Ref. Ares(2016)5813888 - 07/10/2016

Document Properties

Document Number:	D 2.6
Document Title:	**NetIDE APIs and Integrated Platform v2**
Document Responsible:	Telcaria
Document Editor(s):	Elisa Rojas (TELCA)
Authors:	Elisa Rojas (TELCA), Pedro A. Aranda (TID), Roberto Doriguzzi Corin (CN), Arne Schwabe (UPB), Kévin Phemius (THALES)
Target Dissemination Level:	PU
Status of the Document:	Final
Version:	1.0

Production Properties:

Reviewers:	Christian Stritzke (IEM), Alec Leckey (Intel), Roberto Doriguzzi Corin (CREATE-NET)

Document History:

Version	Date	Issued by	
1.0	30-June-2016	Elisa Rojas	

Disclaimer:

This document has been produced in the context of the NetIDE Project. The research leading to these results has received funding from the European Community's Seventh Framework Programme (FP7/2010–2013) under grant agreement n° 619543.
All information in this document is provided "as is" and no guarantee or warranty is given that the information is fit for any particular purpose. The user thereof uses the information at its sole risk and liability.
For the avoidance of all doubts, the European Commission has no liability in respect of this document, which is merely representing the authors view.

Abstract:

This deliverable presents the second iteration and status of the NetIDE Application Programming Interfaces and Integrated Platform. The main objective is to analyse some of the last pieces in the architecture such as the Backend+ (deeping into the idea and performing the analysis), the flow abstractions towards a protocol-agnostic Core or the integration of the IDE with the Tools. Some of the topics aim to cover the 2nd year review report comments and some other are direct requirements from the use cases defined in Work Package 5 (particularly in Deliverable D5.4 [1]). The deliverable also compares NetIDE with the most current advances in SDN, specifically the evolution of the NBI and SBI. We examine what are implications in the NetIDE architecture and what can we learn from the project with a future perspective.

Keywords:

Network Engine, Shim, Backend, NetIDE protocol

Contents

List of Figures iv

List of Acronyms vi

Executive Summary 1

1 Introduction 2

2 Towards an integrated platform 3
2.1 Towards a protocol-agnostic core . . . 3
2.2 Evolution of the Shim . . . 4
2.2.1 Shim version 2.0 . . . 4
2.2.2 Analysis of the Shim v2.0 for specific Server Controller platforms . . . 5
2.3 Revisiting the LogPub architecture . . . 9
2.3.1 The LogPub module . . . 9
2.3.2 Message format . . . 10
2.3.3 Improving message monitoring . . . 10
2.3.4 Routing of synchronous messages . . . 11
2.3.5 Section on Core connection with ide . . . 11
2.4 Connecting the IDE with the Tools . . . 12
2.5 Other components . . . 12
2.5.1 BigFlowTable . . . 12

3 Matching advanced NB and SB interfaces within the NetIDE architecture 14
3.1 Intents and NetIDE . . . 14
3.2 OVSDB,NETCONF and NetIDE . . . 14
3.2.1 Open vSwitch Database Management Protocol (OVSDB) and NetIDE . . . 15
3.2.2 Network Configuration Protocol (NETCONF) and NetIDE . . . 16
3.3 Interaction with the Network Modelling (NEMO) language effort . . . 17
3.3.1 Intended use of NEMO . . . 17
3.3.2 NEMO within NetIDE . . . 17

4 Conclusion 19

A NetIDE Protocol Specification v1.4 20
A.1 The NetIDE protocol header . . . 20
A.2 Module announcement . . . 21
A.3 Heartbeat . . . 22
A.4 Handshake . . . 22
A.5 The FENCE mechanism . . . 23
A.6 The OpenFlow protocol . . . 24
A.6.1 Properly handling reply messages . . . 25
A.7 Other SBI protocols . . . 26

Bibliography **27**

List of Figures

2.1 Shim architecture . . . 4
2.2 Detailed view of the architecture with the “new” Shim . . . 5
2.3 Classification of the components in the OpenDaylight (ODL) Beryllium release; source [2] . . . 8
2.4 The location if the LogPub in the NetIDE Core . . . 9
2.5 The inner working of the LogPub in PUB mode . . . 10
2.6 The inner working of the LogPub in SUB mode . . . 10
2.7 Communication of the IDE with the Tools . . . 12

3.1 NetIDE Shim underneath the server controller's NBI . . . 15
3.2 Basic OVSDB architecture . . . 15
3.3 NETCONF Protocol Layers . . . 16

A.1 Fence mechanism workflow. Both `nxid` and `module_id` refer to the NetIDE header fields. . . . 24
A.2 Request/reply message handling. `xid` refers to the OpenFlow header field. . . . 25

List of Acronyms

AMQP	Advanced Message Queueing Protocols
API	Application Programming Interface
BGP-4	Border Gateway Protocol [3]
BOF	Birds-of-a-Feather
GUI	Graphical User Interface
IBN	Intent-Based Networking
IDE	Integrated Development Environment
IETF	Internet Engineering Task Force
IRF	Intermediate Representation Format
LLDP	Link Layer Discovery Protocol
MD-SAL	Model-driven Model-driven Service Abstraction Layer (SAL)
NBI	Northbound Interface
NEMO	Network Modelling
NETCONF	Network Configuration Protocol
NFV	Network Functions Virtualisation
ODL	OpenDaylight
OF	OpenFlow
ONF	Open Networking Foundation
ONOS	Open Network Operating System
OVS	Open vSwitch
OVSDB	Open vSwitch Database Management Protocol
RPC	Remote Procedure Call
SAL	Model-driven Service Abstraction Layer
SBI	Southbound Interface
SDN	Software Defined Networking
UC	Use Case
XML	eXtensible Markup Language [4]
YAML	YAML Aint a Markup Language [5]

Executive Summary

This document describes different analysis concerning the NetIDE architecture, next steps towards an integrated platform and future progress in the State of the Art. It is an evolution of the architectural concepts described in previous deliverables (more specifically Deliverable D2.5 [6]) until the middle of the third year and towards the third and final iteration in the implementation of the NetIDE architecture. Together with future Deliverable D2.7, we plan to give a final overview of the NetIDE architecture and a vision on how the future of SDN could affect it.

1 Introduction

This document presents the current state of the evolution of the NetIDE architecture as well as next steps towards an Integrated Platform. As Integrated Platform we understand the finished architecture of the NetIDE Engine, trying to address different details that appeared after the implementation and testing of Use Cases (UCs). Up to now, we designed independent components of the architecture. With this integrated view, our objective is to give a generalised idea of the NetIDE platform as a whole.
The deliverable also compares NetIDE with the most current advances in SDN, specifically the evolution of the NBI and SBI. The main purpose is to tie up loose ends, addressing at the same time many of the resulting comments after the 2nd year review report.

This deliverable is structured as follows:

- In Chapter 2 we present the architectural analysis towards an Integrated Platform. We address possibilities for a protocol-agnostic Core, the evolution of the Shim layer in relation with the "Backend+" (and the ability of supporting Network Application modules directly in the shim layer), the changes in the LogPub architecture and the IDE and Tools communication.
- In Chapter 3 we discuss how the evolution of NBI and SBI might affect the current NetIDE architecture. We specifically talk about Intents and SBI protocols different from OpenFlow (OF), such as NETCONF and OVSDB.
- in Chapter 4 we draw conclusions and outline the resulting work.

In addition we document the current status of the NetIDE Protocol in Annex A.

2 Towards an integrated platform

In this chapter we analyse the different components of the NetIDE architecture and what features should be ideally implemented to have a complete all-in-one framework.

Firstly, we start analysing the Core and its current representation of the information between the Shim and Backend layers, trying to achieve a more generic view, non dependent on the Southbound Interface. Secondly, we describe the new Shim architecture we have defined with the purpose to compose server controller's SDN modules with the rest of the NetIDE Network Application. Specifically we analyse this module implementation for the three SDN frameworks we have been using as Server Controllers so far: Ryu [7], the Open Network Operating System [8] and OpenDaylight [9], respectively. Thirdly, we revisit the LogPub architecture, as Tool integration activities have become more active in the architecture and have arisen new requirements. Finally, and also related with the Tools, we sketch an initial idea on how to connect the Tools with the IDE and the reasons to do so.

2.1 Towards a protocol-agnostic core

Previously in Deliverable D2.4 [10], we analysed a possible NetIDE Application Programming Interface (API) v2.0 in which NETCONF [11] and OF work together. In order to do so, we compared their primitives to find common denominators to represent the information exchanged between control and data planes. We also considered the ONOS Java API [12] as a means to provide NetIDE such an abstracted, protocol-independent API able to cover both OF and NETCONF.
In this document, and considering the comments from the 2nd year review, we examine the *Flow Objectives* (from ATRIUM [13]) abstractions, how they would fit in a theoretical protocol-agnostic Core and their implications in a possible NetIDE API v2.0.

ATRIUM is an Open Networking Foundation (ONF) project that aims at providing an integrated platform to reduce the entry barrier to new users of SDN. In their first release (in 2015), the project provided an Open Network Operating System (ONOS) based Border Gateway Protocol (BGP-4) router application that would allow end users to implement the routing control plane in the SDN controller and use a range of switching devices including OpenFlow switches to implement the data plane. In recent releases, they have included ODL to the supported SDN frameworks.

