SDNRG
Internet-Draft
Intended status: Standards Track
Expires: November 5, 2015

Y. Xia, Ed.
S. Jiang, Ed.
T. Zhou, Ed.
S. Hares
Huawei Technologies Co., Ltd
May 4, 2015

NEMO (NEtwork MOdeling) Language
draft-xia-sdnrg-nemo-language-02

Abstract

The North-Bound Interface (NBI), located between the control plane and the applications, is essential to enable the application innovations and nourish the eco-system of SDN.

While most of the NBIs are provided in the form of API, this document proposes the NEtwork MOdeling (NEMO) language which is intent based interface with novel language fashion. Concept, model and syntax are introduced in the document.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at http://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on November 5, 2015.

Copyright Notice

Table of Contents

1. Introduction

While SDN (Software Defined Network) is becoming one of the most important directions of network evolution, the essence of SDN is to make the network more flexible and easy to use. The North-Bound Interface (NBI), located between the control plane and the applications, is essential to enable the application innovations and nourish the eco-system of SDN by abstracting the network capabilities/information and opening the abstract/logic network to applications.

The NBI is usually provided in the form of API (Application Programming Interface). Different vendors provide self-defined API sets. Each API set, such as OnePK from Cisco and OPS from Huawei, often contains hundreds of specific APIs. Diverse APIs without consistent style are hard to remember and use, and nearly impossible to be standardized.

In addition, most of those APIs are designed by network domain experts, who are used to thinking from the network system perspective. The interface designer does not know how the users will use the device and exposes information details as much as possible. It enables better control of devices, but leaves huge burden of selecting useful information to users without well training. Since the NBI is used by network users, a more appropriate design is to express user intent and abstract the network from the top down.

To implement such an NBI design, we can learn from the successful case of SQL (Structured Query Language), which simplified the complicated data operation to a unified and intuitive way in the form of language. Applications do not care about the way of data storage and data operation, but to describe the demand for the data storage and operation and then get the result. As a data domain DSL, SQL is simple and intuitive, and can be embedded in applications. So what we need for the network NBI is a set of "network domain SQL". [I-D.xia-sdnrg-service-description-language] describe the requirements for a service description language and the design considerations.

This document will introduce an intent based NBI with novel language fashion.

2. Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC2119] when they appear in ALL CAPS. When these words are not in ALL CAPS (such as "should" or "Should"), they have their usual English meanings, and are not to be interpreted as [RFC2119] key words.

Network service also "service" for short, is the service logic that contains network operation requirements;

Network APP also "APP" for short, is the application to implement the network service;

Network user also "user" for short, is the network administrator or operator.

3. Requirements for the Intent Based NBI Language

An intent based NBI language design contains following features:

- Express user intent

  To simplify the operation, applications or users can use the NBI directly to describe their requirements for the network without taking care of the implementation. All the parameters without user concern will be concealed by the NBI.

- Platform independent

  With the NBI, the application or user can description of network demand in a generic way, so that any platform or system can get the identical knowledge and consequently execute to the same result. Any low-level and device/vendor specific configurations and dependencies should be avoided.

- Intuitive Domain Specific Language (DSL) for network

  The expression of the DSL should be human-friendly and be easily understood by network operators. DSL should be directly used by the system.

- Privilege control

  Every application or user is authorized within a specific network domain, which can be physical or virtual. While different network domains are isolated without impact, the application or user may have access to all the resource and capabilities within its domain. The user perception of the network does not have to be the same as the network operators. The NBI language works on the user’s view so the users can create topologies based on the resources the network-operators allow them to have.

- Declarative style

  As described above, the NBI language is designed to help defining service requirement to network, detailed configurations and instructions performed by network devices are opaque to network operators. So the NBI language should be declarative rather than imperative.

4. Related work

YANG [RFC6020] is a data modeling language used to model configuration and state data manipulated by the Network Configuration Protocol (NETCONF) [RFC6241], NETCONF remote procedure calls, and NETCONF notifications.

