17-03-2015

Open Call Deliverable OCQ-DS3.1 Final Report Multi-domain OpenFlow Topology Exchange (MOTE)

Open Call Deliverable OCQ-DS3.1

Grant Agreement No.:	605243
Activity:	NA1
Task Item:	10
Nature of Deliverable:	R (Report)
Dissemination Level:	PU (Public)
Lead Partner:	University of Amsterdam
Document Code:	GN3PLUS14-1297-30
Authors:	Miroslav Živković, Paola Grosso

The research leading to these results has received funding from the European Community's Seventh Framework Programme (FP7 2007–2013) under Grant Agreement No. 605243 (GN3plus).

Abstract
This document is the final deliverable of the GN3plus MOTE project. It discusses the issues and possible solutions of the topology exchange between multi-domain OpenFlow and Network Service Interface domains. It presents the topology exchange architecture, the topology representation within the OpenFlow domains and extensions within OpenFlow domains that support these.

Table of Contents

Table of Figures

Table of Tables

Executive Summary

One of the most promising Software-defined Networking (SDN) technologies currently is based on OpenFlow protocol and the adoption of multi-domain solutions is one of the main challenges. The OpenFlow-based networking solutions do not properly support multi-domain scenarios. On the other hand, an upcoming solution for automated inter-domain network services is the Open Grid Forum (OGF) Network Service Interface (NSI). This standardized interface allows communication between different networks bandwidth-on-demand or circuit reservation systems.

To allow the Network Services Interface to support OpenFlow or other SDN technologies, there must be a method for describing and exchanging topologies of these kinds of networks. This will allow the NSI to become aware of these networks, and to allow reservations on it. Besides, the network topologies are not static, as networks are constantly upgraded and maintained, which results in ever changing topologies. This requires from the topology exchange solutions to automatically propagate topology changes between different domains.

Within the project MOTE we have addressed a number of challenges and issues. The NSI lacks proper topology exchange architecture, and we first addressed this. Besides the fact that the proposed topology exchange architecture is generic and could be used in any networked environment to exchange documents, it fulfils many requirements for the successful implementation within NSI. From the perspective of MOTE project, it supports push/pull topology update mechanism, which is one of the basic principles for the automated topology exchange solution.

Next we addressed the challenge of the adequate topology description within the OpenFlow domains. The OpenFlow domains are governed by the controllers, which are logically centralized control plane elements within the network domain. There are many different implementations of the OpenFlow controllers, with many different approaches and solutions for the topology representation. For the implementation purposes we selected a particular controller implementation (Floodlight), and investigated how to implement architecture components within Floodlight that allow us to integrate the OpenFlow domains into the proposed architecture. We also addressed controller extensions in order to create the topology documents within the domain and how to update these documents automatically, i.e. upon changes in network topology.

As we investigated different OpenFlow controllers and the topology representations by these, it became clear that we need a unified way to present the OpenFlow domains' topologies. The OGF Network Markup Language (NML) schema is a standard for describing multi-layer multi-technology computer networks. At the time of its inception OpenFlow did not exist, yet the NML standard has attempted to define a generic way of describing computer networks in a technology independent way.

As the documents exchanged within NSI are using NML standard format, it goes without saying that the OpenFlow domains should export their topologies in this format. The conclusion was that NML is suitable to model OpenFlow topologies. Therefore, we implemented functionality that allows the conversion of the topology instance within Floodlight controller to the NML format.

The proof-of-concept implementation verified the main architecture concepts and how to create automated topology updates within the OpenFlow domains. We implemented a path computation application that uses OpenFlow controller, which is a pre-requisite in order to allow end-to-end network resource reservations. The main issues and challenges we encountered were the topology discovery mechanisms within OpenFlow, as these cannot resolve inter-domain loops. We supplied a solution that could be improved in the following research. The biggest issue that needs to be addressed in the future is how to address the issue of a plethora of different OpenFlow implementations.

This project resulted also resulted in a publication of the short article within CONNECT magazine, as well as submission of an article to the 10th edition of the IEEE TRIDENTCOM conference. A successful demo of the basic implementation of the topology exchange architecture was presented at the 2014 SuperComputing conference. Besides, topology exchange architecture demo and pitch talk are confirmed for the GEC22 and US Ignite Demo Night during the coming GENI22 conference. The topology exchange architecture is proposed to the OGF standardization body, NSI Working Group (WG).

This document is the final deliverable of the Multi-domain OpenFlow Topology Exchange (MOTE) GN3plus project.

1 Introduction

The services and capabilities offered by the large--scale multi--domain networked environment implies that management of these networks is essential. The (controlling) network elements need to have sufficient information to take management decisions in an optimal and cooperative way. This information therefore needs to be exchanged between different domains in an efficient and secure way.

One of the most promising SDN technologies currently is based on OpenFlow protocol and the adoption of multi-domain solutions is one of the main challenges. The OpenFlow-based networking solutions do not properly support multi-domain scenarios, although there are some solutions such as FlowVisor [FlowVisor] that can partition multiple OpenFlow-enabled switches over several users. However, this solution is applicable to switches that belong to a single domain only.

On the other hand, an upcoming solution for automated inter-domain network services is the Open Grid Forum [OGF] Network Service Interface (NSI). This standardized interface allows communication between different networks bandwidth-on-demand or circuit reservation systems. Several supporting services are also defined in NSI, e.g. a Discovery Service. The NSI Framework [NSI-FW] provides a way to do automated inter-domain bandwidth reservations. These inter-domain network services are mainly implemented on VLANs. However, the NSI Framework is initially derived without support for OpenFlow.

In Section 2 we addressed the topology exchange architecture within NSI. Besides the fact that the proposed topology exchange architecture is generic and could be used in any networked environment to exchange documents, it fulfils many requirements for the successful implementation within NSI. From the perspective of MOTE project, it supports push/pull topology update mechanism, which is one of the basic principles for the automated topology exchange solution. This means that notifications about any relevant domain topology updates would be propagated. The main requirement for the OpenFlow domains is therefore to implement both Topology Client and Topology Provider architecture components.

In Section 3 we described different experiments performed with topology software. We first described the MOTE OpenFlow testbed. Then we briefly introduced the main topology discovery and topology representation at OpenFlow controllers, and explain why the Floodlight is the controller of our choice for the implementation. In order to understand the topology discovery and representation within the Floodlight we conducted the topology discovery experiments using MOTE testbed and how this information is presented at OpenFlow controllers. It helped us to clearly identify the issues with LLDP topology discovery and we described how to address some of them.

As the documents exchanged within NSI are using NML standard format, it seems natural that the OpenFlow controllers should export their domains' topologies using NML format. In Section 4 we summarized the investigation of applicability of the NML to model OpenFlow domain topologies.

Finally, in Section 5 we describe the implementation of the extensions of the Floodlight controller in order to facilitate topology exchange between NSI and OpenFlow domains. We briefly describe the implemented architecture components and controller extensions that facilitate automated conversion of the Floodlight topology instances to/from NML format. We also describe a basic implementation of a path finding for a Floodlight controller, which is implemented as topology client.

Our architecture has provided to facilitate easy and rapid integration of different topology clients and flawlessly supports exchange of topologies specified in NML (or other agreed upon) format. The NML also proved to be suitable for standardizing topology instances maintained by versatile OpenFlow controllers.

However, automated topology discovery within OpenFlow domains requires points of attention. First, topology discovery within OpenFlow domains is based on LLDP. In order to use this protocol to discover links to other (non-OF) domains LLDP has to be enabled by switches at these domains.

Second, to easily identify the neighbouring domains, an extension of the LLDP port description may be necessary. This LLDP modification has to be supported by the controller, and the switches that are deployed in other domains.

The main issue encountered with topology discovery is the one with the loops, i.e. the situation when two ports that belong to the same OF domain are presented as connected through an external link. The proposed solution would work for Floodlight controllers exclusively. This requires attention for other controllers available.

2 Topology Exchange Architecture

In this section we present our Topology Exchange Architecture. The presented architecture addresses the lack of similar approaches within the NSI community. Although a Topology Service has been specified within the NSI, it was not a mature solution that would be able to address the very versatile requirements brought forward within the NSI community. Besides, it lacked capabilities to support automatic exchange of network topology updates between different domains. The procedure that has been deployed within the NSI community so far has been to upload topology descriptions to the centralized repository "server". The topology retrieval was based on usage of this known location from which the topologies could be obtained. This has been done, in general, lacking basic security functionality (e.g. authentication), and the solution proposed supported only a pull method. The parties that would like to obtain updates could do this only by (periodically) pulling the topologies from the given repository. The topologies stored at the repository were specified using the OGF NML standard [NML] that is accepted by the NSI community.

One of the goals of the MOTE project was to allow for automatic creation of the topology documents within the OpenFlow domains. The second goal was to examine the distribution and exchange of topology information in inter-domain scenarios including OpenFlow networks. Clearly, the automatic generation of the topology documents within the OpenFlow domains would be of no use if these documents could not be distributed across other NSI domains.

Therefore, in order to achieve the goals of the MOTE project we first addressed the Topology Exchange Architecture that would extend currently deployed NSI solutions, and would make the topology exchange feasible. In this section we present the complete architecture, including e.g. security considerations. This gives a better impression of the solution that has been proposed for standardization within OGF.

We will first describe briefly the architecture requirements we addressed with our proposal. After that, the architecture, and its principal components will be introduced, as well as the interaction between these components. Next, we present the minimal functionality (interfaces) these components should support. We also show how our solution addresses the topology exchange requirements recently introduced by the NSI group, which represent a foundation for the evaluation of the topology exchange proposals within it. Finally, we explain which components should be implemented by the OpenFlow domains.

This section is based on the following publications/submissions to the standards groups, journals and conferences:

- Architecture proposal to the OGF NSI-WG Standards group [TOPO-PROPOSAL]. Currently, it is one of the three topology exchange proposals, of which one (or combination of these) will be accepted as the NSI standard.
- The paper [CONNECT-paper] which has been accepted for publication at the next edition of the Connect Journal.
- Paper [TRIDENTCOM-paper] which has been submitted to the TRIDENTCOM conference.

2.1 Topology Exchange Architecture Requirements

In order to provide the end-to-end connectivity within a multi-domain environment, one needs to calculate the (optimal) end-to-end path. This is typically performed by path-finding applications, which, on the other hand, may need to get how the topology of the individual networks look like and how these networks are interconnected. This means that the path computation element needs to obtain, in a secure and trustworthy way, the topologies of the domains.

The information exchanged cannot be tampered with, and it includes the mutual authentication between the involved parties. Further, the topology description should be unified, and it relies upon the underlying information model. The authors of [NW-MODELLING] present a state-of-the-art of the network topology information models.

We summarize the architecture requirements for topology exchange as the following:

1. Every administrative domain controls and shares its own topology information
2. Domains provide topology representations based on:
    a. the requesting party
    b. any existing bilateral agreements between neighbours
    c. other policy information
3. The system supports multiple topology information disclosure levels
4. Topology information is exchanged in a secure way between domains
5. All domains share information using the same data model

Based on these requirements, we now present the main components of the architecture.

