Design Patterns and Extensibility of REST API for Networking Applications

Li Li, *Member, IEEE*, Wu Chou, *Fellow, IEEE*, Wei Zhou, *Member, IEEE*, and Min Luo, *Senior Member, IEEE*

Abstract**—REST architectural style has become a prevalent choice for distributed resources, such as the northbound API of Software-Defined Networking (SDN). As services often undergo frequent changes and updates, the corresponding REST APIs need to change and update accordingly. To allow REST APIs to change and evolve without breaking its clients, a REST API can be designed to facilitate hypertext-driven navigation and its related mechanisms to deal with structure changes in the API. This paper addresses the issues in hypertext-driven navigation in REST APIs from three aspects. First, we present REST Chart, a Petri-Net based REST service description framework and language to design extensible REST APIs, and it is applied to cope with the rapid evolution of SDN northbound APIs. Second, we describe some important design patterns, such as backtracking and generator, within the REST Chart framework to navigate through large scale APIs in the RESTful architecture. Third, we present a client side differential cache mechanism to reduce the overhead of hypertext-driven navigation, addressing a major issue that affects the application of REST API. The proposed approach is applied to applications in SDN, which is integrated with a generalized SDN controller, SOX. The benefits of the proposed approach are verified in different conditions. Experimental results on SDN applications show that on average, the proposed cache mechanism reduces the overhead of using the hypertext-driven REST API by 66%, while fully maintaining the desired flexibility and extensibility of the REST API.**

Index Terms**— SDN, Controller, Northbound API, OpenStack, Neutron, REST API, Hypertext Driven, REST Chart, Petri-Net, Differential Cache**

I. Introduction

SOFTWARE-DEFINED NETWORKING (SDN) decouples the data and control planes of a network to provide enhanced flexibility, programmability, and openness to applications. This decoupling allows a logically centralized controller to control the network based on a global view of the network resources. As shown in Figure 1, at the center of the control plane is an SDN controller, which controls the behaviors of the underlying data forwarding elements through some southbound APIs, e.g. OpenFlow [1][2]. The SDN controller also provides a programmable interface for applications to observe and change the network states dynamically. This interface is called the Northbound API of SDN, shorthanded as NBI (North Bound Interface).

In SDN, the data plane and the control plane can interact within 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 and the resource information of the network; and 3) the data plane executes the operations which can change the network states, e.g. data path.

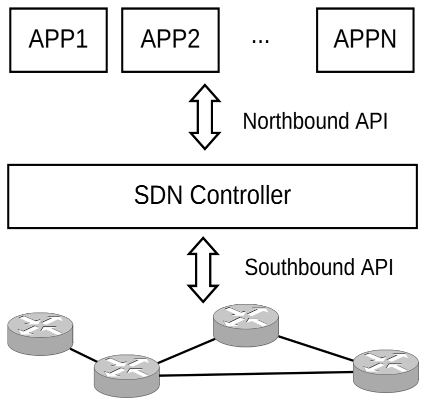


Figure 1. The architecture of Software-Defined Network.

A network is a complex distributed system whose structure, function, resource, and behavior can change dynamically. As the network structure evolves and the application paradigm shifts, the NBI of the network has to be flexible enough to accommodate these changes. As such, RESTful NBI has become a prevalent choice for the Northbound Interface (NBI) in SDN, where a well-designed REST API can be highly extensible and maintainable for managing distributed resources, such as data networking. Extensibility in REST means that a REST API can provide different functions at the same time and it can also make certain changes to those functions over time without breaking its clients. A REST API achieves such flexibility through the mechanism of *hypertext-driven navigation* of connected resources, which is also described as "hypertext as the engine of application state" in [3][4].

In particular, a REST API consists of connected REST

The paper is submitted for publication in April 2015 as a full paper for *IEEE Transactions on Network and Service Management (TNSM)*.

Li Li is with Shannon (IT) Lab, Central Research, Huawei, USA (e-mail: li.nj.li@huawei.com)

Wu Chou is with Shannon (IT) Lab, Central Research, Huawei USA (e-mail: wu.chou@huawei.com)

Wei Zhou was with Shannon (IT) Lab, Central Research, Huawei China when this work was done. He is currently with Autonomous System, Digital Productivity Flagship, CSIRO. (e-mail: wei.zhou@csiro.au)

Min Luo is with Shannon (IT) Lab, Central Research, Huawei USA (e-mail: min.ch.luo@huawei.com)

resources that provide different services through uniform interfaces. In case of SDN, it includes services from both data and control planes, such as switches, routers, subnets, networks, NAT devices, and controllers.

There are plenty of cases that the REST APIs need to undergo rapid changes and updates as evidenced by many open source development projects. For example, OpenStack [5], which is an IaaS platform, currently supports 14 REST APIs [6], implemented by over 30 components that manages compute, storage, network, VM image, and identity services. Moreover, OpenStack development follows a very rapid release cycles of 6-11 months [7], and each new release may introduce changes to REST APIs. To maintain backward compatibility, OpenStack simultaneously supports different versions of the same API. For example, there are 3 versions of Compute API, 2 versions of Block Storage API, 2 versions of Identity API, and 2 versions of Image API. On the other hand, Docker, another major open source development project on Linux container, has already published 6 versions of the Docker Remote REST API (v1.14-v1.19) [8] within a 2 year period, or one new version every 2 months on average. Moreover, many new versions, including OpenStack Neutron REST API [9], introduce incompatible changes. As a consequence, it has become an acute problem to increase the extensibility while maintaining the backward comparability of a REST API under rapid changes and updates – a situation which is typical in large scale open source developments.

Hypertext-driven navigation offers an effective means for clients to cope with certain changes in a REST API automatically. To facilitate hypertext-driven navigation, we need to design our REST API following the REST principles [10] and convey our design to the users precisely so that they can create the well-designed clients accordingly. Furthermore, we need to address the overhead associated with hypertext-driven navigation to balance the flexibility and efficiency of a REST API.

This paper presents an extensible REST API framework for producing a hypertext-driven REST API for SDN NBI. The framework consists of REST Chart, a Petri-Net based XML language that describes a REST API, and several automated tools to compare, generate, and prototype REST APIs based on REST Chart. This paper applies and extends our earlier work on REST Chart based modeling framework to the design and implementation of SDN NBI, with a focus on dealing with rapid REST API changes and updates. In addition, this paper proposes five new design patterns to address the common issues in large scale REST APIs and a new layered differential cache mechanism to reduce the overhead of hypertext-driven navigation. The proposed REST API for SDN NBI has been implemented based on Huawei SOX SDN Controller [11]. The benefit of the hypertext-driven navigation is demonstrated through a live REST API migration without interrupting the services to its clients, which is critical for large scale cloud based deployment. The advantage of the proposed cache mechanism is further demonstrated in SDN applications by reducing the average overhead of hypertext-driven navigation by 66%.

The rest of this paper is organized as follows. Section II presents related work. Section III discusses how hypertext-driven navigation is enabled in a layered REST API architecture. Section IV presents the REST Chart and its extensibility under hypertext-driven navigation, with examples from SDN Northbound REST API based on OpenStack Neutron [9]. In Section V, we describe and analyze some important navigation patterns in REST API. Section VI describes a novel cache mechanism to reduce the overhead of hypertext-driven navigation. Section VII presents the experimental results based on an integration of SOX, an SDN controller developed in [11], and OpenStack Neutron, and the paper is concluded in Section VIII.

II. Related Work

SDN is aimed at a flexible and programmable network, and it is different from the traditional network architecture where the data plane, control plane and management plane (SNMP [12], NETCONF [13]) are distributed and integrated in network devices. SDN decouples the control plane from the data plane and centralizes the programmability of the network through either the low-level southbound API to the data plane, e.g. OpenFlow, or through the high-level northbound API to SDN controller.

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

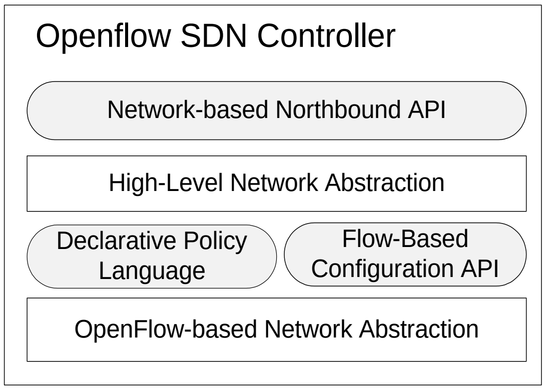

Openflow SDN Controller
Network-based Northbound API
High-Level Network Abstraction
Declarative Policy Language
Flow-Based Configuration API
OpenFlow-based Network Abstraction

Figure 2. Architecture of a typical OpenFlow SDN controller.