UML (Unified Modeling Language) is a powerful modeling language, which is domain agnostic. YANG and UML all focus on syntax specification which formulate grammatical structure of NBI language, however, they do not have the ability to express users' real semantics. NBI language should facilitate users to express their own intent explicitly, instead of general complying with grammar syntax. So YANG and UML is appropriate to describe the model behind the NBI language not the NBI itself.

With the emergence of the SDN concept, it is a consensus to simplify the network operation, which leads to many cutting-edge explorations in the academic area.

Nick McKeown from Stanford University proposed the SFNet [TSFNet], which translated the high level network demand to the underlying controller interfaces. By concealing the low level network details, the controller simplified the operation of resource, flow, and information for applications. The SFNet is used for the SDN architecture design, and does not go into the NBI design.

Jennifer from Princeton University designed the Frenetic [Frenetic] based on the OpenFlow protocol. It is an advanced language for flow programming, and systematically defines the operating model and mode for the flow. However, the network requirement from the service is not only the flow operations, but also includes operations of resource, service conditions, and service logic.

In the book [PBNM], John Strassner defined the policy concept and proposed the formal description for network operations by using the policy. The method for querying network information is absent in the book. Virtual tenant network and operations to the tenant network are not considered.

All these investigations direct to the future SDN that use simple and intuitive interfaces to describe the network demands without complex programming.

5. The NEMO Language Specifications

NEMO language is a domain specific language (DSL) based on abstraction of network models and conclusion of operation patterns. It provides NBI fashion in the form of language. Instead of tons of abstruse APIs, with limited number of key words and expressions, NEMO language enables network users/applications to describe their demands for network resources, services and logical operations in an intuitive way. And finally the NEMO language description can be explained and executed by a language engine.

5.1. Network Model of the NEMO Language

Behind the NEMO language, there is a set of basic network models abstracting the network demands from the top down according to the service requirement. Those demands can be divided into two types: the demand for network resources and the demands for network behaviors.

The network resource is composed of three kinds of entities: node, connection and flow. Each entity contains property and statistic information. With a globally unique identifier, the network entity is the basic object for operation. Users can construct their own topology or network traffic arbitrarily with these basic objects without considering about real physical topology. In addition, NEMO Engine also has the ability of obtaining available resources automatically as operation objects when users don’t define them.

- Node model: describes the entity with the capability of packet processing. According to the functionality, there are two types of node
  - The function node (FN) provides network services or forwarding with user concern, such as, firewall, load balance, vrouter, etc.
  - The business node (BN) describes a set of network elements and their connections, such as subnet, autonomous system, and internet. It conceals the internal topology and exposes properties as one entity. It also enables iteration, i.e., a network entity may include other network entities.
- Connection model: describes the connectivity between node entities. This connection is not limited at the connectivity between single entity and single entity, but it also can express the connectivity between single entity and multiply entities, or multiply entities and multiply entities.

- Flow model: describes a sequence of packets with certain common characters, such as source/destination IP address, port, and protocol. From the northbound perspective, flow is the special traffic with user concern, which may be per device or across many devices. So the flow characters also include ingress/egress node, and so on.

Network behavior includes the information and control operations.

The information operation provides two methods to get the network information for users.

- Query: a synchronous mode to get the information, i.e., one can get the response when a request is sent out.

- Notification: an asynchronous mode to get the information, i.e., with one request, one or multiple responses will be sent to the subscriber automatically whenever trigger conditions meet.

The NEMO language uses policy to describe the control operation.

- Policy: control the behavior of specific entities by APP, such as flow policy, node policy. All the policies follow the same pattern "when <condition>, to execute <action>, with <constraint>", and can be applied to any entity. But some of policy elements can be omitted according to users' requirement.

5.2. Notation

The syntactic notation used in this specification is an extended version of BNF ("Backus Naur Form" or "Backus Normal Form"). In BNF, each syntactic element of the language is defined by means of a production rule. This defines the element in terms of a formula consisting of the characters, character strings, and syntactic elements that can be used to form an instance of it. The version of BNF used in this specification makes use of the following symbols:

< >

Angle brackets delimit character strings that are the names of syntactic elements.

