2014 28th International Conference on Advanced Information Networking and Applications Workshops

REST API Design Patterns for SDN Northbound API

Wei Zhou, Li Li, Min Luo, Wu Chou
Shannon (IT) Laboratory
Huawei Technologies Co., Ltd.
{sky.zhouwei, li.nj.li, min.ch.luo, wu.chou}@huawei.com

***Abstract*—REST architectural style gains increasing popularity in the networking protocol design, and it has become a prevalent choice for northbound API of Software-Defined Networking (SDN). This paper addresses many critical issues in RESTful networking protocol design, and presents a framework on how a networking protocol can be designed in a truly RESTful manner, making it towards a service oriented data networking. In particular, we introduce the HTTP content negotiation mechanism which allows clients to select different representation formats from the same resource URI. Most importantly, we present a hypertext-driven approach, so that hypertext links are defined between REST resources for the networking protocol to guide clients to identify the right resources rather than relying on fixed resource URIs. The advantages of our approach are verified in two folds. First, we show how to apply our approach to fix REST design problems in some existing northbound networking APIs, and then we show how to design a RESTful northbound API of SDN in the context of OpenStack. We implemented our proposed approach in the northbound REST API of SOX, a generalized SDN controller, and the benefits of the proposed approach are experimentally verified.**

Keywords—SDN, Controller, Northbound API, OpenStack, Quantum, REST API, Hypertext Driven

I. INTRODUCTION

Software-Defined Network (SDN) decouples the data and control planes, in which a logically centralized controller controls the network behaviors based on global network information across various networking elements. As shown in Fig. 1, at the center of SDN is an SDN controller, which controls the behaviors of underlying data forwarding elements through some southbound APIs, e.g. OpenFlow [12]. On the other hand, the controller, either implemented in a centralized or distributed manner, also provides an abstraction of the network functions with a programmable interface for applications to consume the network services and configure the network dynamically. This interface is called the northbound API of SDN.

In SDN, the data plane and the control plane are typically connected by a closed control loop: 1) the control plane receives network events from the data plane; 2) the control plane (the SDN controller and applications) computes some network operations based on the events for the data plane; and 3) the data plane executes the operations which can change the network states, e.g. data path. The role of SDN northbound API is to provide a high-level API between the controller and the applications to facilitate step 2 in the control loop.

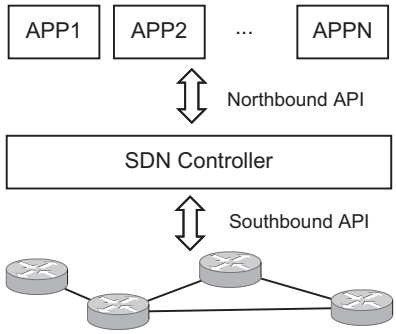

Fig. 1. The architecture of Software-Defined Network.

REST is an architecture style for designing networked applications. As REST architectural style has gained more popularity in implementing loosely-coupled systems, RESTful services are becoming the style of choice for northbound API and gaining increasingly importance in SDN architecture. Adopting REST for the SDN northbound API within this control architecture has the following benefits:

1) Decentralized management of dynamic resources: REST does not use any centralized resource registry but relies on connections between resources to discover and manage them as a whole. REST allows network elements, such as routers, switches, middle boxes (e.g. NAT and DPI devices), to be dynamically deployed and changed in a distributed fashion.
2) Heterogeneous clients: because 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.
3) Service composition: the current trend in SDN is to use programming composition to achieve functional flexibility, such as Click [19] for data plane compositions and Pyretic [20] for control plane compositions. REST can provide service-oriented compositions that are independent of programming languages and can run on different platforms.
4) Localized migration: since the functions of SDN are fast evolving, the northbound APIs of SDN controllers will 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. Combined with uniform

978-1-4799-2652-7/14 $31.00 © 2014 IEEE
DOI 10.1109/WAINA.2014.153

interface and hypertext-driven service discovery, it can ease the tension between the new service deployments and backward compatibility.

5) Scalability: REST achieves server scalability by keeping the server stateless and improves server performance through layered caches. This feature will become useful, when an SDN controller needs to support a large number of concurrent host-based applications and to use network resources in an efficient way.

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. One of the grounding principles of REST is "hypertext as the engine of application state" [1], and a REST API should be driven by nothing but hypertext. This constraint is often ignored by some REST API designs using some out-of-band mechanisms. Such design inevitably creates fixed resource names, types, and hierarchies that violate the REST design principles prescribed by Roy Fielding [2]. Violations of REST design principles result in APIs that may not be as scalable, extensible, and interoperable as promised by REST.

