Document Properties

Document Number:	D 4.1
Document Title:	**NetIDE App Engine 1st release**
Document Responsible:	Roberto Doriguzzi Corin (CREATE-NET)
Document Editor:	Roberto Doriguzzi Corin (CREATE-NET)
Authors:	Roberto Doriguzzi Corin (CREATE-NET) Pedro A. Aranda Gutiérrez (TID) Alec Leckey (Intel) Carmen Guerrero (IMDEA)
Target Dissemination Level:	PU
Status of the Document:	Final
Version:	2.0

Production Properties:

Reviewers:	Elisa Rojas (Telcaria), Arne Schwabe (UPB)

Document History:

Disclaimer:

Abstract:

The purpose of this document is to provide an overview of the first release of the so-called NetIDE Network Engine. In particular, this document gives a brief description of the Network Engine architecture and how it has been implemented. Shortly, the architecture is based on the WINE concept [1] where Microsoft Windows applications (such as Microsoft Office) can run on top of POSIX operating systems thanks to the libraries and the APIs provided by Wine.
In a similar way, Ryu [2] (or Floodlight [3], or others) applications can run on top of the OpenDaylight (ODL) [4] (or others) framework thanks to the Network Engine modules and APIs. In this document we summarize how this goal has been achieved and how to test the Network Engine.

Keywords:

software defined networks, networking, network application

Contents

List of Figures

List of Tables

List of Acronyms

API	Application Programming Interface
JSON	JavaScript Object Notation
ODL	OpenDaylight
OF	OpenFlow
OSGi	Open Services Gateway Initiative
POM	Project Object Model
POSIX	Portable Operating System Interface for Unix
SAL	Service Abstraction Layer
SBI	Southbound Interface
SDN	Software Defined Networking
TCP	Transport Control Protocol
WINE	Wine Is Not an Emulator

Executive Summary

The present document accompanies the first software release of the NetIDE Network Engine. An overview of the NetIDE framework can be found in Deliverable D2.2v2 [5], however some architectural details of the Network Engine have been also included in the Introduction of this document. In particular, we recall the concepts of Client and Server Controllers, as defined by the Open Networking Foundation, and how these concepts are applied to the Network Engine to achieve the runtime approach.
With this prototype we release a first implementation of the so-called *NetIDE API Interceptor* (the core component of the Network Engine) that has been designed to allow Client and Server Controllers to communicate with each other. The document provides architectural and implementation details of the API Interceptor, the installation procedure for each software module and the road-map for the future work.
This second and revised version is motivated by the 1^{st} Yearly Review of the project. We have included footnotes to highlight changes introduced in the text as a result of comments or requests stemming from it.

1 Introduction

The high-level goal of the NetIDE Network Engine (Fig. 1.1) is to enable Network Applications to be executed, systematically tested, and refined on a variety of concrete Software Defined Networking (SDN) target platforms. To achieve this goal, the Network Engine has been envisioned as a runtime system by following the WINE [1] paradigm. I.e. the Network Engine provides a compatibility layer (called NetIDE API Interceptor) capable of translating the Network Application calls into calls for the SDN so-called Server Controller platform [1].

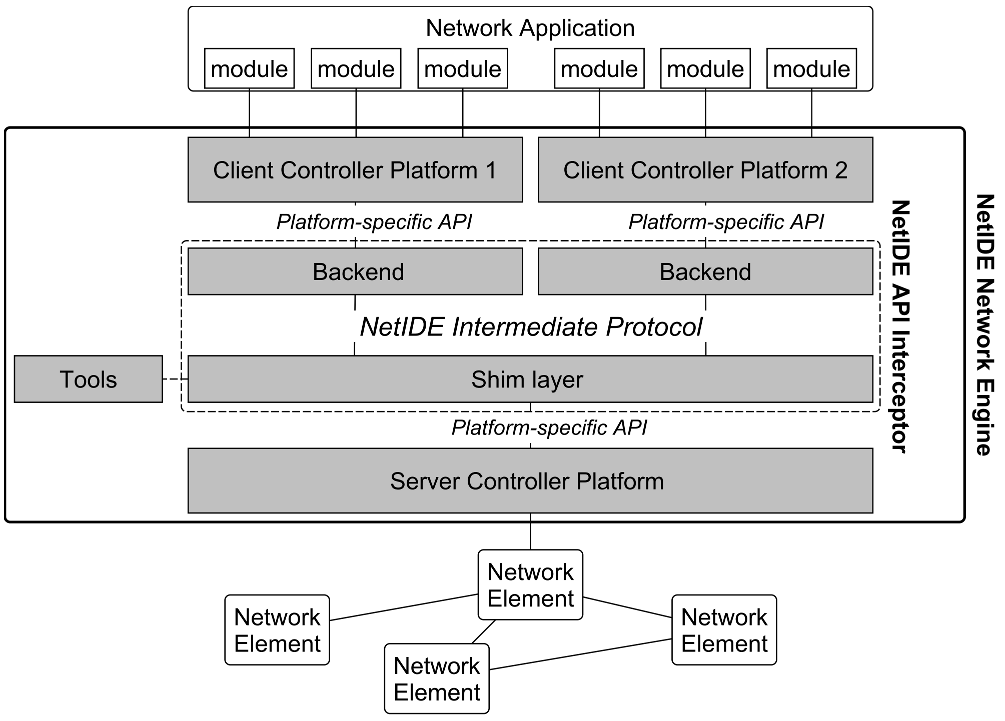

Figure 1.1: Architecture of the Network Engine.