Problem statement: As the data plane devices supported by the project have different pipeline implementations, applications written for a specific data plane device will not necessarily work on other data plane devices. To be able to write a routing application that can drive data plane devices that implement different paradigms, there needs to be an abstracted, device independent API.

Solution used in Atrium: The device-independent API defined by Atrium are the *Flow Objectives*. The project claims that this is a way of expressing the intent behind the flow installation. The flow installation objective is translated to flow rules using the Device Drivers. As of writing, the Atrium project has defined three *Flow Objectives*: (i) Next (to define the next hop for forwarding or OF group and similar), (ii) Forwarding, and (iii) Filter (to implement *permit* or *deny* actions on packet fields like, e.g. MAC or IP address or VLAN tag id).

Application in NetIDE : Flow objectives are implemented as Remote Procedure Calls (RPCs) in ODL. An extension of the NetIDE API to support flow objectives is feasible and could be beneficial as far as it would abstract the implementation details of the underlying switching hardware. However, flow objectives are currently designed for the pro-active mode only. This is the biggest gap with our current line of thought, where we have stressed on the reactive operation and grouping of network events and the resulting commands into transactions.

2.2 Evolution of the Shim

In Deliverable D2.5 [6] we introduced the so-called "Backend+" as a means to compose client and server controller application modules into a single Network Application. The Backend+ was defined as an additional SBI for the server controller connected to the Core via the NetIDE Intermediate Protocol. Differently from a common Backend implemented for client controllers, the Backend+ hides the other SBIs to the application modules running on the server controller, so that they cannot interact with the network by-passing the Core, hence the composition and conflict resolution mechanisms.
However, after analysing different controller frameworks such as Open Network Operating System, Ryu and OpenDaylight and aiming to simplify the overall design of the architecture, we moved the aforementioned function of the Backend+ to the Shim.
The architecture of the extended Shim is described in the next Section.

2.2.1 Shim version 2.0

Figure 2.1 shows the internal architecture of the Shim integrated in a server controller platform. With respect to the architecture presented in Deliverable D2.5, it includes hooks placed inside SBIs (small grey boxes in the figure) to route all messages from the network to the Shim itself, overriding the server controller's processing logic.

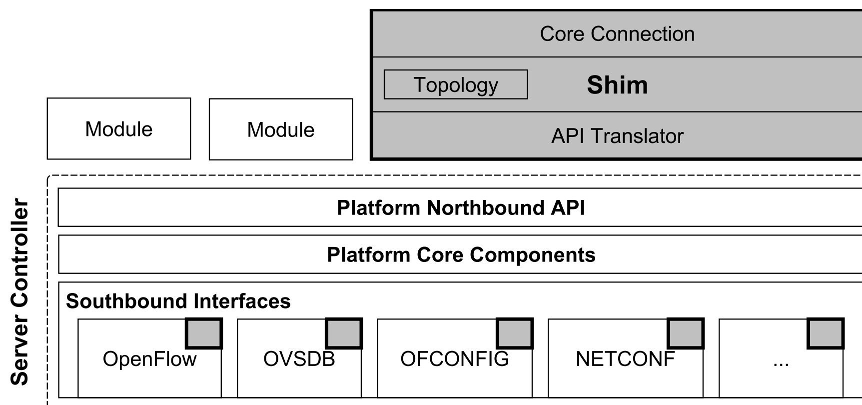

Figure 2.1: Shim architecture

The *Core Connection* component handles the communication with the Core through the NetIDE Intermediate Protocol. The *API Translator* converts between NetIDE protocol and platform-specific NBI messages. The *Topology* module provides Network Engine's upper layers with access to network topology details and asynchronous updates in case of changes in the network. Topology is discovered either by interacting with network through the SBI of the server controller (e.g. by means of the Link Layer Discovery Protocol (LLDP) or by means of specific control messages such as OpenFlow's `OFPMP_PORT_DESCRIPTION` requests) or by retrieving the topology information collected by

specific Core components available in advanced controller platforms (e.g. ONOS and OpenDaylight). Topology details include `datapath_ids`, number of supported flow tables, (administrative and operational) port information and other properties of the network elements.
The functionalities of the *Handshake handler* component included in the previous versions of the Shim have been moved to the Core, as explained in Appendix A.

Figure 2.2 shows a detailed view of the Network Engine's architecture with the new version of the Shim. With respect to the architecture proposed in D2.5, it includes the version 2.0 of the Shim, and the Backend+ has been replaced by a common Backend acting as an additional SBI for the server controller.

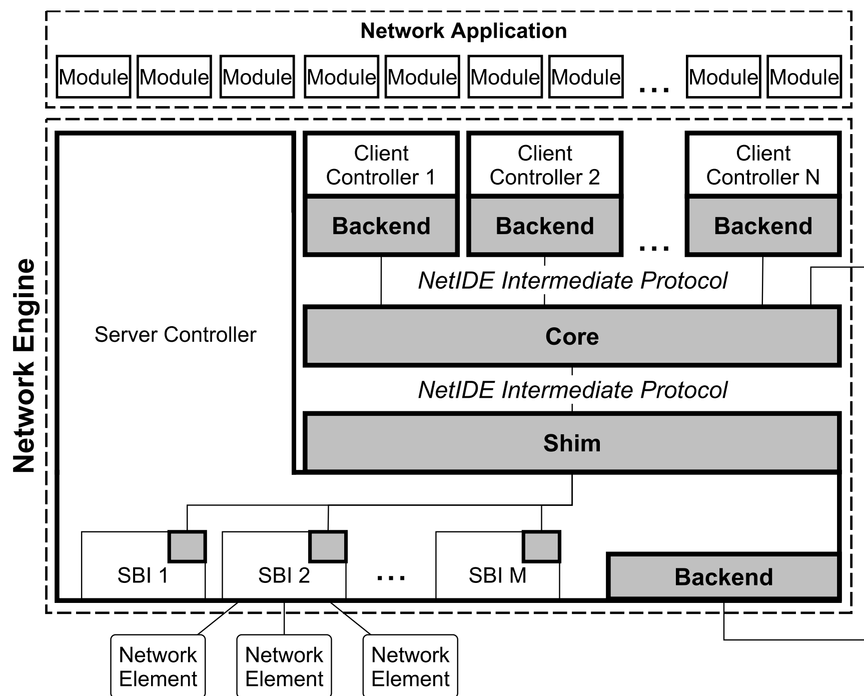

Figure 2.2: Detailed view of the architecture with the "new" Shim

When an event is generated by the network and arrives to the server controller, this event is intercepted by the Shim which relays it to the Core, overriding the original execution logic of the server controller. The Core determines the final destination for the event based on the composition specification and sends it to one or more modules via Backends (more details on composition and conflict resolution mechanisms can be found in Section 2.2.3 of Deliverable D2.4 [10]), no matter whether they are running on top of the server or client controllers.
Vice-versa, action messages generated by the modules are intercepted by the Backends (in case of the server controller, the Backend is the only active SBI available for those modules, as the others are intercepted by the Shim) which relay them to Core. The Core checks the messages for possible conflicts and composes them with other messages received from other modules (if any). Finally, the resulting control message is sent to the network via Shim and server controller's SBIs.

2.2.2 Analysis of the Shim v2.0 for specific Server Controller platforms

In the following sections we analyse a possible implementation of the new Shim for three SDN frameworks: Ryu, ONOS and ODL. We analyse ONOS and ODL because they are the two selected server controllers for the NetIDE architecture. For such SDN platforms, we already implemented the first version of the Shim. As explained above in this Section, to allow their application modules

to be composed with modules running on client controllers into a single Network Application (like depicted in Figure 2.2), we need to implement a Backend and enhance the Shim for both platforms. We also analyse Ryu because we implemented the two first pieces of the NetIDE Engine (Backend and Shim v1.0) using this framework and it is the only SDN framework for which we have them both. Therefore Ryu can help us analysing the implications of linking the Shim v2.0 with the server controller's SBIs. Notice we will not analyse other SDN frameworks as Floodlight because we are not using it as Server Controller.

2.2.2.1 Shim v2.0 implementation for Ryu

By default Ryu distributes a network event (new flow, port status, etc.) to all the application modules registered to that type of event. When Ryu is used as a server controller, there are at least two active SBIs: the Backend and the one connected to the network elements (e.g. OpenFlow, NETCONF). The modules composing the Network Application running on Ryu should only get the network events coming from Core and Backend, not from the other SBIs.

Implications for the Ryu architecture The code of the `controller.py` file needs to be modified in order to ensure that all the updates coming from the network are only sent to the Shim. Specifically, we need to modify the `recv_loop` method so that only the handlers implemented in the *Shim* will get the events. Listing 2.1 shows the relevant part of the controller.py code. In particular, the loop at lines 5 and 6 should be replaced by a single call `handler(ev)`, where `ev` is a network event and `handler` points to the handler implemented in the Shim.

```
1 dispatchers = lambda x: x.callers[ev.__class__].dispatchers
2 handlers = [handler for handler in
3                          self.ofp_brick.get_handlers(ev) if
4                          self.state in dispatchers(handler)]
5 for handler in handlers:
6         handler(ev)
```

Listing 2.1: Excerpt of controller.py code

From our previous experience in implementing the Backend, we know that this simple trick also prevents application modules on sending control messages to the network via `controller.py` (i.e. by-passing the Core). As they only receive network updates via Backend, they will only use the interfaces exposed by such a SBI.