::=

The definition operator. This is used in a production rule to separate the element defined by the rule from its definition. The element being defined appears to the left of the operator and the formula that defines the element appears to the right.

[ ]

Square brackets indicate optional elements in a formula. The portion of the formula within the brackets may be explicitly specified or may be omitted.

{ }

Braces group elements in a formula. The portion of the formula within the braces shall be explicitly specified.

|

The alternative operator. The vertical bar indicates that the portion of the formula following the bar is an alternative to the portion preceding the bar. If the vertical bar appears at a position where it is not enclosed in braces or square brackets, it specifies a complete alternative for the element defined by the production rule. If the vertical bar appears in a portion of a formula enclosed in braces or square brackets, it specifies alternatives for the contents of the innermost pair of such braces or brackets.

!!

Introduces ordinary English text. This is used when the definition of a syntactic element is not expressed in BNF.

5.3. NEMO Language Overview

NEMO language provides 5 classes of commands: model definition, resource access, behavior, connection management, transaction to facilitate the user intent description.

```
<NEMO_cmd> := <model_definition_cmd> | <resource_access_cmd> |
              <behavior_cmd>
<model_definition_cmd> := <node_definition> | <connection_difinition> |
                          <action_deifinition> | <model_description>
<resource_access_cmd> := <node_cu> | <node_del> | <connection_cu> |
                         <connection_del> | <flow_cu> | <flow_del>
<behavior_cmd> := <query_cmd> | <policy_cu> | <policy_del> |
                  <notification_cu> | <notification_del>
<connection_mgt_cmd> := <connect_cmd> | <disconnect_cmd>
<transaction_cmd> := <transaction_begin> | <transaction _end>
```

NEMO language provides limited number of key words to enables network users/applications to describe their intent. The key words supported by the language are as follows:

```
 <key_word> := Boolean | Integer | String | Date | UUID | EthAddr |
               IPPrefix | NodeModel | ConnectionModel | FlowModel |
               ActionModel | Description | Porperty | Node | Connection|
               Flow | No | EndNodes | Type | NW | Match | List |
               Range| Query | From | Notification | Listener |
               Policy | ApplyTo | Priority | Condition | Action |
               Connect | Disconnect | Address | Port | Transaction |
               Commit
```

5.4. Model Definition

5.4.1. Data Types

NEMO language provides several build-in data types:

Boolean This data type is used for simple flags that track true/ false conditions. There are only two possible values: true and false. The Boolean literal is represented by the token <boolean>.

Integer A number with an integer value, within the range from -(2^63) to +(2^63)-1. The Integer literal is represented by the token <integer>.

String A sequence of characters. The string is always in the quotation marks. The String literal is represented by the token <string>.

Date A string in the format yyyy-mm-dd hh:mm:ss, or yyyy-mm-dd, or hh:mm:ss. The Date literal is represented by the token <date>.

UUID A string in the form of Universally Unique IDentifier [RFC4122], e.g. "6ba7b814-9dad-11d1-80b4-00c04fd430c8". A typical usage of the UUID is to identify network entities, policies, actions and so on. The UUID literal is represented by the token <UUID>.

EthAddr A string in the form of MAC address, e.g. "00:00:00:00:00:01". The EthAddr literal is represented by the token <eth_addr>.

IPPrefix A string in the form of IP address, e.g. "192.0.2.1". The mask can be used in the IP address description, e.g. "192.0.2.0/24". The IPPrefix literal is represented by the token <ip_prefix>.

The token <data_type> can be defined as follows:

```
<data_type> := Boolean | Integer | String | Date | UUID |
               EthAddr | IPPrefix
```

And a generic <data_type> literal is represented by the token <value>

```
<value> := <boolean> | <integer> | <string> | <date> | <UUID> |
           <eth_addr> | <ip_ prefix>
```

5.4.2. Model Definition and Description Statement

In addition to default build-in network models, NEMO language facilitates users to define new model types.