The benefits of the NetIDE approach can be summarized as follows[2]:

- Network Applications written for many different controller frameworks, e.g. implemented in the past for different environments/needs, can be re-used and executed on top of the controller

[1]The Network Engine follows the layered SDN controller approach proposed by the Open Networking Foundation [6]. It integrates a Client Controller layer that executes the modules that compose a Network Application and interfaces with a Server SDN Controller layer that drives the underlying infrastructure.

[2]The description of the benefits of the NetIDE framework has been added in response to review comment C#4. In particular, our aim is to explain that implementing and using a "dummy" SDN controller instead of "real" SDN controllers on the Server side is not a viable solution. Moreover, we provide the relation between the NetIDE approach and the Use Cases UC1 and UC3. We also state that we are considering enhancing the Shim Layer in order to expose high level services provided by advanced Server Controller platforms to the Client Controllers.

framework that is currently managing a given network infrastructure (e.g., in Use Case UC1 [7] we envision a Data Center operator that needs to migrate a SDN infrastructure to an infrastructure based on a different SDN framework) or, on the other way around, a well-tested Network Application can be ported and executed "as is" on a second network controlled by another controller framework (as envisioned in the Use Case UC3 "Hybrid Environment" [7], where a Floodlight application implemented for the main office branch must be also deployed in a remote branch controller by OpenDaylight).

- The Network Engine can be exploited to empower any SDN platform with the advanced functionalities and richer southbound interfaces offered by other (more powerful) platforms such as ONOS or OpenDaylight. This goal can achieved by enhancing the Server Controller Platform specific implementations of the Shim Layer with the support for the needed services.

The NetIDE API Interceptor (introduced in Chapter 2) is the core component of the Network Engine and is a generalization of an early proof-of-concept (described in Section 4.5 of the Deliverable D2.1 [8]) where SDN applications implemented for Pyretic [9] could be executed on top of the Ryu framework [2].
The API Interceptor is composed of two modules, the *Backend* and the *Shim Layer*. The first, along with the above Client Controller platform, can be compared to the Win32 sub-system of WINE [10] that re-creates the native environment for the Windows applications. In the same way, the platform and the Backend allow the Network Applications to run in their native environment. The latter can be compared to the WINE drivers, that allow the Win32 sub-system to communicate with the hardware (in our case with the switches) through the Unix system libraries and drivers (in our case the Server Controller platform APIs and OpenFlow libraries).
As shown in Fig. 1.1, the Network Engine architecture foresees applications written for different control frameworks to run simultaneously and controlling/configuring the same physical infrastructure. To this purpose, one of the main objectives for the next release of the Network Engine is the definition and the implementation of a specific mechanism that is able to handle the conflicts that may occur with multiple Client Controllers running in parallel (a well-know and non-trivial problem already investigated in, e.g., [11], [12] and a few other works). We are also considering the case when a Network Application is written for the Server Controller platform and is deployed beside the Shim Layer. In this specific scenario, this application would by-pass the conflict resolution logic applied by the Shim Layer. To this purpose, we are investigating whether it is possible to enhance the Shim Layer to intercept the interaction between the application and the Server Controller[3].
Finally, the Network Engine can be extended with additional tools (e.g. debugger, logger, resource manager, etc.) that are plugged to the API Interceptor through a publish/subscribe messaging system where the Shim Layer is the publisher and the tools are the subscribers.

[3]This paragraph is intended as a response to Review Request R#10

The architecture we have outlined above has a rather high ambition level. To get a first understanding on feasibility, we started our practical work with a restricted version of the architecture as depicted in Figure 1.2. Specifically, we made the following main assumptions:

- We only considered two Java-based and two Python-based controller frameworks, namely OpenDaylight, Floodlight, Ryu and Pyretic/POX.
- We considered the Pyretic protocol as a preliminary intermediate protocol between Backend and Shim Layer (details in Section 2.1).
- We only considered the single client controller scenario, no conflict resolution mechanism.