2.2 The Architecture

As sharing of the topology information provides more flexibility than sharing of reachability information, we propose a hybrid approach to exchange topologies: the topology information is stored in a decentralized manner augmented by a centralized index to quickly point to where the topology information is stored. Because topologies are provided by

As sharing of the topology information provides more flexibility than sharing of reachability information, we propose a hybrid approach to exchange topologies: the topology information is stored in a decentralized manner augmented by a centralized index to quickly point to where the topology information is stored. Because topologies are provided by the domains themselves it is up to a particular domain what topology information it discloses.

Our architecture distinguishes the three main topology exchange components

Topology Index (TI) is a database that holds pointers to the topology providers and relevant domain information. Topology index is discussed in detail in subsection 2.3

Topology Provider (TP) hosts the topology descriptions, as detailed in subsection 2.4

Topology Consumer (TC) is the component that typically uses topologies. A plethora of topology consumers may exist and some examples of topology consumers are

- Path finding component
- Monitoring component
- Topology validation component
- Lookup component

Topology consumers are further described in subsection 2.5

A single domain may run these components or share them with other domain(s). All components take security into account; by default all communication is signed and encrypted. Our system supports Public Key Infrastructure (PKI) as a way to share public keys.

In the following subsections we elaborate on the main components of our architecture and their interaction is discussed in subsection 2.6.

2.3 Topology index

Topology index is a (logically) centralized index that holds the time stamped pointers to the topology description files in the various domains. Domains are the only holders of (complete) topology information. The *topologies are* therefore *not stored* and *maintained* by the central index. There is a

single entry per domain (i.e. topology provider) as shown at Table 2-1. This also represents a data model of the topology entry within the TI, which is used for the implementation of the components.

Domain	Version	Public key	Topology location	Neighbours	Foreign domains	Signature

Table 2-1: Topology index entry (data model)

The various entry fields for a domain ***D*** have the following meaning:

- **Domain** is the domain name of the topology provider.
- **Version** of the topology file or a time stamp.
- **Public key** of the domain originating the information. The public key plays an essential role in our infrastructure as it allows the verification of digital signatures and encryption of the messages so these can be only decrypted by the intended recipient(s).
- **Topology location** is a pointer (usually an URL) to the location of the topology file. The URL of the file has to be reachable by the topology index before the respective entry is added to the list or updated.
- **Neighbours** represent a list of domains that are directly connected to domain ***D***. In other words, the domain ***D*** has peering relationships with the neighbours.
- **Foreign domains** represent a list of domains without their respective entries in the index but with data plane connections to a domain ***D***. In other words, this is a list of domains external to the presented system. We can use foreign domains to indicate connectivity to domains that use different systems for exchanging topologies. In addition to this, a topology provider can share a link to (partial) topology information of the foreign domain.
- **Signature** contains the digital signature of the stored information. As well-known, the signature is created using the private key of the domain.

2.3.1 Populating the Topology Index

Figure 2-1 shows an example network for which the content stored at the topology is given in Table 2-2. In this example we excluded the key and signature fields. Domains D_1, D_2 and D_3 are topology providers registered in the topology index and therefore only these domains can be shown in the neighbour list. The other domains D_4 and D_5 do not run a topology provider registered to this topology index but are connected to D_1, D_2 and D_3. Therefore they are listed in the foreign domains list, respectively.

To create the foreign domain list, the topology provider sends a notification to the topology index that the topology is updated, and augments it with all domains it connects to on the data plane. The

topology index receives this full list, and compares it with the domains known to the system. The domains that are known will be listed in the neighbour's field and all the others end up in the foreign domains field. If a new domain registers at the topology index, it checks all foreign domains fields and will move them to the “neighbours” field when applicable.

Remark: A foreign domain may be represented as a tuple containing the domain name and the URL for additional topology information, e.g., ["d4.net","http://d4.net/topo/"]. The additional topology information is optional.

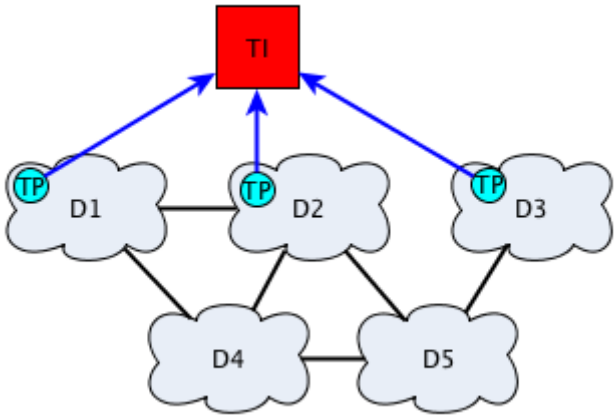


Figure 2-1: Illustrative network example.

Domain	Version	Topology Location	Neighbours	Foreign domains
D1	Version 1	https://d1.org/topo/	***D2***	***D4***
D2	Version 2	https://d2.org/topo/	***D1***	***D4 D5***
D3	Timestamp 12569537329	https://d3.org/topo/		***D5***

Table 2-2: Content of Topology Index for illustrative network example.

2.3.2 Security concerns

The topology index is considered a trusted third party. In this way it can also be used to share public keys amongst domains. These keys can be used by the topology consumer to verify the signed information it retrieves from topology providers or to encrypt the data that is being sent from the provider to the consumer.

The public keys can be exchanged out-of-band, e.g. by email. As no secret information is shared, this can be done via an unsecured connection. Naturally, there is a possibility of impersonating the index or provider when no PKI infrastructure is in place. When a domain hosts both a topology consumer

and a topology provider it will use the same private/public key for both components; this allows a topology provider to look up the key of a consumer and we do not have to maintain a second registry.

Updates or insertion of data in the table need to be signed by the originating domain with its own private key. The topology index will update the table content related to the originating domain only when the digital signature is verified by the public key of this domain.

Topology index lists the signature over the data so consumers can verify this. This will prevent malicious domains to tamper with information of other domains. To prevent false index information updates the topology index must verify who pushes the index updates. This can be done in two ways:

- The topology index maintainer first verifies and obtains both the domain name and public key via an out-of-band channel and adds both of these to the system. Any updates can now be verified by simply checking a signature.

- The public key can be distributed using for example the Domain Name System (DNS), as illustrated in Figure 2-2. The **A** or ***AAAA*** record of the server can be augmented by a ***CERT*** record containing the public key. This trust is not ultimate unless the domain uses DNSSec [RFC-4641] to sign their zone files. The advantage of this method is there's no manual intervention needed to add a domain to the system and you can verify the domain keys without using the topology index as a trusted party. Also the domain name used here must match the domain name used in the DNS system.

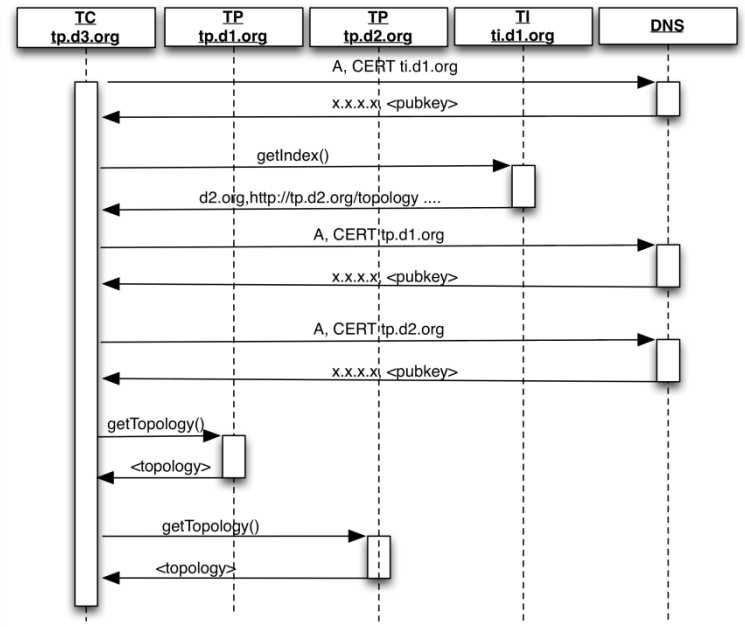


Figure 2-2: Retrieving keys using DNS

2.3.3 Synchronization and fail over

The topology index is a single point of failure. When the topology index becomes unreachable, the topology consumer can still retrieve them based on cached information. This is an unwanted situation. One approach to this problem is to replicate the index. A particular topology index can be a consumer of another topology index. This allows topology index A to receive updates whenever topology index B changes and merge this information when the version is newer. In case of a conflict, index A might have to ask the conflicting topology providers to re-send their summary information. Then topology index A sends out update notifications and topology consumers including topology index B can update their information. Topology consumers of course need to be aware of this backup server and know its public keys.

2.4 Topology Provider

The ***topology provider*** is the service that provides a topology for a specific network or domain. Topology provider shares a topology file to topology consumers and sends information to the topology index in case of an update. When necessary more advanced functionality can be implemented, like different topologies based on the consumer or decisions on via what domain the consumer comes from.

A ***topology file*** contains the topology information relevant for a given domain. The format can be arbitrary as long as it can be interpreted by the consumer. The domain decides which information it shares. The most common use will be sharing inter-domain topologies, this is a summarized version of the full topology which only contains the service endpoints and connections between domains though there is no restriction on sharing the full topology. A topology provider has to share a topology file by default this is what is supplied to any topology consumer making a request this is the default topology file.

Based on the requesting topology consumer the topology provider can supply different topology files. Therefore, different topologies of the same network may be presented to different parties. This may be used to reflect the respective policy information and to remove links that cannot be used by the consumer. If the topology consumer knows roughly what path it wants to take or knows the neighbouring domain it wants to traverse through the consumer can supply this information in the request. Based on this more filtering of topology information can be done.

This filtering is not intended to enforce policy or restriction, yet reflecting the policies in the topology file will aid the path finder skipping links that cannot be used later on, making the overall process more efficient.

2.5 Topology Consumer

The ***topology consumer*** is the component that performs an operation on the topology information. Depending on whether the topology index is open, first the consumer needs to exchange public keys

with the topology index. Only then the consumer can consult the topology index. The consumer looks for the domains it is interested in, processes the summary information when necessary and then requests the topology files from the topology providers.

2.5.1 Update notifications

A consumer can subscribe to the topology index in order to obtain the topology updates of other domains. Topology consumer can register to obtain the topology updates of certain subset of domains only. When a topology provider publishes a new or updated topology to the index, a broadcast message is sent to all subscribed consumers with the information that the new topology has been added or updated. This message is optionally signed by the topology index.

The consumers could then determine whether they want to update potentially cached information by retrieving new information from the topology index. For example adding a new domain to their list of domains, or updating existing topology information for known a domain.

In order to receive update notifications the consumer should provide a callback address to which those notifications should be sent. These services can be different implementations ran by different domains. The only prerequisite is that there is a trust relation between the consumer and the topology index.

2.5.2 Authentication

It is not necessary for the consumer to have a private and public key pair to retrieve topology information. It can verify the topology information with the public key of the provider and use the public key to securely send a symmetric session key to the provider so secure communication can take place. However, if some topology providers want to customize topologies based on the requesting domain the consumer should still be able to authenticate this with the private key of the requesting domain.