Under this generic architecture, several northbound APIs have been developed before the RESTful approach is taken. In particular, OpenFlow controller, NOX [14], 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 [15], Onix [16] and Maestro [17] 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 built-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 [18].

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) [19], Procera [20], and Frenetic [21], are proposed to expose network programmability by expressing the semantics of network policies declaratively, and 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 APIs have shown their effectiveness in some applications, they are not as extensible as RESTful services.

Floodlight [22] is a popular open-source SDN controller, providing a built-in virtual-network module, which exposes a REST API through OpenStack Neutron [9]. Meridian, implemented as a module inside Floodlight, also provides a REST API for managing virtual networks [23]. However, different from the Floodlight module for Neutron, 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 [24], which aims to provide a network abstraction for applications with heavy east-west traffics.

OpenStack [5] is a popular open source software package for building IaaS clouds. It provides an extensible infrastructure using REST APIs to control and manage the distributed resources. Neutron [9] is the component in OpenStack providing networking services such as grouping virtual machines into isolated virtual networks and assigning IP addresses to them.

The Neutron Core API consists of three types of resources: network, port, and subnet. A network resource represents a virtual network with ports, to which virtual machines can attach. A subnet resource in a network represents an IP address block allocated for the ports. A port resource receives IP addresses when it joins a subnet.

The Neutron Core API allows the clients to manage the network resources of OpenStack, in conjunction to the Compute and Storage resources managed by different OpenStack APIs. Neutron employs a plugin-based architecture, which allows different network vendors to incorporate their own network technologies into OpenStack. A plugin must implement the Neutron Core API and may implement any number of extensions defined by OpenStack or by the third-party network vendors. Neutron API server transforms client requests into calls to the responsible plugin, which interacts with the network elements (e.g. switches and routers) accordingly. To support SDN, the plugin interacts with the associated SDN controller over northbound API which controls the network switches through the southbound API. For example, the Floodlight plugin named RESTProxy allows Neutron API server to interact with a Floodlight controller that controls Open vSwitches using the OpenFlow protocol.

Despite the increasing popularity of REST API, we do not have a standard and machine readable language to describe a REST API at the level to match the status of WSDL [25] for RPC style web services. Over the years, several REST description languages have been proposed, including WADL [26], RAML [27], Swagger [28], RSML [29][31], API Blueprint [30], RADL [32] and REST Chart [33]. All these languages describe the structure and behavior of a REST API to its clients by answering the following 4 basic questions: 1) where are the hyperlinks in a hypertext; 2) what service does each hyperlink provide; 3) how to interact with a hyperlink; and 4) what hypertext will the interaction produce. However, REST Chart differs from the others as it is based on Petri-Net [34][35] and it does not explicitly define resource identifications and connections. Instead, it defines resource representations and interactions, while leaving the identifications and connections to be determined by hypertext-driven navigation at runtime. More detailed comparisons between these languages can be found in paper [36].

YANG [37] from IETF is a hierarchical data modeling language that describes the structure, operations, and contents of data stores on network devices to be configured by remote clients through RPC or HTTP protocols. A YANG model consists of modules, and a module is made up by statements, in which the data statements define the structure of a data store in terms of primitive types, and the operation statements define the operations (query, create, delete, insert, merge, modify, etc.) in terms of input and output messages. YANG also includes a set of advanced features such as conditional extensions and data constraints, which can be used in combination to create complex and dynamic relations in data models. YANG can be mapped to XML syntax called YIN [37].

Based on the YANG modeling language, RESTCONF [38] from IETF specifies a HTTP protocol to access the data stores on network devices, as a compatible alternative to the NETCONF RPC protocol [50]. To access a data store, a client first discovers the entry URI to the RESTCONF API from the predefined URI on the device. From the entry URI, the client can retrieve the entire data store, which is essentially a tree with different types of nodes, as well as the YANG modules that describe the data store. The client can determine the URI to any node in the data store by combining the entry URI with the relative path to the node and the HTTP operations on the node according to the YANG modules and the capabilities of the data store, which can be discovered by the client. RESTCONF is included in OpenDaylight [39], an open source SDN Controller platform.

RESTCONF adopts a top-down design where the resource model (resource types and relationships) of a data store is predefined and made available to the clients as YANG modules. Exposing all the resource connections of a data store to a client is necessary in this case because modifying a node in the data store may affect its children. Using a path, instead of a hyperlink, to identify a node in the data store, is efficient if resource identifications and connections do not evolve independently. For example, moving a <container> node to a different path always creates a new <container> resource and therefore a new URI.

Top-down design is generally suitable for a stable REST API whose resource model evolves slowly over time. However, as discussed in Section I, the resource model of SDN NBI tends to change frequently as its popularity grows. As more network devices and functions (e.g. switches, routers, NAT, FW and ID, etc.) are virtualized and can be connected and disconnected in almost any order in seconds, it is difficult and unnecessary for a REST API to fix a resource model for the clients. To cope with such rapid changes without frequent updates to the clients, a *bottom-up* design is more effective, where a client can interact with a resource without knowing all its connections. To follow this design, a REST API only fixes the hypertext formats at design time, and the clients can use hypertext-driven navigation to discover the resource identifications and connections at runtime.

III. Hypertext-Driven Navigation Principle

A primary goal of hypertext-driven navigation is to cope with REST API changes (discussed in Section I) by reducing the design time dependency between the clients and a REST API, such that a client can navigate to the target resources as driven by the hypertext from interacting with the REST API. Hypertext-driven navigation relies on 5 layers of a REST API described below:

1. Connection: links between the resources implemented in any programming languages and running on any devices.
2. Interactions: methods/protocols (e.g. HTTP) to interact with the resources.
3. Identification: identifiers (e.g. URI) that identify the resources.
4. Representation: hypertexts (e.g. XML) sent to and obtained from the interactions.
5. Description: descriptions (e.g. REST Chart) about the possible resource representations.

At the design time, a client is programmed against a description of a REST API, without knowing how the resources are identified and connected. At the runtime, the client becomes an automaton that starts from the entry point and navigates to the target resource driven by hypertext representation. During the navigation, the client moves from layer 4 down to layer 1 and back up in cycles. In each cycle, the client uses representations to determine identifications, identifications to determine interactions, interactions to determine connections and representations.

Besides following hyperlinks, content negotiation and redirection can also be used to facilitate hypertext-driven navigation. In content negotiation, the URI does not reveal the media type [46] of the representation, but suggests an interaction (e.g. HTTP) that can negotiate and determine the appropriate representation. As the result, the representation of a resource can change without changing its identification – a loosely coupled architecture for distributed resources. In HTTP redirect, the URI does not need to identify the target resource, but it initiates an interaction that in turn determines the correct connection to the target resource. As the result, a resource can change its connections without changing its identification. As URI resolution depends on the hypertext in which a URI occurs, it is also possible to change the interaction with a resource without changing its identification. Hypertext-driven navigation, therefore, makes it possible to allow certain independent changes in these layers, as long as those changes are permitted by the description layer.

Based on hypertext-driven navigation principle, defining media types in URI, such as /network/port/json and /network/port/xml would not be a good practice, because it makes representation dependant on and tightly coupled with the identification. Consequently, it is difficult to evolve the representations without changing the identification, and the client is deprived of the right to negotiate for the optimal media type that best fits its capabilities. As a SDN controller can be accessed over a variety of network resources, including switches, routers, and devices, separating identification from representation allows a client to select the most efficient representation.

Moreover, due to the similar reason, exposing a set of fixed URIs to clients at design time is not a good practice either, as it ties identification with connection, and any changes in the connections can invalidate the URIs. For example, URI /networks/1/ports/2/ implies there is a connection from the networks resource to the ports resource. If this connection changes, then the URI becomes invalid. In SDN, such resource reorganization is common and critical, because SDN controllers are expected to provide a more dynamic, open, and programmable network, where the network can be dynamically configured and planned against the network traffics to support various applications and QoS requirements.