The token <naming> is a string that MUST start with a letter and followed by any number of letters and digits. More specific naming can be defined as follows:

```
<node_type> := <naming> !!type name of the node model
<connection_type> := <naming> !!type name of the connection model
<flow_type> := <naming> !!type name of the flow model
<entity_type> := <node_type> | <connection_type> | <flow_type>
<action_type> := <naming> !!type name of the action model
<model_type> := <entity_type> | <action_type>
<property_name> := <naming> !!name of the property in a model
```

The <node_definition> statement is used to create a node model:

```
<node_definition> := NodeModel <node_type>
                     Property { <data_type> : <property_name> };
```

The NodeModel specifies a new node type.

The Property is followed by a list of "<data_type> : <property_name>" pairs to specify properties for the new node type. Since belonging network is the intrinsic property for a node model, there is no need to redefine the belonging network in the property list.

Example:

NodeModel "DPI" Property String : "name", Boolean : "is_enable"; The statement generates a new node model named DPI with two properties, "name" and "is_enable".

The <connection_definition> statement is used to create a connection model:

```
<connection_definition> := ConnectionModel <connection_type>
                   Property { <data_type> : <property_name> };
```

The ConnectionModel specifies a new connection type.

The Property is followed by a list of "<data_type> : <property_name>" pairs to specify properties for the new connection type. Since end nodes are intrinsic properties for a connection model, there is no need to redefine the end nodes in the property list.

The <flow_definition> statement is used to create a flow model:

```
<flow_definition> := FlowModel <flow_type>
                  Property { <data_type> : <property_name> };
```

The FlowModel specifies a new flow type.

The Property is followed by a list of "<data_type> : <property_name>" pairs to specify fields for the new flow type. The <action_definition> statement is used to create an action model:

```
<action_definition> := ActionModel <action_type>
                    Property  { <data_type> : <property_name> };
```

The ActionModel specifies a new action type.

The Property is followed by a list of "<data_type> : <property_name>" pairs to specify properties for the new action.

NEMO language also supports querying the description of a defined model by using the <model_description> statement:

```
<model_description> := Description <model_type>;
```

The keyword Description is follow by a model type name. The description of the queried model will return from the language engine.

5.5. Resource Access Statements

In NEMO language, each resource entity instance is identified by a <UUID>. We use the following token to indicate the identifier given to the resource entity instance.

```
<node_id> := <naming> !! name to identify the node instance
<connection_id> := <naming> !! name to identify the connection instance
<flow_id> := <naming> !! name to identify the flow instance
<entity_id> := <node_id>|<connection_id>|<flow_id>
```

5.5.1. Node Operations

The <node_cu> statement is used to create or update a node instance:

```
<node_cu> := Node <node_id> Type  <node_type>
                            NW <node_id>
                            [Property {<property_name>: <value>}];
```

The Node is followed by a user specified <node_id>. If the <node_id> is new in the system, a new node will be created automatically. Otherwise, the corresponding node identified by <node_id> will be updated with the following information.

The Type specifies the type of the node to operate.

The NW specifies the dependence where the node is located.

The Property is an optional keyword followed by a list of "<property_name>: <value>" pairs. Multiple "<property_name>: <value>" pairs are separated by commas. The <property_name> MUST be selected from the property definition of the corresponding node definition.

```
Node "Headquater"
     Type      "logicnw"
     NW        "LN-1"
     Property  "location" : "Beijing";
```

The statement creates a switch type node that is located in the logical network "LN-1".

The <node_del> statement is used to delete a node instance:

```
<node_del> := No Node <node_id>;
```

The No Node is to delete a node in user’s network.

5.5.2. Connection Operations

The <connection_cu> statement is used to create or update a connection:

```
<connection_cu> := Connection <connection_id>
             EndNodes <node_id>, <node_id>
             [Property {<property_name>: <value>}];
```

The Connection is followed by a user specified <connection_id>. If the <connection_id> is new in the system, a new connection will be

created automatically. Otherwise, the corresponding connection identified by the <connection_id> will be updated with the following information.

