A semantic model for complex computer networks : the network description language

van der Ham, J.J.

Publication date
2010

Chapter 4

NDL Applications

This chapter is based on *Semantics for Hybrid Networks Using the Network Description Language* by J.J. van der Ham, P. Grosso and C.T.A.M. de Laat[2], *Using the Network Description Language in Optical Networks* by J.J. van der Ham, P. Grosso, R. van der Pol, A. Toonk and C.T.A.M. de Laat[3] and *A Distributed Topology Information System for Optical Networks Based on the Semantic Web* by J.J. van der Ham, F. Dijkstra, P. Grosso, R. van der Pol, A. Toonk and C.T.A.M. de Laat[5].

4.1 Introduction

In the previous chapter we have introduced the Network Description Language. We have shown an example of how it can describe devices, and link between different domains.

One of the advantages of using NDL as the language for description of hybrid networks is the availability of semantic web tools for RDF that can parse and consume the information in each NDL file. This means that extracting the information needed for network management, and in our specific case lightpath provisioning, is straightforward and simple.

In this chapter we show several examples of how we have applied the language to solve many of the operational issues that operators and users face in hybrid

optical networks, and other kinds of networks.

Network operators and users often use maps of the network to make sense of the topology, and to support them in diagnostics or manual pathfinding. Having up to date maps is very important, but creating and updating these maps is also very difficult and labour intensive. In section 4.2 we show our graph generation application.

Before we can start working on the operational issues, we need to gather data, preferably in an automatic way. In section 4.3 we present how we gather data from networks.

Besides extracting the visual aspect of the network description, we can also perform other queries on the data. Section 4.4 shows how we perform queries on the data, and our applications for lightpath planning both in a single network, as well as over multiple networks.

After developing several of these applications we have found that it is helpful to have a toolkit available to support the most common tasks performed with NDL files. In section 4.5 we discuss the Python NDL Toolkit (pynt).

Another application that we have developed is a network emulation toolkit called Virtual Network Experiments (VNE). We discuss this toolkit in section 4.6.

Finally in section 4.7 we summarize our work, and discuss our findings and results with the different applications.

4.2 Network Graph Generation

Our first application of the language has been the visualization of network topologies. Given that most lightpaths are still provisioned manually, at least when they involve crossing organization boundaries, maps become the visual aid used by network engineers to setup the circuits. The information about the connection between domains must be up to date, accurate and consistent, because mistakes in lightpath provisioning can have impact on other lightpaths and in the case of hybrid networks also on the regular traffic.

There are certainly many ways to create a graphical overview of a network, and when working in a single network domain plenty of tools to choose from. But in multi-domain environments, such as the cooperating universities and research institutes in GLIF, we need to take into account that the information for each domain is not centrally maintained and there is a big potential for inconsistencies. The manual creation of these large-scale topology overviews requires a tedious conversion of gathered data to a consistent representation,

and the verification of consistent information at the boundaries. After these steps one can then feed the information to the graphing tools. Requesting this data from the different networks in GLIF and processing that is a process which can easily take months.

NDL provides a way for operators to interoperably and independently publish their network topology, including references to other networks. Starting from a network description in NDL format, we extract the connections between the devices and their names. Using a small script, this data is then converted to serve as input to GraphViz, an open source graph visualization tool[71]. An example of such a graph is shown in figure 4.1. This is a map of NetherLight[72], one of the network domains participating in GLIF.

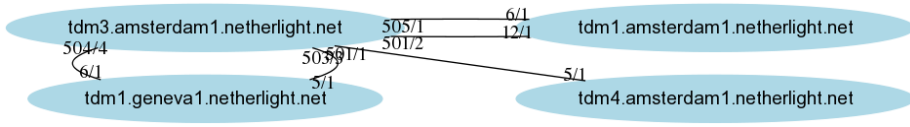

Figure 4.1: *A graph of NetherLight resources (generated from NDL file)*

The graphs generated using GraphViz are not always ideal, they are generated automatically, which means that a small change in the topology can radically change the generated image. For larger networks another option is to embed GPS coordinate information in the NDL documents using the standard `geo` RDF namespace. It then becomes possible to use Google Maps[73], for example, to display networks on the map.