IV. REST Chart Model

To provide a structural model for hypertext-driven navigation in REST API, we adopt the framework of REST Chart, a Petri-Net based model and language as the foundation for REST API modeling. REST Chart models a REST API as a special type of Colored Petri Net that consists of places and transitions connected by arcs. Resource representations are modeled as tokens stored in the places. The colors of a place designate the schemas that define the representations (tokens) accepted by the place. A transition binds a hyperlink between two places to a protocol that interacts with the hyperlink. A REST Chart describes the resource representations and interactions of a REST API at design time as well as the hypertext-driven navigation of its client at runtime.

Figure 3 illustrates a basic REST Chart with three places (schemas) and one transition that describe a REST API to login an online account with username and password. To use this REST API, a client selects the login hyperlink *k* from token *x1* in the login place, and deposits token *x2* (username and password) in the credential place. With these two tokens in the input places, the transition will fire as defined by Petri-Net semantics, where firing a transition here models the interaction with the remote resource using the protocol defined by the transition, shown here as the dotted arrows between the transition and the remote resource, which is not part of, but is implied by, the REST Chart. If the interaction is successful, the resource will return the account information in a response, which is modeled as token *x3* in the account place.

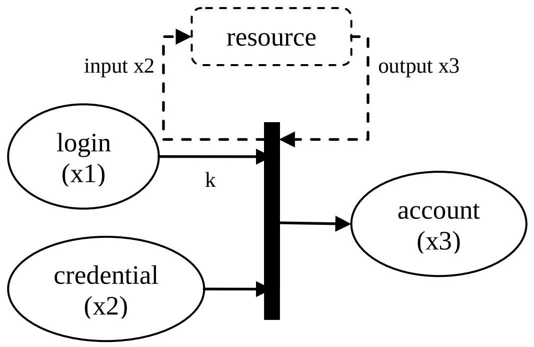


Figure 3: Illustration of a basic REST Chart

Hypertext-driven navigation of a REST API can be determined by the reachability of the corresponding Petri-Net, when the client states are modeled as token makings of the Petri-Net. In this example, the initial state of the client is (*x1, 0, 0*), meaning it has token *x1* in the place login, but no tokens in the credential and account places. From this initial state, the REST Chart indicates that the final state (*0, 0, x3*) is reachable as follows:

$$(x_1, 0, 0) \rightarrow (x_1, x_2, 0) \rightarrow (0, 0, x_3).$$

Not all possible states are reachable from the initial state, and such unreachable states include (*x1, 0, x3*) and (*0, x2, x3*). State (*x1, 0, x3*) means that a client can access the account without valid credential, and state (*0, x2, x3*) means that the client can access the account without following the hyperlink. An interesting property of REST Chart is that the set of reachable states defines the possible resource connections taken by the clients following hypertext-driven navigation. If a client can enter an unreachable state, it means that the REST API description violates the hypertext-driven navigation principle [33].

REST Chart is based on a layered and modular architecture to model the extensibility of a REST API. As shown in Figure 4, a REST Chart allows extensions at several layers with local changes. A local change at a module only expands the layers below it without modifying its interface to the upper layer. For example, a REST Chart can make local changes to the schemas of a place without affecting the places or the transitions. Similarly, a REST Chart can make local changes to the network protocols of a transition without affecting the transitions or the places.

The *hyperlink decoration* mechanism in REST Chart separates hyperlinks (resource identification) from schemas (resource representation), such that we can define hyperlinks independently from schemas. A hyperlink decoration has two parts: 1) a hyperlink; and 2) a mapping from the hyperlink to the schema. A hyperlink also has two parts: 1) a URI that identifies the service provided by the resource; and 2) a URI Template [40] that describes the locations of the resource in the REST API.

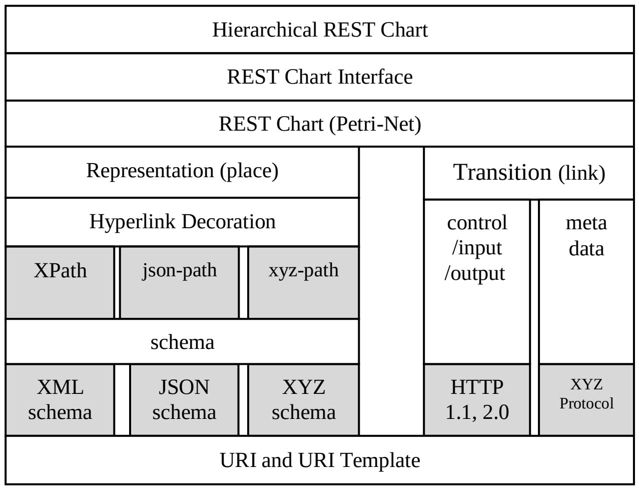


Figure 4. Layered modular architecture of REST Chart.

For example, a hyperlink decoration *k* in Figure 3 could be defined as an XML element, where we use short service names to illustrate the idea:

```
<link id="k">
  <rel value="urn:login" />
  <href value="http://bank.com/{path}/{userid}" />
</link>
```

The <rel> element defines the service and the <href> element defines the actual location of the service. Two hyperlinks are *compatible* if they have the same service URIs. A hyperlink path is a sequence of place-hyperlink pairs, where hyperlink is a [service, location] pair as defined above. Two hyperlink paths are *compatible* if for each corresponding place-hyperlink pairs, the place is identical and the hyperlinks are *compatible*.

For example, the hyperlink path for the REST Chart in Figure 3 could be:

```
login-[urn:login, http://bank.com/users/1]
account-[]
```

The first place-hyperlink pair corresponds to token markings *(x1, 0, 0)*→*(x1, x2, 0)*, while the second one corresponds to token marking *(0, 0, x3)*, i.e. the final state. If a REST API changes the location URI but keeps the service URI, it produces a compatible hyperlink path:

```
login-[urn:login, http://bank.com/accounts/1]
account-[]
```

A client that navigates a REST API based on compatible hyperlinks can follow compatible hyperlink paths without any change using the decision rule: in place *x*, follow hyperlink with service *y*, regardless of location *z*. In other words, as long as the relations between x and y in pairs *x-[y, z]* remain the same, a REST API is free to change *z* anywhere along a path without breaking any hypertext-driven clients.

Not all connection changes produce compatible hyperlink paths. Figure 5 shows two incompatible paths between Neutron versions 1.0 and 2.0 depicted by REST Charts in Figure 6 and Figure 7 respectively. These hierarchical REST

Charts [47] have places that contain nested REST Charts so that we can organize a complex REST API as modular REST Charts. For example, the REST Chart in Figure 6 contains the REST Chart in Figure 8. We refer readers interested in hierarchical REST Chart to the reference [36].

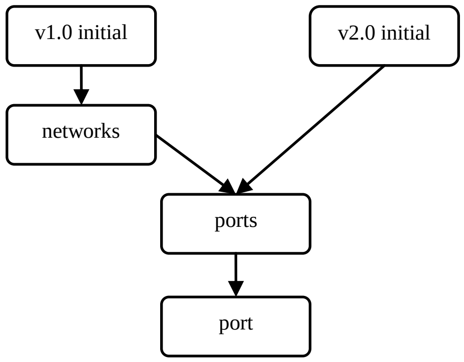


Figure 5. Version 1.0 (left) and version 2.0 (right) paths

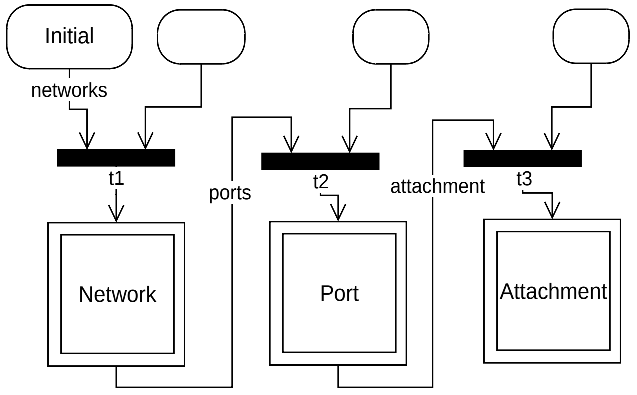


Figure 6. Part of REST Chart for Neutron REST API v1.0.