2.6 Interaction between the main architecture components

In this section we describe the basic interaction between the main architecture components. We assume that each participating domain has a single topology provider. The basic system operation can be summarized as the following:

When the provider detects an updated topology it sends a notification to the topology index, which checks the version updates the data of the respective domain (neighbours, foreign domains) when newer. The index then sends a notification to the subscribed consumers so they can contact the providers to update their topologies. Both subscribed and non-subscribed consumers poll the index regularly to avoid possible inconsistencies.

Besides, in order to exchange topology information in a secure way the following prerequisites have to be met:

- Topology index needs to be known and accessible by all consumers and topology providers.
- Topology providers have to be accessible by the topology index.
- Topology index and each topology provider need a respective private/public key pair.
- Topology providers always present their respective public keys to the topology index.
- Topology consumers may (optionally) possess a private/public key pair.

2.6.1 Topology update and retrieval

Figure 2-3 shows that when an update is made to the topology the provider notifies the index and provides some summary information (number 1 at the Figure). The Topology index will in turn notify all registered consumers that there is an updated index (number 2).

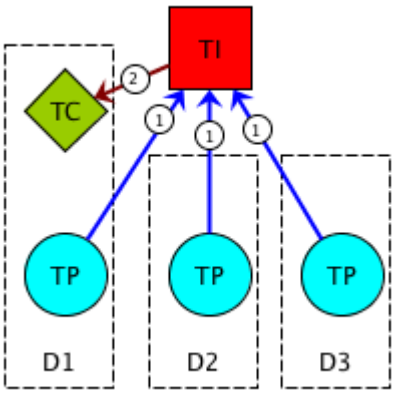


Figure 2-3: An illustration of topology update.

In Figure 2-4 the topology consumer contacts the central index and looks for updates from the topology providers it is interested in and gets their public key (1). The topology consumer contacts all the providers it is interested in and retrieves the topology files (2).

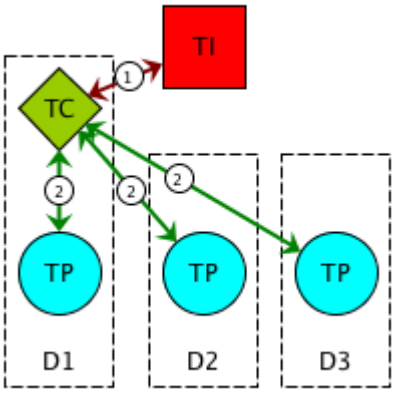


Figure 2-4: Topology retrieval

An example of a topology consumer is the path finder: First the path finder fetches all the information from the central index. This information includes pointers to topology files and summary information. The path finder creates a network graph topology, and it can prune the domains it cannot use to find the solution. Then path finder requests the topology for the relevant domains and uses the retrieved topologies to create a more detailed network graph. Finally the path finder prunes out mismatches and unusable links such that they will not be considered during the routing process and the path finder tries to find the most suitable paths and returns the optimal one.

2.6.2 Key distribution

Figure 2-5 shows the key distribution during the bootstrap phase:

- The topology providers in D_2 and D_3 send their domain's public key to the index and the index verifies the providers.
- The topology index sends its public key to consumers in D_1 and D_2, the consumers may verify the index.
- The topology consumer optionally sends its domain key to the index. In case of D_2 this is already done since it also runs a provider. In this way a consumer doesn't have to be known by the index to request information.

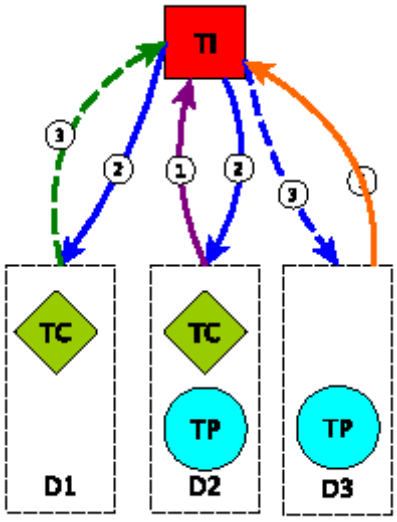


Figure 2-5: Key distribution at bootstrap phase

If a topology provider or a topology consumer cannot reach the topology URL of a particular domain, it can request from the topology index to retrieve it on its behalf. In such a scenario, a topology index acts as a proxy. After bootstrap phase, there is a trust at the Topology Index. This can be further used for key exchange during the operational phase, as illustrated in Figure 2-6. The key exchange comprises the following two steps:

- **Step 1:** Topology consumer requests the public keys of the provider domains from the index. After this step the consumer contacts the providers directly.

- **Step 2:** Topology providers may request the consumer's key from the index if they want to authenticate.

Topology providers may try multiple times to update the index when the first attempt fails. Topology index follows a more strict approach on sending its updates. For each of the updates at topology index there is a maximum number of re-attempts. When all attempts to update the topology index fail, it continues with the update of the next subscribed topology client.

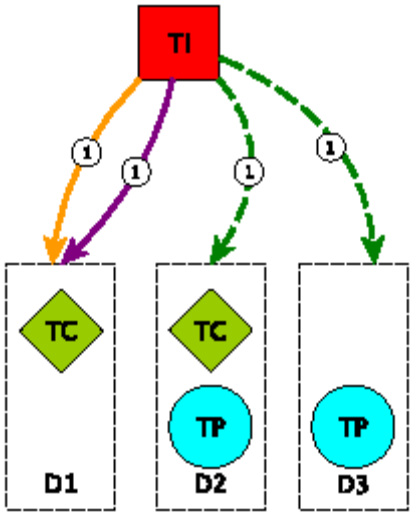


Figure 2-6: Key distribution during operational phase

2.6.3 Validation of topologies

Authenticity of information: The topology document of a domain is stored by the domain itself and it can be validated by other domains in order to identify tampering with the topology document. In other to ensure the authenticity of the information the following steps are taken:

- All topologies are digitally signed by private keys of the providing domain.
- The domain sending the topology will encrypt the topology using a shared secret that the requester shares with the sending domain. The shared secret is communicated to the sending domain by encrypting this with its public key which was retrieved from the topology index.
- Optionally the sending domain can sign the request. In this way the sending domain can be authenticated by the provider.

Correctness of format: Both the topology provider and topology consumer can validate the correctness of the topology information. When the topology information of a domain is not formatted correctly, the topology consumer (e.g. path finder) should treat this particular domain as non-existent and cannot consider the domain for path-related calculations.

Correctness of information: Only the topology provider knows whether supplied domain information is correct, and it is a sole responsibility of a topology provider to verify topology information before publishing it. Mistakes can be made and this can become an issue when a mismatch of information regarding a connection between domains occurs. In the majority of cases this needs to be solved by human intervention of the domains network engineers.

Path finders should be resilient to these mismatches and not provide them as an option to the user. This means mismatches in topology information go unnoticed to the user unless you specifically make a request to use such a link.

A way to mitigate this is to provide a separate service that contacts the topology index to request the topologies of a domain and its neighbours and to perform cross checks to aid in identifying topology mismatches. The domains network engineers can then act upon this and update their topologies accordingly.

2.7 Architecture components - Interfaces

In this subsection we describe the design of the architecture components previously discussed. We therefore specify the components' interfaces, i.e. the minimum set of functions that should be supported by each of the three main architecture components.

2.7.1 Topology Index – interface design

The following functions represent the minimal functionality that should be implemented by Topology Index:

- **getIndex()** returns the complete topology index to the requesting domain. The returned object contains information about all entries within the topology index. This method could be used by the requesting party in order to e.g. create the complete network graph.

- **updateIndex(domainId, topologyData)** updates the data in the index for the given domain. It requires the authentication of the domain's topology provider and data verification as explained in subsection 2.6.3. The update takes place only when both authentication and data verification succeed.

- **subscribeForUpdates(domainId, callbackURL)** allows a topology consumer (domain) to subscribe to the topology index for topology information updates. These updates are provided to the consumer via the supplied callback URL.

- **unsubscribe(domainId)** removes a topology consumer (domain) from the list of domains registered to obtain topology information updates.

2.7.2 Topology Provider – interface design

The following functions represent the minimal functionality that should be implemented by Topology Provider:

- **getTopology(requester)** returns customized topology that depends on the identity of the topology consumer (the requester). Depending on enforced policies between domains (of the

topology provider and the requester) the returned topology may contain more information than otherwise. In case no policy is specified for the requester, a default topology is returned.

- **getTopology(requester, route)** returns a customized topology that depends upon the following two conditions:
    - the policy between the topology provider and the consumer (requester) or
    - A list of domains supplied as route which identifies domains that the end-to-end connection traverses. The former condition precedes is latter one, i.e. in case there is a peering agreement between the topology provider and the requester, this function returns a customized topology view. In case there is no such peering agreement, the topology provider takes into account agreements with domains included in supplied route list in the same manner as **getTopology(requester).**

Remark: Peering agreements can make path finding difficult. Conflicts can be introduced that make path finding next to impossible. Local policy conflicts can be resolved by the topology provider itself before returning a customized topology result to the requester. However, it becomes complex when there are conflicts on a global level. Therefore peering agreements should be used with care, simple, and avoided when possible.

2.7.3 Topology consumer – interface design

The topology consumer may optionally implement callback function **topologyUpdate(domains)** which is used by the Topology Index to notify the consumer when there is an update in the index (e.g. a new provider or updated topology). The domains parameter is a list of domains for which a topology update has occurred. After the notification is obtained, the list of domains may be analysed by the topology consumer, and topologies from (a subset of) domains' list may be requested. This improves the efficiency of the implementation of the topology consumer, and reduces a number of requests made by the consumers. In order to receive updates the consumer needs to subscribe to the topology index with this function as callback.

2.8 Evaluation of the proposed Topology Exchange Architecture using the requirements specified by the OGF NSI working group (NSI-WG)

In this subsection we show how our solution addresses the topology exchange requirements recently introduced by the NSI group, which represent a foundation for the evaluation of the topology exchange proposals within it. This is done in the following three tables, in which column requirement specifies the requirements presented by the NSI group, and columns supported and explanation give more details whether a particular requirement is supported by our architecture, and briefly explain the solution we offer.

Table 2-3 presents the mandatory requirements, i.e. those based on core NSI principles. In the Table 2-4 the requirements that are highly desirable by NSI are presented. Those requirements resulted from the discussions within the NSI group. The last group of requirements, given in Table 2-5, present the requirements that are not yet (fully) formalized, but are put forward to NSI in order to be answered in the future.