2.2.2.2 Leveraging the Shim v2.0 to compose ONOS application modules

Enabling the composition of ONOS modules with other modules running on top of client controllers implies the implementation of a Backend and the enhancement of the current Shim.

Implications for the ONOS architecture Like for Ryu, the implementation of the Shim v2.0 consists of intercepting the process that distributes the events to all the registered listeners. For instance, in case of the OpenFlow, this process is implemented by the `OpenFlowControllerImpl` class, which implements the OpenFlow provider. The method that handles the network events is called `processPacket` and, for each event, it goes through all the registered listeners and passes them the event. Listing 2.2 shows how `PACKET_IN` messages are handled. Specifically, the loop at lines 4-6 must be replaced with a single `p.handlePacket(pktCtx)`, where `p` points to the packet listener implemented by the Shim. Therefore, only the Shim will handle `PACKET_IN` messages.
The same trick applies to other events as well (e.g. `FEATURES_REPLY`, `PORT_STATUS`, etc.).

```
1 case PACKET_IN:
2         OpenFlowPacketContext pktCtx = DefaultOpenFlowPacketContext.
3         packetContextFromPacketIn(this.getSwitch(dpid),(OFPacketIn) msg);
4         for (PacketListener p : ofPacketListener.values()) {
5                 p.handlePacket(pktCtx);
6         }
7         break;
```

Listing 2.2: Excerpt of OpenFlowControllerImpl.java code

The Backend can be implemented as a *Provider*. In ONOS, Providers are components that interface the ONOS platform with the network. In our case, the Backend connects ONOS with the NetIDE Core via the NetIDE protocol. Since the other providers are intercepted by the Shim, the Backend is the only ONOS SBI that serves the network events to the application modules (via ONOS core layer) based on the directives it receives from the NetIDE Core.

2.2.2.3 Leveraging the Shim v2.0 to compose ODL application modules

Like in the case of ONOS, enabling the composition of modules running directly on the ODL controller framework with modules running on top of client controllers implies the implementation of a Backend for ODL and the enhancement of the current Shim.

Implications for the ODL architecture The implementation of the Backend and of the new functionality of the Shim has a significant impact on the overall OpenDaylight architecture:

- The message exchange with the SBI interfaces that handle the network devices needs to be intercepted and these messages need to be diverted to the Shim
- The principle behind the Backend is that we have access to message handling loop of the controller to implement the *fence* mechanism.

This implies significant modifications to the Model-driven SAL (MD-SAL), which one of the *offset 0* or *kernel* components of the ODL distribution (cf. Figure 2.3 [2]).

Feasibility: The extreme modularity of the ODL controller should allow this approach. In fact, some demos in the SDN and NFV Europe'2015 conference in Düsseldorf showed smaller projects that had tackled specific challenges this way. In addition, the Backend is conceptually yet another (extended) SBI and ODL is one of the SDN platforms that supports the widest range of Southbound Interfaces.

Sustainability beyond the lifetime of the project: The modifications that need to be done on the MD-SAL in order to support the new Shim and the Backend are substantial. This would imply a significant effort on the side of the project in order to convince the community of their usefulness. Additionally, since the discussion period for the Boron release is closed as of writing this deliverable and the planning for releases beyond will happen once NetIDE has finished.

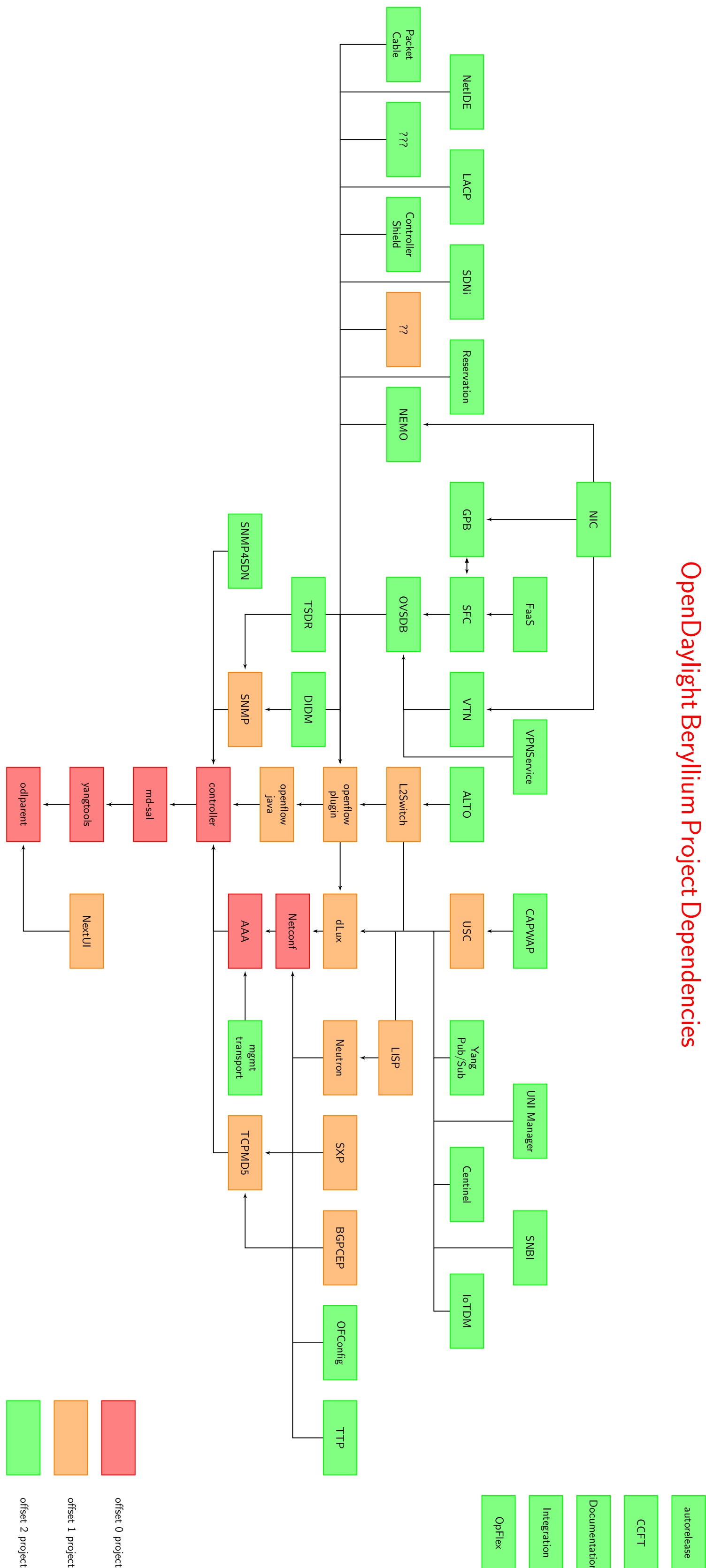

Figure 2.3: Classification of the components in the ODL Beryllium release; source [2]

2.3 Revisiting the LogPub architecture

2.3.1 The LogPub module

The LogPub module is one of the modules of the NetIDE Core. It acts as an interface between the tools and the Core in two ways:

1. it transmits messages originated by the Shim Layer/Backend(s) to the tools
2. it receives messages from the tools and sends them to the Shim Layer/Backend(s)

It used to rely on the core.api module (see Fig. 2.4), to grab all messages sent by either the Shim layer or the Backend(s) and replies of messages sent by the tools. The revisited LogPub will use the core.routing module to simplify the architecture (see 2.3.4). The LogPub works using a PUB/SUB paradigm and thus exposes two queues to the tools.