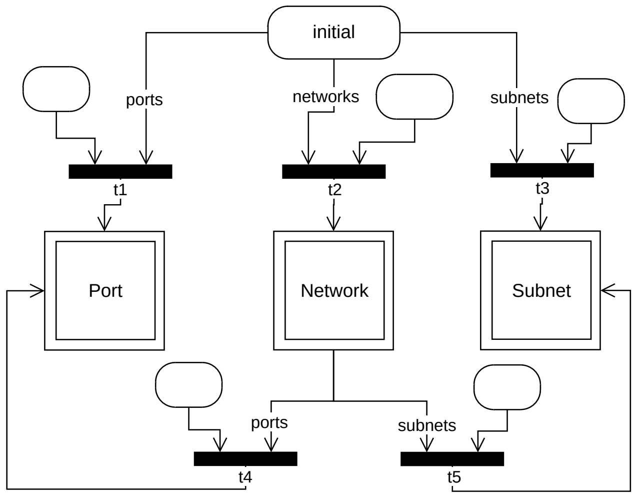


Figure 7. Part of REST Chart for Neutron REST API v2.0.

To facilitate machine processing, REST Charts are encoded as machine readable XML. For example, the networks-ports connection of version 1.0 REST Chart in Figure 6 is represented as follows (all places are represented by <representation> elements in REST Chart):

```
<representation id="networks" >
   <link id="ports_v1.0">…</link>
  <schema>…</schema>
</representation>
<transition>
  <input>
    <representation ref="network" link="ports_v1.0" />
  </input>
  <output>
    <representation ref="ports" />
  </output>
</transition>
```

Similarly, the initial-ports connection of version 2.0 REST Chart in Figure 7 is a follows:

```
<representation id="initial" >
   <link id="ports_v2.0">…</link>
  <schema>…</schema>
</representation>
<transition>
  <input>
    <representation ref="initial" link="ports_v2.0" />
  </input>
  <output>
    <representation ref="ports" />
  </output>
</transition>
```

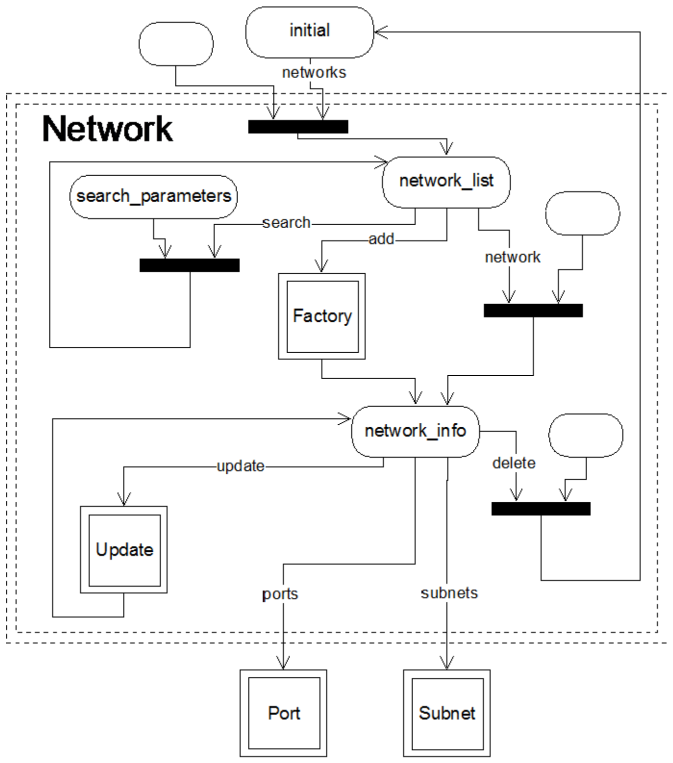


Figure 8. REST Chart nested inside the network place.

Table I lists the main URI namespaces of Neutron REST API versions 1.0 and 2.0, which contains both compatible and incompatible hyperlink paths.

TABLE I. NAMESPACES OF TWO VERSIONS OF NEUTRON API

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

To identify the invariants and changes between two versions of a REST API, such as the one in Figure 5, we developed a Petri-Net based algorithm and tool [41] to compare two REST Charts, e.g. Figure 6 and Figure 7. The algorithm emulates a hypertext driven client navigating through the REST Chart of Figure 7 with only the knowledge about the REST Chart defined by Figure 6, and it outputs all the places it can reach in Figure 7 when assisted by the places in Figure 6. In other words, the places it can reach in Figure 7 are the services that are compatible to those in Figure 6, while the places it cannot reach in Figure 7 are the new services that have no counterparts in Figure 6.

In this case, it identifies that to migrate a v1.0 client to v2.0 REST API, the minimum change to a v1.0 client is to modify the decision rule at the *initial* place to reach the *port* place, because if we align place *initial* in v2.0 with place *networks* in v1.0, then the following two paths become compatible:

v1.0: initial-[networks,_], networks-[ports,_], ports-[port,_]
v2.0: initial-[ports, _], ports-[port, _].

To cope with compatible and incompatible changes to a REST API, we propose a more structured way to break a REST client into two cooperative functional components, i.e. client oracle and client agent. In this approach, the client oracle is responsible for selecting hyperlinks and the client agent is responsible for exchanging protocol messages based on the selected hyperlinks. By this structural separation, we can reuse the client agent and/or the client oracle. Table II summarizes the possible changes to a REST API and the corresponding changes required to the client components.

TABLE II. REST API AND CLIENT CHANGES.

client changes		**API message changes**	
		compatible	**incompatible**
API path changes	**compatible**	no	agent
	incompatible	oracle	oracle, agent

Since the optimal oracle can be generated automatically from a REST Chart [42], our approach automates part of client implementation and migration as well.

V. Hypertext-Driven Navigation Design Patterns

In order to effectively use the REST Chart framework to support hyperlink-driven navigation in large scale REST APIs, a REST API should be designed and implemented following certain patterns, each addressing a common problem in REST APIs. In addition to the 5 basic patterns in [49], this section considers 5 new practical patterns to facilitate hypertext-driven navigation. As each design pattern has advantages and disadvantages, a REST API can combine them based on its needs.

A. Tree Pattern

If we have 10^4 types of resources, we can organize them as an n-ary balanced tree to reduce the cost of hypertext-driven navigation. In a 10-ary balanced tree, the number of messages to access any resource is bounded by O(logN), which is 4 for $N=10^4$.

The advantage of this approach is that we can reduce the overhead of hypertext-driven navigation. The disadvantage is that we may not find a balanced tree for all domains.

B. Backtracking Pattern

Once a client navigates to a resource following a hypertext path, it can remember the hyperlinks in caches and use them to backtrack a REST API when the connections change. For example, after a client navigates a hyperlink path:

initial-[networks, _], networks-[ports,_], ports-[port, _],

it can cache the places {initial, networks, ports}, and the hyperlinks {networks, ports, port} separately to allow them to be recombined in different ways. When getting stuck, the client can backtrack the path as follows: at place x, if hyperlink h is not available, go back to x's parent and select hyperlink h again. If the REST API reorganizes the resource as follows:

initial-[ports,_], ports-[port, _],

then the client can no longer find the ports hyperlinks from the networks place. However, the client can backtrack to place initial, the parent place of networks, and rediscover the hyperlinks for ports.

In order to support backtracking, the service URI of a hyperlink should be independent of places, although its location URI can be dependent on the place. Furthermore, REST API should avoid HTTP cookie as it may make the client's interactions non-repeatable. With HTTP cookie, a client may get different tokens when it backtracks to the same place. An approach that avoids the side-effect of HTTP cookie is to use session resources [43].

The advantage of backtracking is that it can rediscover the relocated resource. Its disadvantage is that a client can only find relocated resource with compatible hyperlinks reachable from the hyperlink path it has visited. For example, if a SDN REST API changes the original hyperlink path to:

initial-[subnets, _], subnets-[ports,_], ports-[port, _],

then the client will not be able to find the ports hyperlinks, because subnets-[ports, _] is a new place that the client has never visited before, and consequently, the backtracking will fail.

C. Redirection Pattern

HTTP 1.1 redirection [44] can be used to inform a client the existence of new service and/or new location of a resource. For example, if a client follows a hyperlink path: network-[urn:update, u1], and receives from u1 a HTTP 3xx response with new hyperlink pair [urn:set, u2], then the client can learn that this change, even it is not specified in REST Chart:

network-[urn:update, u1]→network-[urn:set, u2].

One primary advantage of redirection is that a REST API does not have to change its REST Chart. The disadvantage is that the REST API cannot delete the previous resource as it is used for redirection. However, it is possible to delete the previous resources after a certain transition period, so that clients that use the REST API frequently can learn the new locations from redirection, and the casual clients can learn from a new REST Chart published later on.