This paper presents a REST Chart model based approach for designing a scalable and extensible RESTful networking protocol and in particular, the northbound API of SDN. The proposed approach addresses many critical networking protocol design issues observed in current SDN controller applications, and provides a generic framework on how the RESTful networking protocols, including northbound API of SDN, can be designed in a truly RESTful manner. Our main contributions of this paper are as follows:

1) A theoretical model and a design framework for RESTful networking protocol design based on the approach of the REST Chart model. In particular, we extended the framework of the REST Chart model to a new structure - Hierarchical REST Chart based on a special kind of Hierarchical Petri-Net.
2) A loosely coupled and resource oriented RESTful networking protocol design pattern that supports resource discovery, late binding, filtering, pattern matching and search. The proposed design pattern can provide the networking protocols with the capability of dynamic service updates, seamless failover for networking services, smooth protocol version transitions, and optimized reuse of protocol implementations – features which are critically needed but missing in many current designs.
3) Decouple the tight coupling between the media types and resource URIs with the introduction of media negotiation that removes any media type from the resource URI. The proposed HTTP content negotiation mechanism allows clients to request different media types from the same resource URI that best fit their capabilities and needs – a loosely coupled structure for service optimality.
4) Most importantly, we present a hypertext-driven approach using hypertext links defined between REST resources to guide clients to identify the right resources rather than relying on fixed resource URIs. The proposed approach has been applied to the design of the northbound API in SDN. It overcomes many protocol design issues in some existing protocols. We successfully implemented the new northbound API of SDN in the context of data networking with OpenStack. The advantages of the proposed approach, e.g. media negotiation, resource discovery, dynamic service and resource updates, protocol version transitions, etc., are observed and verified by experimental results.

The rest of this paper is organized as follows. Section II presents related work. Section III describes our HTTP content negotiation mechanism and hypertext-driven framework to northbound RESTful API design, taking REST design issues in some existing (Floodlight) northbound REST API as example. Section IV describes our REST Chart model based approach to design northbound RESTful API and the embodiment of the proposed approach in SDN with OpenStack Quantum [13] (now renamed to Neutron). Section V shows evaluation results based on integration of SOX, an SDN controller developed in Huawei [18] with Quantum, and the paper is concluded in Section VI.

II. Related Work

SDN is different from the traditional network management protocols, such as SNMP and more recent NETCONF. It decouples the data plane and the control plane so that the programmability of the network can be realized through either the low-level southbound API to the data plane, e.g. OpenFlow, or through the high-level northbound API of the software-based controller.

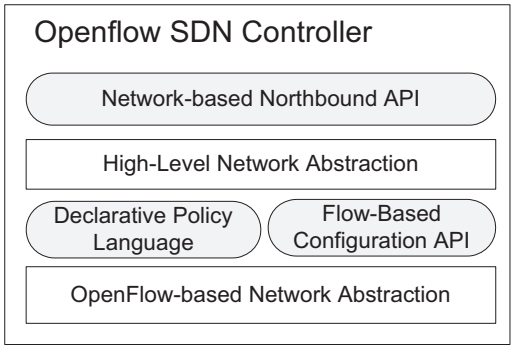

Fig. 2. Internal architecture of a typical OpenFlow SDN controller.

The SDN controller provides a logically-centralized network abstraction for applications to manage the network in an efficient and flexible manner. Fig. 2 illustrates the internal architecture of a typical SDN controller based on the model of OpenFlow. At the bottom, it is the OpenFlow-based network abstraction. It maintains a global topology of OpenFlow network and provides a configuration API to manage the flow tables in the underlying OpenFlow devices. Based on this configuration API, build-in applications can implement high-level network abstractions, such as virtual networks, that provide an efficient way to manage modern networks, especially for the large scale data networks in data centers and cloud computing platforms.

In particular, OpenFlow controller, NOX [3], provides the flow-based network configuration API in both Python and C++. Applications can add/remove flow entries as well as handling events, such as packet-in, from the OpenFlow devices. Some other controllers, such as Devoflow [6], Onix [15] and Maestro [4] focusing on improving the performance and scalability in SDN, also provide such network configuration APIs with similar functionalities. These controllers usually offer the network configuration API only for build-in applications. However, recently, POX, a Python implementation of NOX, exposes the configuration API as JSON-RPC web services together with a web messenger for sending OpenFlow events to applications [5].

The OpenFlow configuration API requires applications to devise flow entries across various OpenFlow devices (e.g. switches/routers). In contrast, a number of declarative configuration languages, such as Flow-based Management Language (FML) [7], Procera [8] and Frenetic [9], are proposed to expose network programmability by expressing the semantics of network policies declaratively rather than actually implementing them through the southbound API. The policy layer, which is typically built on top of existing controllers, such as NOX, will compile the high-level policies into the flow constraints to be enforced by the controller. Although these languages have shown their effectiveness in some applications, they are still under development and not directly related to the design of RESTful networking protocols.