The EndNodes specifies the two end nodes, identified by "<node_type> : <node_id>", of a connection. The Property is an optional keyword followed by a list of "<property_name>: <value>" pairs. Multiple "<property_name>: <value>" pairs are separated by commas. The <property_name> MUST be selected from the property definition of the corresponding connection definition.

Example:

```
Connection "connection-1"
     EndNodes "S1", "S2"
     Property "bandwidth" : 1000, "delay" : 40;
```

The statement creates a connection between two nodes, and sets the connection property.

The <connection_del> statement is used to delete a node instance:

```
<connection_del> := No Connection <connection_id>;
```

The No Connection is to delete a connection in user’s network.

5.5.3. Flow Operations

The <flow_cu> statement is used to create or update a flow:

```
<flow_cu> := Flow <flow_id> Match {<property_name>: <value>
                                  | Range (<value>, <value>)
                                  | List({<value>})}
```

The Flow is followed by a user defined <flow_id>. If the <flow_id> is new in the system, a new flow will be created automatically. Otherwise, the corresponding flow identified by the <flow_id> will be updated with the following information.

The Match specifies a flow by indicate match fields. NEMO language also provides two keywords to assist the expression of values:

- The List is used to store a collection of data with the same data type.

- The Range is used to express a range of values.

Example:

```
Flow "flow-1"
      Match "src_ip" : Range ("192.0.2.1", "192.0.2.243");
```

The statement describes a flow with the source IP address ranging from 192.0.2.1 to 192.0.2.243.

The <flow_del> statement is used to delete a flow instance:

<flow_del> := No Flow <flow_id>;

The No Flow is to delete a flow in user's network.

5.6. Behavior Statements

5.6.1. Query Behavior

The query statement is to retrieve selected data from specified model object.

The <query_statement> generate a query:

```
<query_cmd> := Query {<property_name>}
               From {<entity_id>|<policy_id>}
```

The Query is followed by one or more <property_name>s which are defined properties of the object to be queried.

The From is followed by the one or more queried objects. NENO language support query operation to network entities and the policy.

5.6.2. Policy Behavior

In NEMO language, each policy instance is identified by a <naming>

<policy_id> := <naming> !! name to identify the policy instance

Create and update a policy

```
<policy_cu> := Policy <policy_id> ApplyTo <entity_id>
               Priority <integer>
               [Condition {<expression>}]
               Action {<action_type> : {<value>}}
               [Constraint {<expression>}];
```

The Policy is followed by a user defined <policy_id>. If the <policy_id> is new in the system, a new policy will be created automatically. Otherwise, the corresponding notification identified by the <policy_id> will be updated with the following information.

The ApplyTo specifies the entity to which the policy will apply.

The Priority specifies the globe priority of the policy in the tenant name space. The <value> with lower number has a higher priority, i.e. priority 0 holds the highest priority.

The Condition is an optional keyword follow by an expression. It tells your program to execute the following actions only if a particular test described by the expression evaluates to true. And users also can define which objects won't need to execute these actions with Constraint.

A NEMO language expression is a construct made up of variables, operators, and method invocations, which are constructed according to the syntax of the language and evaluates to a single value. NEMO language supports many operators to facilitate the construction of expressions. Assume variable A holds 10 and variable B holds 0, then:

Operator	Description	Example
&&	Called Logical AND operator. If both the operands are non-zero, then the condition becomes true.	(A && B) is false.
||	Called Logical OR Operator. If any of the two operands are non-zero, then the condition becomes true.	(A || B) is true.
!	Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true then Logical NOT operator will make false.	!(A && B) is true.
==	Checks if the values of two operands are equal or not, if yes then condition becomes true.	(A == B) is not true.
!=	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.	(A != B) is true.
>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	(A > B) is not true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	(A >= B) is not true.
<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	(A < B) is true.
<=	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	(A <= B) is true.

The Action specifies the execution when conditions meet.