D.Search Pattern

Using hypertext-driven navigation to find a resource among thousands of resources (e.g. networks or ports) one by one is not effective. To address this problem, our REST API provides a search resource that accepts URI query string: ?attributes={name1,…,nameN} and returns a small list of resources that match the criteria. An illustrative example is shown below where the <ports> XML in the response message contains a hyperlink for search.

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

```
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?{k1}={v1}...{kN}={vN}" />
</ports>
```

The advantage of this approach is that a client can reach any resources with two interactions: search and visit. The disadvantage is that a client can only search resources that exist, and the client has to know some information about the resources it searches for.

E. Generator Pattern

If a REST API wants to exchange a large number of hyperlinks with a client, it can include a small generator to create those hyperlinks, instead of the hyperlinks themselves, in a message. A generator defines a list of hyperlinks with a URI template and functions f_1-f_n that instantiate the variables v_1-v_n in the URI template [45]:

$$[\text{URI\_template}(v_1,\ldots v_n)|v1\leftarrow f_1, \ldots, v_n\leftarrow f_n].$$

For example, the following is a generator that generates the hyperlinks to network ports for a tenant tid:

$$[\text{/networks/\{nid\}/ports/\{pid\}}|[\text{nid, pid}]\leftarrow f(\text{tid})].$$

If f(1234)={[1,a], [2, b], [3, c]}, then the generated list is [/networks/1/ports/a, /networks/2/ports/b, /networks/3/ports/c]. A REST API can send the generator functions to a client using a mechanism called Code On-Demand [3], in the similar way that Web servers deliver JavaScript code to Web browsers.

The advantage of this approach is that the REST API can convey a large number of items with a small message, and a client has the freedom to select the items to fit the need. The disadvantage is that the client has to execute code from a REST API, which increases the security risks in the REST API.

F. Factory Chain Pattern

In many situations, hypertext-driven navigation can be the only way to access resources that have not been created in a REST API. The hyperlink path in this case can be a chain of factory resources, where one factory resource creates another factory resource in the chain, and the final factory creates the target resources. Figure 9 shows the REST Chart model of a factory resource that accepts a form and creates a network, from which a client creates subnets, and from subnets the ports. The example HTTP request and response to interact with the factory resource is shown below Figure 9. Here the form marks the required attributes by making attribute required=true, while it provides default values for some other elements. In particular, 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.

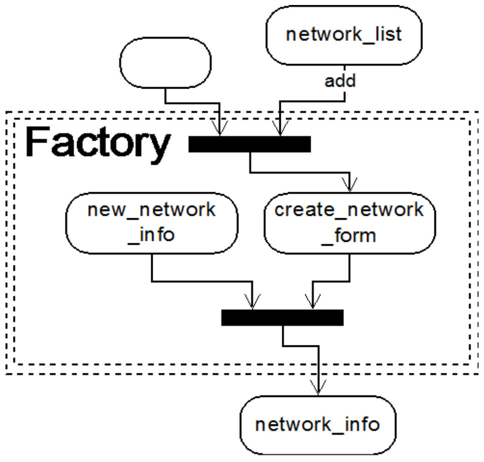


Figure 9. REST Chart model of creating a new network.

```
GET /neutron/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>
```

A chain of factories defines a hyperlink path that can be navigated by a client only once and in the fixed order. To support backtracking along this hyperlink path, each factory resource should also support a delete operation, such that a client can always go back to the parent and recreate a child resource that is not working.

The advantage of this approach is that a REST API has the freedom to reorganize a factory chain, such as inserting and removing factories, in the same way as it reorganizes any resources, but with the additional semantics that a creation operation must be executed at most once. For this reason, a client can deal with changes to factory chains using any patterns discussed herein. The disadvantage of this approach is that it does not support transaction that can automatically abort all the previous creations if the last one fails. REST transaction is still a research area [4].

VI. HYPERTEXT-DRIVEN CLIENT DESIGN

Clients visiting the hypertext-driven REST API for the first time should start from the entry URI and then follow the returned hyperlinks to access the resources. Compared to the fixed-URI scenario where the clients access the target resource directly, this could introduce considerable performance overhead if the target resources take many rounds of interactions to reach from the initial resource. This section presents a client cache mechanism to reduce the performance overhead when performing the hypertext-driven interactions. Consequently, a REST API and its clients can take advantages of hypertext-driven navigation without sacrificing the performance.

A. Issues of Caching for REST Service Clients

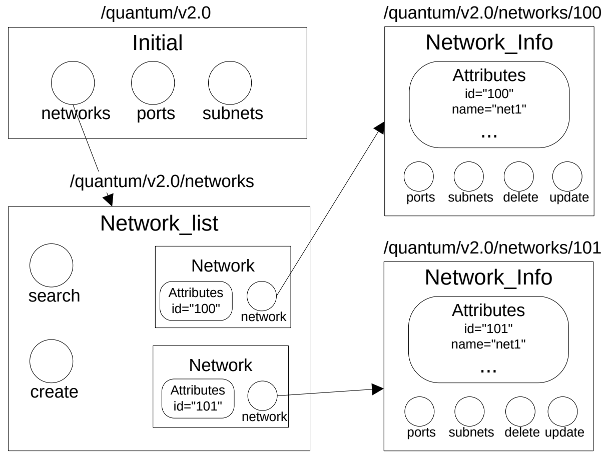
The basic idea of caching for REST service clients is to save clients from additional rounds of interactions in future accesses. As the current REST APIs are commonly implemented over HTTP, we started by considering deploying a client-side classical Web cache based on HTTP 1.1 cache controls [44], which stores the complete HTTP response, including all the attributes, entities, and hyperlinks contained in the representation.

Figure 10 (a) shows an example of such Web cache. Within the cache, the initial representation contains the hyperlinks only, while the network_list representation contains not only hyperlinks, but also a set of network entities. Each network entity has its ID attribute and a network hyperlink.

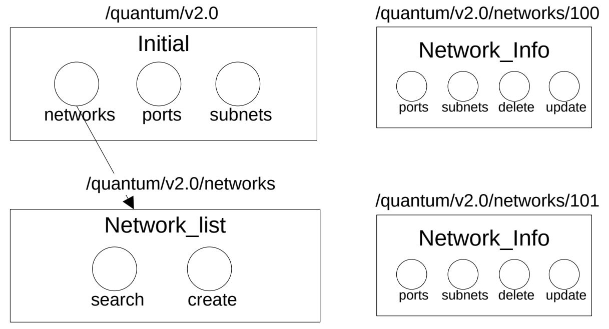
Such a Web cache could effectively improve the performance of clients for retrieving representations of resource states. However, many hypertext-driven interactions with a REST API are for retrieving the hyperlinks rather than the resource state. These hyperlinks are more stable than the resource states, especially in SDN. Therefore, caching stable hyperlinks with unstable states in the same way would cause unnecessary cache invalidations. For example, as shown in Figure 10 (a), to add a new network, clients follow the create hyperlink in the network network_list representation. Although this hyperlink remains constant, the entire representation may become invalid when a network resource is added or removed.

A closer examination of SDN REST API shows that a resource representation typically contains three types of disjoint content:

1. entity hyperlinks that point to unstable entity resources;
2. action hyperlinks that point to stable action resources;
3. elements describing the state of a resource.



(a) A classical Web cache contains an entire representation, including attributes, elements and hyperlinks



(b) Our hyperlink cache contains only the stable hyperlinks.

Figure 10. The differences between a classical Web cache and our hyperlink cache.

For example, as shown in Figure 8, the network hyperlink is an entity hyperlink pointing at a network resource, while other hyperlinks are action hyperlinks providing various services related to the network. In general, the action hyperlinks are defined at design time and tend to be stable. In contrast, the entity hyperlinks tend to change with the resource state. For example, the create action hyperlink in Figure 8 is always present in the network_list representation, whereas the network entity hyperlink will change as networks are added or removed.

If we apply the same cache control to the representation, we are likely to cache the entity hyperlinks too long but the action hyperlinks too short. A cache saved too long will not only waste storage, but also give clients incorrect representations about the resources, whereas a cache saved too short will cause undue misses and decrease the performance.

These observations suggest that we should cache entity hyperlinks, action hyperlinks, and elements differently to maximize the client performance while reducing the rate of cache invalidation.

