NEtwork MOdeling (NEMO) Language

An Application API for Intent Driven Networking:


Version 0.7

Nemo Project Team
main author: Tianran Zhou
NeMo team: Cao Wei, Susan Hares, Jiang Sheng
November 10, 2014
Table of Contents

NeMo Language: Network Modeling (NEMO) Language ........................................... 1
An Application API for Intent Driven Networking: ................................................. 1
Technical Reference Manual.................................................................................. 1
1 SDN NBI Challenges .......................................................................................... 4
2 NBI Language: A Network Domain SQL.............................................................. 4
3 Requirements for the Intent Based NBI Language ................................................ 4
4 Introducing NEMO Language ............................................................................... 5
5 NEMO Language Architecture ............................................................................ 6
6 Main NEMO Language Use Cases ........................................................................ 7
   6.1 Virtual WAN and Traffic Engineering ............................................................. 7
   6.2 VPC and Service Chaining ............................................................................. 8
7 NEMO Language Specifications ............................................................................ 9
   7.1 Basic Network Models .................................................................................... 9
   7.2 Notation .......................................................................................................... 10
   7.3 NEMO Language Overview .......................................................................... 11
   7.4 Model Definition ............................................................................................. 12
      7.4.1 Data Types ............................................................................................... 12
      7.4.2 Model Definition and Description Statement .......................................... 13
   7.5 Resource Access Statements ......................................................................... 15
      7.5.1 Node Operations ...................................................................................... 16
      7.5.2 Link Operations ....................................................................................... 16
      7.5.3 Flow Operations ....................................................................................... 17
   7.6 Behavior Statements ....................................................................................... 18
      7.6.1 Query Behavior ......................................................................................... 18
      7.6.2 Policy Behavior ......................................................................................... 19
      7.6.3 Notification Behavior ............................................................................... 20
   7.7 Connection Management Statements ............................................................ 21
   7.8 Transaction Statements ................................................................................... 22
8  Examples.................................................................22

8.1  Dynamic Link Load Balancing ......................................22

9  Appendix (TBD)...........................................................25

9.1  Reference Network Model ...........................................25

  9.1.1  Logical Node.........................................................25

  9.1.2  Switch ................................................................25

  9.1.3  Router ................................................................25

  9.1.4  Access Link............................................................25

  9.1.5  Flow ..................................................................25

  9.1.6  Node Action...........................................................25

  9.1.7  Link Action............................................................25

  9.1.8  Flow Action............................................................25
1 SDN NBI Challenges

As cloud computing and virtualization becomes popular, networks are increasingly asked to provide agility and flexibility without compromising performance, security, scalability, and stability. By decoupling the network control from data forwarding, SDN (Software Defined Networking) simplified network control and explored a lot of novel capabilities and opportunities.

The North-Bound Interface (NBI), located between the control plane and the applications, is essential to enable application innovations and nourish the eco-system of SDN by abstracting the network capabilities/information and opening the abstract/logical 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 standardize.

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 a huge burden of selecting useful information to untrained users. 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.

2 NBI Language: A Network Domain SQL

To implement a novel 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 mechanisms of data storage and data operation, but want to simply request a data transaction that requires data storage and computation and then get the result. As a data domain DSL (Domain Specific Language), SQL is simple and intuitive, and can be embedded in applications. So what we need for the network NBI is a "network domain SQL".

3 Requirements for the Intent Based NBI Language

An intent based NBI language design contains following features:

Express user intent

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

**Platform independent**

With the NBI, the application or user can describe the network request in a generic way, so that any platform or system can get the identical information and consequently achieve the same result. Any low-level and device/vendor specific configurations and dependencies should be avoided.

**Intuitive 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 resources 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 define service requirements for network, detailed configurations, and instructions performed by network devices that are opaque to network operators. So the NBI language should be declarative rather than imperative.

### 4 Introducing NEMO Language

The NEMO language is a 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, the NEMO language enables network users/applications to describe their requests 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.

As an NBI language, the NEMO language provides a uniform framework to bridge the SDN application layer and control layer with super compatibility. On one hand, NEMO language engine supports multiple interfaces and model adaption to the build-in NEMO web client, third party applications, and orchestrators. On the other hand, NEMO language engine interacts with multi-vendor SDN controllers. And with the help of a virtual network manager (VNM) engine, the NEMO language supports the execution of hybrid physical-virtual network.
5 NEMO Language Architecture

NEMO enables the capability for users to describe their intent. An end user or application developer can describe their network with resource and behavior requirements. This description generates a NEMO script and is sent to a language engine, where user intent is interpreted and mapped to optimal solutions automatically.