Requirement	Supported	Explanation
Solution must allow the topology information to be shared between NSAs	Yes	Each NSA can implement topology provider (TP) and therefore share its topology information with other NSAs that would act as topology clients (TCs)
Solution **must** allow AG NSAs to aggregate topology	Yes	The AG NSA serves as TC towards NSAs that it aggregates (which are in this case TPs). The AG NSA than acts as TP on behalf of aggregated NSAs towards NSAs other than aggregated. This does not require topology index at AG NSA, but AG NSA knows each one of aggregated NSAs.
Solution **must** support chain based path signalling	Yes	The proposed topology architecture does not impact signaling at all.
Solution **must** support tree based path signalling	Yes	The proposed topology architecture does not impact signaling at all.
Solution **must** support centralized path finding for source-based routing decisions	Yes	The multiple PCE algorithms are supported by the architecture, including the source-based ones. This means that centralized path finding takes place at “source-based” PCE that relies upon “source-based” TC to obtain required topology information
Solution **must** support distributed path finding for hop-by-hop routing decisions	Yes	The multiple PCE algorithms are supported by the architecture, including the greedy ones (i.e. “hop-by-hop” algorithms).
Solution must allow the creation of a full view of network topology to perform advanced "intelligent" routing decisions	Yes	It is possible to obtain topologies from all TPs that have shared information with TI. Based on this a full network graph can be created, and used further for e.g. path-finding.

Table 2-3 NSI: Mandatory requirements

Requirement	Supported	Explanation
Topology **must** contain versioning information (e.g. Creation/modify time stamp used to determine if an older version of the topology can be replaced with a newer one)	Yes	Topology Index data model has versioning field.
Topology must be verifiable (e.g. Topology must be signed by the author before uploading to topology server. Note: Key signing infrastructure and trust relationships must be in place for this to work and only complete topologies can be signed)	Yes	Topology Index data model contains signature field as well. Topology Providers can sign the topology documents, provided that key-signing infrastructure is in place. Furthermore, TI signs with its own key information that is distributed, which can be validated by either TP or TC as they possess the corresponding keying material necessary to verify the signature. TI can apply the asymmetric key solutions in order to provide confidentiality on top of other security mechanisms, which means that only the intended recipient could derive topology information.
Topology service **should** support temporal related queries (e.g. give me anything new you have learned since <date/time>)	Not at the moment	The decision not to implement this is due to the fact that we want to keep the data model at TI as simple as possible, as well as the components within our infrastructure. Our understanding is that timestamp information could be added to the TI data model, and used to obtain the topologies changed after certain point of time in the past. However, if the intention is to obtain **difference** ("diff") from the previous topologies (e.g. ports removed/added in the meantime) we do not support this at the moment, and extension of the TI would be necessary.
Topology service **should** support stand-alone model: single server serving single topology	Yes	
Topology service **should** support central model: single server serving multiple topologies	Yes	
Topology service **should** support distributed model: topology information can be "flooded" between topology servers	No	We have conceptually different approach because the flooding model leads to inconsistencies in the overall network topology representations (due to timing issues). We have implemented callback mechanism instead in which each topology client could register to obtain updates.
Should support getTopology - allows a user (which can be behind a firewall) to get one or more topologies	Yes	This is implemented by TCs and TPs.
Should support putTopology - allows a service provider to	Yes/No	We do not have a centralized topology server, instead TP notifies TI about the topology (URL, version, etc).

Requirement	Supported	Explanation
upload its topology onto the topology server		
Should support subscribeTopology - allows a user to subscribe to receive any changes in the topology information, resulting in a "push" of the new data	Yes	
Should support unsubscribeTopology - allows a user to delete their subscription	Yes	The entry is removed from TI

Table 2-4 NSI: Highly desirable requirements

Requirement	Supported	Explanation
Support distribution of topology for purposes other than path-finding, e.g. monitoring, measurements, and visualization	Yes	We have demonstrated how our solution could be used for e.g. monitoring. Besides, our architecture could be used to distribute any type of documents within the network (e.g. resource documents)
Support separate topology views for different user groups	Yes	Each user group could have its own TC.
Support distribution of documents other than topology (e.d. NSA description document, SLAs, etc)	Not clear	The architecture is generic and can be used to distribute any kind of documents. However, a separate implementation of the components would be necessary to support other documents.
Support application/project/deployment specific path finding for different user groups	Yes	There could be PCEs per user groups
Support expiration or revocation of topology	Yes	Supported by TPs themselves.
Support for applying local policy via modification of the topology/reachability information	Yes	TP decides, based on the requesting party which information to share.

Requirement	Supported	Explanation
Supports policy-based routing enforcement	Partially	Depends on policies in place. All policies that could "translate" to one of the following: - include/exclude domain(s) (could be ordered list) - include/exclude link(s) are supported by our solution. However, an implementation of the policy evaluation is necessary.
Support for notification and removal of misbehaving topology services	Yes	TI could delete the key of the TP, as well as data entries relating to this misbehaving TP.

Table 2-5 NSI: Not yet formalized requirements

2.9 Topology Exchange Architecture – Conclusions and Recommendations

In this section we have presented architecture for topology exchange within the NSI. Although this solution is generic (i.e. it could be used in any networked environment), it fulfils many requirements for the successful implementation within NSI. From the perspective of MOTE project, it supports push/pull topology update mechanism, which is one of the basic principles for the automated topology exchange solution. This means that notifications about any relevant domain topology updates would be propagated. The main requirement for the OpenFlow domains is therefore to implement both Topology Client and Topology Provider architecture components.

As the documents exchanged within NSI are using NML standard format, it goes without saying that the OpenFlow domains should export their topologies in this format. Therefore, in the next sections, we investigate the topology information within OpenFlow domains and whether NML format can be used to model OpenFlow topologies. Subsequently we analyse how to convert the topology representation within OpenFlow into the NML format, and how to exchange this topology information.

3 Experiments

In this section we describe different experiments performed with topology software. We first describe the setup that has been used for some of the experiments, the MOTE OpenFlow testbed. Then we briefly introduce the main topology discovery and topology representation at OpenFlow controllers, and explain in more detail how to access topology information from the Floodlight, the OpenFlow controller of our choice. Next we describe the OpenFlow topology discovery experiments (based on LLDP) performed using MOTE testbed and how is this information presented at OpenFlow controllers. We discuss the main issues with LLDP topology discovery and how to address them. Next, we describe the implementation and experiments with the topology exchange architecture. Then we discuss the extensions required at OpenFlow domains in order to allow topology exchange between OpenFlow and NSI domains. We conclude this section with recommendations and observations from our experiments.

3.1 MOTE Testbed

The MOTE project planning states that multiple connected OpenFlow networks with separate controllers are required are required to perform the experiments. The minimal required environment was specified to consist of the two OpenFlow domains, each running its own OpenFlow controller. These domains were deployed at SURFsara and UvA, respectively. However, the actual deployment of the testbed exceeded the planning, as a total of eight OpenFlow-enabled switches were obtained for the testbed. This allowed a lot of freedom for re-configuring the network, and deployment of multiple (sub)-domains within the testbed itself. Hence there was no need to extend the testbed including other NRENs. The deployed testbed at UvA is shown in Figure 3-1. A similar testbed setup was deployed at SURFsara, and hence is not shown. Figure also illustrates the two sites were connected with a deployed (optical) link between them.

The switches that are used within MOTE testbed are: Pica8 P-3290, HP 2920-24G (J9726A), Dell Networking N2024, Dell Networking N3024. The reasoning for this deployment was to use as many as possible OpenFlow-enabled switches from different vendors in order to detect any compatibility issues (e.g. with LLDP).

At Figure 3-1 an overview of the ports used to connect switches is also shown. Besides, the two Dell switches for this particular experimental setup were configured as “non-OpenFlow-enabled” switches. Different controllers were deployed for Pica8 and HP switches, respectively, thus resulting in two OF domains (1 and 2). Finally, we configured three virtual bridges at the Pica8 switch within OF domain 1. This testbed configuration allowed us to experiment with setups in which OF domains (“islands”)

are connected through non-OF domains and to observe the topologies discovered by the controllers at OF domains.

The complete UvA MOTE testbed setup, with an overview of the controllers, hosts and switches, and a (partial) mesh network topology is shown in Figure 3-2.

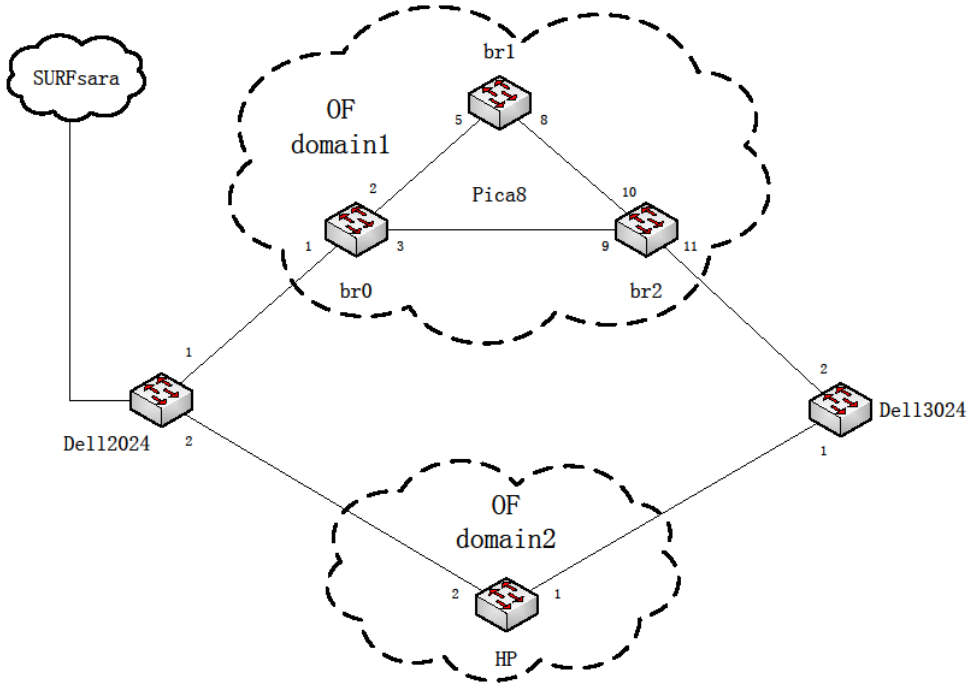
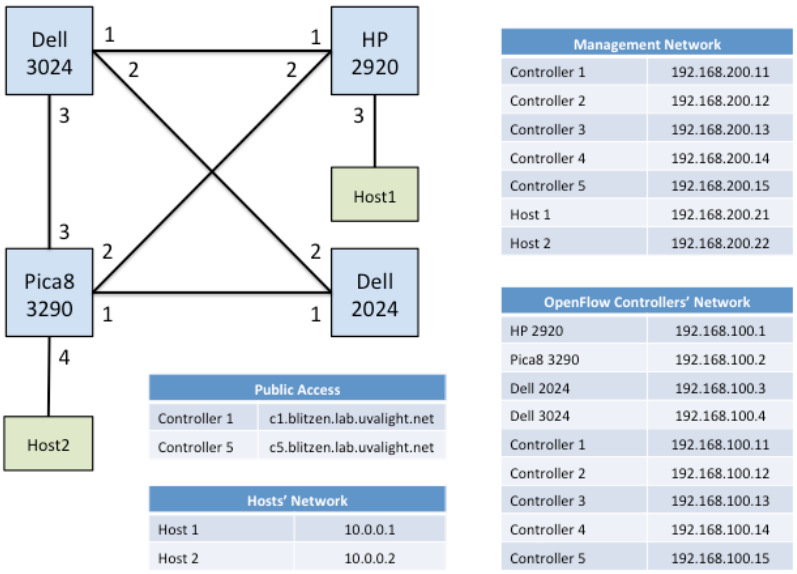


Figure 3-1: MOTE testbed example deployment at UvA.