Saving the complete hyperlinks in cache may waste a lot of spaces if a large number of hyperlinks are instances of a few URI templates. Applying the idea of Generator Pattern in Section V to caches, we only save the distinct URI templates and the variable values. Since in our system, a common URI template and a distinct resource ID will be sufficient to identify most resources, we can save cache space by maintaining a mapping from the distinct IDs to URI templates.

B. Differential and Layered Cache at Clients

Our solution to cache entity hyperlinks, action hyperlinks and elements differently is a *differential and layered cache* mechanism. The differential cache control is a fine-grained cache control that allows a REST API to specify which elements of a representation can be cached for how long, which is in addition to representation level HTTP 1.1 cache control. The proposed differential cache control adapts HTTP 1.1 max-age cache directive [44] to individual elements of a representation as a special attribute. For example, the `cache-control` attribute in the link element: `<link cache-control="max-age:{delta}" />` indicates the link element can be cached for {delta} seconds, which overwrites any cache control at the representation level. When applied to an element with children, the attribute indicates that the children share the same cache control, unless a child element has its own cache control.

This approach is domain and language independent as it can specify cache controls in different mark-up languages at different granularity. For this reason, the differential caches can be maintained by clients or trusted proxies who have access to the representations.

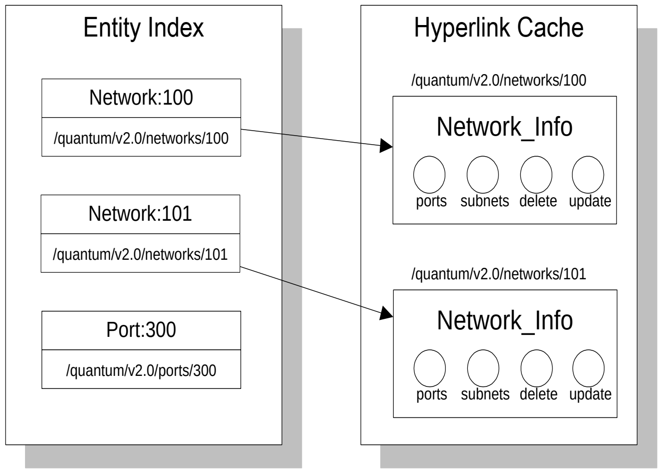


Figure 11. The layered caches: bookmark, hyperlink and representation.

Based on the differential cache control, the action hyperlinks are cached in the hyperlink cache, which contains *rel→(hyperlink, expiry)* mappings, where the hyperlink points to entries in the representation cache, which contains *hyperlink→(representation, expiry)* mappings.

The entity hyperlinks are saved in a bookmark style cache, called Entity Index. The Entity Index table saves *(rel, ID)→(hyperlink, expiry)* mappings, where the *rel,* ID and *hyperlink* are extracted from the *rel*, *id* and *href* attributes of entity hyperlinks respectively. The Entity Index acts like bookmarks that offer clients shortcuts to access the resources. Combined with the hyperlink cache, clients can perform operations on the entity resources with much fewer interactions.

These caches are populated and updated at clients according to the differential cache controls and the regular HTTP 1.1 cache controls, whenever a fresh representation is retrieved from the origin REST API server.

Figure 11 shows an example of the Entity Index with three entries extracted from the representations of the previous hypertext-driven interactions. To update the network of ID 100, the client first constructs the resource URI from the Entity Index and then uses this URI to lookup the updated hyperlink from the hyperlink cache. As all these operation happens at the caches, the client can find the URI to the target resource without any interaction with the REST API.

C. Cache Replacement

To prevent the cache overflow, new entries should replace old entries when the cache reaches the size limit. The problem of cache replacement has been well-studied. The efficiency of the replacement algorithm depends heavily on the workload. Finding the most efficient cache replacement algorithm is out of the scope of this paper, we consider that the classical LRU (least recently used) algorithm should match the data locality well for many cases.

Furthermore, the sizes of entries in the Entity Index are mostly very similar. This allows the lightweight LRU type algorithm to perform well rather than having to adopt more complex non-uniform size cache replacement algorithms, such as Greedy Dual-Size (GDS) algorithm [48]. This observation also applies to the hyperlink cache, as its entries contain the static hyperlinks only, of which the size differs much less comparing to the case of including all the attributes and entities in the cached representations.

The hyperlink cache and Entity Index carry out cache replacement independently on their own buffers, because the size of an entry in the Entity Index is typically several times smaller than the size of an entry in the hyperlink cache.

VII. Implementation and Evaluations

We developed a Java tool to automatically generate JAX-RS compliant Java code from REST Chart [51] that returns canned messages with valid hyperlinks, such that a developer can deploy and navigate the REST API prototype without writing one line of code. At this stage, the REST Chart XML files were composed manually but validated automatically against its XML Schema. We hope to improve the REST Chart composition process in the future, by taking advantage of the graphical representations of Petri-Net.

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 [11]. We also implemented an OpenStack Neutron 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 [46]: JSON and XML. SOXProxy prefers JSON using content negotiation described in Section III, because the dict data type of Python fits JSON naturally.

Figure 12 illustrates the integrated system of OpenStack and

SOX in two physical machines. Each machine was 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 were interconnected by a local Gigabit LAN. The Round-Trip Time (RTT) between the two machines was less than 0.2ms on average.

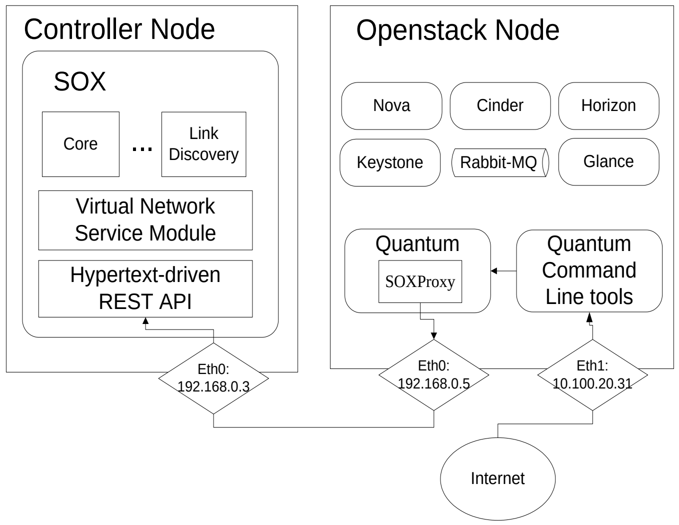


Figure 12. The evaluation system configuration.

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

We evaluated the northbound REST API of SOX in two ways: 1) transparent upgrade to OpenStack Neutron under REST API change; and 2) the performance and the cost comparison with 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 OpenStack Neutron without interrupting its service executions. This feature is critical, because otherwise, an update of the REST API or a network control switch-over from one set of controllers to another can result in a total service breakdown, a situation that is costly and disastrous for large scale distributed network systems. To demonstrate this critical feature in our approach, we prepared two versions A and B of SOX controllers, where A is modeled after the OpenStack Neutron REST API v1.0 and B after the REST API v2.0. The client first interacts with SOX-A using the Factory Pattern to create a port, and then switch over to SOX-B to delete the port using the Backtracking Pattern, while OpenStack Neutron is running and not aware of the switch-over.

Figure 13 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 URIs to their resources. Since Neutron backtracks to 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.

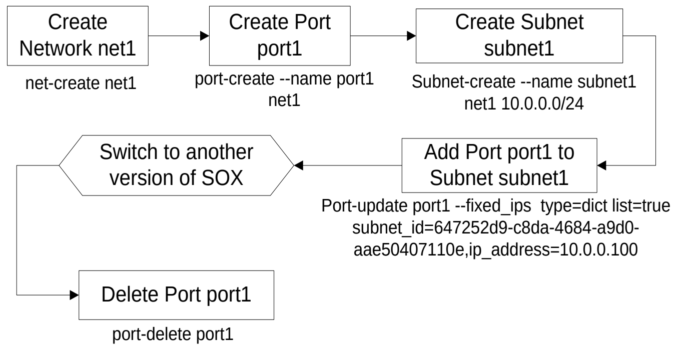

Create Network net1
net-create net1
Create Port port1
port-create --name port1 net1
Create Subnet subnet1
Subnet-create --name subnet1 net1 10.0.0.0/24
Add Port port1 to Subnet subnet1
Port-update port1 --fixed_ips type=dict list=true subnet_id=647252d9-c8da-4684-a9d0-aae50407110e,ip_address=10.0.0.100
Switch to another version of SOX
Delete Port port1
port-delete port1

Figure 13. The switch-over sequence and CLI commands

The following tables (Table III and Table IV) recorded the URI paths traversed by Neutron before and after the switch-over. If the REST API exposed fixed URIs, Neutron will get HTTP 404 after the switch-over.