The NEMO language engine has a flexible framework for model extension to facilitate solutions and devices from multiple vendors. The network service provider can use NEMO to define a new service model and plugin the network function implementation.
6 Main NEMO Language Use Cases

NEMO language can be broadly applied across a range of devices and scenarios, but several use cases stand out as points of customer interest. The following common use cases for NEMO language are not exhaustive, but can be combined to create more complex use cases.

6.1 Virtual WAN and Traffic Engineering

A user has two geographically distributed branch offices in Beijing and Guangzhou. The user can plan and create a virtual network in WAN by specifying several routers and the inter-connections. Traffic engineering can also be generated dynamically by indicating a specific flow goes through a planned path.
6.2 VPC and Service Chaining

Because of many demands on performance, security, scalability, and user experience, application traffic typically needs to be processed by various services in the network, such as WAN optimization, caching, VPN, NAT, load balancing, firewall, IPS (Intrusion Prevention System) and QoS control. In a virtual private cloud, a user needs to generate a local network to connect virtual machines. Additionally, the user has the requirement to generate service chains for specified flows and dynamically insert, remove, modify network services.
7 NEMO Language Specifications

7.1 Basic Network Models

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 requests can be divided into two types: the requests for network resources and the requests for network behaviors.

The network resource is composed of three kinds of entities: nodes, links, and flow. Each entity contains property and statistic information. With a globally unique identifier, the network entity is the basic object for operation.

- **Node model**: describes the entity with the capability of packet processing. According to the functionality, there are three types of node
  - The forwarding node (FN) only deals with L2/3 forwarding. It forwards packets according to the forwarding table and modifies packet heads.
  - The processing node (PN) provides L4-7 network services, and will modify the body of packets.
  - The logical node (LN) describes a set of network elements and their links, 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.

- **Link model**: describes the connectivity between node entities.

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

Network behavior includes the information acquisition and control operations.
The information acquisition 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 use 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 "with <condition>, to execute <action>", and can be applied to any entity

### 7.2 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.

### 7.3 NEMO Language Overview

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

\[
\text{<NEMO\_cmd>} := \text{<model\_definition\_cmd>} \mid \text{<resource\_access\_cmd>} \mid \text{<behavior\_cmd>}
\]

\[
\text{<model\_definition\_cmd>} := \text{<node\_definition>} \mid \text{<link\_definition>} \mid \text{<action\_definition>}
\]

\[
\text{<resource\_access\_cmd>} := \text{<node\_cu>} \mid \text{<node\_del>} \mid \text{<link\_cu>} \mid \text{<link\_del>} \mid \text{<flow\_cu>} \mid \text{<flow\_del>}
\]

\[
\text{<behavior\_cmd>} := \text{<query\_cmd>} \mid \text{<policy\_cu>} \mid \text{<policy\_del>} \mid \text{<notification\_cu>} \mid \text{<notification\_del>}
\]

\[
\text{<connection\_mgt\_cmd>} := \text{<connect\_cmd>} \mid \text{<disconnect\_cmd>}
\]

\[
\text{<transaction\_cmd>} := \text{<transaction\_begin>} \mid \text{<transaction\_end>}
\]

The following figure shows a state machine of the NEMO language engine.
NEMO language provides limited number of key words to enable network users/applications to describe their requests for network resources, services and logical operations in an intuitive way. The key words supported by the language are as follows:

\[
\text{<key_word>} := \text{Boolean} \mid \text{Integer} \mid \text{String} \mid \text{Date} \mid \text{UUID} \mid \text{EthAddr} \mid \text{IPPrefix} \mid \text{NodeModel} \mid \\
\text{LinkModel} \mid \text{FlowModel} \mid \text{ActionModel} \mid \text{Description} \mid \text{Property} \mid \text{Node} \mid \\
\text{Link} \mid \text{Flow} \mid \text{No} \mid \text{EndNodes} \mid \text{Type} \mid \text{NW} \mid \text{Match} \mid \text{List} \mid \text{Range} \mid \text{Query} \mid \\
\text{From} \mid \text{Notification} \mid \text{Listener} \mid \text{Policy} \mid \text{ApplyTo} \mid \text{Priority} \mid \text{Condition} \mid \\
\text{Action} \mid \text{Connect} \mid \text{Disconnect} \mid \text{Address} \mid \text{Port} \mid \text{Transaction} \mid \text{Commit}
\]

7.4 Model Definition

7.4.1 Data Types

NEMO language provides several built-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 [RFC-4122], e.g. f0ba7814-9dad-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. f0:00:00:00:00:01 The EthAddr literal is represented by the token <eth_addr>.