Management Network	
Controller 1	192.168.200.11
Controller 2	192.168.200.12
Controller 3	192.168.200.13
Controller 4	192.168.200.14
Controller 5	192.168.200.15
Host 1	192.168.200.21
Host 2	192.168.200.22

OpenFlow Controllers' Network	
HP 2920	192.168.100.1
Pica8 3290	192.168.100.2
Dell 2024	192.168.100.3
Dell 3024	192.168.100.4
Controller 1	192.168.100.11
Controller 2	192.168.100.12
Controller 3	192.168.100.13
Controller 4	192.168.100.14
Controller 5	192.168.100.15

Public Access	
Controller 1	c1.blitzen.lab.uvalight.net
Controller 5	c5.blitzen.lab.uvalight.net

Hosts' Network	
Host 1	10.0.0.1
Host 2	10.0.0.2

Figure 3-2: Complete MOTE testbed at UvA

3.2 Topology Discovery at OpenFlow domains

Network topology discovery is the process of automatically generating a network topology. Models that describe the working of computer networks often distinguish different functional layers. As a consequence, Network Topology Discovery can be done on different layers of the network. The physical topology, for example, gives an overview of the physical interconnections between all devices in the network. The logical topology displays the data flow between devices according to the protocols that are used on the different functional layers. Examples are Ethernet, or Internet Protocol (IP).

We define the topology discovery within OpenFlow domain as the discovery of switches, links, and hosts within this domain. A switch is initially configured with a master controller IP address and a set of slave controller IP addresses. When the switch establishes a (TCP) connection to a controller, controller sends a feature request message to the switch and waits for a reply. When the reply reaches to the controller, controller gets informed about the features provided by the switch, for instance, the datapath ID (i.e., DPID), list of ports, etc. The link discovery is based on LLDP and BDDP protocols.

Currently there is a plethora of the (open source) OpenFlow controllers which means different topology discovery implementations by different controllers and, more important, (very) different topology formats may be used by these controllers. A brief overview of the main features of some of the most popular controllers is as the following:

A RYU controller has a basic topology discovery application which monitors datapaths and launches topology events when topology changes occur. The current controller implementation provides a parser skeleton used only to debug topology-relating events. This requires extension of the supplied

topology discovery application and is not clear how would this integrate with topology exchange components. Similar observations could be made for POX controller.

Floodlight controller provides, in our opinion, the most suitable topology discovery functionalities. This controller is developed using a modular architecture and the modules are either controller or application ones. It allows for easy and fast integration of developed modules which we consider important as it reduces the development cycle. Besides, in order to establish end-to-end connectivity, it is important to have functionality that supports static flow pushing. This has been made available in Floodlight via REST API. Due to the fact that our solution requires that topology client and topology provider are implemented for each controller, and taking into account limited effort allocated for this activity, we have chosen to perform topology experiments using exclusively Floodlight controller.

3.2.1 Relevant modules within Floodlight Controller

The modular architecture of Floodlight controller is shown in Figure 3-3. From the topology perspective, the most important modules are Link Discovery Manager, Topology Service and Device Manager. The Device Manager keeps track of the devices or hosts via OF PacketIn requests. By using these requests it can also learn to which port of which switch a particular device is connected. The Topology Service computes topologies based on link information from the Link Discovery Manager. A term “OpenFlow island” is used to indicate a group of connected OpenFlow-enabled switches that are managed by the same instance of an Floodlight. In addition the islands can be interconnected using non-OpenFlow switches on the same layer 2 domain.

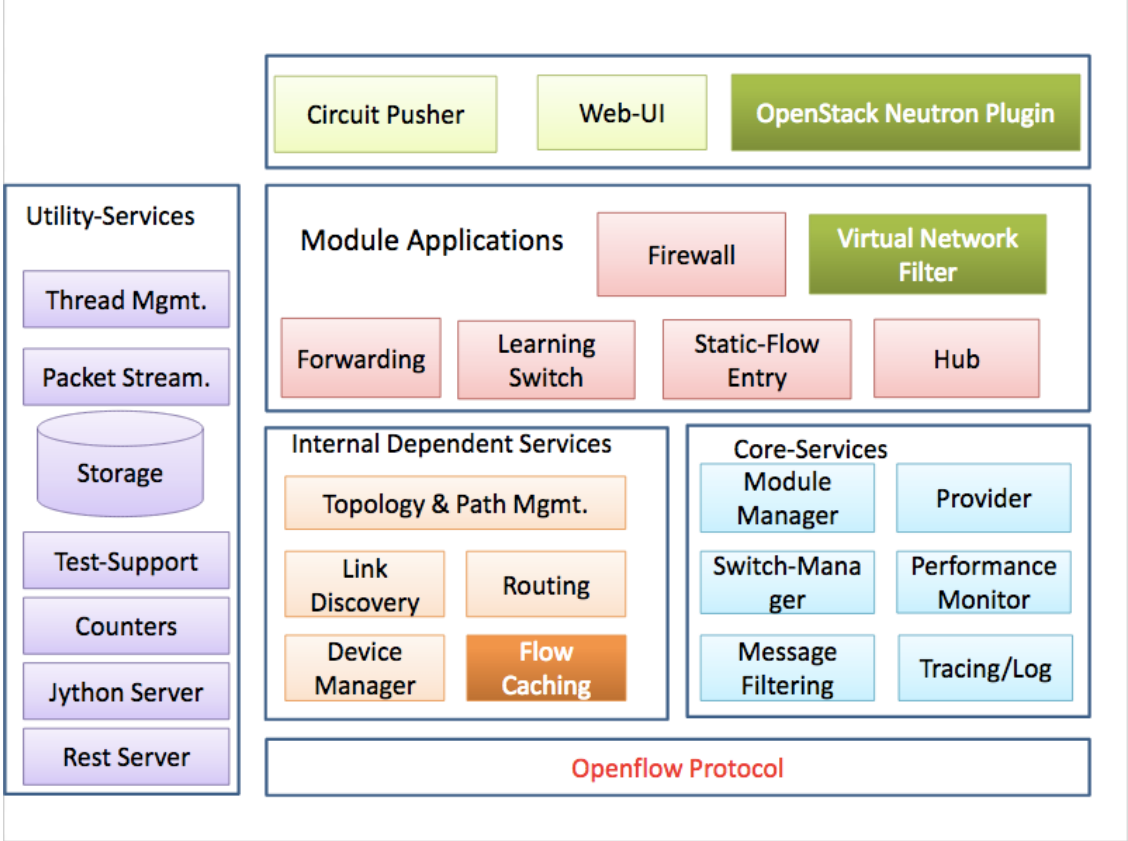


Figure 3-3: Floodlight Modular Architecture

The Topology Service computes topologies based on link information it learns from the Link Discovery Service. All the information about the current topology is stored in an immutable data structure called the ***topology instance***. If there is any change in the topology, a new instance is created and the topology changed notification message is called.

We need to implement Topology Client and Topology Provider, as well as path computation element (PCE) in order to provide for automatic topology exchange and, accordingly path re-calculations based on these updates. We have decided to implement these as Floodlight modules. These modules need to listen for changes in topology so they need to implement the ITopologyListener interface.

3.3 Topology Discovery at OpenFlow domains – Experiments

We investigated different topology setups at MOTE testbed and here we show the most insightful experiments. For each setup, a Floodlight topology representation (switches, links, etc.) has been obtained using REST API.

3.3.1 Experimental setup 1

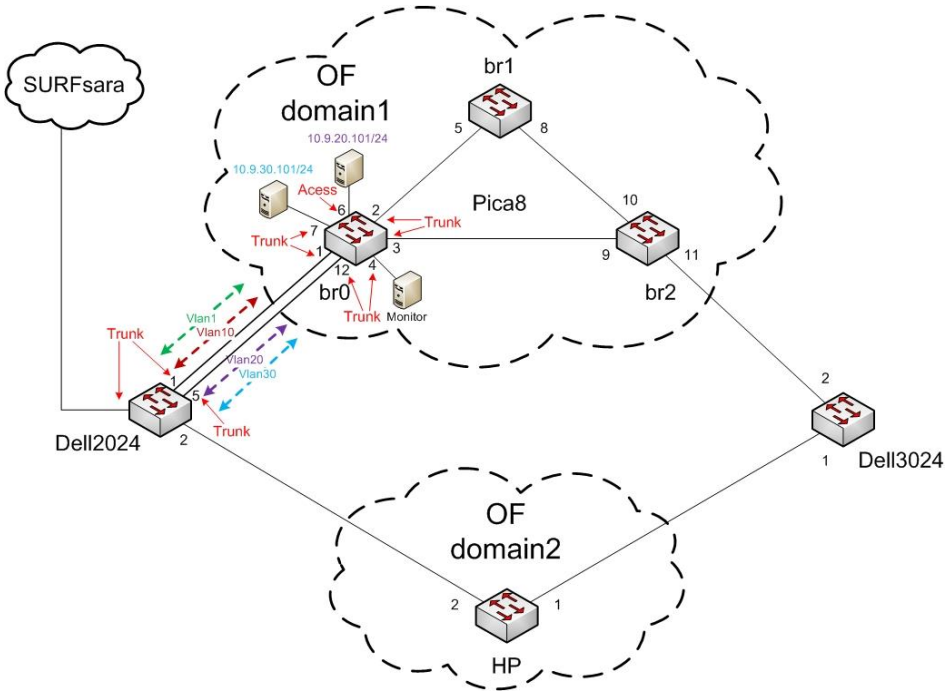


Figure 3-4: Topology discovery setup one

The first setup is shown at Figure 3-4. This setup consists of two OF domains that are interconnected through “conventional” Dell switches. The default configuration of each one of the conventional (Dell) switches does not have LLDP support enabled, so we had to enable it. The OF domains (HP and Pica8, respectively) in this setup are not the OF islands, as we configured these to be in two actual separate domains. Each OF domain is controlled by a separate Floodlight controller. Besides, we have configured three virtual bridges within Pica8 switch. Using the REST API and standard curl scripts, we have obtained the results from the Pica8 Floodlight controller. These results are shown in compressed format in Appendix A.

The results show that the Pica8 controller obtains erroneous information about the network topology, i.e. port 12 from Pica8 is “linked” to port 11 of the same switch, and this type of link is *external*. On the other hand, port 12 appears to be linked to port 5 of the Dell 2024 switch, and this type of link is external as well. This is correct information. On the other hand side, the Pica8 switch cluster contains

correct information about the switches (bridges br0, br1 and br2) that are within the same domain. Similar conclusions could be drawn for ports 1 and 12 of Pica8 switch. So, in order to remove these incorrectly identified loops during the topology discovery the procedure is as the following:

1. Identify switches from the same cluster

2. Remove any “external link” entry between the switches from the same cluster.

The main issue here is that this procedure would work for Floodlight controllers only. It is highly likely this solution is not applicable to OpenFlow domains that have other type(s) of controllers deployed. Therefore, for each of these domains, a fresh look into this issue is mandatory.