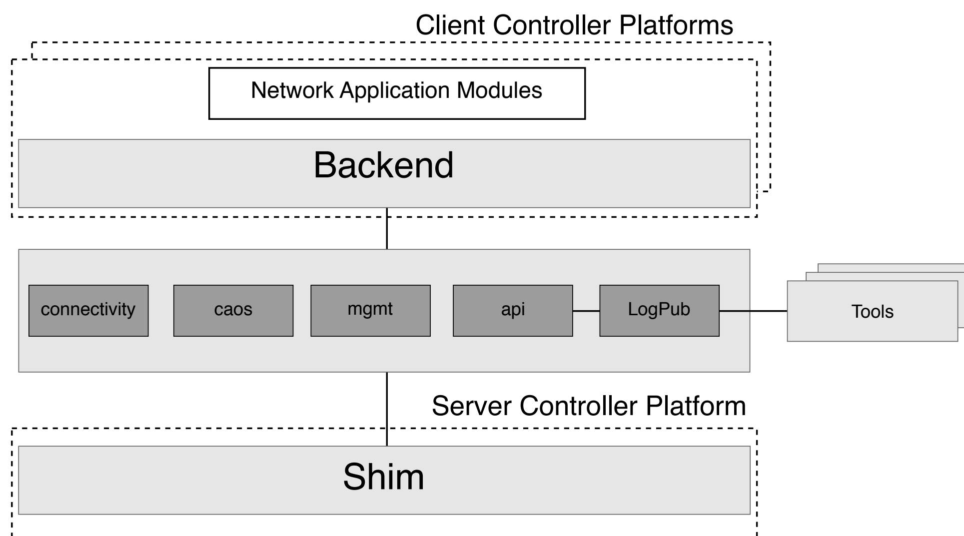

Figure 2.4: The location if the LogPub in the NetIDE Core

The LogPub module receives the messages from the Shim Layer and Backend(s) using the Java interface *core_connector* which is used by the Core.routing module to send messages to the LogPub. The main receiving loop of the LogPub being in a Thread (to allow concurrent access between the different queues), there is an initial process to deposit the message in the (ØMQ [1]) `in_process` queue (as depicted in Fig. 2.5). This prevents the Thread from hanging as ØMQ handles all the communications from/to the process. The messages are then classified by the `Identifier`; its role is to assign them a topic; when the messages are dispatched in the **PUB** queue, the tools will only receive the messages they are subscribed to (see Fig. 2.5). For example the *Logger* is supposed to get all messages and will subscribe to all topics (represented by the `*` mnemonic) while the IDE will subscribe to the *topology update* topic.

Moving from the previous implementation, the LogPub does not keep in internal representation of packets sent from the **SUB** queue in a hash table. This functionality is now handled in another Core module. Thus in SUB mode, the LogPub only had to call the Routing module to send the message either to the Shim or a Backend (see Fig. 2.6).

[1] We already introduced the LogPub architecture in Deliverable D2.5 [6], while in Deliverable D2.3 [14] we described the two Advanced Message Queueing Protocols (AMQP) options we evaluated to implement it: RabbitMQ and ØMQ

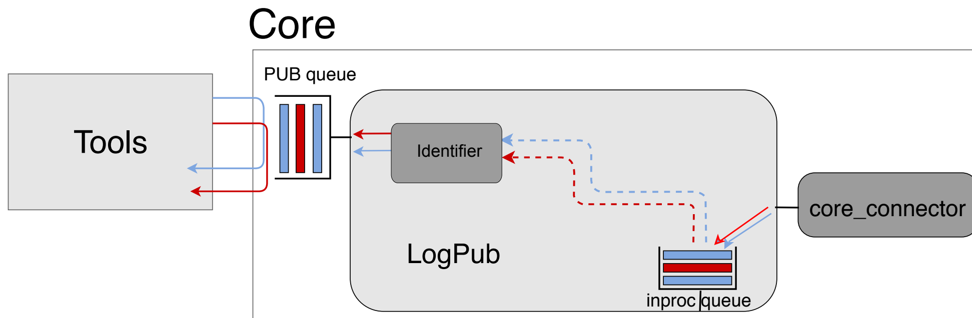

Figure 2.5: The inner working of the LogPub in PUB mode

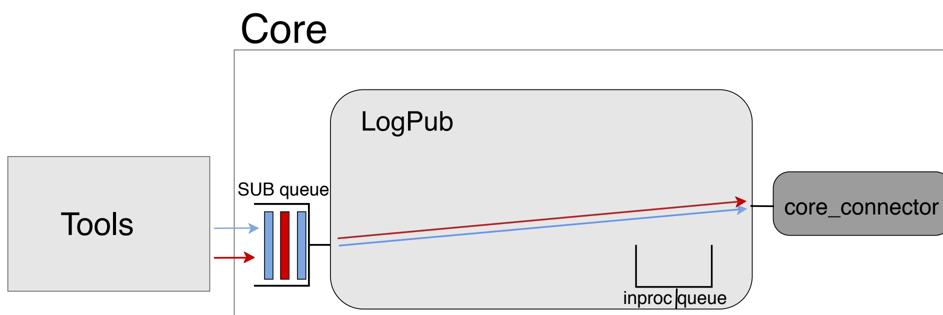

Figure 2.6: The inner working of the LogPub in SUB mode

2.3.2 Message format

Messages exchanged between the LogPub and the tools must use the format inside a ∅MQ multi-part message shown in Listing 2.3. The *location* field contains a string identifying the destination or source of the message (Shim or Backend) and the *message* field contains a *netide_message*:

- In the Tool ⟶ LogPub direction, the *location* field indicates whether the message is intended for the Shim Layer or a Backend (eg. {shim,message}).
- In the LogPub ⟶ Tool direction, the *location* field indicates whether the message originates from the Shim Layer or a Backend. As mentioned earlier, the message will also have a topic (eg. {topic,shim,message}).

Listing 2.3: ∅MQ multi-part message header

```
struct logpub_message{
        struct zeromq_message location;
        struct zeromq_message message;
};
```

2.3.3 Improving message monitoring

Currently, the LogPub can monitor everything that is coming *from* the Shim and the Backends. The messages the Core sends *to* the Shim and Backend are currently not monitored. In the past when the Core did have limited functionality and packet mangling this was not a problem since most messages passed out of the Core were identical to the monitored messages passed into the Core. With the functionality of the Core and utilities becoming more advanced this gap widens.

To alleviate this problem, we are implementing listener interfaces for the outgoing interfaces. This will enabled the LogPub module to monitor all traffic from the Core.

2.3.4 Routing of synchronous messages

Currently, when a Backend or the LogPub module sends a synchronous message to a switch (like an OpenFlow STATISTICS_REQUEST) the Core will pass the answer (a STATISTICS_REPLY) to all Backends and the LogPub module and not only to the one that requested the statistics. This "broadcast" of the answer is not only extremely inefficient (the reply is forwarded to **every** connected Backend and the LogPub) there could be unintended effects.
We intend to rectify this problem by implementing a special `core.routing` module that remembers the original Backend ID (from the routing module point of view, the LogPub module is handled like a Backend) and will send the response only to that module.

Backend message : If a Backend sends a statistics request, the LogPub will get the message too (for example the Logger is interested in such messages). The reply need to be forwarded to the Backend which requested the statistics as well as the LogPub (again, for the Logger) but no other Backend.

Tool message : If a Tool sends a statistics request, the LogPub will forward it to the core.routing module and the reply will only be forwarded to the original Tool (via the core.routing and the LogPub). This method is much more efficient and can improve performance by limiting the number of message duplication.

2.3.5 Section on Core connection with ide

The NetIDE Graphical User Interface (GUI) (which belongs to the NetIDE IDE) has the task of displaying the data to the user, which is a direct requirement from Work Package 5 and specifically from UC2. To implement this task, various data needs to be forwarded to the GUI or actively collected by the GUI. Since we already designed and use the LogPub interface for the tools to allow them to query/collect information, we will leverage this interface as much as possible for the GUI.

One type of information the GUI needs to fulfill its role is a representation of the topology of the network. The IDE uses this information to match the deployment with the actual topology running. Topology is already collected by most of SDN controllers[2], by the use of Link Layer Discovery Protocol (LLDP) [15] packets and, specifically, the topology is readily available in all server controllers we support. We therefore plan to implement special messages that allow requesting the topology from the Shim from Tools/the GUI, which now sit together in the architecture. The Shim will send the topology information in a standardised NetIDE message to the Core, which forward the message to the requesting entity.

The conclusion is to leverage the SDN frameworks services (Topology, Stats, etc.), to avoid repeating a mechanism in the Engine. If we did not do so, we could end up having services repeated even more than three times. For example, LLDP could be executed in the Server Controller (e.g. ONOS), but also in the Client Controller (e.g. Ryu) and in the Core (requested by the GUI), or even more times if we have more than one Client Controller. Leveraging the Server Controller services we avoid repetition, although not all. It remains as future work how NetIDE should coordinate SDN services to avoid duplicates.

[2] We ignore here the toy examples like a simple layer2 switch without loop detection/spanning tree implementation

2.4 Connecting the IDE with the Tools

As explained in Section 2.3, the Tools and the NetIDE IDE are located in the same part of the architecture. They are both fed by the PUB queue of the LogPub module and can send messages to the Engine with the SUB queue. Currently, we have shown different communication possibilities: between Backend and Shim, between Shim and some of the Tools, etc. Thus there is one use case left and it is related to the communication of the IDE with the Tools, independently of the Engine (understanding Engine as Backends, Core and Shim).

So far, the IDE acted as a passive reader of the Tools outputs. For example, the Debugger generated a `*.txt` and a `*.pcap` file after execution in a `results` folder, which later the user of the IDE could read and analyse with Wireshark [16]. This approach has some problems in the case we want the Tool to be proactive during runtime. This is the case of the Verificator (to be further explained in future WP4 deliverables), which detects malfunctions of the network during runtime and needs to indicate them to the IDE right when they occur.