Example:

```
Policy "policy-1"
        ApplyTo "flow-1"
        Priority 100
        Condition ("time">"18:00") || ("time"<"21:00")
        Action  "gothrough" : "backup_connection";
```

The statement creates a policy which indicates the flow to go through backup connection from 18:00 to 21:00.

Delete a policy:

<policy_del> := No Policy <policy_id>;

The No Policy is to delete a policy in user’s network.

5.6.3. Notification Behavior

In NEMO language, each notification instance is identified by a <naming>

```
<notification_id> := <naming> !! name to identify the notification
                     instance
```

Create and update a notification

```
<notification_cu> := Notification <notification_id>
                     [(Query {<property_name>}
                     From {<entity_id>})]
                     Condition {<expression>}
                     Listener <callbackfunc>;
```

The Notification is followed by a user defined <notification_id>. If the <notification_id> is new in the system, a new notificaiton will be created automatically. Otherwise, the corresponding notification identified by the <notification_id> will be updated with the following information.

The Query clause is nested in the notification statement to indicate the information acquisition.

The Condition clause is the same as in policy statement, which triggers the notification.

The Listener specifies the callback function that is used to process the notification.

Delete a notification:

<notification_del> := No Notification <notification_id>;

The No Notification is to delete a notification in user’s network.

5.7. Connection Management Statements

In NEMO language, each connection instance is identified by a <naming>

```
<conn_id> := <naming> !! name to identify the connection instance
```

Setup a connection to the NEMO language engine:

```
<connet_cmd> := Connect <conn_id> Address <ip_prefix>
                Port <integer>
```

The Connect is followed by a user defined <conn_id>. If the <conn_id> is new in the system, a new connection will be created automatically. Otherwise, the corresponding connection identified by <conn_id> will be updated with the following information.

The Address and Prot specify the IP address and the port of the NEMO language engine to connect separately.

Disconnect the connection to the NEMO language engine:

```
<disconnect_cmd> := Disconnect <conn_id>
```

The Disconnect is to remove the connection with an ID equals to <conn_id> from the NEMO language engine.

5.8. Transaction Statements

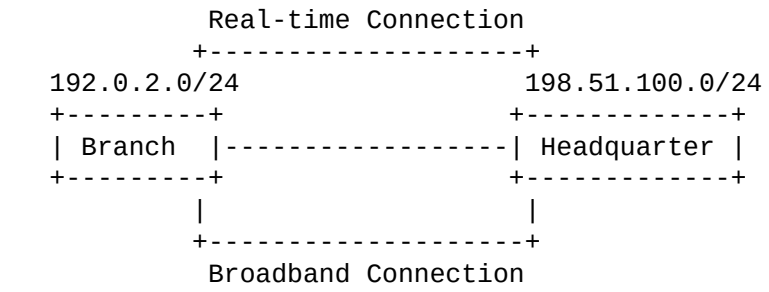
```
<transaction_begin> := Transaction
<transaction_end> := Commit
```

The keywords Transaction and Commit are used to tag begin and end of a transaction. The code between the two key words will be interpreted as a transaction and executed by the NEMO language engine.

6. The NEMO Language Examples

An enterprise with geographically distributed headquarter and branch sites has the requirement to dynamically balance the backup traffic.

In order to implement this scenario, the virtual WAN tenant creates two logicnw, and generates two connections with different SLA to carry diverse service flows. One connection has 100M bandwidth with less than 50ms delay, which is used for normal traffic. And the other connection has 40G bandwidth with less than 400ms delay, which is used for backup traffic after work (from 19:00 to 23:00). With self defined flow policy, the tenant can manage the connection load balancing conveniently.