TABLE III. URIs ACCESSED TO CREATE PORT1 IN SOX-A.

	Accessed URI
1	GET /Neutron/v2.0/**tenants**/46f684a7-dcc0-478c-b4a8-313d4f768344
2	GET /Neutron/v2.0/**tenants**/46f684a7-dcc0-478c-b4a8-313d4f768344 /**networks**/93d3c03f-5199-48e8-8596-fd365dc8e273/**ports**/7a02dea5-41e7-48f4-8de2-3e3ba7c24f08
3	GET /Neutron/v2.0/**tenants**/46f684a7-dcc0-478c-b4a8-313d4f768344 /**networks**/93d3c03f-5199-48e8-8596-fd365dc8e273/**ports**/7a02dea5-41e7-48f4-8de2-3e3ba7c24f08/**update**
4	POST /Neutron/v2.0/**tenants**/46f684a7-dcc0-478c-b4a8-313d4f768344 /**networks**/93d3c03f-5199-48e8-8596-fd365dc8e273/**ports**/7a02dea5-41e7-48f4-8de2-3e3ba7c24f08/**update**

TABLE IV. URIs ACCESSED TO DELETE PORT1 IN SOX-B.

	Accessed URI
1	GET /Neutron/v2.0/**tenants**/46f684a7-dcc0-478c-b4a8-313d4f768344
2	GET /Neutron/v2.0/**tenants**/46f684a7-dcc0-478c-b4a8-313d4f768344/**ports**/7a02dea5-41e7-48f4-8de2-3e3ba7c24f08
3	DELETE /Neutron/v2.0/**tenants**/46f684a7-dcc0-478c-b4a8-313d4f768344/**ports**/7a02dea5-41e7-48f4-8de2-3e3ba7c24f08

B. Performance Evaluation

To evaluate the performance, we deployed an emulator in the OpenStack machine to simulate the workload from OpenStack Neutron as shown in Figure 14.

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 to create, retrieve, update or delete a network at 500ms intervals to the controller and recorded the average response times of all requests at 5 minutes intervals.

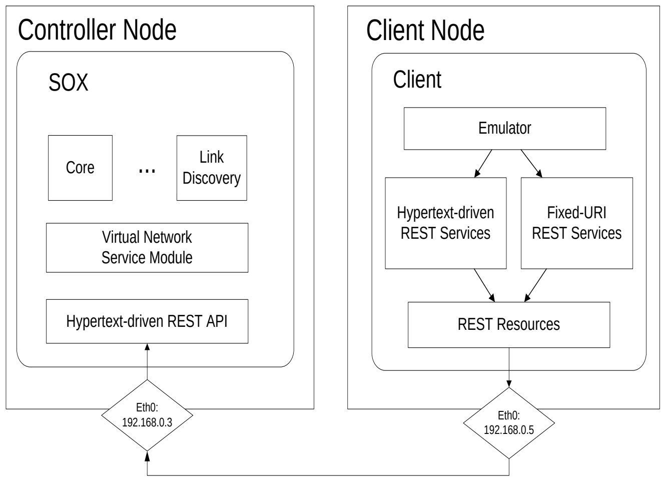


Figure 14. Experiment setup for client-side hyperlink caching.

Under this test plan, we evaluated REST API performance under four client configurations:

1. Fixed URI: the emulator accesses the target resources directly by URIs obtained through out-of-band mechanism without using any cache (this configuration violates REST as pointed out in Section III, but it is used here only as a performance benchmark because it represents the minimal number of client-server interactions).
2. No cache: the emulator does not use any cache and always navigates to the target resources from the initial REST API entry URI to the target goal place.
3. Cache only: the emulator uses the hyperlink cache during the interactions.
4. Cache and index: the emulator uses both hyperlink cache and bookmark cache (Entity Index) during the interactions.

Figure 15 shows the average round-trip times measured at the emulator under the above four configurations. In Figure 15, the "fixed URI" configuration achieved the best performance, with an overall average response time of 10.35ms. In contrast, the "No cache" performed the worst with an overall average response time of 46.82ms, about 4.52 times of "Fixed URI" configuration. When using "Cache and index", the average response time was 15.80ms, only 1/3 of the average "No cache" configuration response time, and an increase of 5ms compared to the "Fixed URI" configuration. The "Cache only" configuration achieved 27.92ms, which increased the response time of the "Cache and index" configuration by 12ms or about 76%. Overall, the proposed "Cache and Index" approach reduced the response time of "No cache" by 66% (from 46.82ms down to 15.8ms).

These tests show that without cache, the performance of the REST API is not acceptable, but with the proposed differential caches, the performance was very close to the optimal (but not RESTful) performance achieved by the "Fixed URI" configuration.

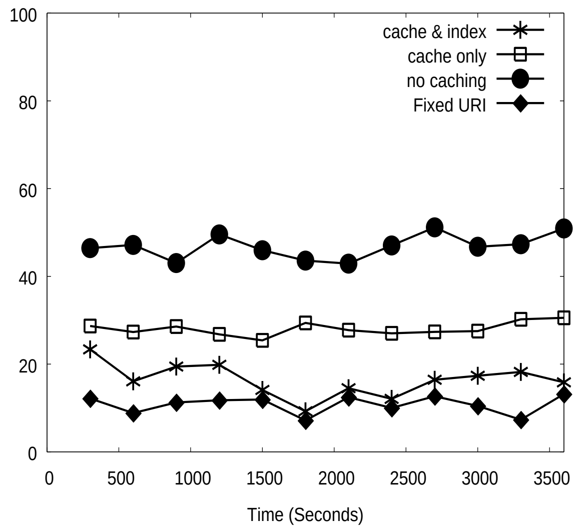


Figure 15. Average round-trip response time in a 5 minute interval.

To find out reasons behind this speed-up in response time, we measured the average number of interactions needed to complete an operation under the 4 test configurations, and the results are shown in TABLE V.

TABLE V. AVERAGE NUMBER OF INTERACTIONS PER OPERATION.

Configuration	Avg. No. of Interactions
Fixed URI	1.37
No Cache	3.98
Cache Only	2.68
Cache and Index	1.70

Noted that to create and update entities in any configuration, the client must retrieve the form first, this is why even the "Fixed URI" configuration had greater than 1 average number of interactions. The "Cache and Index" saved about half of the interactions compared to "No cache" configuration, which contributed to the significant speed-up in response time. Compared to the "Fixed URI" configuration, the "Cache and Index" configuration increased the average number of interactions by less than 40%, which explains why it only increased the response time slightly (5ms). Compared to No Cache, the Cache and Index approach reduces the overhead by 57% of interactions.

VIII. CONCLUSION

In this paper, we presented algorithms and methods to materialize the extensibility of REST API under hypertext-driven navigation. Firstly, we described a Petri-Net based REST Chart framework that provides a formal model to extend, change, and update the REST API without breaking its clients. Secondly, we described and characterized a set of important hypertext-driven design patterns that facilitate the hypertext-driven navigation for a REST API as well as the implementation of its clients. Thirdly, we presented a novel differential cache mechanism that can significantly reduce the overhead of clients in hypertext-driven navigation.

The proposed approaches have been successfully applied to the RESTful design of a northbound API of SDN in the context of cloud computing using OpenStack. It overcame some critical REST API design and performance deficiencies in previous approaches. Moreover, we demonstrated the advantages of the proposed hypertext-driven REST API approach through a transparent migration between two versions of a RESTful SDN APIs in OpenStack without interrupting its service executions, a critical feature for large scale distributed systems, such as cloud computing and SDN. To improve the system efficiency, the proposed differential layered cache mechanism was applied to SDN applications to manage the data networking. The performance evaluation results showed that, to reach the same target resources, a REST client based on the proposed approach on average reduced the overhead of the hypertext-driven navigation by 66%, and the average response time could be maintained well below 20ms in the tested networking applications.

REFERENCES

[1] N. 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.

[2] Open Networking Foundation, OpenFlow Switch Specification, Version 1.5.0, December 19, 2014, ONF TS-020.

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

[4] Leonard Richardson, Sam Ruby, RESTful Web Services, O'Reilly, 2007.