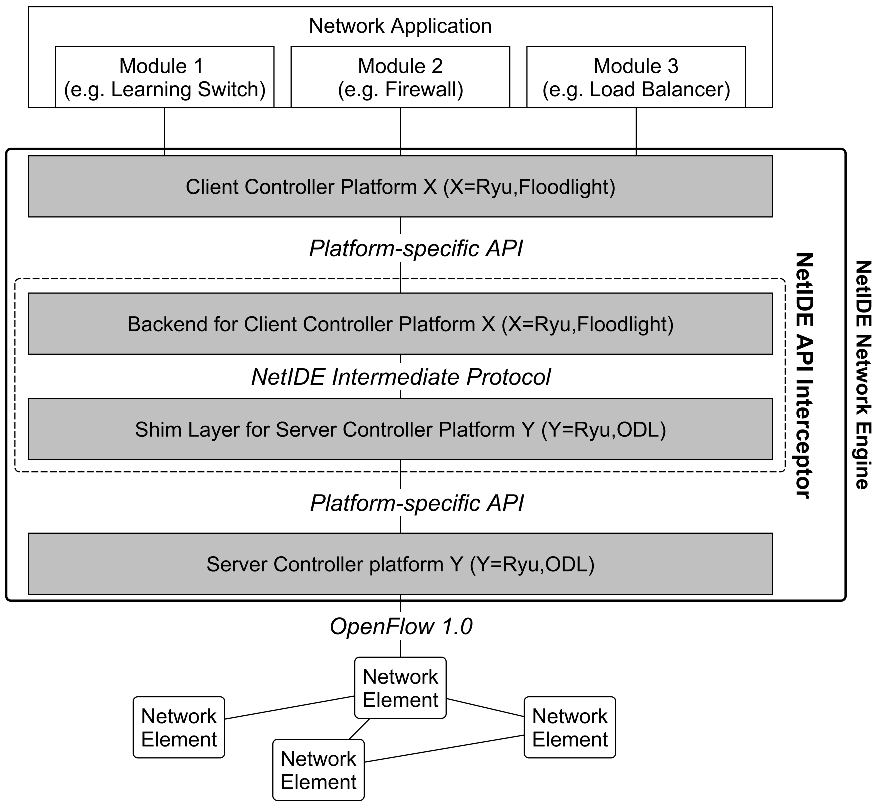
Network Application
Module 1 (e.g. Learning Switch)
Module 2 (e.g. Firewall)
Module 3 (e.g. Load Balancer)
Client Controller Platform X (X=Ryu,Floodlight)
Platform-specific API
Backend for Client Controller Platform X (X=Ryu,Floodlight)
NetIDE Intermediate Protocol
Shim Layer for Server Controller Platform Y (Y=Ryu,ODL)
Platform-specific API
Server Controller platform Y (Y=Ryu,ODL)
NetIDE API Interceptor
NetIDE Network Engine
OpenFlow 1.0
Network Element
Network Element
Network Element
Network Element

Figure 1.2: Architecture of the first prototype of the Network Engine.

In the rest of the document we describe the implementation details of the first version of the Network Engine. In particular the document is organized as follows: Chapter 2 provides an overview of the NetIDE API Interceptor. Chapter 3 describes the installation and configuration procedures of the two implementations of the Shim Layer (ODL and Ryu), while Chapter 4 describes the installation and configuration procedures of the two implementations of the Backend (Floodlight and Ryu). Conclusions and next steps are provided in Chapter 5.

2 The NetIDE API Interceptor

The API Interceptor is a component of the NetIDE architecture that allows SDN applications written for different SDN controllers to be executed on top of ODL or other Server Platforms like Ryu. The API Interceptor is composed of two different modules:

i) The *Shim layer*, which is an application for the Server Platform (in this first release ODL or Ryu)

ii) The *Backend layer* implemented for the SDN platform of the user's application (currently Ryu and FloodLight are supported).

The two modules (Backend and Shim Layer) talk each other through a TCP socket by using the so-called *NetIDE Intermediate Protocol*. To summarize with an example: when a `packet_in` message arrives to ODL from a switch, the message is converted by the ODL Shim Layer into a intermediate serialized message and then sent to the above Backend module of the, e.g., Ryu controller. Finally the message is passed to the application that eventually decides how to forward the related flow through either a `packet_out` or a `flow_mod` answer.