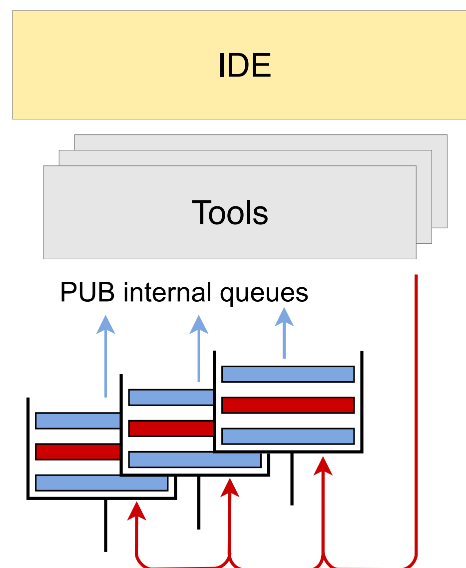

Figure 2.7: Communication of the IDE with the Tools

Therefore, for tools such as the Verificator, we designed a new pool of queues to exchange messages between the IDE and the Tools (Figure 2.7). Initially these queues are defined using ports from 5560 and a single one could be enough to fit all purposes. Currently there is only one queue being used for the Verificator to publish information to the IDE. But in the future, there could be more than one or even a single one defining different topics (similarly to the LogPub PUB queue).

Ideally, these queues should be merged with the PUB queue of the LogPub in the future.

2.5 Other components

2.5.1 BigFlowTable

The work on the BigFlowTable is progressing and we added the first fragment of code that currently record and save the installed flows but without merging them to "big flows". The collected flows can already be displayed with a Karaf command.

The BigFlowTable is a crucial element for composition of proactive rules. In Deliverable D2.5 [6] we analysed how to compose reactive rules, which we installed at the same time based on a previously received event. However, proactive rules come at any time and we need to synchronize them with the current status of the network. This status is represented by the BigFlowTable.

3 Matching advanced NB and SB interfaces within the NetIDE architecture

In this chapter we analyse how the evolution of north-bound and south-bound interfaces might affect the NetIDE architecture. We first take a look at the NBIs and then to the SBIs. Finally, we analyse the status and interaction of the NetIDE project with the definition activities of NEMO language and the IBNEMO project in ODL.

3.1 Intents and NetIDE

Back in Deliverable D2.3 [14], we introduced the concept of Intent-Based Networking (IBN) [17] and how the ONF[1] was coordinating many of the activities related to it in communities such as ONOS or ODL. Some specific examples are the ONOS *Intent* Framework [19] or the Boulder project [20, 21], which is designed for Intent-based Northbounds of SDN controllers, focusing on semantics and information models to enable applications to tell networks what to do and providing *Intent* portability across different controller environments.

In the case of NetIDE, Intent-Based Networking is a good example of how to translate network behaviour into a common language, which can be particularly useful when coordinating different types of application modules. However, it also represents the evolution of the SDN controllers' NBIs to be really independent of their respective SBIs and this might break our NetIDE architecture.

To understand what we mean by *breaking* the NetIDE architecture, we first need to understand that our Core sits between the Server Controller's NBI and the different Client Controllers' SBIs (see Fig. 2.2). Therefore, basically NetIDE *translates* SBIs messages back into NBI messages and viceversa. Initially, NBI were still dependent on the SBI (e.g. the `SimpleSwitch` application in Ryu is written once per each OF version supported). Later on, a common API was created in some SDN controllers to use the same application for different versions of the SBI protocol (e.g. the `l2switch` project in ODL is valid both for OF 1.0 and OF 1.3). However, they were still very related to the SBI protocol (not on the version, but on the primitives) and that is when *Intents* come into place, and who knows what will come next (because current *Intents* are still a bit limited, e.g. in ONOS, *Intents* are still very related to the concept of *flows* in the network).

Therefore, little by little, small abstraction layers are being added to the NBI to be as independent as possible of the SBIs. If we keep the Shim layer at the top of the Server Controller, as depicted in 2.1, this translation between NBI and SBIs starts being complicated or even impossible. For this reason, we would need to integrate the Shim layer *inside* the NBI controller, as depicted in 3.1, or even below, as part of the core services of the Server Controller. The idea is to place the Shim underneath the NBI, so that we can ignore the evolution of the NBI since we are by-passing it when communicating with the Core.

3.2 OVSDB,NETCONF and NetIDE

Previously, in Deliverable D2.4 [10], we analysed how to match different SBI protocols together with OF. Specifically, we focused on the different versions of OF and the NETCONF [11] protocol,

[1] One of the first documents using the term *Intent* is "ONF: Northbound Interfaces" [18]

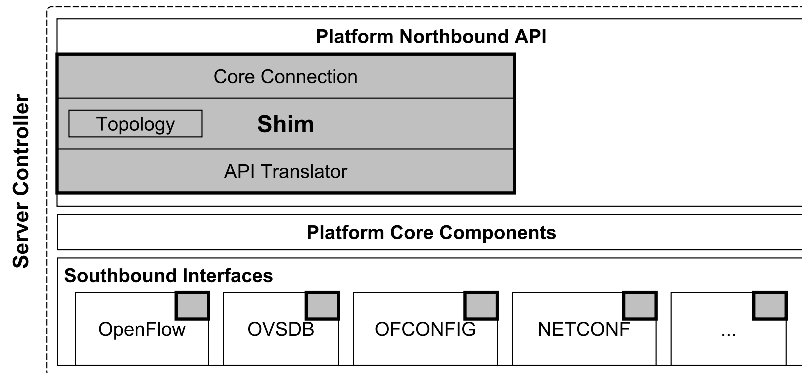

Figure 3.1: NetIDE Shim underneath the server controller's NBI

looking for common primitives that could match them. However, we ignored how these protocols could affect the inner workings of NetIDE .

Initially, NetIDE will not have problems with different versions of OF, as OF always works when the network is running (run-time). But NETCONF or OVSDB [22] are used at deployment time as well, and this could affect the deployment of NetIDE . In the following paragraphs we show some examples.

3.2.1 OVSDB and NetIDE

The Open vSwitch Database Management Protocol (OVSDB) [22] was originally conceived as a management protocol for the Open vSwitch (OVS) [23], an OF switch implementation for Linux on general purpose processors. It has been adopted by OF switch manufacturers. OVSDB implements the network element configuration procedures that were not foreseen in the OF specification. The basic architecture assumed is shown in Figure 3.2. Although OVSDB and OF address different concerns, i.e. OVSDB takes care of the configuration of the network element *before* it can actually talk to the OF, there are scenarios where OVSDB will interfere seriously with OF.

More specifically, the *management and control cluster* module in Figure 3.2 consists of some number of managers and controllers. Managers use the OVSDB management protocol to manage OVS instances, while controllers use OF to install flow state in OF switches. For example, the ONOS SDN framework can implement both a manager and a controller for an OVS at the same time; the manager is implemented as a driver, which can execute commands such as `getControllers`, `setControllers` or `addBridge`.

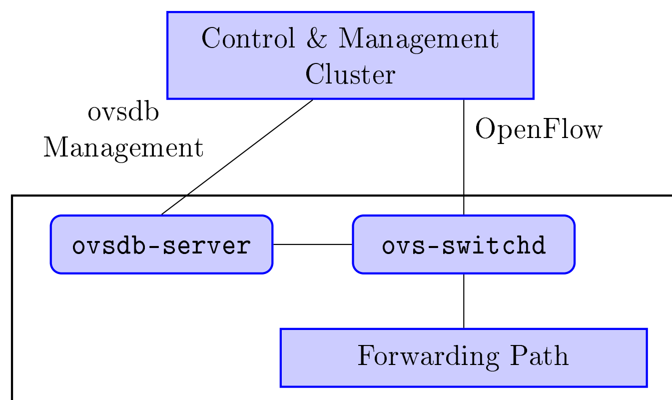

Figure 3.2: Basic OVSDB architecture

One example is setting up a new SDN controller for some network devices: OVSDB could decide to set a different SDN controller for some part of the network so that they follow a different SDN application or module. In NetIDE , this is translated into a change of the composition file. First of all, both modules need to be active, i.e. (1) the one running in which OVSDB makes the decision and (2) the new one assigned. Secondly, the network administrator should write a composition file indicating that all network device events should go to the first module. Finally, when OVSDB sends the instruction, it should not go to the network, but instead change the composition file in NetIDE so that the network devices specified by OVSDB send their events to the second module now (and not to the first one).

A second example would be the opposite one, the case in which OVSDB requests the controllers assigned to specific network devices. If the OVSDB module was expecting different modules controlling different parts of the network, it might expect as well to get different SDN controllers assigned. However, NetIDE unifies this and would give a single instance of a controller: NetIDE . We should then provide this information to the OVSDB module that requested it by taking the composition file deployed in the system.

These two problems get even more complex if the OVSDB module contains hardcoded SDN controllers. For example, imagine that it wants to set the controller of one network device to port 6646, and the network administrator was not aware of this and deployed the module in NetIDE , how can NetIDE translate that port into one of the modules running in NetIDE? Or moreover, how can NetIDE distinguish if the module *really* wants to set *that* controller, which contains a module outside NetIDE (in the case that NetIDE is not the only platform controlling the network)?