Floodlight [10], a popular open-source SDN controller, provides a built-in virtual-network module, which exposes a REST API to the application of OpenStack Quantum [13]. Meridian, implemented as a module inside Floodlight, also provides a REST API for managing virtual networks [16]. However, different from the Floodlight module for Quantum, Meridian provides a virtual-network model from the perspective of application operators. Another example of REST-based northbound API is the Application Layer Traffic Optimization (ALTO) protocol [14], which aims to provide the right network abstraction for applications with heavy east-west traffics.

III. BASIC REST API DESIGN PRINCIPLES

However, the abovementioned northbound REST APIs are not truly RESTful. To illustrate the design issues in REST API, we take a look at an existing open-source SDN northbound REST API, Floodlight, which provides various services such as retrieving the network topology and statistics, and managing firewalls. The current REST API design does not comply with REST principles in two ways: 1) the API exposes media type in URIs; 2) the API exposes fixed resource URIs. In the following, their adverse effects will be discussed respectively, and a method is described that fixes these two issues without reducing the network services it provides.

A. *Violation 1: Exposing Media Type in URI*

In the current Floodlight REST API, many URIs include a suffix `/json`, to indicate that URIs point to a JSON representation, as shown below.

```
http://localhost:8080/wm/firewall/module/status/json
```

This approach does not separate identification from representation. As the result, it limits the client's and the server's abilities to evolve independently. If the server decides to switch from JSON to XML, the JSON URI will become invalid and the clients have to rediscover the new XML URI. If a client passes a JSON URI to another client that only understands XML, the second client cannot access the resource even if the server provides XML representation for the resource.

```
GET /wm/firewall/module/status    HTTP/1.1
Host: localhost:8080
Accept: application/json
```

```
GET /wm/firewall/module/status    HTTP/1.1
Host: localhost:8080
Accept: text/xml
```

A loosely coupled and service oriented way is to remove any media type from the URI, and use HTTP 1.1 content negotiation mechanism [11] to request different media types from the same URI. The above boxes show HTTP requests that uses `Accept` header to retrieve JSON and XML representations from the same URI.

B. *Violation 2: Exposing Fixed Resource URI*

The Floodlight REST API documentation exposes a fixed set of URIs to clients as shown below.

```
/wm/firewall/rules/json
/wm/firewall/module/status/json
/wm/firewall/module/enable/json
```

This approach violates the REST principle that the REST API must be hypertext driven such that the URIs are discoverable from an entry URI. When the URIs are predefined by some out-of-band mechanism, the controller loses the freedom to change the URIs in order to relocate the resources, because any such change will break the clients that were bound to the fixed URIs. In SDN, such resource reorganization is critical, as the controllers are expected to evolve and migrate rapidly to support various applications. A real-world example in Section IV will be presented to further illustrate this point.

To address this problem, we remove any fixed URI from our REST APIs, except a single entry URI to the API. In this hypertext-driven approach, the meaning of each URI is defined by the hypertext in which it occurs, and its value can be changed by the controller without changing its meaning, thus leading to a loosely-coupled REST architecture. A common way to assign meaning to a URI is to use the `rel` attribute of the link element. The `rel` attribute can be absolute URI as well. The values of these attributes will be defined by the REST API and cannot be changed by implementations. Based on `rel` attributes and the hypertext structure, a client can select the correct resource URI to follow and at the same time allow the controller to change the resource URI.

This approach is simple enough to be applicable to many languages, including XML and JSON as shown below.

```
HTTP/1.1 200 OK
Content-Type: text/xml
<firewall>
   <link rel="modules" href="/wm/firewall/123/modules"/>
   <link rel="rules" href="/wm/firewall/123/rules"/>
</firewall>
```

```
HTTP/1.1 200 OK
Content-Type: application/json
{
   "firewall": {
      "links": [
         {"rel":"module",
          "href":"/wm/firewall/123/modules"},
         {"rel":"rules",
          "href":"/wm/firewall/123/rules"}
      ]
   }
}
```

IV. REST API Model and Design Patterns

In this section, we present our REST Chart model for RESTful API design and apply this design framework in the design of northbound RESTful API of SDN with OpenStack Quantum. We describe how the hypertext-driven approach can be applied as well as its benefits over the fixed-URI scenario in the case of service evolvement.

A. *OpenStack Quantum Overview*

OpenStack is a composition of software components for building private or public cloud infrastructure. Quantum is the component providing networking services such as grouping virtual machines into isolated virtual networks and assigning IP addresses to them.