Figure 4.2 shows a subset of the GLIF resources on a global Google Map plotted using NDL information. Each of the pointers in the figure is a separate NDL file. The links between these points are defined by both sides, including `seeAlso` pointers to the other file. The inter-domain topology is created by crawling the distributed descriptions.

4.3 Automatic Generation of Network Descriptions

In the early stages we created network descriptions by hand. This soon became very tedious, and error-prone. A first simple method of automating NDL generation is by using forms. We created a simple webform (see [74]), which allowed us to quickly create a description for a number of locations and devices with their interfaces, and connections between them.

Figure 4.2: *A subset of the GLIF resources on a Google Map generated from NDL files as demonstrated at SC06*

Network descriptions based on real world data are harder to create. Unfortunately, as we have discussed in chapter 2 there is no standard way of representing management data in network hardware. Different vendors use different techniques for storing management information. Over the years we have created several different modules to cope with different sorts of hardware. Examples are hand-coded Python scripts for reading specific devices (Force10 E-Series, Glimmerglass), Python and Perl modules for input through TL1, and a Python module for input from OSPF and OSPF-TE traffic.

When reading information from separate devices, the information must be correlated using external information. That is, the information about the cables connecting devices must be provided in some way. Even in dynamic networks, these cables are often very static, most changes to the network topology are done by changing device configurations. For example in our test network, all machines are connected to an optical patch-panel, and several ports of this patch-panel are connected to a switch. This allows a very dynamic topology where connections between devices can either be made directly only passing through the patch-panel, or through the switch, possibly using VLANs.

In larger networks where OSPF is used to manage routing, it is possible to automatically gather topology data. The routers in such a network use a link-state algorithm. This means that each router monitors the links to its neighbours, and distributes this information to other routers in the network. The distribution is done in such a way that each router has full knowledge of the whole topology in its link state database. We wrote a program which requests a copy of this database, and translates the relevant information to NDL, resulting

in an automatically generated description of the network. Unfortunately, OSPF only exchanges information about connectivity at the IP layer, so the topology description will be limited to that layer.

An extension to OSPF, called OSPF Traffic Engineering (OSPF-TE), is used in the GMPLS protocol suite. OSPF-TE defines several new types of messages. These can contain information about layers and different kinds of labelling, so that different transport techniques can be combined. The new messages also allow the topology information to be exchanged out-of-band, i.e. the messages about the network can be exchanged on a separate network. This allows it to be used in optical networks where in-band management is not possible.

4.3.1 Topology Generation for TITAAN

The TITAAN network is military network of the Royal Netherlands Army (see section 1.4), using commercial off-the-shelf hardware. The network nodes have been configured such that it allows for quick plug-and-play operations, requiring minimal configuration in the field. All this has been done (just) within the boundaries of normal networking standards and capabilities.

While the configurations have allowed for quick setup in the field, some of the complexity moved to managing and monitoring the network at the network operations centre. The engineers have had trouble getting a good overview of the network, because all available network management tools can not make sense of the configuration.

We have developed a proof-of-concept application for the TITAAN network which extracts the topology data from the OSPF network. The topology is then exported to NDL, so that it can be used in other applications, such as showing which path will be used between two nodes.

The application has provided the engineers with a valuable source of information with which they can easily see which route traffic will take through the network. The topology data can also easily be used as input information for other applications, so that for example they can gauge the current state of the network and adapt their behaviour to it.

4.3.2 Topology Generation from OSPF-TE

We have also extended our work on topology extraction from OSPF to OSPF-TE. Messages in OSPF-TE are exchanged out-of-band, the control-plane network runs regular OSPF, which is then supplemented with Opaque LSAs. These

are a special kind of LSAs that can carry different kinds of contents describing the topology and details of devices and interfaces on the data plane.

Unfortunately, there are not many production networks using GMPLS, but we have successfully tested this with some experimental setups using DRAGON. DRAGON, Dynamic Resource Allocation over GMPLS Optical Networks, is an open-source implementation of OSPF-TE and other GMPLS protocols[75]. We have also successfully used the exporting from OSPF-TE in network emulations, which we discuss in section 4.6.

Technical details about OSPF and OSPF-TE and how the information in LSAs maps to NDL descriptions can be found in Appendices A and B.