Finally, another potential problem would be instantiating new network devices via OVSDB and then setting a controller to them. Now we have mixed problems, we do not know if there is another controller outside NetIDE , or to what module we should send the traffic in case it is inside NetIDE and we need to obtain the datapath ID assigned to the new device to be able to add it to a new composition file.

3.2.2 NETCONF and NetIDE

In the case of NETCONF this can be even more difficult as we need to parse the XML content exchanged between the NETCONF and the network devices. NETCONF provides mechanisms to install, manipulate, and delete the configuration of network devices. Its operations are realized on top of a RPC layer as depicted in 3.3. At the same time, contents in NETCONF could be expressed in YANG [24] modeling language or other (although YANG was specifically designed for this purpose).

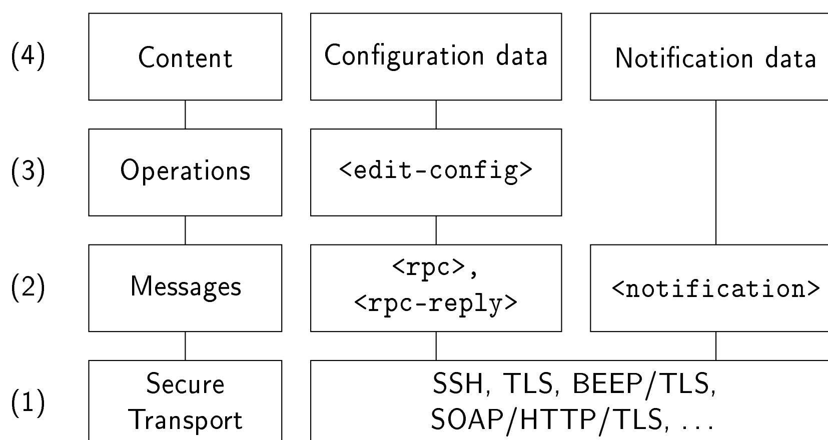

Figure 3.3: NETCONF Protocol Layers

As for how NETCONF could affect NetIDE , for example, at operation level, NETCONF provides different configuration tags: $<get/edit/copy/delete-config>$. Therefore, NetIDE should be able to analyse the different parts of the message and get what one could affect internal components (such as the composition of the Core) and which not. This configuration tags mean the same problems than OVSDB with the set/get controller actions, but extended to a variety of configuration possibilities.

Also, NETCONF modules can dynamically subscribe to specific notifications in the network. This imply modifying the composition file on runtime according to the XML content of the subscriber, because it might choose to what network devices listen or not, and to what specific events.

3.3 Interaction with the Network Modelling (NEMO) language effort

The Network Modelling (NEMO) language [25] is a human readable specification language to express networking scenarios. It has been a low-profile, long-term activity which started in the early days of NetIDE in the context of the Internet Engineering Task Force (IETF). This collaboration has mainly involved TID and Huawei and the project has been informed of all steps.

3.3.1 Intended use of NEMO

NEMO is used in the IBNEMO project of OpenDaylight to drive a transaction-based NBI which allows applications to use intent-based policy to create network scenarios that consist of virtual networks comprised of nodes with policy-controlled flows. This intent based policy is prescriptive rather than descriptive, thereby abstracting the implementation details of the network. The IBNEMO project implements a Northbound Interface for the OpenDaylight controller. It is minimalistic in that it operates with a very reduced set of commands to define models (e.g. `NodeModel` for network elements, network functions, etc., `LinkModel` for connections between them, . . .) for basic network objecs and their implementations.

3.3.2 NEMO within NetIDE

NEMO is an overarching concept that we have tested for different purposes in the scope and lifetime of the project. The rationale behind this activity has been to use NEMO as a high-level language that would allow us to express the composition of the pre-existing applications. To this avail, we would encapsulate them as `NodeModels` in the NEMO description. The resulting `NodeModels` would then be new building blocks that we would incorporate into new NetIDE Network Applications.

NEMO in the context of the Intermediate Representation Format (IRF) In the first phase of the project, we evaluated whether NEMO could encapsulate the low-level details of different SDN programming frameworks we contemplated in the project. We used the same methodology we used for other SDN languages, with Use Case 1 as a proof-of-concept to evaluate the features provided by NEMO[2].
Our main aim was to exploit the fact that NEMO is a programming language that is not only human readable (eXtensible Markup Language (XML) [4], YAML Aint a Markup Language (YAML) [5] are other examples of human readable languages) but also easily understandable.

NEMO in the context of the Network Engine After we abandoned the IRF-based approach, we continued considering NEMO as a means to express the relationships between the modules in a Network Application. Our first take was that it could steer the deployment of the different

[2] Compare with the implementation of UC1 using YANG models we provided in Deliverable D2.1 [26]

modules in the NetIDE Network Engine and translate into the module composition specification. In this scenario, we have explored the possibility of using the IBNEMO module which is provided in the Beryllium release of OpenDaylight as an alternative driver to deploy modules when we deploy OpenDaylight as the Server Controller of our Network Engine.
The main advantage of this approach is that it would allow us to reuse *tested* code that is already shipped with OpenDaylight. The main disadvantage of this approach is that it binds the NetIDE Network Engine too much to OpenDaylight and effectively creates branches that would diverge significantly (e.g. in the case we use ONOS as the Server Controller).

Generic activities to support the NEMO effort We contributed to an effort to raise awareness of NEMO in the IETF and helped organise and took part in two Birds-of-a-Feather (BOF) sessions. As a result, we published a report in the IETF Journal [27]. The language has evolved during the lifetime of NetIDE and was refined and presented in the hackathon organised for the IETF94 [28].

4 Conclusion

After the second iteration of the architecture in Deliverable D2.5 [6], we face the next steps towards an Integrated Platform and, in general, towards the future of the NetIDE architecture. For this deliverable we have focused on the direct requirements requested by Work Package 5, for example in the case of the communication of the IDE and the Core or the Tools. We aim to cover up the last steps for a third and final iteration of the NetIDE architecture, specifically focused on the finalization and evaluation of the Work Package 5 UCs.

Finally, regarding the future of the NetIDE project, we have sketched different aspects that should be taken into consideration when integrating other NBI and SBI.

A NetIDE Protocol Specification v1.4

The Intermediate protocol accomplishes the following functions: (i) to carry management messages between the Network Engine's layers (Core, Shim and Backend); e.g., to exchange information on the supported SBI protocols, to provide unique identifiers for application modules, (ii) to carry event and action messages between Shim, Core, and Backend, properly demultiplexing such messages to the right module based on identifiers, (iii) to encapsulate messages specific to a particular SBI protocol version (e.g., OpenFlow 1.X, NETCONF, etc.) with proper information to recognize these messages as such.
Messages of the NetIDE protocol contain two basic elements: the NetIDE header and the data (or payload). The NetIDE header, described below in Section A.1, is placed before the payload and serves as the communication and control link between the different components of the Network Engine. The payload carries management messages or the SBI messages issued by either client controllers or network elements.

A.1 The NetIDE protocol header

The NetIDE header can be represented in a `C-style` coding format as follows:

```
struct netide_header{
        uint8_t netide_ver;
        uint8_t type;
        uint16_t length;
        uint32_t nxid
        uint32_t module_id
        uint64_t datapath_id
};
```

The `netide_ver` is the version of the NetIDE protocol (the current version v1.4 which is identified with value 0x05), `length` is the total length of the payload in bytes and `type` contains a code that indicates the type of the message according with the following values[1]:

```
enum type{
        NETIDE_HELLO             = 0x01,
        NETIDE_ERROR             = 0x02,
        NETIDE_MGMT              = 0x03,
        NETIDE_MODULE_ANN        = 0x04,
        NETIDE_MODULE_ACK        = 0x05,
        NETIDE_HEARTBEAT         = 0x06,
        NETIDE_TOPOLOGY          = 0x07,
        NETIDE_FENCE             = 0x08,
        NETIDE_OPENFLOW          = 0x11,
        NETIDE_NETCONF           = 0x12,
        NETIDE_OPFLEX            = 0x13,
        NETIDE_OFCONFIG          = 0x14,
        NETIDE_OTHER             = 0xFF
};
```

[1] `NETIDE_MGMT` and `NETIDE_TOPOLOGY` message types are not documented in the current specification. They have been introduced to allow future extensions of the Network Engine capabilities.

`datapath_id` is a 64-bits field that uniquely identifies the network elements. `module_id` is a 32-bits field that uniquely identifies Backends and application modules running on top of each client controller. The composition mechanism in the Core layer leverages on this field to implement the correct execution flow of these modules. Finally, `nxid` is the transaction identifier associated to the each message. Replies must use the same value to facilitate the pairing.

A.2 Module announcement

The Core executes composition and conflict resolution operations based on a configuration file which specifies how the applications modules cooperate in controlling the network traffic. In particular, configuration parameters determine the way the Core handles the messages received from the applications modules running on top of the client controllers. To this purpose, each message is encapsulated with the NetIDE header containing a `module_id` value that identifies the module that has issued the message.
`module_id` values are assigned by the Core during the modules announcement/acknowledge process described in this Section. As a result of this process, each Backend and application module can be recognized by the Core through an identifier (the `module_id`) placed in the NetIDE header.