The core data model of Quantum consists of three types of entities: `network`, `port` and `subnet`. A `network` represents a virtual network, to which virtual machines attach by their `ports`. A `subnet` in a `network` represents an IP addresses block. A `port` receives IP addresses when it joins these `subnets`. The core functionality of Quantum is to manage these entities in the data model. Quantum employs a plugin-based architecture, which transforms user requests into calls to a responsible plugin. The plugin then interacts with the associated networking controller to perform the corresponding network operations. For example, Quantum interacts with Floodlight controller via a plugin named RESTProxy that accesses the REST API of Floodlight controller.

B. *REST Chart Model*

To design the Northbound REST API based on the Quantum architecture without violating the REST constraints [17], we adopt the REST Chart model [17] for describing general REST API. The REST Chart models a REST API as a special type of Colored Petri Net where the places represent media type schemas, the transitions represent links between the schemas, and the tokens in a place represent resource representations of a particular schema. REST Chart uses a XML dialect to encode a REST API such that all the REST constraints are enforced by the XML or can be verified automatically through algorithms. REST Chart can support efficient content negotiation and reuse hybrid representations to broaden design choices. A major feature of REST Chart is to combine the static aspects of a REST API, e.g. schemas and link relations, with the dynamic aspects of the API, e.g. the interactions with representations, into one coherent model.

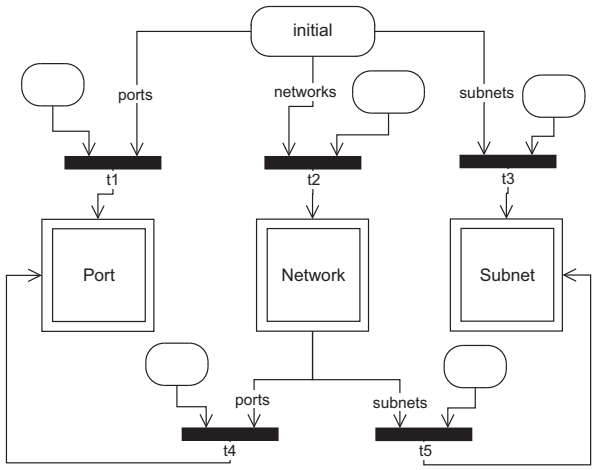

Fig. 3. A REST Chart for Quantum REST API .

Fig. 3 illustrates a REST Chart for part of the Quantum REST API, where places are represented by curved rectangles, transitions by solid bars, and nested charts by double framed rectangles. The entry URI is represented by the `initial` place, which contains three hyperlinks related to ports, networks, and subnets. Each transition has two incoming places, one for the link and one for the request to dereference the link. Each transition has at least an outgoing place denoting the response. To interact with the REST API, the client first obtains a token for the initial place, i.e. a representation from the entry URI. If the client deposits a "networks" token (request) at the empty place to fire transition t2, the API will return a token (response) in the network place. From this token, if the client deposits a "ports" token to the empty place to fire transition t4, it will receive a token in the port place. This process repeats until the client obtains the desired tokens from the API.

1) *URI Pattern*

URI is an important part of a REST API. Good URI namespace supports readability and extensibility. The URI namespace design we follow is called "type/variable" or "collection/member" pair, which prefixes each variable by a type. When used repeatedly in one URI, it gives flexible and consistent extension points to a URI namespace.

An example is to use types "tenant" and "networks" in one URI to identify the networks owned by a tenant: {entry}/tenants/{tid}/networks/{nid}. Although this pattern tends to create longer URI that seems unnecessary when the URI namespace is small, our experiences show that not

following this pattern limits extensibility when the namespace has to grow to accommodate more resources.

1) *Navigation Pattern*

The most basic interaction with a REST API is to navigate from an entry URI to a desired resource to obtain its current representation. This pattern is well-supported by the REST Chart. Fig. 4 shows the REST Chart model for accessing the `network` entity. The `network_list` representation contains a list of `network` hyperlinks by which clients can access a specific network. Listing 1 shows the HTTP request and responses for the `network_list` representation, where the links consist of relative URIs.