4.4 Extracting Data from Network Descriptions

As we described in the previous chapter NDL is based on RDF. This has several advantages, one of which is that we can make use of generic RDF tools and standards. An important tool for RDF is the SPARQL Protocol and Query Language for RDF (SPARQL)[53], which is an SQL-like query language for RDF. It uses a simple syntax to specify variables and triplet templates for retrieving information from a repository. An example is shown in Listing 4.1.

```
1 PREFIX ndl: <http://www.science.uva.nl/research/sne/ndl#>
2 PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
3 SELECT ?hostname ?locationname
4 WHERE { ?host rdf:type ndl:Device .
5       ?host ndl:locatedAt ?location .
6       ?host rdf:label ?hostname .
7       ?location rdf:label ?locationname .
8     }
```

Listing 4.1: *Example of a SPARQL query.*

The example shows a query to select hostnames and their location names, the variables `?hostname` and `?locationname`. The values must satisfy the constraints in the `WHERE` clause. These constraints are expressed using two other variables, `?host` and `?location`. The `?host` must be of type `Device`, and must be `locatedAt` a `?location`. Then with the `label` property the names of both objects are found. In summary this query will return all host and location pairs, if the host has a location defined.

4.4.1 Lightpath Planning in SURFnet6

SPARQL allows for much more complex queries, and we have used it in a lightpath planning application for SURFnet6. SURFnet6 is the Dutch national research and education network. SURFnet6 is a hybrid network, offering both IP services and lightpath services. Toonk and Van der Pol of the Dutch national super-computing centre SARA have written a tool for planning new lightpaths based on NDL and SPARQL [76, 77] .

Part of the SURFnet6 hybrid network is formed by a collection of Nortel OME6500 Time-Division Multiplexing (TDM) nodes. We obtain the topology information by using the neighbour knowledge from each of these devices. We gather the data by periodically sending and receiving discovery messages on the control plane. We use NDL to describe the topology of the TDM layer in files.

```
1  my $query = new RDF::Query ( <<"END", undef, undef, 'sparql' );
2  PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
3  PREFIX ndl: <http://www.science.uva.nl/research/air/ndl#>
4  SELECT ?device1 ?device2 ?if1 ?if2
5  WHERE {
6        ?d1 ndl:hasInterface ?x .
7        ?d1 ndl:name ?device1 .
8        ?x ndl:connectedTo ?y .
9        ?x ndl:name ?if1 .
10       ?d2 ndl:hasInterface ?y .
11       ?d2 ndl:name ?device2 .
12       ?y ndl:name ?if2
13 }
14 END
15
16 $ne1 = $result->[0]->getValue;
17 $ne2 = $result->[1]->getValue;
18 $if1 = $result->[2]->getValue;
19 $if2 = $result->[3]->getValue;
20
21 $g->add_edge("$ne1-$if1", "$ne2-$if2");
22
23 my @V = $g->ShortestPath("$v1", "$v2");
```

Listing 4.2: *A SPARQL Query implemented in Perl*

Listing 4.2 shows the SPARQL query that is used in the planning application. Line 6 gets all interfaces of a device and line 7 gets the name of the device. Line 8 gets the neighbour of an interface and finally line 10-11 get the name of the device to which the neighbour interface belongs.

Line 16-19 store the device and interface names of both ends of a link (con-

nectedTo). These variables are used to build a graph. Line 21 builds the edges of the graph. This implicitly adds the vertices `$v1` and `$v2` to the graph. Finally, in line 23 a shortest path is computed by the method `ShortestPath`, which is a standard Dijkstra Shortest Path implementation.

Additionally, a network state database holds the cross-connect information for each network element in the network, that is, information about currently provisioned lightpaths. This enables the application to determine the amount of time-slots still available on each interface. Combining the NDL topology information and the database time-slot information we can find a shortest path through the network that has enough free time-slots to accommodate a new user request. To find this path we use a constraint based shortest path algorithm.

The result of the path calculations are then implemented by human operators who provision the lightpaths through SURFnet6. This application is a first step towards completely automatic lightpath provisioning. Before creating this application, the engineers had to piece together the time-slot information themselves, making the provisioning process a very time-consuming procedure.

4.4.2 Lightpath Planning in GLIF

We have also extended the lightpath planning application to GLIF, which constitutes an ideal environment to see NDL at work in a multi-domain and multi-administrator setup. The most important domains of the GLIF network are the lightpath exchanges, also called GLIF Open Lightpath Exchanges (GOLEs).