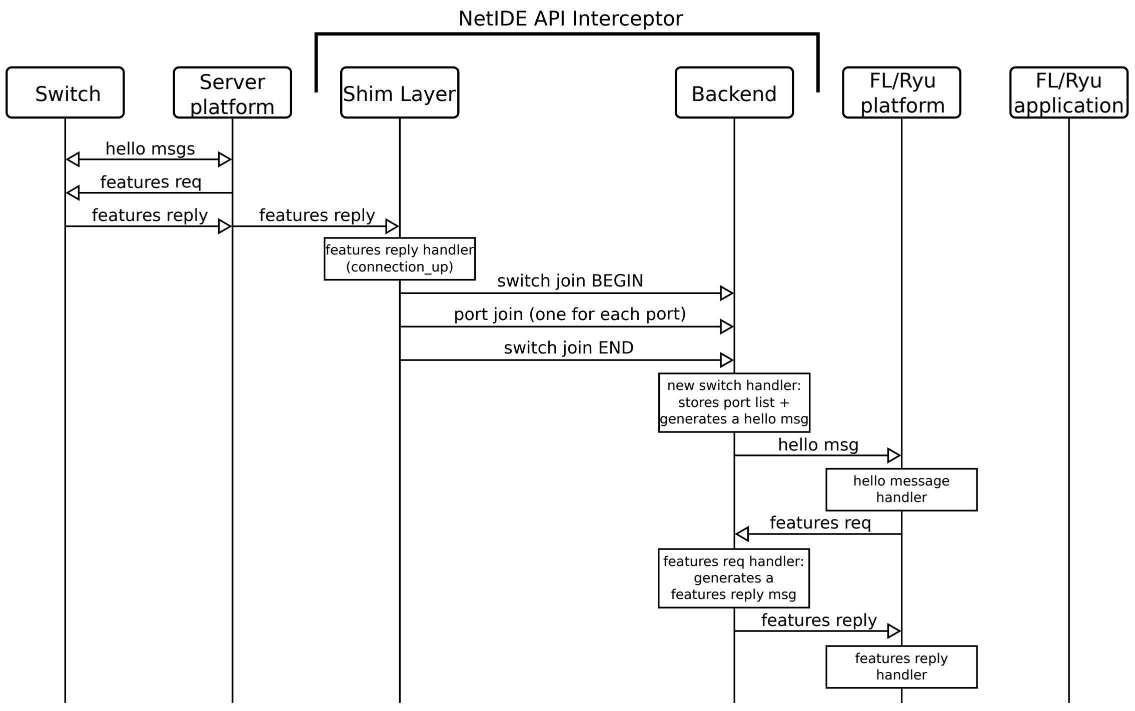

Figure 2.1: NetIDE API Interceptor. Handshake workflow.

In Fig. 2.1 the workflow of the handshake stage is represented. It is worth to notice that the Backend not only forwards the messages coming from the Shim Layer to the applications and vice-versa, but it also generates the `hello` and `feature_reply` messages based on the information of the physical topology received from the Shim Layer. This operation is needed to simulate the handshake session that happens when an OpenFlow switch connects to a controller.

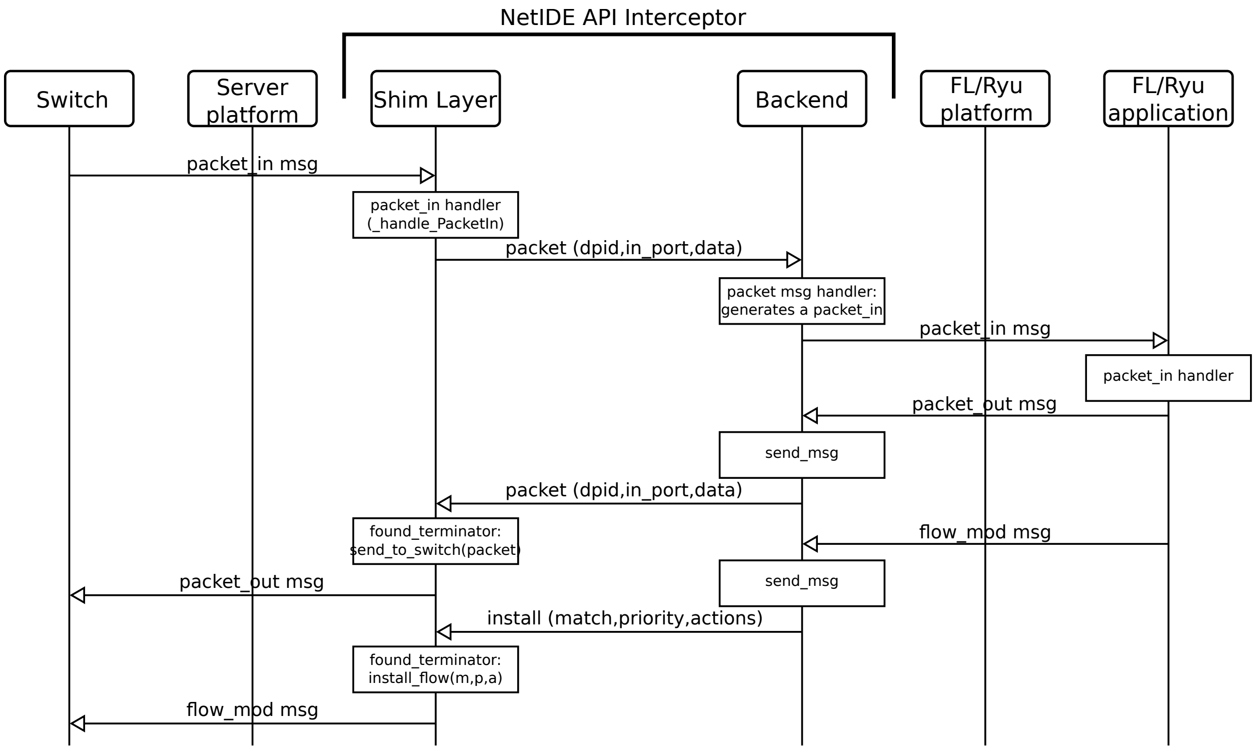
NetIDE API Interceptor
Switch
Server platform
Shim Layer
Backend
FL/Ryu platform
FL/Ryu application
packet_in msg
packet_in handler (_handle_PacketIn)
packet (dpid,in_port,data)
packet msg handler: generates a packet_in
packet_in msg
packet_in handler
packet_out msg
send_msg
packet (dpid,in_port,data)
found_terminator: send_to_switch(packet)
flow_mod msg
packet_out msg
send_msg
install (match,priority,actions)
found_terminator: install_flow(m,p,a)
flow_mod msg