**IPPrefix:**
A string in the form of IP address, e.g. f0:0.0.1.1 The mask can be used in the IP address description, e.g. f0:0.0.1.0/24 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>
```

### 7.4.2 Model Definition and Description Statement

In addition to default built-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

\[
\begin{align*}
\text{<node_type> := <naming>} & \quad \text{!!type name of the node model} \\
\text{<link_type> := <naming>} & \quad \text{!!type name of the link model} \\
\text{<flow_type> := <naming>} & \quad \text{!!type name of the flow model} \\
\text{<entity_type> := <node_type> | <link_type> | <flow_type>} \\
\text{<action_type> := <naming>} & \quad \text{!!type name of the action model} \\
\text{<model_type> := <entity_type> | <action_type>} \\
\text{<property_name> := <naming}> & \quad \text{!!name of the property in a model}
\end{align*}
\]

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

\[
\begin{align*}
\text{<node_definition> := NodeModel \text{ <node_type>}} \\
\quad \text{Property} \{ \text{ <data_type> : <property_name> } \} ;
\end{align*}
\]

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 to a network is the intrinsic property for a node model, there is no need to redefine the belonged to network in the property list.

**Example:**

`NodeModel  \text{ DPI}  Property  \text{ String : name, Boolean : is_enable}`

The statement generates a new node model named DPI with two properties, `name` and `is_enable`.

The `<link_definition>` statement is used to create a link model:

\[
\begin{align*}
\text{<link_definition> := LinkModel \text{ <link_type>}} \\
\quad \text{Property} \{ \text{ <data_type> : <property_name> } \} ;
\end{align*}
\]

The `LinkModel` specifies a new link type.

The `Property` is followed by a list of `<data_type> : <property_name>` pairs to specify properties for the new link type. Since end nodes are intrinsic properties for a link 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:

```xml
<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:

```xml
<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:

```xml
<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.

### 7.5 Resource Access Statements

In the 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.

```xml
<node_id> := <naming>       !! name to identify the node instance
<link_id> := <naming>       !! name to identify the link instance
<flow_id> := <naming>       !! name to identify the flow instance
```
7.5.1 Node Operations

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

```
<nod...ce> := 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.

**Example:**

```
Node switch-1
    Type switch
    NW LN-1
    Property ip_version: 4;
```

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:

```
<nod...e> := No Node <node_id>;
```

The `No Node` deletes a node in user's network.

7.5.2 Link Operations

The `<link_cu>` statement is used to create or update a link:
<link_cu> := Link <link_id> Type <node_type>

EndNodes <node_id>, <node_id>

[Property {<property_name>: <value>}];

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

The Type specifies the type of the link to operate.

The EndNodes specifies the two end nodes, identified by $<node_type>: <node_id>$, of a link.

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 link definition.

Example:

Link $\tilde{\text{link-1}}$ Type $\tilde{\text{General}}$

EndNodes $\tilde{\text{S1}}, \tilde{\text{S2}}$

Property $\tilde{\text{bandwidth}}: 1000, \tilde{\text{delay}}: 40$;

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

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

<link_del> := No Link <link_id>;

The No Link deletes a link in user's network.

7.5.3 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**  ņflow-1ô

**Match**  ņsrc_ipô: **Range** (ň92.168.1.1ô, ť92.168.1.243ô);

The statement describes a flow with the source IP address ranging from 192.168.1.1 to 192.168.1.243.

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

<flow_del> := **No Flow** <flow_id>;

The **No Flow** deletes a flow in userô's network.

### 7.6 Behavior Statements

#### 7.6.1 Query Behavior

The query statement is used to retrieve selected data from the 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 supports query operations to network entities and policies.
7.6.2 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> } };

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 policy 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. 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:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;&amp;</td>
<td>Called Logical AND operator. If both the operands are non-zero, then the condition becomes true.</td>
<td>(A &amp;&amp; B) is false.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>!</td>
<td>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.</td>
<td>!(A &amp;&amp; B) is true.</td>
</tr>
<tr>
<td>==</td>
<td>Checks if the values of two operands are equal or not, if yes then condition becomes true.</td>
<td>(A == B) is not true.</td>
</tr>
<tr>
<td>!=</td>
<td>Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.</td>
<td>(A != B) is true.</td>
</tr>
<tr>
<td>Operator</td>
<td>Description</td>
<td>Example</td>
</tr>
<tr>
<td>----------</td>
<td>-------------</td>
<td>---------</td>
</tr>
<tr>
<td>&gt;</td>
<td>Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.</td>
<td>(A &gt; B) is not true.</td>
</tr>
<tr>
<td>&gt;=</td>
<td>Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.</td>
<td>(A &gt;= B) is not true.</td>
</tr>
<tr>
<td>&lt;</td>
<td>Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.</td>
<td>(A &lt; B) is true.</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.</td>
<td>(A &lt;= B) is true.</td>
</tr>
</tbody>
</table>

The **Action** specifies the execution when conditions meet.

**Example:**

**Policy**

