the controller would use to provide these notifications. The responses to these notifications could specify any actions required by the controller (e.g. dropping a packet or inserting a new flow). Borrowing representations from OpenFlow wherever possible (perhaps with some modifications/abstractions) would ensure familiarity with people already comfortable with OpenFlow.

Once these RESTful interfaces have been defined fully, the next step will be to implement shim implementations that utilise two existing, but currently incompatible, SDN controllers and have them both interact with a single implementation of an SDN application. This proof of concept could then be extended to further controllers and existing SDN applications could even be ported to this new interface by placing a translator component between the controller using this new interface and the application.

SDN applications are often tied to a specific SDN controller and porting these applications between different implementations is complex and time-consuming. The main advantage of our proposal is to improve this compatibility, so developers can create an application once and deploy it to any SDN controller. The disadvantage of our proposal is due to the existing incompatible implementations of NBIs in SDN. To overcome this, we recommend the development of a conversion layer to translate from our open and extendable NBI to the closed proprietary interfaces of existing controllers.

Funding Open Access funding enabled and organized by CAUL and its Member Institutions. Abdullah Alghamdi's PhD study is supported by the Saudi Arabia Cultural Mission.

Declarations

Conflict of Interest On behalf of all authors, the corresponding author states that there is no conflict of interest.

References

1. Al Shuhaimi F, Jose M, Singh AV. Software defined network as solution to overcome security challenges in IoT. in 2016 5th International Conference on Reliability, Infocom Technologies and Optimization (Trends and Future Directions)(ICRITO). 2016. IEEE. https://doi.org/10.1109/icrito.2016.7785005.
2. Akcay H, Derya Y-K. Web-based user interface for the floodlight SDN controller. Int J Adv Netw Appl. 2017;8(5):3175–80.
3. Jain S, et al. B4: Experience with a globally-deployed software defined WAN. ACM SIGCOMM Comput Commun Rev. 2013;43(4):3–14. https://doi.org/10.1145/2486001.2486019.
4. Shahid A, Fiaidhi J, Mohammed S. Implementing innovative routing using software defined networking (SDN). Int J Multimedia Ubiquitous Eng. 2016;11(2):159–172. https://doi.org/10.14257/ijmue.2016.11.2.17.
5. Open Networking Foundation. 2021; Available from: https://opennetworking.org/.
6. McKeown N, et al. OpenFlow: enabling innovation in campus networks. ACM SIGCOMM Comput Commun Rev. 2008;38(2):69–74. https://doi.org/10.1145/1355734.1355746.
7. Tijare P, Vasudevan D. The northbound APIs of software defined networks. Int J Eng Sci Res Technol. 2016;5(10):501–13.
8. Alghamdi A, Paul D, Sadgrove E. A RESTful northbound interface for applications in software defined networks. in Proceedings of the 17th International Conference on Web Information Systems

& Technologies. 2021. SciTePress. https://doi.org/10.5220/0010713300003058.
9. Haji SH, et al. Comparison of software defined networking with traditional networking. Asian J Res Comput Sci. 2021;1–18.
10. Rekhter Y, Li T, Hares S. A border gateway protocol 4 (BGP-4). 1994, ISI, USC Information Sciences Institute. https://doi.org/10.17487/rfc1654.
11. Baker F, Coltun R. OSPF version 2 management information base. 1991, RFC 1253, ACC, Computer Science Center. https://doi.org/10.17487/rfc1252.
12. Rosen E, Viswanathan A, Callon R. Multiprotocol label switching architecture. 2001. https://doi.org/10.17487/rfc3031.
13. Braun W, Menth M. Software-defined networking using OpenFlow: Protocols, applications and architectural design choices. Future Internet. 2014;6(2):302–36. https://doi.org/10.3390/fi6020302.
14. Latif Z, et al. A comprehensive survey of interface protocols for software defined networks. J Netw Comput Appl. 2020;156:102563. https://doi.org/10.1016/j.jnca.2020.102563.
15. Kreutz D, et al. Software-defined networking: a comprehensive survey. Proc IEEE. 2014;103(1):14–76. https://doi.org/10.1109/jproc.2014.2371999.
16. Shin MK, Nam KH, Kim HJ. Software-defined networking (SDN): A reference architecture and open APIs. in 2012 International Conference on ICT Convergence (ICTC). 2012. IEEE. https://doi.org/10.1109/ictc.2012.6386859.
17. Goransson P, Black C, Culver T. Software defined networks: a comprehensive approach. 2016: Morgan Kaufmann.
18. Banse C, Rangarajan S. A secure northbound interface for SDN applications. in 2015 IEEE Trustcom/BigDataSE/ISPA. 2015. IEEE. https://doi.org/10.1109/trustcom.2015.454.
19. Zhou W, et al. REST API design patterns for SDN northbound API. in 2014 28th international conference on advanced information networking and applications workshops. 2014. IEEE. https://doi.org/10.1109/waina.2014.153.
20. Janz C et al. Intent nbi–definition and principles. Open Networking Foundation, Version, 2015;2.
21. Hoang DB, Pham M. On software-defined networking and the design of SDN controllers. in 2015 6th International Conference on the Network of the Future (NOF). 2015. IEEE. https://doi.org/10.1109/nof.2015.7333307.