3.3.2 Experimental setup 2

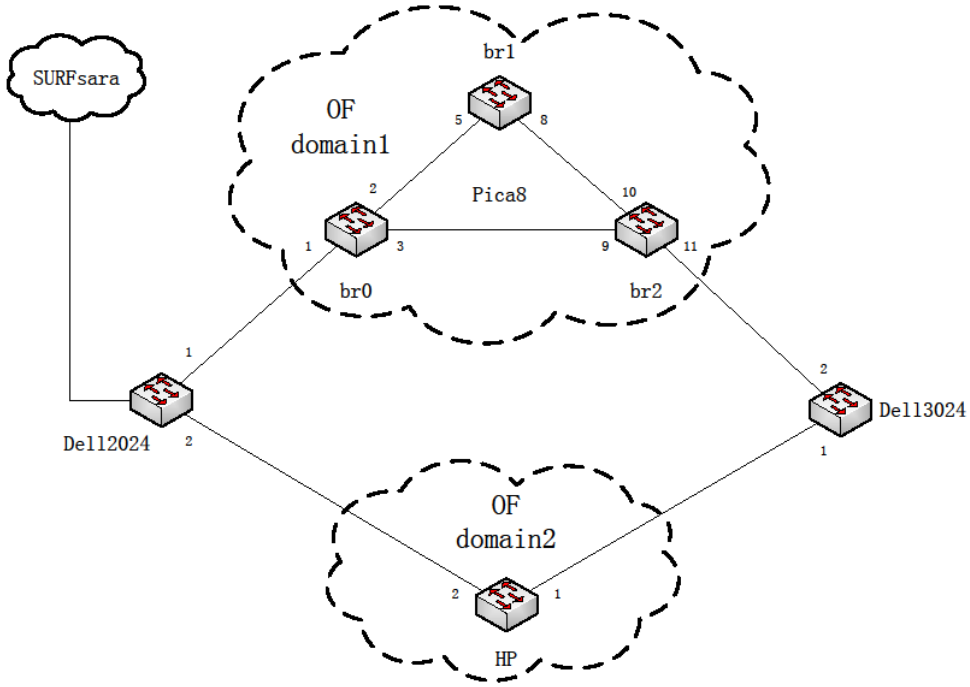


Figure 3-5 Topology discovery setup two

The first setup is shown at Figure 3-5. Different from previous one, there is a single link between any two given switches in the domain. The similar loop issue as identified in the previous setup. Yet another issue may be illustrated here. It can be resolved that Pica8 port 1 is connected to a different domain, i.e. the one that contains switch Dell 2024. In order to automatically create topology description within NML format, a domain ID/remote port ID resolution is necessary. In this particular case, Dell domain ID should be known by Pica8. We suggest to configure the LLDP port description of the switches for this purpose. *This might not be supported by every switch.*

A snippet of the configuration script for Dell 2024 that allows this is as the following:

```
interface Gi1/0/1
     description "MOTE Dell 2024 port 1"
     switchport mode trunk
     switchport trunk allowed vlan 1,10,20,30
     lldp transmit-tlv port-desc sys-name sys-desc
sys-cap
     lldp transmit-mgmt
     lldp notification
exit
```

This LLDP modification worked flawlessly on all switches we have applied this solution (Dell and Pica8). However, the question is whether this applies to every single switch that is deployed nowadays. This is therefore yet another point of attention for any further deployments of the MOTE solution.

3.3.3 Recommendations and Conclusions

The topology discovery experiments clearly indicated some of the most important issues and some points of attention:

1. The “off-the-shelf” configuration of the switches need to be checked, as we have encountered the issue that LLDP was not enabled by default (Dell switches in our testbed)

2. In order to facilitate the resolution of the remote domain to which a port is attached to, modification of the LLDP port description may be necessary. This LLDP modification has to be supported by the controller, i.e. the controller needs to extract this information from LLDP packets. This is true for the Floodlight controller, but need to be verified for other controllers as well. Besides, the switches that are deployed in other domains need to support LLDP port description configuration as well.

3. The main issue encountered is the one with the loops, i.e. that two ports that belong to the same OF domain are presented as connected through and external link. The proposed solution to look in the topology discovery results applies to the Floodlight controllers only. This may not be a solution for any OF domain with different type of controller. Therefore, further investigation of this issue for different controllers is mandatory.

4. The Floodlight controller has very rich topology representation format, and offers a solid REST-based API to automatically deduce intra-domain topology, as well as resolution of the local port – link – remote port/domain constructions.

3.4 Topology Exchange Architecture experiments

We deployed our proof-of-concept architecture implementation within the Automated GOLE environment. The proof-of-concept was also demonstrated during the SuperComputing conference in November 2014. The implementation consists of (a single) Topology Index, multiple Topology Providers, and different Topology Consumers (e.g. lookup service, path finder). No security (e.g. authentication, signatures) was implemented. The Automated GOLE testbed shown in Figure 3-6 consists of dedicated switching hardware at the various GLIF Open Lightpath Exchanges (GOLEs). The testbed represents a worldwide implementation of the NSI standard in multiple Open Exchanges. The links in between GOLEs provide a certain amount of bandwidth, either as a full link dedicated to the Automated GOLE, or sub--rate circuits that may be allocated.

Figure 3-6: Automated GOLE testbed.

3.4.1 Implementation Details

The topology exchange implementation for the Automated GOLE is written in Python using Flask [FLASK]. The architecture components act as web services and exchange messages in JSON format on top of HTTP. The components ran together with a GUI on a virtual machine with two cores and 2 GB of RAM. In total there were 24 domains and 265 network endpoints shared as topology documents specified in NML. The memory usage by different components remained stable regardless the number of topology updates, and was at most 30 MB for topology provider. Flask uses about 14 MB of memory and all the components are built on top of this. The memory usage of the components also depends on the size of the topology it serves.

Figure 3-7 shows a picture of the topology exchange running on top of the Automated GOLE. The participating networks currently provide their topology documents using the Document Discovery Service (DDS). All individual topologies are aggregated and presented as a single XML document. The

Automated GOLE uses the standardized NML format to describe their topologies. Therefore, we developed a simple NML parser and integrated it into the Topology Provider.

Normally each network would run its own topology provider - however we used the aggregated DDS document and extracted the NML documents for each domain. For every domain we launched a separate topology provider. The topology providers checked the Netherlight aggregator at a regular interval for new versions of topology documents.

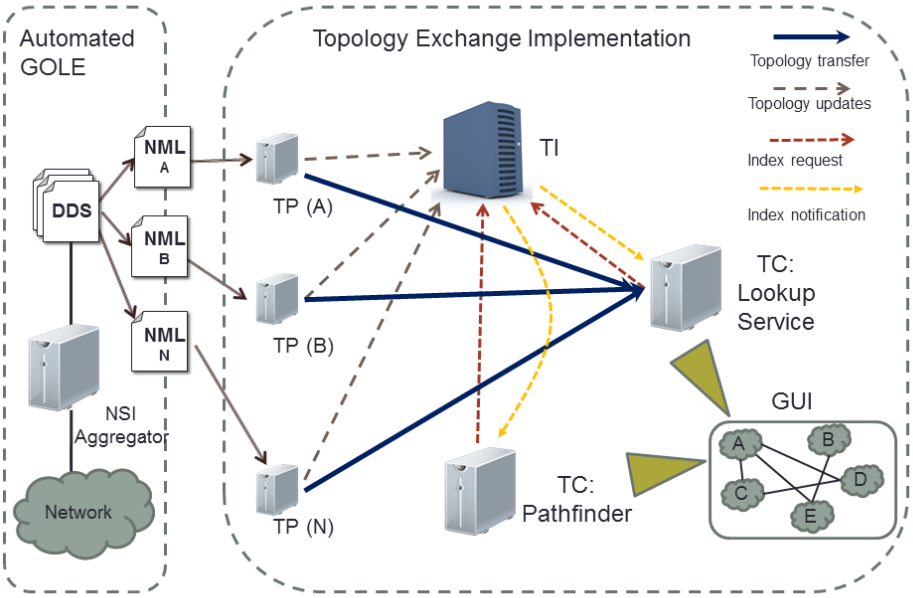


Figure 3-7: Topology exchange implementation

A couple of topology consumers were implemented, namely a Lookup Service (LS) to find what domain belongs to an endpoint, and a Path Finder to calculate the shortest path between two endpoints. All consumers subscribe to the topology index and receive and process the incoming updates. The lookup service keeps tracks of all topologies. In addition, to avoid inconsistency as a result of lost updates, it regularly requests updates from the topology index. The path finder operates differently. To reduce overall load it keeps (in memory) the topologies of the domains previously used and updates them when notified.

When a new request comes in, the path finder retrieves the topologies for the domains that are not stored in memory and does the path calculation. The resulting path is then stored in memory. The newly retrieved topologies are also kept in memory and are being tracked. To optimize response time on frequently requested paths, the path finder provides the previously calculated path to all requests within certain time interval that can be specified. After the expiration of such time interval, that particular path is re-calculated again. This may have a penalty of possible inconsistency, which needs to be addressed in the future work.

4 NML representation of the OpeFlow domain

In this section we briefly summarize the main conclusions and recommendations from the MOTE MS 1.1 deliverable [MOTE-MS1.1]. The Network Markup Language (NML) base schema defines a generic structure for layered networks, based on the concepts defined in ITU-T G.800 [G.800]. The base schema is technology-agnostic. In order to describe a specific network layer, an extension is required.

In the current implementation of Network Service Interface (NSI) connection service, each Service Termination Point (STP) is technically a client VLAN (C-VLAN). The proposed NML-Ethernet extension defines the subset of the Ethernet standards that is sufficient for the NSI use case. Since OpenFlow is also able to router flows based on client VLANs, this allows NSI and OpenFlow to interwork. Both OpenFlow as well as NSI connection service use VLANs as the de-facto resource labels for inter-domain traffic engineering, although both are designed to cope with other labels in the future (e.g. MPLS labels). It is trivial to extend NML with such support.

The main conclusion is that the existing NML base standard is sufficient to describe OpenFlow network topologies. This is impressive result taking into account that NML has been proposed long time before OpenFlow concept has been introduced at all.

5 Extensions to the Floodlight Controller

In this section we briefly describe the main features of the Topology Provider and Topology Client when deployed at the Floodlight controller. Both components are deployed as Floodlight modules.

5.1 Topology Provider

As already discussed, the Floodlight internally discovers and maintains the network topology. The Topology Service computes topologies based on link information it learns from the Link Discovery Service. All the information about the current topology is stored in an immutable data structure called the topology instance. If there is any change in the topology, a new instance is created and the topology changed notification message is called. Floodlight exposes, among others, API for topology queries, as well is topology listener interface.

The Topology Provider needs to listen for changes in topology which means it needs to implement ITopologyListener interface. Once a change occurs, TP obtains the topology from controller, resolves the loops issue (as explained in subsection 3.3.1), and then uses NMLBuilder to create topology model in NML format. Once this is done, TP notifies Topology Index about the new version of the OF-domain topology. The rest of the functionality has already been explained.