Figure 2.2: NetIDE API Interceptor. Main workflow.

Fig. 2.2 shows the workflow of the management of the most common control messages: `packet_in`, `packet_out` and `flow_mod`. This workflow (along with the one represented in Fig. 2.1) summarizes the status of the NetIDE API Interceptor released with this Deliverable. This means that the modules described in the following sections are all able to complete the handshake session and to handle `packet_in` messages coming from the network by replying with `packet_outs` and/or `flow_mods`.

2.1 The Intermediate Protocol

To speed up the initial development process, where multiple developers were working in parallel on different components of the NetIDE API Interceptor, it has been agreed to use, as Intermediate Protocol, the same protocol (and related API) used by Pyretic between its Backend sub-module and the so-called POX client. Although the Pyretic API has several limitations already highlighted in Deliverable D2.1 [8] (e.g.: only OpenFlow 1.0, no idle and hard timeouts for the flow entries, etc.), it has been adopted for two main reasons:

1. Both Shim Layer and Backend implementations could be developed in parallel with a common API that, although with some limitations, have proved to be advanced enough for the implementation of the "*Vendor Agnostic Data-Centre Evolution*" use case (see Appendix A.1 of Deliverable D5.1 [13] and Section 3.1 of Deliverable D5.2 [7]).

2. The Shim Layer could be tested during the development phase with the original Pyretic Backend. On the other hand, the new Backends for Ryu and Floodlight could be tested with the POX client included within the Pyretic source code. Therefore, both Shim Layer and Backend could be developed in parallel by different partners.

2.1.1 The Shim Layer northbound interface

The Shim Layer receives the OpenFlow messages coming from the switches and translates them into an intermediate format where the original OpenFlow messages are translated into simple tuples (see Table 2.1).

OpenFlow protocol	Intermediate message	Comment
features_reply	[‘switch’,‘join’,datapath.id,’BEGIN’] [‘port’,‘join’,datapath.id, port.port_no, CONF_UP, STAT_UP, PORT_TYPE] [‘switch’,‘join’,datapath.id,’END’]	The *features_reply* message is forwarded to the Backend as a set of messages “switch join” and “port join”.
packet_in	[‘packet’,received]	*received* is a tuple containing the datapath ID of the switch, the input port of the flow and the Ethernet frame
lldp (packet_in)	[‘link’, originatorDPID, originatorPort, event.dpid, event.port]	
port_status	[‘port’,‘join’ or ‘mod’,event.dpid, port.port_no, CONF_UP, STAT_UP, PORT_TYPE]	New port or notification of the new status of a port. CONF_UP and STAT_UP contain the PORT_DOWN config value and the LINK_DOWN state value respectively
flow_stats_reply	[‘flow_stats_reply’, dpid, flow_stats]	
switch disconnection	[‘switch’,‘part’,event.dpid]	Message sent to the Backend when the controller loses the contact with a switch.

Table 2.1: List of the shim-to-backend messages.

2.1.2 The Backend southbound interface

The Backend receives the OpenFlow messages coming from the applications and translates them into an intermediate format where the original OpenFlow messages are translated into simple tuples (see Table 2.2).