As a first step, Backends register themselves by sending a module announcement message (message type `NETIDE_MODULE_ANN`) to the Core containing a human-readable identifier such as: `backend-<platform_name>-<pid>`. Where `platform_name` is the name of the client controller platform (*ryu*, *onos*, *odl* and *floodlight* can be used) and `pid` is the process ID of the instance of the client controller which is performing the registration. The format of the message is the following:

```
struct NetIDE_message{
        netide_ver              = 0x04
        type                    = NETIDE_MODULE_ANN
        length                  = len("backend-<platform_name>-<pid>")
        nxid                    = 0
        module_id               = 0
        datapath_id             = 0
        data                    = "backend-<platform_name>-<pid>"
}
```

The answer generated by the Core (message type `NETIDE_MODULE_ACK`) includes a `module_id` value and the Backend name in the payload (the same indicated in the `NETIDE_MODULE_ANN` message):

```
struct NetIDE_message{
        netide_ver              = 0x04
        type                    = NETIDE_MODULE_ACK
        length                  = len("backend-<platform_name>-<pid>")
        nxid                    = 0
        module_id               = BACKEND_ID
        datapath_id             = 0
        data                    = "backend-<platform_name>-<pid>"
}
```

After this step, all the messages generated by the Backend (e.g., heartbeat and hello messages described in the following Sections) will contain the `BACKEND_ID` value in the `module_id` field of the NetIDE header. Furthermore, `BACKEND_ID` is used to register the application modules that are running on top of the client controller:

```
struct NetIDE_message{
        netide_ver              = 0x04
        type                    = NETIDE_MODULE_ANN
```

```
        length                  = len("module_name")
        nxid                    = 0
        module_id               = BACKEND_ID
        datapath_id             = 0
        data                    = "module_name"
}
```

where `module_name` is the name of the module under registration. The module's name can be assigned by the Backend or retrieved from the module itself via API calls. The Core replies with:

```
struct NetIDE_message{
        netide_ver              = 0x04
        type                    = NETIDE_MODULE_ACK
        length                  = len("module_name")
        nxid                    = 0
        module_id               = MODULE_ID
        datapath_id             = 0
        data                    = "module_name"
}
```

After this last step, the Backend allows the application modules to control the network. In particular, network commands sent towards the network (e.g. OpenFlow `FLOW_MOD`s, `PACKET_OUT`s, `FEATURES_REQUEST`s) are intercepted by the Backend, which encapsulates them with the NetIDE header containing the `MODULE_ID` value. Such a value is then used by the Core to recognize the sender of the message and to properly feed the composition and conflict resolution operators.

A.3 Heartbeat

The heartbeat mechanism has been introduced after the adoption of the ZeroMQ messaging queuing library [29] to transmit the NetIDE messages. Unfortunately, the ZeroMQ library does not offer any mechanism to find out about disrupted connections (and also completely unresponsive peers). This limitation can be an issue for the Core's composition mechanism and for the tools connected to the Network Engine, as they are not able to understand when an client controller disconnects or crashes. As a countermeasure, Backends must periodically send (let's say every 5 seconds) a "heartbeat" message to the Core. If the Core does not receive at least one "heartbeat" message from the Backend within a certain timeframe, the Core considers it disconnected, removes all the related data from its memory structures and informs the relevant tools. In order to minimize the service disruption, the Core applies default policies as specified in the composition specification (e.g. a "drop all" action in case of disconnected firewall module).
The format of the message is the following:

```
struct NetIDE_message{
        netide_ver              = 0x04
        type                    = NETIDE_HEARTBEAT
        length                  = 0
        nxid                    = 0
        module_id               = BACKEND_ID
        datapath_id             = 0
        data                    = 0
}
```

A.4 Handshake

Upon completion of the connection with the Core (and of the module announcement/acknowledge process for the Backends), Shim and Backends must immediately send a `hello` message with the list

of the supported control and/or management protocols. The format of the message is the following:

```
struct NetIDE_message{
        netide_ver              = 0x04
        type                    = NETIDE_HELLO
        length                  = 2*NR_PROTOCOLS
        nxid                    = 0
        module_id               = BACKEND_ID
        datapath_id             = 0
        data                    = [list of supported protocols]
}
```

Where `data` contains one 2-byte word (in big endian order) for each protocol, with the first byte containing the code of the protocol according to the above `enum`, while the second byte indicates the version of the protocol (e.g. according to the ONF specification, 0x01 for OpenFlow v1.0, 0x02 for OpenFlow v1.1, etc.). NETCONF version is marked with 0x01 that refers to the specification in the RFC6241 [11], while OpFlex version is marked with 0x00 since this protocol is still in work-in-progress stage [30].
The Core responds with another `hello` message containing the following:

```
        struct NetIDE_message{
        netide_ver              = 0x04
        type                    = NETIDE_HELLO
        length                  = 2*NR_PROTOCOLS
        nxid                    = 0
        module_id               = BACKEND_ID
        datapath_id             = 0
        data                    = [list of supported protocols]
}
```

if at least one of the protocols requested by the client is supported by the composition mechanism. In particular, `data` contains the codes of the protocols that match the client's request (2-bytes words, big endian order). The `backend_id` value is used in the NetIDE header to allow the Core to forward the reply to the Backend that started the handshake. If none of the requested protocols is supported, the header of the reply is as follows:

```
        struct NetIDE_message{
        netide_ver              = 0x04
        type                    = NETIDE_ERROR
        length                  = 2*NR_PROTOCOLS
        nxid                    = 0
        module_id               = BACKEND_ID
        datapath_id             = 0
        data                    = [list of supported protocols]
}
```

where the payload of the message `data` contains the codes of all the protocols supported by the server controller (2-bytes words, big endian order).

A.5 The FENCE mechanism

An application module may respond to a given network event (e.g., an OpenFlow `PACKET_IN`) with a set of zero, one or multiple network commands (e.g., OpenFlow `FLOW_MOD`s and `PACKET_OUT`s). The so-called FENCE mechanism is a means for the Core to correlate events and commands and to know when the module has finished processing the input event.
This mechanism is implemented through the message type `NETIDE_FENCE` which is sent by the Backend to the Core once a module has finished processing a network event. Within the same

transaction, FENCE message, the network event and related network commands use all the same `module_id` and `nxid` values in the NetIDE header so that the Core can correlate them.

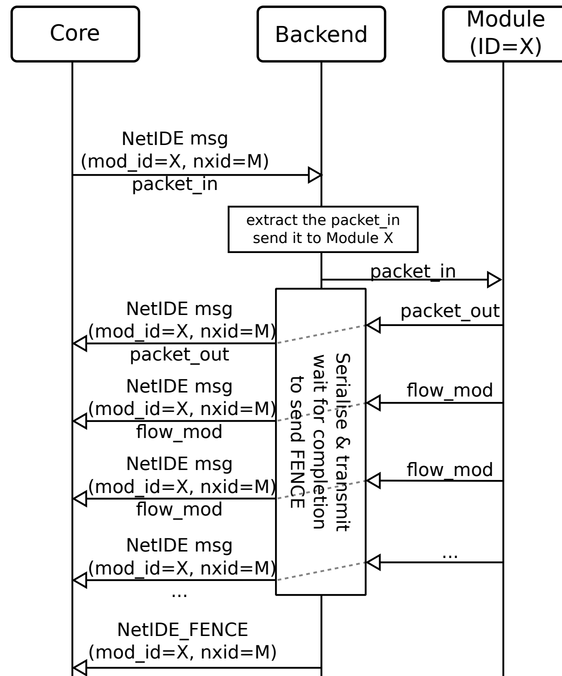

Figure A.1: Fence mechanism workflow. Both `nxid` and `module_id` refer to the NetIDE header fields.

The process is represented in Fig. A.1, where a `PACKET_IN` event is encapsulated with the NetIDE header by the Core with values `module_id=X` and `nxid=M` and finally sent to the Backend. The Backend removes the NetIDE header and forwards the `PACKET_IN` the the application module `X`. The module reacts with zero, one or multiple network commands (represented by OpenFlow messages `PACKET_OUT` and `FLOW_MOD` in the figure). Each network command is encapsulated by the Backend with the NetIDE header re-using the same `module_id` and `nxid` values received from the Core with the `PACKET_IN`. Therefore, the Core uses the `nxid` value to pair the network commands generated by the module and the previous network event.
Once the module's event handling function returns, the Backend issues a FENCE message to signal the completion of the transaction to the Core.

A.6 The OpenFlow protocol

In this specification, the support for all versions of OpenFlow is achieved with the following:

```
struct netide_message{
        struct netide_header header;
        uint8 data[0]
};
```

Where `header` contains the following values: `netide_ver=0x05`, `type=NETIDE_OPENFLOW` and `length` is the size of the original OpenFlow message which is contained in `data`.