As a proof of concept, we created an abstraction of several GOLEs of the GLIF network, where each GOLE is described as a virtual device with several interfaces. These interfaces connect the GOLE to other GOLEs. To correlate the abstracted descriptions of each individual GOLE with each other we use the `seeAlso` property of RDF. Using these links, a linked web of descriptions is formed. This provides a global view of the network, where each domain maintains the description for its own GOLE. This network is also shown in figure 4.2.

```
1 <ndl:Device rdf:about="#netherlight">
2    <rdf:label>Netherlight</rdf:label>
3    <ndl:locatedAt rdf:resource="#NetherLight"/>
4    <ndl:hasInterface rdf:resource="#netherlight:if1"/>
5    <ndl:hasInterface rdf:resource="#netherlight:if5"/>
6    <ndl:hasInterface rdf:resource="#netherlight:if6"/>
7    <ndl:hasInterface rdf:resource="#netherlight:if10"/>
8 </ndl:Device>
9
```

```
10 <ndl:Interface rdf:about="#netherlight:if1">
11    <rdf:label>if1</rdf:label>
12    <ndl:connectedTo rdf:resource="http://networks.internet2.edu/manlan/manlan.rdf#
          manlan:if1"/>
13    <ndl:capacity rdf:datatype="http://www.w3.org/2001/XMLSchema#float">1.2E+9</
          ndl:capacity>
14 </ndl:Interface>
15
16 <ndl:Interface rdf:about="http://networks.internet2.edu/manlan/manlan.rdf#
       manlan:if1">
17    <rdfs:seeAlso rdf:resource="http://networks.internet2.edu/manlan/manlan.rdf"/>
18 </ndl:Interface>
```

Listing 4.3: *Part of the abstracted Netherlight GOLE description*

Listing 4.3 shows an excerpt of the abstract description for the Netherlight GOLE. Line 1 describes which virtual device (GOLE) this is. Line 4 to 7 describe which interfaces the (virtual) device has. Interface `netherlight:if1` is described on line 10 to 16. line 11 describes its name and line 12 describes where it is connected to. Finally, line 14 describes the capacity (total bandwidth) of the interface. Lines 18 to 20 provide a pointer for the description of the other side of the connection using the `seeAlso` property.

To find a path within the GLIF the first step is to read this NDL file. The description for the other GOLEs is found by crawling the links contained in the `seeAlso` properties. This way we can determine if a path exists and can be provisioned in the GLIF network.

During SuperComputing 2006 we have shown an application that gathered all the NDL files from the different GOLEs. Using a web interface, a user can select two endpoints from a list, which is generated from the gathered NDL information. After the two endpoints are selected, the application applies the Dijkstra algorithm to find the shortest path between the two endpoints. The network is displayed using Google Maps, as described in section 4.2. The shortest path through the network is then also drawn in the same figure, using highlighted links. A list of hops is also provided next to the map. Figure 4.3 shows the example output for a path between Seattle and Geneva.

4.4.3 Lightpath Monitoring in NetherLight

NDL can play an important role in lightpath monitoring as well. SARA developed Spotlight, a tool for lightpath monitoring in SURFnet6 and in NetherLight. To monitor the lightpaths, SARA uses NDL to specify their topology

Figure 4.3: *Pathfinding in GLIF, presented in Google Maps.*

details, and actively query the network elements involved. The output is stored in a network state database with alarm and configuration information.

The Spotlight application gathers all the information on the lightpaths in one place, providing engineers with a single overview of all lightpaths, along with their status. The user can then click on a specific lightpath to see an overview of the configuration, and the status of each of the segments of the lightpath. If a failure is detected somewhere in the lightpath route, this will be clearly indicated using a visualization of the lightpath.

The Spotlight tool has made monitoring of lightpaths in SURFnet6 and Netherlight a lot simpler. Engineers now have a single place where configuration and alarm data is correlated into a single view, allowing them to more quickly and accurately pinpoint the problem with a lightpath. The Spotlight application is available online, see [78].

4.5 Python NDL Toolkit

The applications that we have discussed have been mostly developed in isolation. We soon realised that much can be gained by providing a common toolkit for parsing and creating NDL files.