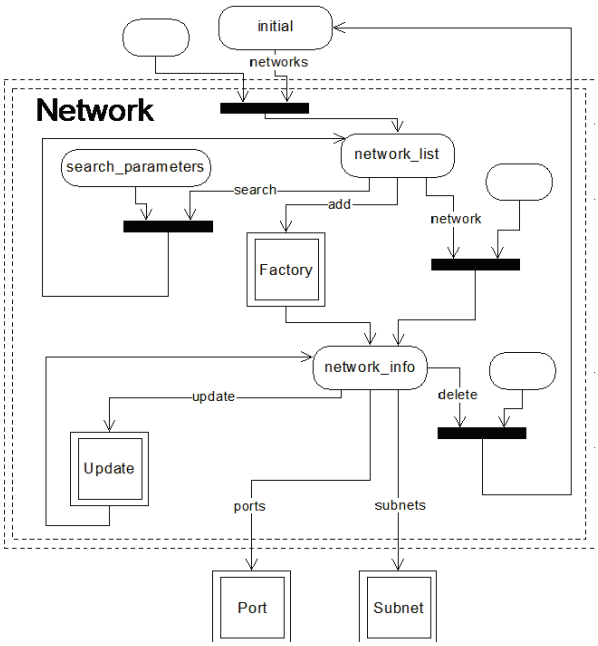

Fig. 4. REST Chart model of accessing network entity.

```
GET /quantum/v2.0/tenants/t100/networks        HTTP/1.1
Host: localhost:8080
Accept: text/xml, application/json
```

```
HTTP/1.1 200 OK
Content-Type: text/xml
<networks>
   <link rel="network"
         href="/net1" />
   <link rel="network"
         href="/net2"/>
   <link rel="network"
         href="/net3"/>
    <link rel="add"
          href="/factory" />
   <link rel="search"
          href="/search?{key1}={value1}&...&{keyN}={valueN}" />
</networks>
```

Listing 1: Navigation to networks of tenant 100.

1) *Filter and Search Patterns*

In Listing 1, the representation contains only the `network` hyperlink for each listed `network` entity. This can effectively reduce the representation size, especially when the number of `network` entities is large. This however may not be efficient for other use cases. For example, clients may need to retrieve the names of all networks but not the entire network representations. With current design, this is impossible because the client has to retrieve the entire network representation to get the name in it.

To address this problem, our REST API allows client to request additional content using `?attributes={name1,…,nameN}` URI query string, as shown in Listing 2, where the client request the name and id elements in addition to the default content.

```
HTTP/1.1 200 OK
Content-Type: text/xml
<ports>
   <network>
      <name>myNet</name>
      <id>net1</id>
      <link rel="network"
          href="/net1" />
   </network>
    <link rel="add"
         href="/factory" />
   <link rel="search"
         href="/search?{key1}={value1}&...&{keyN}={valueN}" />
</ports>
```

Listing 2: Filter response to networks.

If a tenant has a large number of networks, it would be very inefficient for the client to use the navigation pattern to locate a specific network. To address this problem, our REST API provides a `search` hyperlink containing a URI template to allow clients to submit queries consisting of key-value pairs.

```
GET /quantum/v2.0/tenants/t100/networks?attributes=name,id
HTTP/1.1
Host: localhost:8080
Accept: text/xml, application/json
```

1) *Factory and Update Patterns*

In addition to finding information through the REST API, clients can also create and change resources. To support these functions, the REST API links a factory resource to any representation where new resources can be created. The factory resource returns a form-like representation to specify what information is required from the client.

Fig. 5 shows the REST Chart model of creating a `network` entity and Listing 3 shows the example HTTP request and response for retrieving such a form. The form marks required attributes by attribute `required=true` and provides default values for some elements. The attribute `method` of the form element indicates that the client should submit the filled-out form to the factory resource by the HTTP command POST.

Similar to creating a resource, a client can also use a form to update a resource with HTTP PUT method.

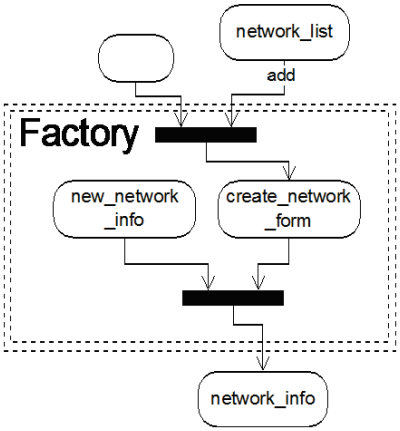

Fig. 5. REST Chart model of creating a new network.

```
GET /quantum/v2.0/tenants/t100/networks/factory    HTTP/1.1
Host: localhost:8080
Accept: text/xml, application/json
```

```
HTTP/1.1 200 OK
Content-Type: text/xml
<form method="POST">
   <network>
      <id required="true" />
      <name />
      <admin_state_up>true</admin_state_up>
      <shared>true</shared>
      <tenant_id required="true" />
   </network>
</form>
```

Listing 3: Factory form to create network in XML

C. *API Migration Patterns*

In the recent years, the functionality of Quantum has been enhanced rapidly from its early release. These enhancements lead to changes to its REST API. TABLE I. shows the main changes between two versions of Quantum's REST API in terms of resource organizations.

TABLE I. QUANTUM REST API EVOLVEMENT.