A.6.1 Properly handling reply messages

The NetIDE protocol helps the Network Engine in pairing OpenFlow reply messages with the corresponding requests issued by the application modules running on top of it (e.g. statistics, feature requests, configurations, etc., thus the so-called "controller-to-switch" messages defined in the OpenFlow specifications). In this context, the `xid` field in the OpenFlow header is not helpful, as may happen that different modules use the same values.
In the proposed approach, represented in Fig. A.2, the task of pairing replies with requests is performed by the Core which replaces the `xid` of the OpenFlow requests with new unique values and stores the original `xid` and the `module_id` it finds in the NetIDE header. As the network elements use the same `xid` values in the replies, the Core can easily pair requests and replies and can use the saved `module_id` to send the reply to the right application module.
The diagram in Fig. A.2 shows how the Network Engine handles the controller-to-switch OpenFlow messages. The workflow starts with an application module that issues an OpenFlow request with `xid=N`. The Backend relays the message to the Core by encapsulating it with the NetIDE header by using `module_id=X` previously assigned to the application module (see Section A.2). Once the Core receives this message, it computes a new OpenFlow `xid` value `M` (e.g. by using a hashing algorithm) and ensures that such a value is not being used in other existing transactions.

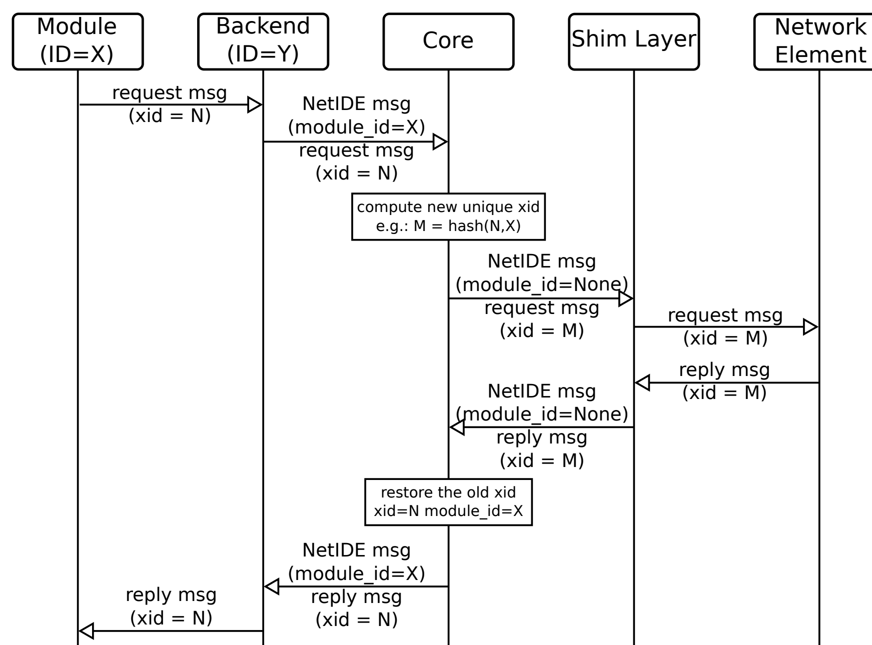

Figure A.2: Request/reply message handling. `xid` refers to the OpenFlow header field.

Before sending the reply to the Backend, the Core restores the original `xid` in the OpenFlow reply (the application module expects to find in the reply the same `xid` value that was used for the request) and inserts the `module_id` previously saved in the NetIDE header. The Backend will use this information to forward the message to the right application module.

Asynchronous OpenFlow messages generated by the network elements are ignored by the above described tracking mechanism. They are simply relayed to the Backends that eventually forward them to the relevant application modules based on the composition and topology specifications. Currently defined OpenFlow asynchronous messages are the following:

Message Type	ID	Description	OF Version
OFPT_PACKET_IN	10	New packet received by a switch	1.0-1.5
OFPT_FLOW_REMOVED	11	Flow rule removed from the table	1.0-1.5
OFPT_PORT_STATUS	12	Port added, removed or modified	1.0-1.5
OFPT_ROLE_STATUS	30	Controller role change event	1.4-1.5
OFPT_TABLE_STATUS	31	Changes of the table state	1.4-1.5
OFPT_REQUESTFORWARD	32	Request forwarding by the switch	1.4-1.5
OFPT_CONTROLLER_STATUS	35	Controller status change event	1.5

A.7 Other SBI protocols

The NetIDE intermediate protocol can easily support other SBI protocols, such as NETCONF [11], OF-Config [31] or OpFlex [30].
While OF-Config configurations are only encoded in eXtensible Markup Language (XML), NetConf and OpFlex specifications are more flexible and support both XML and JavaScript Object Notation (JSON) encoding formats. For this reason, we need an additional field in the NetIDE header to indicate the format of the message contained in `data` and to allow the recipients to correctly handle it. To this purpose, when transmitting NetConf or OpFlex messages, the sender must set `type=NETIDE_OTHER` in the NetIDE header to indicate the presence of an additional 16-bits field at the end of the header. This field, named `ext_type`, specifies the SBI protocol and the format of the message carried by `data`:

```
struct netide_message{
        struct netide_header header;
        uint16_t ext_type;
        uint8_t data[0];
};
```

Where `header` contains the following values: `netide_ver=0x05`, `type=NETIDE_OTHER` and `length` is the size of the original SBI message carried by `data`. The value of `ext_type` indicates the SBI protocol in the most significant byte (as specified in Section A.1) and the format of the message (either `0x00` for XML or `0x01` for JSON) in the least significant byte.

Bibliography

[1] The NetIDE consortium. D5.4 - Use case design: 2nd release. Technical report, The European Commission, 2016.

[2] OpenDaylight Beryllium Project list. https://wiki.opendaylight.org/view/Project_list, apr 2016.

[3] Y. Rekhter, T. Li, and S. Hares. A Border Gateway Protocol 4 (BGP-4). Technical Report 4271, IETF Secretariat, January 2006.

[4] Extensible Markup Language (XML) 1.0 (Fifth Edition). https://www.w3.org/TR/xml/, nov 2008.

[5] Oren Ben-Kiki and Clark Evans. YAML Ain't a Markup Language. http://www.yaml.org/spec/1.2/spec.html, oct 2009.

[6] The NetIDE Consortium. NetIDE deliverable 2.5 - NetIDE Runtime Architecture Consolidation, March 2016.

[7] Ryu SDN Framework. https://osrg.github.io/ryu/.

[8] ONOS - Open Network Operating System. http://onosproject.org/.

[9] The OpenDaylight Platform. https://www.opendaylight.org/.

[10] The NetIDE Consortium. NetIDE deliverable 2.4 - NetIDE Core concepts and Architecture v2, Sep 2015.

[11] R. Enns, M. Bjorklund, J. Schoenwaelder, and A. Bierman. Network Configuration Protocol (NETCONF). Technical Report 6241, IETF Secretariat, June 2011.

[12] ONOS Java API. http://api.onosproject.org/, Nov 2015.

[13] ONF - Introducing ATRIUM. https://www.opennetworking.org/?p=1757&option=com_wordpress&Itemid=316.

[14] The NetIDE Consortium. NetIDE deliverable 2.3 - NetIDE IRF APIs Integrated Platform v1, Apr 2015.

[15] IEEE standard for local and metropolitan area networks – station and media access control connectivity discovery.

[16] Wireshark. Network protocol analizer for Unix and Windows. https://www.wireshark.org/.

[17] ONF Blog - Intent: What. Not How. https://www.opennetworking.org/?p=1633&option=com_wordpress&Itemid=101.

[18] ONF: Northbound Interfaces. https://www.opennetworking.org/technical-communities/areas/services/1916-northbound-interfaces.

[19] ONOS: Intent Framework. https://wiki.onosproject.org/display/ONOS/Intent+Framework.

[20] Project Boulder: Intent Northbound Interface (NBI). http://opensourcesdn.org/projects/project-boulder-intent-northbound-interface-nbi/.

[21] GitHub: Boulder, Intent-based NBI development. https://github.com/OpenNetworkingFoundation/BOULDER-Intent-NBI.

[22] Ben Pfaff and Bruce Davie. The Open vSwitch Database Management Protocol. RFC 7047, December 2013.

[23] Open vSwitch. http://openvswitch.org/.

[24] Martin Bjorklund. YANG - A Data Modeling Language for the Network Configuration Protocol (NETCONF). RFC 6020, October 2015.

[25] Yinben Xia, Sheng Jiang, Tianran Zhou, Susan Hares, and Yali Zhang. NEMO (NEtwork MOdeling) Language. Internet-Draft draft-xia-sdnrg-nemo-language-04, Internet Engineering Task Force, April 2016. I-D Exists.

[26] The NetIDE consortium. D2.1 - NetIDE Architecture. Technical report, The European Commission, 2014.

[27] Bert Wijnen, Tianran Zhou, Susan Hares, and Dr. Pedro A. Aranda Gutiérrez. Intent-based Network Modeling. *IETF Journal*, 11(2):23, 2015.

[28] IETF 94 Hackathon. https://www.ietf.org/registration/MeetingWiki/wiki/94hackathon, nov 2015.

[29] 0MQ Distributed Messaging. http://zeromq.org/, 2014.

[30] M. Smith et al. OpFlex Control Protocol. Technical report, IETF, November 2014.

[31] OpenFlow Management and Configuration Protocol (OF-Config 1.1). https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow-config/of-config-1.1.pdf, Jun 2012.