2.1.3 The “serialize” method

The *serialize* method is used by both Shim Layer and Backend to format the messages exchanged between each other. Basically, the serialize method takes a tuple containing one of the intermediate messages listed in Tables 2.1 and 2.2 and returns a messages in JavaScript Object Notation (JSON)

OpenFlow protocol	Intermediate message	Comment
packet_out	['packet', packet]	The first element is a string (like for all the other messages), while the second is a dictionary that contains, among other items, the output port for the flow.
flow_mod (add)	['install', match, priority, action_list]	
flow_mod (delete)	['delete', match, priority]	
flow_mod (delete all)	['clear', dpid]	Removes all the entries from the flow tables of the switch with datapath ID == 'dpid'
flow_stats_request	['flow_stats_request', dpid]	
barrier	['barrier', dpid]	
lldp (packet_out)	['inject_discovery_packet', dpid, port]	

Table 2.2: List of the backend-to-shim layer messages.

format [14]. The method, inherited from the Pyretic's code, is required for all the implementations of both Shim Layer and Backend.
To better understand how a message is processed by this method, lets analyze how a new forwarding rule is sent down from the Backend to the Shim Layer (which eventually converts the message to a `flow_mod` for the switch):

```
['install', {'dstip': 10.0.0.1, 'protocol': 1, 'srcmac': 12:93:2c:4a:48:24, 'tos': 0,
    'dstmac': 6a:2d:7f:29:c8:61, 'inport': 1, 'switch': 2, 'ethtype': 2048, 'srcip':
    10.0.0.2, 'dstport': 0, 'srcport': 0}, 0, [{'outport': 2}]]
```

The sample message contains:

- `'install'` is a string that is used to identify the message type at the other side of the communication (in this case the Shim Layer)
- the match: `'dstip': 10.0.0.1, 'protocol': 1, 'srcmac': 12:93:2c:4a:48:24, 'tos': 0, 'dstmac': 6a:2d:7f:29:c8:61, 'inport': 1, 'switch': 2, 'ethtype': 2048, 'srcip': 10.0.0.2, 'dstport': 0, 'srcport': 0`
- the priority: `0`
- the list of actions (in this example we only have an output action `['outport': 2]`)

In this case the application (a simple learning switch application) wants to install a new entry on the `switch 2` that sends out to port 2 all the packets with header matching the match.
It is worth noting how the API allows the specification of the priority of the rule but no fields are foreseen for neither idle nor hard timeouts (this is one of the limitations that we should overcome in the future versions of this API).
The message is processed in several steps. One of these steps converts the values of the match into ASCII values (e.g. '1' = 49, '0'=48, '.'=46 etc.). The final result is a message in JSON format as follows:

```
["install", {"dstip": [49, 48, 46, 48, 46, 48, 46, 49], "protocol": 1, "srcmac": [49,
    50, 58, 57, 51, 58, 50, 99, 58, 52, 97, 58, 52, 56, 58, 50, 52], "tos": 0, "dstmac":
    [54, 97, 58, 50, 100, 58, 55, 102, 58, 50, 57, 58, 99, 56, 58, 54, 49], "inport": 1,
    "switch": 2, "ethtype": 2048, "srcip": [49, 48, 46, 48, 46, 48, 46, 50], "dstport":
    0, "srcport": 0}, 0, [{"outport": 2}]]
```

3 The Shim Layer

The Shim Layer represents the lowest part of the NetIDE API Interceptor (gray box in Fig. 3.1). It is implemented as an application for the Server Controller Framework (by using its native API) and exposes the Intermediate API as a northbound interface (see Section 2.1) for the communication with one of the Backends (see Chapter 4).
This chapter describes the architecture, installation and execution procedures of different implementations of the Shim Layer for the following controller frameworks acting as server controller: OpenDaylight, Ryu and POX. While the first two implementations were achieved within the NetIDE development activities, the last one was ported from the source code of Pyretic and included into the NetIDE code repository for testing purposes.

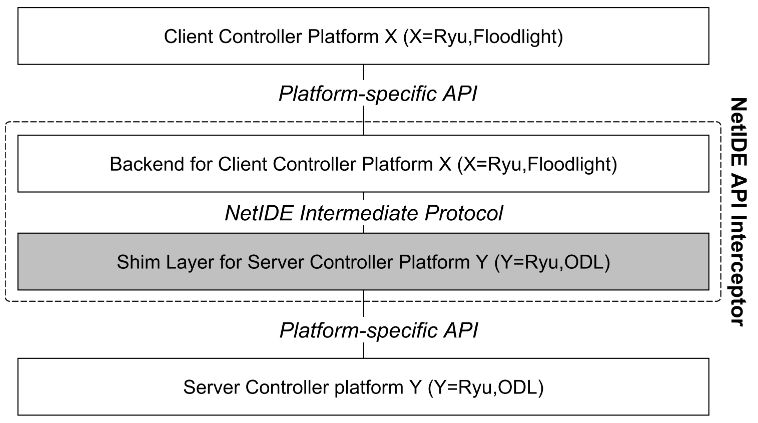
Client Controller Platform X (X=Ryu,Floodlight)
Platform-specific API
Backend for Client Controller Platform X (X=Ryu,Floodlight)
NetIDE Intermediate Protocol
Shim Layer for Server Controller Platform Y (Y=Ryu,ODL)
Platform-specific API
Server Controller platform Y (Y=Ryu,ODL)
NetIDE API Interceptor