Core Resources of Quantum REST API v1.0
/networks/{network-id} /networks/{network-id}/ports/{port-id} /networks/{network-id}/ports/{port-id}/attachment
Core Resources of Quantum REST API v2.0
/networks/{network-id} /ports/{port-id} /subnets/{subnet-id} /devices/{device-id}

For comparison with Fig. 4 which is the REST API v2.0, Fig. 6 shows the REST Chart for REST API v1.0. From the two REST Charts, we can identify the invariant paths between versions. For example, both REST Charts contains `initial→networks→ports→port` path. Furthermore, the internal structures of the `network` and `port` nested charts are almost the same between the two versions. This means, if the two versions use the same rel attributes to annotate links, the v1.0 clients can use the v2.0 API portion that accesses the `network` and `port` resources with no or minor modifications.

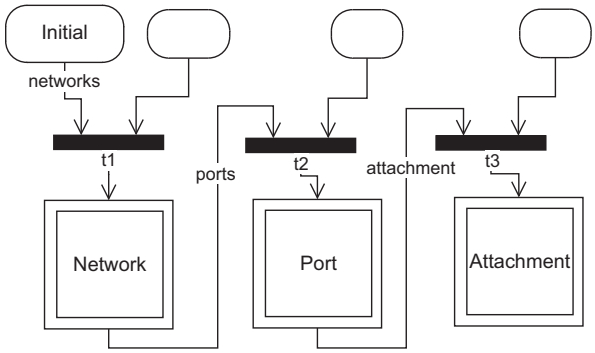

Fig. 6. Overall REST Chart Model of the hypertext-driven version of Quantum REST API v1.0.

The v2.0 REST API also changes the structure of resources. For example, ports are located under network in v1.0 and they are moved to the top level in v2.0. It is easy to devise an algorithm to traverse two REST Charts to identify such movements if the rel attributes remain the same. In this case, the algorithm will identify `initial→networks→ports→port` in v1.0 as equivalent to `initial→ports→port` in v2.0. These equivalent paths will help us to update the v1.0 clients to use v2.0 API.

The v2.0 REST API also changes resource relations while the functions of the resources remain the same. For example, v1.0 attachment resource is renamed to v2.0 device resource. In this case, we need to provide a mapping between "attachment" and "device" and use these mappings to find equivalent paths.

The only case where there is no code reuse is when v2.0 API adds new resources not defined in v1.0, such as subnets. By comparing two REST Charts, we can also identify such resources.

D. *Implementation Details*

We successfully applied the described design patterns to the REST API of the virtual network service module in SOX, an SDN controller developed in Huawei [18]. We also implemented a Quantum plugin SOXProxy in Python to access this REST API of SOX, to let OpenStack manage the virtual networks through SOX. SOX is implemented in JAVA and its REST API supports two media types: JSON and XML. SOXProxy prefers JSON using content negotiation described in Section III, since the `dict` data type of Python fits JSON naturally.

Fig. 7 illustrates the integrated system of OpenStack and SOX in two physical machines. Each machine is a Huawei Tecal RH2285 V2 server, with the same configuration of 2

Intel Xeon 2.4GHz 4-Core E5620, 48 GB of memory and 8 1-TB SATA hard drives. The two machines are interconnected by a local Gigabit LAN.

A single-node OpenStack environment with the OpenStack Grizzly release was deployed on one machine, which also contains our SOXProxy plugin to access the SOX controller, deployed on another machine. The OpenStack machine allows remote user management through a separate Internet connection. In the experiments, we used the Quantum Command-Line Interfaces (CLI) tools to remotely control Quantum, which in turn invokes the SOX REST API through the SOXProxy.

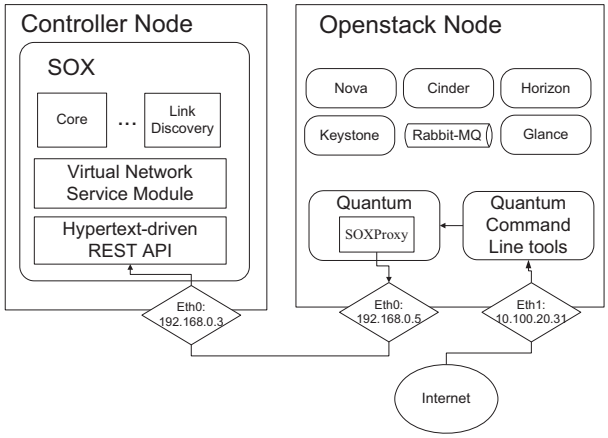

Fig. 7. The evaluation system configuration.

V. Evaluation Results

We evaluated the northbound REST API of SOX in two ways: 1) transparent upgrade to OpenStack Quantum under REST API change; and 2) the performance and the cost comparing to the fixed-URI scenario.