The pynt[79] provides a complete object model of NDL in Python. The toolkit allows a developer to grab an NDL file, request the toolkit to parse it and directly have all the objects available that were described. The properties of these instances can then be easily queried and updated. The updated description can then be easily exported to an NDL file.

The toolkit consists of six main components:

`pynt` is the basis of the package that has the definitions of the object model, and namespaces.

`pynt.input` contains modules that can parse from the RDF XML syntax, directly from nodes, or from OSPF (see section 4.3).

`pynt.output` provides an output mechanism to the RDF XML syntax, but also to graphs, and VNE configurations (see section 4.6).

`pynt.protocols` is a set of generic modules that provide base 'protocols' for interacting with command-lines, OSPF, or through TL1.

`pynt.technologies` contains a set of pre-defined technology descriptions to make it easier to describe nodes and interfaces of these technologies. Objects from these modules predefine layer properties, and make it easy to use well-known adaptations between layers.

`pynt.algorithms` implements different pathfinding algorithms, such as Dijkstra's shortest path algorithm, which can be used in single layer topologies, and a breadth-first pathfinding algorithm that is suitable for multi-layer pathfinding[13].

4.6 Virtual Network Experiments

Virtual Network Experiments (VNE)[6],[80] is a Python application that we have developed to allow users to easily create a virtual experimental network. The nodes in the network are implemented using User-Mode Linux (UML)[81], which allows for a lightweight virtualization, yet allows users to run their own applications.

VNE takes a topology description in XML. This XML file can be written manually or exported from an NDL file using pynt. VNE launches instances of UML following the provided configuration. These instances are provided with the proper configuration to create the network topology as described in the configuration file. The emulated network is provided using the `vde-switch` tool of the Virtual Distributed Ethernet project[82].

The contribution of VNE is that it makes it greatly simplifies the configuration and management of an emulated network. Configuring and launching UML instances, along with the right network topology is a very complicated task. VNE uses a clear XML syntax to define the configuration and network, allowing users to easily create and modify complex topologies, ready for experimentation.

We have also extended VNE to integrate it with DRAGON. The configuration file and VNE provide the proper configuration for the DRAGON toolkit, so that it is simple to create an emulated network that can be used to easily experiment with GMPLS. VNE was invaluable for us to gain understanding of GMPLS, and to test and validate the OSPF-TE export.

We can combine the OSPF-TE export to NDL, and the export from NDL to VNE. This allows us to sample the topology from any GMPLS network, and then create an emulated version of it using VNE. Both exports have been validated by completing a full circle, i.e. creating a VNE configuration, running that emulation, exporting the OSPF-TE data from that to NDL, and then create a VNE configuration from that again.

4.7 Conclusion

In this chapter we have shown the applications that we have developed that use NDL topology descriptions.

A first contribution of NDL has been that it allowed us to quickly create up to date maps of networks. The maps can be made abstractly using GraphViz, or if Global Positioning System (GPS) information is available, using Google Maps.

We have shown that descriptions in NDL can easily be generated using webforms, or by using our toolkit. The toolkit can also extract topology information in more automatic ways, either by logging into machines and parsing the command-line output, or by extracting it from management traffic. This makes it possible to provide a real-time view of the network.

The information retrieved from the network can support engineers with light-

path planning. With SARA we have developed an application that correlates the NDL topology information with the database of timeslot usage, this allows an engineer to quickly find an available path through the SURFnet6 network.

The configuration information of these lightpaths is also used to provide the engineers with a status overview. The configuration and state of lightpaths is presented in a graphical overview, allowing support engineers to quickly identify the root cause of incidents in the network, and make a better assessment of the consequential loss of services caused by this incident.

Developing the applications above has also inspired us to create the Python NDL Toolkit. The toolkit has made it very easy for us to test new ideas and create new applications. Combined with the Virtual Network Experiments application we can quickly create network topologies and test applications on them.

The applications described in this chapter have provided us with valuable insights about the possible use of NDL descriptions. They have allowed us to test NDL in real-world scenarios and solve problems. Creating the applications has helped us to gain a better understanding of the use-cases for the descriptions, and provided valuable feedback for the NDL schemata.

The applications in this chapter have been mostly aimed at single layer descriptions, multi-layer descriptions are a fairly recent development. Not many networks have described their multi-layer network in detail yet, and we are currently still exploring the possibilities.