Figure 3.1: The Shim Layer

The source code of the software modules presented in this Chapter and released with this Deliverable can be downloaded from the NetIDE Github repository (the links are listed in the Appendix A). Alternatively, to get the latest version of the code, clone the repository by using the following command:

```
git clone https://github.com/fp7-netide/Engine/
```

3.1 The OpenDayLight Shim Layer

One of the candidates for the NetIDE server controller is OpenDaylight. The main features that make it a good candidate to implement the server controller are (see also Chapter 4 of Deliverable D2.2v2 [5]):

- Modularity: ODL is highly modular
- Module life-cycle control: ODL includes Open Services Gateway Initiative (OSGi)
- Supports multiple and heterogeneous Southbound Interfaces

3.1.1 Architecture[1]

Figure 3.2 shows the high-level architecture for NetIDE instantiated in the case of OpenDaylight. It reflects the currently implemented functionality. It positions the Shim Layer in the context of OpenDaylight and the interactions with components of the ODL framework. We include the client controller with its Backend for better understanding.

The ODL Shim Layer subscribes to events from the Service Abstraction Layer (SAL) which are passed using the NetIDE API to the client controller Backend. In turn, it receives messages from the Backend and generates events on the SAL which are consumed by the different Southbound Interface (SBI) modules installed in ODL. Currently, the assumption is that ODL has, at least, the OpenFlow v1.0 SBI module installed and activated.

Interactions with other ODL modules are still to be defined and depend on:

1. the architectural choice regarding conflict resolution within NetIDE
2. ODL projects like the recently created NEMO project [15] or the network intent project [16] and their success in the ODL community

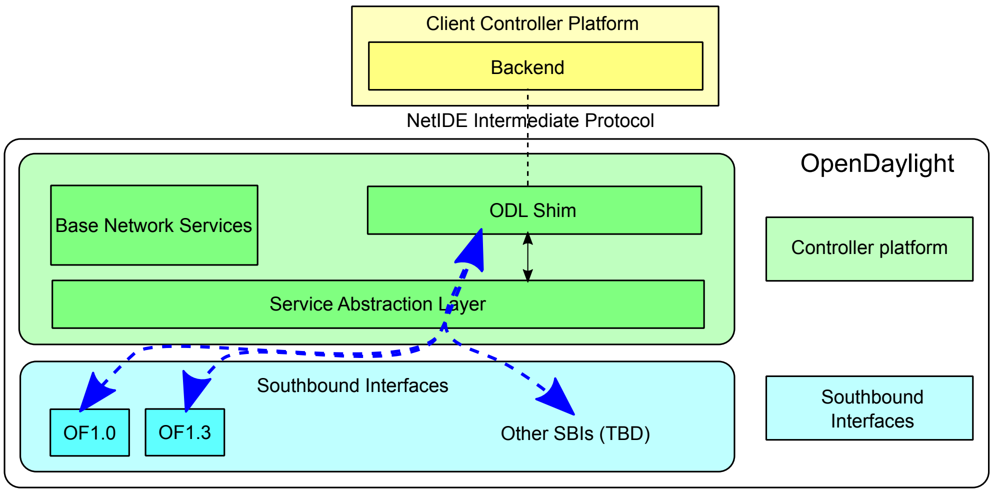

Figure 3.2: OpenDaylight as a server controller – locating the Shim Layer in the overall architecture

Internally, the ODL consists of two modules, the **BackendChannel** and the **ODLHandler**, as shown in Figure 3.3. The **BackendChannel** module implements the communication with the Backend in the Client Controller using the NetIDE API. It receives messages from the Backend and passes them to the **ODLHandler** and sends the messages generated by the **ODLHandler** to the Backend.

The **ODLHandler** interacts with the SAL. It receives SAL events generated by the Southbound Interfaces and translates them into NetIDE API messages. Additionally, it translates NetIDE API events received from the client controller to SAL events.

3.1.2 Installation and deployment

The current version of the OpenDaylight (ODL) Shim Layer is prepared to work with the Helium version of ODL [17], and particularly using the Karaf OSGi framework [18].