A. *Live Transparent Upgrade*

This experiment is to show that SOX REST API can migrate to a different URI namespace transparently to Quantum without interrupting its execution. To demonstrate this effect, we prepared two versions A and B of SOX controllers, where A is modeled after the REST API v1.0 and B after v2.0. We first start with SOX-A, and then switch over to SOX-B while Quantum is running and not aware of the switch-over.

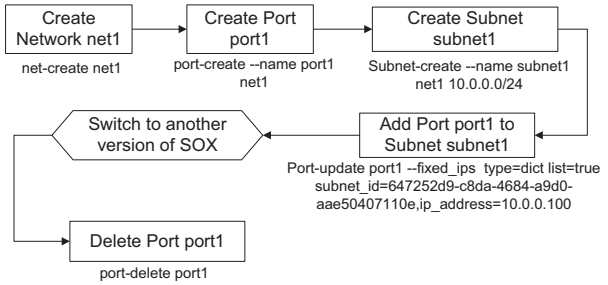

Fig. 8. The switch-over sequence and CLI commands

Fig. 8 shows the switch-over point between the interactions with SOX-A and SOX-B. The only common URI for SOX-A and SOX-B is the entry URI and they assigned different URI to the same resource. Since Quantum always starts from the entry URI and follows hyperlinks (the rel attributes) for each operation, it can navigate its way to the port1 resource via different URI path, even if SOX-A and SOX-B assign different URI to port1.

The following tables record the URI paths traversed by Quantum before and after the switch-over. If the REST API exposed fixed URIs, Quantum would get HTTP 404 after switch-over.

TABLE II. URIs accessed to update port1 in SOX-A.

Step	Source	Accessed URI
1	Fixed Entry URI	GET /quantum/v2.0/tenants/46f684a7-dcc0-478c-b4a8-313d4f768344
2	Link(Ports)	GET /quantum/v2.0/tenants/46f684a7-dcc0-478c-b4a8-313d4f768344/ports/7a02dea5-41e7-48f4-8de2-3e3ba7c24f08
3	Link(Port)	**GET /quantum/v2.0/tenants/46f684a7-dcc0-478c-b4a8-313d4f768344/ports/7a02dea5-41e7-48f4-8de2-3e3ba7c24f08**
4	Link(Update)	GET /quantum/v2.0/tenants/46f684a7-dcc0-478c-b4a8-313d4f768344/ports/7a02dea5-41e7-48f4-8de2-3e3ba7c24f08/update
5	Form(POST)	POST /quantum/v2.0/tenants/46f684a7-dcc0-478c-b4a8-313d4f768344/ports/7a02dea5-41e7-48f4-8de2-3e3ba7c24f08/update

TABLE III. URIs accessed to delete port1 in SOX-B.

Step	Source	Accessed URI
1	Fixed Entry URI	GET /quantum/v2.0/tenants/46f684a7-dcc0-478c-b4a8-313d4f768344
2	Link(Ports)	GET /quantum/v2.0/tenants/46f684a7-dcc0-478c-b4a8-313d4f768344/ports/7a02dea5-41e7-48f4-8de2-3e3ba7c24f08
3	Link(Port)	**GET /quantum/v2.0/tenants/46f684a7-dcc0-478c-b4a8-313d4f768344/networks/93d3c03f-5199-48e8-8596-fd365dc8e273/ports/7a02dea5-41e7-48f4-8de2-3e3ba7c24f08**
4	Link(Delete)	DELETE /quantum/v2.0/tenants/46f684a7-dcc0-478c-b4a8-313d4f768344/networks/93d3c03f-5199-48e8-8596-fd365dc8e273/ports/7a02dea5-41e7-48f4-8de2-3e3ba7c24f08/update

B. *Performance Evaluation*

In this evaluation, we deploy an emulator in the OpenStack machine to simulate the workload from OpenStack Quantum.

The emulator first added 100 initial virtual networks, to simulate a medium-sized tenant scenario. It then performed the REST API coverage tests for several rounds, each lasting for 1 hour. In each round, the emulator repeatedly sent random REST API requests at 500ms intervals to the controller and recorded the average response times of all requests at 5 minutes intervals. We also recorded at Quantum the total number of interactions made for completing each operation in the following table.