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

[6] OpenStack API Complete Reference: http://developer.openstack.org/api-ref.html.

[7] OpenStack Releases: https://wiki.openstack.org/wiki/Releases, last access: 12/16/2014.

[8] Docker Remote REST API: https://docs.docker.com/reference/api/docker_remote_api/.

[9] OpenStack Networking API: http://developer.openstack.org/api-ref-networking-v2.html.

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

[11] Min Luo, Yingjun Tian, Quancai Li, Jiao Wang, and Wu Chou: SOX - a generalized and extensible smart network openflow controller, The First SDN World Summit, Germany, October 2012.

[12] D. Levi et al (ed): Simple Network Management Protocol (SNMP) Applications, RFC3413, https://tools.ietf.org/html/rfc3413, 2002.

[13] R. Enns et al (ed): Network Configuration Protocol (NETCONF), RFC6421, https://tools.ietf.org/html/rfc6241, 2011.

[14] 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.

[15] 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.

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

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

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

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

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

[21] 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.

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

[23] 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.

[24] 20. 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.

[25] Erik Christensen, et al (ed): Web Services Description Language (WSDL) 1.1, W3C Note, 15 March 2001, http://www.w3.org/TR/wsdl.

[26] Marc Hadley, Web Application Description Language, W3C member Submission, 31, August 2009, http://www.w3.org/Submission/wadl/

[27] RAML Version 0.8: http://raml.org/spec.html

[28] Swagger 2.0: https://github.com/swagger-api/swagger-spec

[29] Jonathan Robie, Rob Cavicchio, Rémon Sinnema, Erik Wilde: RESTful Service Description Language (RSDL), Describing RESTful Services Without Tight Coupling, Balisage: The Markup Conference 2013, http://www.balisage.net/Proceedings/vol10/html/Robie01/BalisageVol10-Robie01.html

[30] API Blueprint Format 1A revision 7: https://github.com/apiaryio/api-blueprint/blob/master/API%20Blueprint%20Specification.md

[31] Rosa Alarcon, Erik Wilde: Linking Data from RESTful Services, LDOW 2010, April 27, 2010, Raleigh, North Carolina.

[32] Jonathan Robie: RESTful API Description Language (RADL), https://github.com/restful-api-description-language/RADL, 2014.

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

[34] Christos G. Cassandras and Stephane Lafortune, Introduction to Discrete Event Systems, Springer, 2008.

[35] Tado Murata: Petri Nets: Properties, Analysis and Applications, Proceedings of the IEEE, Vol. 77, No. 4, April 1989.

[36] Li Li, Tony Tang, Wu Chou: Designing Large Scale REST APIs Based on REST Chart, IEEE ICWS 2015, pages 631-638, New York, New York, USA, June 27-July 2, 2015.

[37] M. Bjorklund (ed): YANG - A Data Modeling Language for the Network Configuration Protocol (NETCONF), RFC6020, 2010, https://tools.ietf.org/html/rfc6020.

[38] A. Bierman, et al (ed): RESTCONF Protocol, draft-ietf-netconf-restconf-08, October 2015, https://tools.ietf.org/html/draft-ietf-netconf-restconf-08.

[39] OpenDaylight Platform: https://www.opendaylight.org/.

[40] J. Gregorio et al (ed): URI Template, RFC6570, 2012, https://tools.ietf.org/html/rfc6570.

[41] Li Li, Wu Chou: Compatibility Modeling and Testing of REST API Based on REST Chart, WEBIST 2015 pages 194-201, Lisbon, Portugal, May 20-22, 2015.

[42] Li Li, Wu Chou: Finding Optimal REST Service Oracle Based on Hierarchical REST Chart, Service Computation 2015, pages 21-26, Nice, France, March 22-27, 2015.

[43] Li Li, Wu Chou: Design Patterns for RESTful Communication Web Services, ICWS 2010, pages 512-519, Miami, 5-10 July 2010.

[44] R. Fielding et al, Hypertext Transfer Protocol -- HTTP/1.1, RFC 2616, section 12, June 1999.

[45] Li Li, Tony Tang, Wu Chou: A XML Based Monadic Framework for REST Service Compositions, IEEE ICWS 2015, pages 487-494, New York, New York, USA, June 27-July 2, 2015.

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

[47] Rainer Fehling: A Concept of Hierarchical Petri Nets with Building Blocks, 12th International Conference on Applications and Theory of Petri Nets: Advances in Petri Nets, Pages 148-168, 1993.

[48] Ludmila Cherkasova, Improving WWW Proxies Performance with Greedy-Dual-Size-Frequency Caching Policy, Computer Systems Laboratory HPL-98-69 (R.1), November, 1998.

[49] Wei Zhou, Li Li, Min Luo, Wu Chou: REST API Design Patterns for SDN Northbound API, The 28th IEEE International Conference on Advanced Information Networking and Applications Workshops (AINA-2014), pages 358-365, Victoria, BC, Canada, May 13-16, 2014.

[50] R. Enns et al (ed): Network Configuration Protocol (NETCONF), RFC6241, June 2011, https://tools.ietf.org/html/rfc6241.

[51] Li Li, Tony Tang, and Wu Chou: Automated Creation of Navigable REST services Based on REST Chart, Journal of Advanced Management Science, Vol. 4, No. 5, September 2016, pp. 385-392.

Authors

Dr. Li joined Huawei Shannon IT Lab in 2012 as a principal engineer and his current research interest includes Web services, cloud computing and software-defined networking. He has published over 70 conference and journal papers and coauthored a book on Artificial Intelligence. He currently holds 12 US and international patents. Dr. Li is a member of IEEE and ACM, and he was the editor of 2 ISO/ECMA CSTA III standards and made significant contributions to W3C WS-RA standard suite. Dr. Li received his Ph.D. in computer science from University of Alabama at Birmingham, USA in 1995, M.S. in computational linguistics in 1987 and B.S. in computer science in 1984 from Huazhong University of Sciences and Technology, China.

Dr. Wu Chou is an IEEE Fellow, VP, Chief Technology Officer of Switch and Enterprise Communications at Huawei. He obtained his Ph.D. in Electrical Engineering from Stanford University. After graduated from Stanford with four advanced degrees in science and engineering, he continued his professional career from AT&T Bell Labs to Lucent Bell Labs and Avaya Labs before joining Huawei as the head of Huawei Shannon (IT) Lab. He is an expert in the field of IT, Cloud computing, data networking, SDN/NFV, Internet-of-things (IoT), Big Data, communication, Internet/Web, machine learning, signal processing, speech and natural language processing, multimedia and multimodal interaction, service computing, and Web services. He published over 200 journal and conference papers, holds 42 US and international patents with many additional patent applications pending. He served as an editor and area expert for multiple international standards at W3C, ECMA, ISO, ETSI, etc. He was an editor of IEEE Transactions on Services Computing (TSC), IEEE TSC Special Issue on Cloud Computing, IEEE Signal Processing Special Issue on Speech Technology and Systems in Human-machine Interaction, IEEE Transactions on Audio and Language Processing, and Journal of Web Services Research.

Dr. Zhou Wei is currently a postdoctoral fellow in Data61 of CSIRO, Australia. His research interests include database system, distributed system and cloud computing. He served as a research engineer at Shannon IT Lab of Huawei from 2012 to 2013. He obtained his Ph.D. in computer science from VU University Amsterdam in 2012. He received a M.E. from Tsinghua University in software engineering in 2009 and B.E from Beihang University in computer science and technology in 2003.

Dr. Min Luo is currently the Head and Chief Architect of the Advanced Networking at Huawei’s Shannon (IT) Lab, leading the research and development in Software Defined Networking (SDN) and other future networking initiatives. He served as Chief Architect for IBM’s Global Business Solution Center –GCG, Industry Solutions, and Center of Excellence for Enterprise Architecture and SOA for more than 11 years. He also worked as Senior Operations Research Analyst, Senior Manager and Director of Transportation Network Planning and Technologies for two Fortune 500 companies for 7 Years. He is a senior member of IEEE, he has been serving on the organizing committee for IEEE’s ICWS and SCS/CC Conferences, chaired sessions, presented several tutorials on SOA and Enterprise Architecture and their best practices and gave lectures at the Service University. He has served as adjunct professors in several US and Chinese universities since 1996. He obtained his Ph.D. in Electrical Engineering from Georgia Institute of Technology in 1992. He also held a M.S. (1987) and B.S (1982) in Computer Science.