A code snippet is as the following:

```
public class TopologyProvider implements IFloodlightModule,
ILinkDiscoveryListener {
      protected IFloodlightProviderService floodlightProvider;
      protected ILinkDiscoveryService linkDiscoverer;
      …
      @Override
      public void init(FloodlightModuleContext context) {

            this.floodlightProvider =

       context.getServiceImpl(IFloodlightProviderService.class);
```

```
            this.linkDiscoverer =

            context.getServiceImpl(ILinkDiscoveryService.class);

            //add self as one of the link events listeners
            this.linkDiscoverer.addListener(this);
    …
}
    //Some of the topology queries related to device status

    this.floodlightProvider.getAllSwitchDpids();
    //Get a particular switch
    IOFSwitch switch = this.floodlightProvider.getSwitch(switchID);

    //Get ports on a switch
    Collection<ImmutablePort> ports = switch.getPorts();

    //Some of the code relating to connectivity status
    //Get all links
    Map<Link, LinkInfo> links = this.linkDiscoverer.getLinks();

    //Get end points of a link
    Link link = …;
    long dstDpid = link.getDst();
    long srcDpid = link.getSrc();
    short dstPort = link.getDstPort();
    short srcPort = link.getSrcPort();
}
```

5.2 Path finding (topology client)

The topology client implemented for the OF domains is the path finding element. There is no difference in functionality already explained for the topology architecture. Path finder has to obtain the NML format for its own domain as well, as this is maintained by the topology provider. We have implemented the basic path finding algorithm explained in [TOPO-PROPOSAL]. We briefly state the main features of the path finding algorithm here – omitted details could be found in cited reference.

Though conventional shortest path algorithms such as the Dijkstra's algorithm may be used by the path finder to find a simple, shortest inter-domain path between two distinct domains, a more intelligent algorithm is needed to accept more path requirement details, and provide an inter-domain path that satisfies the given requirements in return.

Examples of path requirements that could be requested by the clients are:

1. A set of domains must not be part of the multi-domain path.

2. A set of inter-domain links must not be part of the multi-domain path.

3. A set of domains must be part of the multi-domain path.

4. A set of inter-domain links must be part of the multi-domain path.

5. A set of domains must be in a predefined sequence in the multi-domain path.

The first two requirements can be fulfilled by pruning out all of the forbidden domains and inter-domain links before creating the topology graph that would be used for path computation. The resultant topology graph would is treated as the input to the algorithm. The last three requirements are considered during the path computation process.

5.3 Experiments

We performed two sets of topology exchange experiments:

1. Experiment that verified topology exchange between two OF domains (illustrated in Figure 5-1). Each of the two OF domains contains a single switch (SW), single host (H), Floodlight controller (FL), and Topology Provider (TP), respectively. Figure also illustrates steps that happen once the topology provider is notified about change in local topology.

2. Experiment that extended the experiment explained in subsection (illustrated in Figure 5-2). While in the first experimental setup we had a path finding component that resides at an OF domain, in this case we have a centralized path finder that is located somewhere in the network.

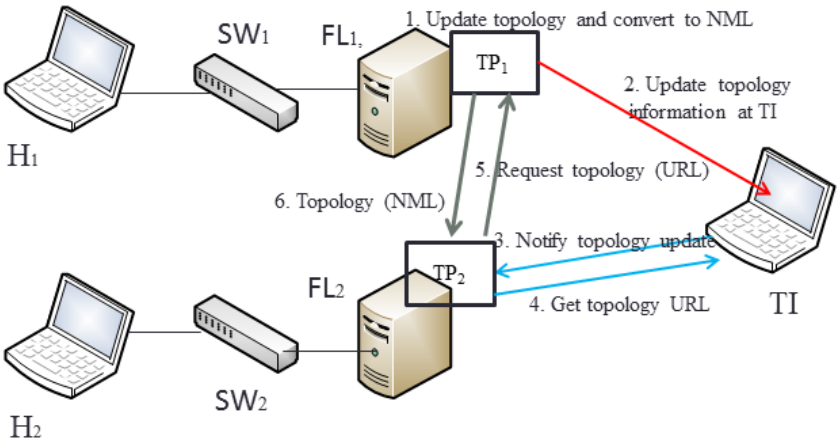


Figure 5-1: Topology exchange between two OF domains.

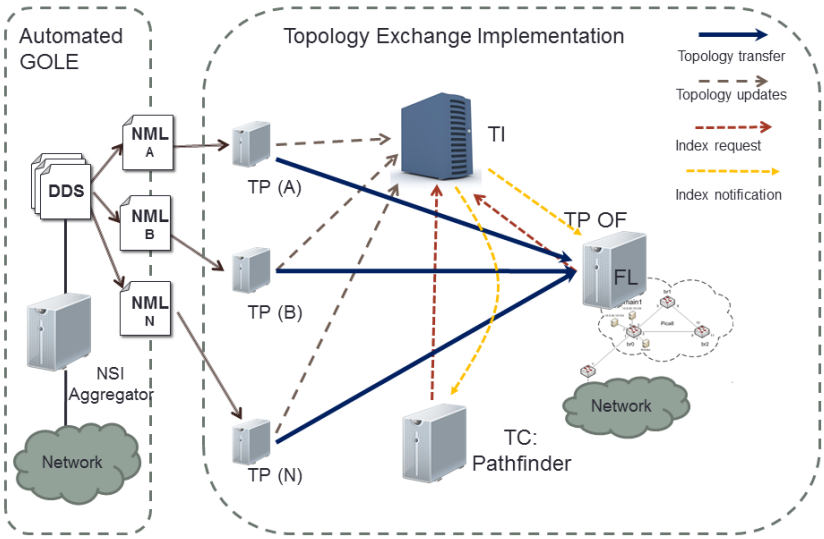


Figure 5-2: Topology exchange between OF domain and Automated GOLE.

Both sets of experiments verified that our topology exchange architecture could be extended with support for OF domains within NSI or could support network that consist of different OF domains only. This holds as long as the controller used by OF domains is Floodlight. In the future, the end-to-end connectivity need to be addressed. In practice, Floodlight is suitable for this implementation of such functionality as well, as it provides for Static Flow Pusher module that is exposed via REST API.

6 Conclusions

The MOTE project had as a goal to research methods for describing and exchanging topologies between NSI and OpenFlow networks. This would allow the NSI to become aware of these networks. On the other hand, network topologies are not static, as networks are constantly upgraded and maintained, which results in ever changing topologies. This would require from the topology exchange solutions to automatically propagate topology changes between different domains.

The topology exchange architecture delivered by the MOTE project provides a method to easily integrate different topology clients and flawlessly supports exchange of topologies specified in NML (or other previously agreed) format between NSI and OpenFlow networks. The NML also proved to be suitable for standardizing topology instances maintained by versatile OpenFlow controllers, as each controller has its own topology format.

The main technical concern is the automated topology discovery within OpenFlow domains. Topology discovery is based on Link Layer Discovery Protocol (LLDP) and, to be of use, this protocol has to be supported (and configured by default) by other (conventional or OpenFlow-enabled) switches. In order to allow us to easily build NML documents within the OpenFlow domains, a way to identify the neighbouring domains is necessary. We have solved this by extending the LLDP port description. Again, this LLDP modification has to be supported by the all switches, thus including the switches that are deployed in other domains.

However, the main issue encountered with topology discovery is the loop issue, i.e. scenario when two ports that belong to the same OF domain are presented as connected through an external link. Careful examinationof Floodlight, the chosen controller to build proof-of-concept allowed us to find a way to resolve this issue. However, the proposed solution is not applicable to other OpenFlow controllers and represents, next to the fact that different controllers use different formats to model topology (if at all!) two challenges that have to be addressed in future extensions of the MOTE work.

Besides these accomplishments, the results of MOTE made impact within NSI community as topology exchange architecture is considered for standardization, and two additional standard proposals are in pipeline. A successful demo of the topology exchange architecture was given at the last edition of SuperComputing, and another one is scheduled for the next edition of the GENI conference. Last, but not least, a MOTE article will appear in the next edition of CONNECT magazine, and a scientific paper is submitted to the well-known conference.

Appendix A Experimental results

A.1 Topology discovery experiment 1

```
pica8@c1:~$ curl -s http://127.0.0.1:8080/wm/device/

[{"entityClass":"DefaultEntityClass","mac":["82:5d:63:8e:df:4f"],"ipv4":["
10.9.30.101"],"vlan":[],"attachmentPoint":[{"switchDPID":"5e:3e:08:9e:01:e
9:95:12","errorStatus":null,"port":12},{"switchDPID":"8f:0e:08:9e:01:e9:95
:12","errorStatus":null,"port":5},{"switchDPID":"e5:e6:08:9e:01:e9:95:12",
"errorStatus":"duplicate-device","port":9}], "lastSeen":1426524419527},

{"entityClass":"DefaultEntityClass","mac":["f8:b1:56:2c:0f:64"],"ipv4":[],
"vlan":[],
"attachmentPoint":[{"switchDPID":"5e:3e:08:9e:01:e9:95:12","errorStatus":n
ull,"port":12}], "lastSeen":1426524320249},

{"entityClass":"DefaultEntityClass","mac":["f8:b1:56:2b:9b:40"],"ipv4":[],
"vlan":[],
"attachmentPoint":[{"switchDPID":"5e:3e:08:9e:01:e9:95:12","errorStatus":n
ull,"port":12},
{"switchDPID":"e5:e6:08:9e:01:e9:95:12","errorStatus":null,"port":11}],"la
stSeen":1426524417066},

{"entityClass":"DefaultEntityClass","mac":["f8:b1:56:2c:0f:63"],"ipv4":[],
"vlan":[],
"attachmentPoint":[{"switchDPID":"5e:3e:08:9e:01:e9:95:12","errorStatus":n
ull,"port":12},{"switchDPID":"e5:e6:08:9e:01:e9:95:12","errorStatus":null,
"port":9}],"lastSeen":1426524420955},

{"entityClass":"DefaultEntityClass","mac":["f8:b1:56:2b:9b:41"],"ipv4":[],
"vlan":[],
"attachmentPoint":[{"switchDPID":"5e:3e:08:9e:01:e9:95:12","errorStatus":n
ull,"port":12},{"switchDPID":"e5:e6:08:9e:01:e9:95:12","errorStatus":null,
"port":9}],"lastSeen":1426524420733},

{"entityClass":"DefaultEntityClass","mac":["36:9d:f1:47:1b:74"],"ipv4":["1
0.9.20.101"],
```

"vlan":[],"attachmentPoint":[{"switchDPID":"5e:3e:08:9e:01:e9:95:12","erro
rStatus":null,"port":12},{"switchDPID":"8f:0e:08:9e:01:e9:95:12","errorSta
tus":null,"port":5}], "lastSeen":1426524421165}]

pica8@c1:~$ curl -s http://127.0.0.1:8080/wm/topology/switchclusters/json

{"e5:e6:08:9e:01:e9:95:12":["e5:e6:08:9e:01:e9:95:12"],
"8f:0e:08:9e:01:e9:95:12":["5e:3e:08:9e:01:e9:95:12","8f:0e:08:9e:01:e9:95
:12"]}