TABLE IV.
PERFORMANCE COSTS OF THE HYPERTEXT-DRIVEN AND THE FIXED-URI

Configuration		Avg. No. of Remote Calls
Fixed URI		1.22
Hypertext Driven	No Caching	3.98
	With Caching	1.70

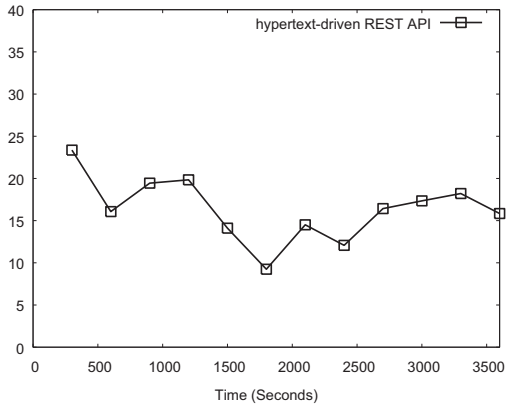

Fig. 9. The average round-trip response time in a 5 minute interval.

To improve performance, we cached the returned hyperlinks at the client (the emulator) so that the client does not have to start over from the entry URI for each operation. With cache, the average number of interactions is about 40% of the number without cache, and is 1.3 times of fixed URI in which resource is tightly bound to the fixed link but represents the minimal number of interactions. Fig. 9 shows that in a 5 minute interval, the average round-trip response times with cache are below 20ms, a result that is satisfactory.

VI. CONCLUSION

We presented a framework on designing northbound API of SDN in a truly RESTful manner through a set of REST API design patterns following the hypertext-driven approach. The proposed approach has been successfully codified by REST Chart and applied to RESTful design of northbound API of SDN, and it overcame some critical deficiencies in previous approaches. We demonstrated the freedom of modifying the hypertext-driven REST API through a transparent migration between different URI namespaces. The performance evaluation result shows a performance cost for achieving a loosely coupled ROA based REST protocol is about 40% more remote calls than using the fixed resource URIs, while the average response time can be maintained below 20ms.

VII. REFERENCES

[1] R. T. Fielding, Architectural styles and the design of network-based software architectures, Ph.D. Disserta-tion, University of California, Irvine, 2000, http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm.

[2] R. T. Fielding, “REST API must be hypertext driven,” 28 October, 2008, http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven.

[3] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker. NOX: towards an operat-ing system for networks. SIGCOMM Comput. Commun. Rev. 38, 3 (July 2008), 105-110, 2008.

[4] Z. Cai, A. Cox, and T. Ng. Maestro: a system for scal-able openflow control. Technical Report TR10-08, Rice University, December 2010.

[5] M. McCauley. POX web interfaces. September 12, 2012. http://www.noxrepo.org/2012/09/pox-web-interfaces/

[6] J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, A. R. Curtis, and S. Banerjee. DevoFlow: cost-effective flow management for high performance enterprise networks. In Hotnets-IX. 2010.

[7] T. L. Hinrichs, N. S. Gude, M. Casado, J. C. Mitchell, and S. Shenker. Practical declarative network man-agement. In ACM WREN '09.

[8] A. Voellmy, H. Kim, and N. Feamster. Procera: a lan-guage for high-level reactive network control. In HotSDN '12.

[9] N. Foster, M. J. Freedman, R. Harrison, J. Rexford, M. L. Meola, and D. Walker. 2010. Frenetic: a high-level language for OpenFlow networks. In PRESTO '10.

[10] Floodlight. Floodlight - an open SDN controller, 2013, http://www.projectfloodlight.org/floodlight/.

[11] N. Freed and N. Borenstein. RFC2046: Multipurpose Internet Mail Extensions (MIME) part two: media types, 1996, http://www.ietf.org/rfc/rfc2046.txt.

[12] 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, 2008.

[13] OpenStack Foundation. OpenStack networking administration guide, Feb 2013, http://docs.openstack.org/trunk/openstack-network/admin/content/index.html.

[14] R. Alimi, R. Penno, and Y. Yang. ALTO Protocol. IETF Internet draft. (work-in-progress), 25 Feb. 2013. http://www.ietf.org/id/draft-ietf-alto-protocol-14.txt.

[15] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. In-oue, T. Hama, and S. Shenker. Onix: a distributed con-trol platform for large-scale production networks. In OSDI, 2010.

[16] M. Banikazemi, D. Olshefski, A. Shaikh, J. Tracey, and G. Wang. Meridian: an SDN platform for cloud network services. IEEE Communications Magazine, 51(2):120-127, Feb 2013.

[17] L. Li and W. Chou, Design and describe REST API without violating REST: a Petri net based approach, Proceedings of the 2011 IEEE International Confer-ence on Web Services, 508-515, 2011.

[18] L. Min, T. Yingjun, L. Quancai, W. Jiao, and W. Chou. SOX - a generalized and extensible smart network openflow controller, October 2012.

[19] Click: http://www.read.cs.ucla.edu/click/click

[20] Pyretic: http://frenetic-lang.org/pyretic/