```plaintext
<policy_id> := "policy-1"

ApplyTo        := "flow-1"
Priority       := 100
Condition      := (time>8:00) || (time<21:00)

Action         := gothrough: List( switch-1, switch-2);
```

The statement creates a policy which indicates the flow is to go through two specified switches from 18:00 to 21:00.

**Delete a policy:**

```plaintext
<policy_del> := No Policy <policy_id>;
```

The **No Policy** is deletes a policy in user's network.

**7.6.3 Notification Behavior**

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

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

**Create and update a notification**

```plaintext
<notification_cu> := Notification <notification_id>  [Query { <property_name> }]
```
The **Notification** is followed by a user defined `<notification_id>`. If the `<notification_id>` is new in the system, a new notification 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:**

```plaintext
<notification_del> := No Notification <notification_id>;
```

The `No Notification` deletes a notification in user’s network.

### 7.7 Connection Management Statements

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

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

**Setup a connection to the NEMO language engine:**

```plaintext
<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 **Port** specify the IP address and the port of the NEMO language engine to connect separately.

**Disconnect the connection to the NEMO language engine:**

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

The **Disconnect** is to remove the connection with an ID equals to `<conn_id>` from the NEMO language engine.
7.8 Transaction Statements

Claim a transaction

<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.

8 Examples

8.1 Dynamic Link Load Balancing

In this section, a concrete example for the use case in section 6.1 is proposed. 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 routers, and generates two links with different SLA to carry diverse service flows. One link has 100M bandwidth with less than 50ms delay, which is used for normal traffic. And the other link 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 link load balancing conveniently.
The NEMO language engine has the capability to automatically generate state machines by extracting events from the user defined NEMO language script. In this example, the time condition described in the flow policy triggers the state transition.

The detailed operation and code are shown as follows.

**Step 1: Create two virtual router nodes in the WAN**

**Node**  
**R1**  
**Type** router  
**NW** LN-1  
**Property** ip_version: 4;

**Node**  
**R2**  
**Type** router  
**NW** LN-1  
**Property** ip_version: 4;

**Step 2: Connect the two virtual nodes with two virtual links with different SLA.**

**Link**  
**broadband_link**  
**Type** General  
**EndNodes** R1, R2  
**Property** bandwidth: 40000, delay: 400;

**Link**  
**realtime_link**  
**Type** General  
**EndNodes** R1, R2  
**Property** bandwidth: 100, delay: 50;

**Step 3: Indicate the flow to be operated.**

**Flow**  
**flow_all**  
**Match** src_ip: 10.0.1.0/24, dst_ip: 10.0.1.0/24;

**Flow**  
**flow_backup**  
**Match** src_ip: 10.0.1.0/24, dst_ip: 20.0.1.0/24, port: 55555;
Step 4: Apply policies to corresponding flows.

P1:
Policy: "policy4all"
ApplyTo: "flow_all"
Priority: 200
Action: "forward_to: realtime_link"

P2:
Policy: "policy4backup"
ApplyTo: "flow_backup"
Priority: 100
Condition: (time >= 9:00:00) || (time <= 3:00:00)
Action: "forward_to: broadband_link"
9 Appendix (TBD)

9.1 Reference Network Model

9.1.1 Logical Node

9.1.2 Switch

9.1.3 Router

9.1.4 Access Link

9.1.5 Flow

9.1.6 Node Action

9.1.7 Link Action

9.1.8 Flow Action