Pica8:~$ curl -s http://127.0.0.1:8080/wm/topology/links/json

[{"src-switch":"5e:3e:08:9e:01:e9:95:12","src-port":2,"src-port-
state":0,"dst-switch":"8f:0e:08:9e:01:e9:95:12","dst-port":5,"dst-port-
state":0,"type":"internal"},

{"src-switch":"5e:3e:08:9e:01:e9:95:12","src-port":12,"src-port-
state":0,"dst-switch":"5e:3e:08:9e:01:e9:95:12","dst-port":1,"dst-port-
state":0,"type":"external"},

{"src-switch":"5e:3e:08:9e:01:e9:95:12","src-port":3,"src-port-
state":0,"dst-switch":"e5:e6:08:9e:01:e9:95:12","dst-port":9,"dst-port-
state":0,"type":"internal"},

{"src-switch":"e5:e6:08:9e:01:e9:95:12","src-port":10,"src-port-
state":0,"dst-switch":"8f:0e:08:9e:01:e9:95:12","dst-port":8,"dst-port-
state":0,"type":"internal"},

{"src-switch":"8f:0e:08:9e:01:e9:95:12","src-port":5,"src-port-
state":0,"dst-switch":"5e:3e:08:9e:01:e9:95:12","dst-port":2,"dst-port-
state":0,"type":"internal"},

{"src-switch":"5e:3e:08:9e:01:e9:95:12","src-port":1,"src-port-
state":0,"dst-switch":"5e:3e:08:9e:01:e9:95:12","dst-port":12,"dst-port-
state":0,"type":"external"},

{"src-switch":"e5:e6:08:9e:01:e9:95:12","src-port":11,"src-port-
state":0,"dst-switch":"5e:3e:08:9e:01:e9:95:12","dst-port":1,"dst-port-
state":0,"type":"external"},

{"src-switch":"5e:3e:08:9e:01:e9:95:12","src-port":1,"src-port-
state":0,"dst-switch":"e5:e6:08:9e:01:e9:95:12","dst-port":11,"dst-port-
state":0,"type":"external"},

{"src-switch":"8f:0e:08:9e:01:e9:95:12","src-port":8,"src-port-state":0,"dst-switch":"e5:e6:08:9e:01:e9:95:12","dst-port":10,"dst-port-state":0,"type":"internal"},

{"src-switch":"e5:e6:08:9e:01:e9:95:12","src-port":9,"src-port-state":0,"dst-switch":"5e:3e:08:9e:01:e9:95:12","dst-port":3,"dst-port-state":0,"type":"internal"},

{"src-switch":"5e:3e:08:9e:01:e9:95:12","src-port":12,"src-port-state":0,"dst-switch":"e5:e6:08:9e:01:e9:95:12","dst-port":11,"dst-port-state":0,"type":"external"},

{"src-switch":"e5:e6:08:9e:01:e9:95:12","src-port":11,"src-port-state":0,"dst-switch":"5e:3e:08:9e:01:e9:95:12","dst-port":12,"dst-port-state":0,"type":"external”}]

A.2 Topology discovery experiment 2

```
pica8@c1:~$ curl -s http://127.0.0.1:8080/wm/device/
```

[{"entityClass":"DefaultEntityClass","mac":["82:5d:63:8e:df:4f"],"ipv4":["10.9.30.101"],"vlan":[],"attachmentPoint":[{"switchDPID":"5e:3e:08:9e:01:e9:95:12","errorStatus":null,"port":1}],"lastSeen":1426526633320},

{"entityClass":"DefaultEntityClass","mac":["f8:b1:56:2c:0f:64"],"ipv4":[],"vlan":[],"attachmentPoint":[{"switchDPID":"5e:3e:08:9e:01:e9:95:12","errorStatus":null,"port":1},{"switchDPID":"e5:e6:08:9e:01:e9:95:12","errorStatus":"duplicate-device","port":11}],"lastSeen":1426526634869},

{"entityClass":"DefaultEntityClass","mac":["f8:b1:56:2b:9b:40"],"ipv4":[],"vlan":[],"attachmentPoint":[{"switchDPID":"e5:e6:08:9e:01:e9:95:12","errorStatus":null,"port":11}],"lastSeen":1426526633799},

{"entityClass":"DefaultEntityClass","mac":["f8:b1:56:2c:0f:63"],"ipv4":[],"vlan":[],"attachmentPoint":[{"switchDPID":"5e:3e:08:9e:01:e9:95:12","errorStatus":null,"port":1}],"lastSeen":1426526633599},

{"entityClass":"DefaultEntityClass","mac":["f8:b1:56:2b:9b:41"],"ipv4":[],"vlan":[],"attachmentPoint":[{"switchDPID":"5e:3e:08:9e:01:e9:95:12","errorStatus":null,"port":1}],"lastSeen":1426526636615},

{"entityClass":"DefaultEntityClass","mac":["36:9d:f1:47:1b:74"],"ipv4":["10.9.20.101"],"vlan":[],"attachmentPoint":[{"switchDPID":"e5:e6:08:9e:01:e9:95:12","errorStatus":null,"port":11}],"lastSeen":1426526636946}]

```
pica8@c1:~$ curl -s http://127.0.0.1:8080/wm/topology/switchclusters/json
```

{"8f:0e:08:9e:01:e9:95:12":["5e:3e:08:9e:01:e9:95:12","e5:e6:08:9e:01:e9:95:12","8f:0e:08:9e:01:e9:95:12”]}

```
pica8@c1:~$ curl -s http://127.0.0.1:8080/wm/topology/links/json
```

[{"src-switch":"e5:e6:08:9e:01:e9:95:12","src-port":9,"src-port-state":0,"dst-switch":"5e:3e:08:9e:01:e9:95:12","dst-port":3,"dst-port-state":0,"type":"internal"},

{"src-switch":"e5:e6:08:9e:01:e9:95:12","src-port":11,"src-port-state":0,"dst-switch":"5e:3e:08:9e:01:e9:95:12","dst-port":1,"dst-port-state":0,"type":"external"},

{"src-switch":"8f:0e:08:9e:01:e9:95:12","src-port":5,"src-port-state":0,"dst-switch":"5e:3e:08:9e:01:e9:95:12","dst-port":2,"dst-port-state":0,"type":"internal"},

{"src-switch":"8f:0e:08:9e:01:e9:95:12","src-port":8,"src-port-state":0,"dst-switch":"e5:e6:08:9e:01:e9:95:12","dst-port":10,"dst-port-state":0,"type":"internal"},

{"src-switch":"5e:3e:08:9e:01:e9:95:12","src-port":1,"src-port-state":0,"dst-switch":"e5:e6:08:9e:01:e9:95:12","dst-port":11,"dst-port-state":0,"type":"external"},

{"src-switch":"5e:3e:08:9e:01:e9:95:12","src-port":3,"src-port-state":0,"dst-switch":"e5:e6:08:9e:01:e9:95:12","dst-port":9,"dst-port-state":0,"type":"internal"},

{"src-switch":"5e:3e:08:9e:01:e9:95:12","src-port":2,"src-port-state":0,"dst-switch":"8f:0e:08:9e:01:e9:95:12","dst-port":5,"dst-port-state":0,"type":"internal"},

{"src-switch":"e5:e6:08:9e:01:e9:95:12","src-port":10,"src-port-state":0,"dst-switch":"8f:0e:08:9e:01:e9:95:12","dst-port":8,"dst-port-state":0,"type":"internal"}]

References

[SDN] https://www.opennetworking.org/images/stories/downloads/sdn-resources/white-papers/wp-sdn-newnorm.pdf
Open Networking Foundation, "Software-Defined Networking: The New Norm for Networks".

[G.800] http://www.itu.int/rec/T-REC-G.800/
"Unified functional architecture of transport networks", ITU-T Recommendation G.800.

[NML] http://www.ogf.org/documents/GFD.206.pdf
J. van der Ham (editor), F. Dijkstra, R. Łapacz, J. Zurawski, "GFD.206: Network Markup Language Base Schema version 1", May 201

[FlowVisor] R. Sherwood, G. Gibb, K.-Kiong Yap, G. Appenzeller, M. Casado, N. McKeown, G. Parulkar† "FlowVisor: A Network Virtualization Layer", Technical report, 2009.

[OGF] https://www.ogf.org/ Open Grid Forum (OGF)

[NSI-FW] http://www.ogf.org/documents/GFD.213.pdf G. Roberts, T. Kudoh, I. Monga, J. Sobieski, C. Guok, J. MacAuley "GFD.213: Network Services Framework v2.0".

[CONNECT-paper] M. Živković, P. Grosso, C. de Laat, F. Dijkstra, D. Vandevenne "MOTE: Multi-Domain OpenFlow Topology Exchange", accepted for CONNECT publication.

[NSI-WG] https://redmine.ogf.org/projects/nsi-wg The Network Services Interface Working Group.

[TRIDENTCOM-paper] R. Koning, S. Konstantaras, M. Živković, F. Iqbal, C. de Laat, P. Grosso "Architecture for Exchanging Topology Information in Multi--domain Environments", submitted to 10th IEEE International Conference on Testbeds and Research Infrastructures for the Development of Networks & Communities (TRIDENTCOM).

[TOPO-PROPOSAL] https://redmine.ogf.org/dmsf_files/13400?download=
R. Koning, M., S. Konstantaras, P. Grosso, F. Iqbal, F. Kuipers: "UvA/TU Delft topology exchange and path finding in NSI environments", Topology Exchange proposal submitted to NSI-WG, February 2015.

[MOTE-MS1.1] F. Dijkstra: Open Call Deliverable MOTE MS1.1: NML Extensions.

[NW-MODELLING] J. van der Ham, M. Ghijssen, P. Grosso, C. de Laat: Trends in computer network modelling
towards the future internet. arXiv preprint arXiv:1402.3951 (2014).

[RFC-4641] http://www.ietf.org/rfc/rfc4641.txt
O. M. Kolkman, R. Gieben: RFC 4641: DNSSEC Operational Practices.

[FLASK] M. Grinberg: Flask Web Development - Developing Web Applications with Python. O'Reilly Media, Inc., 2014.

Glossary

API	Application Programming Interface
BDDP	Broadcast Domain Discovery Protocol
C-VLAN	Client Virtual Local Area Network
DDS	Document Discovery Service
DNS	Domain Name System
DPID	Datapath ID
GLIF	Global Lambda Integrated Facility
GOLE	GLIF Open Lightpath Exchange
GUI	Graphical User Interface
HTTP	Hypertext Transfer Protocol
JSON	JavaScript Object Notation
LLDP	Link-Layer Discovery Protocol
MOTE	Multi-domain OpenFlow Topology Exchange
NML	Network Markup Language
NSA	Network Service Agent
NSI	Network Service Interface
OF	OpenFlow
OGF	Open Grid Forum
PKI	Public Key Infrastructure
REST	Representational State Transfer
RAM	Random Access Memory
SDN	Software-Defined Network/Networking
SDP	Service Demarcation Point
STP	Service Termination Point
TC	Topology Consumer
TI	Topology Index
TP	Topology Provider
VLAN	Virtual Local Area Network
VM	Virtual Machine
XML	Extensible Markup Language