The detailed operation and code are shown as follows.

o Step1: Create two virtual logicnw nodes in the WAN

```
Node "Branch"
      Type  "logicnw"
      NW    "LN-1"
      Property "ipv4Prefix" : 192.0.2.0/24;

Node "Headquarter"
      Type "logicnw"
      NW   "LN-1"
      Property "ipv4Prefix" : 198.51.100.0/24;
```

o Step2: Connect the two virtual nodes with two virtual connections with different SLA.

```
Connection "broadband_connection"
     EndNodes "Branch", "Headquater"
     Property   "bandwidth" : 40000, "delay" : 400;

Connection "realtime_connection"
     EndNodes "Branch", "Headquater"
     Property  "bandwidth" : 100, "delay" : 50;
```

o Step3: Indicate the flow to be operated.

```
Flow "flow_all"
      Match "src_ip" : "192.0.2.0/24", "dst_ip": "198.51.100.0/24";

Flow "flow_backup"
      Match "src_ip" : "192.0.2.0/24", "dst_ip": "198.51.100.0/24",
            "port": 55555;
```

o Step4: Apply policies to corresponding flows.

```
P1:
Policy "policy4all"
       ApplyTo "flow_all"
       Priority 200
       Action  "forward_to": "realtime_connection";
P2:
Policy "policy4backup"
        ApplyTo "flow_backup"
        Priority 100
        Condition ("time">"19:00:00") || ("time"<"23:00:00")
        Action  "forward_to": "broadband_connection";
```

7. Security Considerations

Because the network customers are allowed to customize their own services, they may bring potentially big impacts to a running IP network. A strong user authentication mechanism is needed for the northbound interface of the SDN controller. User authorization should be carefully managed by the network administrator to avoid any dangerous operations and prevent any abuse of network resources.

8. IANA Considerations

This memo includes no request to IANA.

9. Acknowledgements

The authors would like to thanks the valuable comments made by Wei Cao, Xiaofei Xu, Fuyou Miao, Yali Zhang and Wenyang Lei.

This document was produced using the xml2rfc tool [RFC2629].

10. Informative References

[Frenetic] Foster, N., Harrison, R., Freedman, M., Monsanto, C., Rexford, J., Story, A., and D. Walker, "Frenetic: A Network Programming Languages, ICFP' 11".

[I-D.xia-sdnrg-service-description-language] Xia, Y., Jiang, S., and S. Hares, "Requirements for a Service Description Language and Design Considerations", draft-xia-sdnrg-service-description-language-02 (work in progress), May 2015.

[PBNM] Strassner, J., "Policy-Based Network Management: Solutions for the Next Generation, Morgan Kaufmann Publishers Inc. San Francisco, CA, USA.", 2003.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

[RFC2629] Rose, M., "Writing I-Ds and RFCs using XML", RFC 2629, June 1999.

[RFC4122] Leach, P., Mealling, M., and R. Salz, "A Universally Unique IDentifier (UUID) URN Namespace", RFC 4122, July 2005.

[RFC6020] Bjorklund, M., "YANG - A Data Modeling Language for the Network Configuration Protocol (NETCONF)", RFC 6020, October 2010.

[RFC6241] Enns, R., Bjorklund, M., Schoenwaelder, J., and A. Bierman, "Network Configuration Protocol (NETCONF)", RFC 6241, June 2011.

[TSFNet] Yap, K., Huang, T., Dodson, B., Lam, M., and N. McKeown, "Towards Software-Friendly Networks, APSys 2010, pp:49-54, 2010, New Delhi, India.".

Authors' Addresses

Yinben Xia (editor)
Huawei Technologies Co., Ltd
Q14, Huawei Campus, No.156 Beiqing Road
Hai-Dian District, Beijing, 100095
P.R. China

Email: xiayinben@huawei.com

Sheng Jiang (editor)
Huawei Technologies Co., Ltd
Q14, Huawei Campus, No.156 Beiqing Road
Hai-Dian District, Beijing, 100095
P.R. China

Email: jiangsheng@huawei.com

Tianran Zhou (editor)
Huawei Technologies Co., Ltd
Q14, Huawei Campus, No.156 Beiqing Road
Hai-Dian District, Beijing, 100095
P.R. China

Email: zhoutianran@huawei.com

Susan Hares
Huawei Technologies Co., Ltd
7453 Hickory Hill
Saline, CA 48176
USA

Email: shares@ndzh.com