[1]New section in response to review requirement R#11

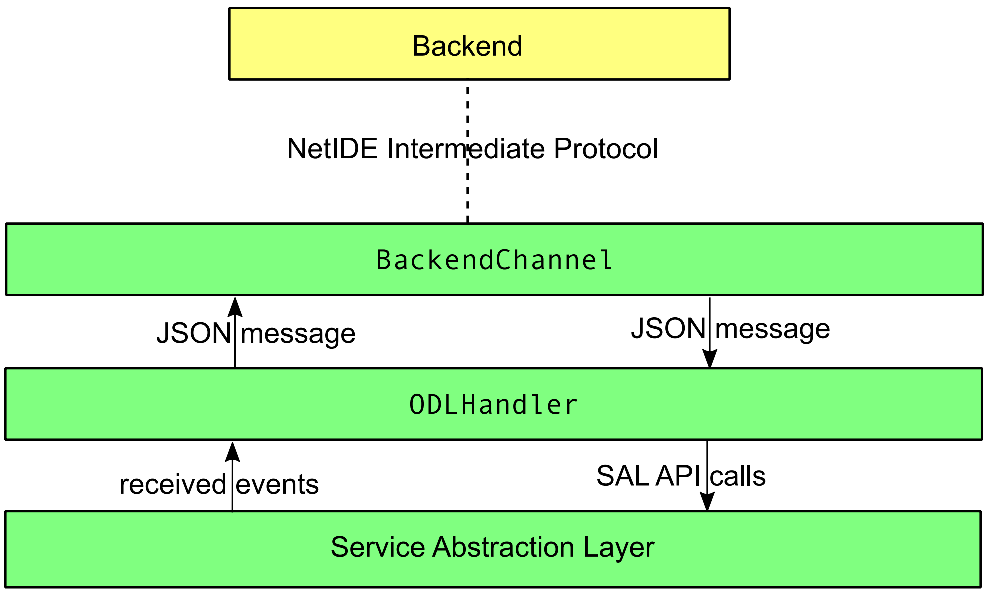

Figure 3.3: Architecture of the OpenDaylight Shim Layer

3.1.2.1 Getting OpenDaylight

First of all, we want to be able to use an ODL which already uses OpenFlow, and for this we will follow the second step from the following guide:

```
https://wiki.opendaylight.org/view/OpenDaylight_OpenFlow_Plugin::
Running_controller_with_the_new_OF_plugin
```

which allows the use of the OpenFlow plugin for versions 1.0 and 1.3. So clone the following repository for getting the OpenDaylight distribution:

```
git clone https://git.opendaylight.org/gerrit/p/openflowplugin.git
```

After that, go to the openflowplugin directory and run `git checkout stable/helium` and `mvn clean install`. If that raises any errors, just run it adding -DskipTests (i.e. `mvn clean install -DskipTests`). If that still raises any errors, run `mvn dependency:tree`, which hopefully will solve all the dependencies
At this point, you have your ODL Helium ready to be run. Now cd to openflowplugin/distribution/karaf/target/assembly/ and here launch Karaf by running `./karaf`.

Note: Each time you recompile the ODL bundle and generate the .jar, all the contents inside the data folder (in openflowplugin/distribution/karaf/target/assembly/) need to be removed. Otherwise, your bundle will automatically be installed inside karaf and you won't be able to see any of the changes.

3.1.2.2 Getting the ODL bundle and running ODL Shim Layer inside Karaf

Now that you have your Karaf distribution running, you will have to clone the odl-shim (read the instructions in the introduction of this Chapter). Enter the `odl-shim` folder and run `mvn clean install`. The .jar file should be in this path:

```
~/.m2/repository/org/opendaylight/openflowplugin/pyretic-odl/0.1.0-SNAPSHOT
```

In addition, it is inside the target folder that has just been created in Engine/odl-shim.
Go to the Karaf console (which you opened before, just after running `./bin/karaf`) and install the JSON bundle (which is a dependency that the ODL Shim Layer has) like this:

```
bundle:install -s mvn:com.googlecode.json-simple/json-simple/1.1.1
```

After that, you can install the ODL Shim Layer bundle just fine:

```
bundle:install -s mvn:org.opendaylight.openflowplugin/pyretic-odl/0.1.0-SNAPSHOT
```

You can avoid installing the JSON bundle if you copy .the jar:

```
.m2/repository/com/googlecode/json-simple/json-simple/1.1.1/json-simple-1.1.1.jar
```

and paste it into openflowplugin/distribution/karaf/target/assembly/deploy. You just have to do this once. (The .m2 refers to linux platforms. If you do not know where that is, find out where Maven creates it in your specific platform).

Note: Remember to perform the bundle:install of the ODL Shim Layer each time you launch Karaf.

Now, when you create a new topology in Mininet and ping between any of the nodes, you should be seeing things happening in the Karaf console.

3.2 The Ryu Shim Layer[2]

The Ryu Shim Layer is implemented as an application for the Ryu controller [2] (see Fig. 3.4) by using the OpenFlow libraries included in the Ryu's source code [19]. The Shim Layer is a single-threaded entity that interacts with the Ryu's internal OpenFlow controller to communicate with the underlying OpenFlow devices. Although Ryu provides libraries for many southbound protocols (such as OF-CONFIG, OVSDB, NETCONF, XFLOW, etc.) and for different versions of OpenFlow (up to v1.4), this release of Shim Layer is implemented specifically for OpenFlow v1.0. The main reason for this limitation is that the current Intermediate API based on Pyretic only supports this version of protocol. However, we plan to extend the Ryu Shim Layer along with an enhanced version of the Intermediate protocol which we are defining with multi southbound protocols support.

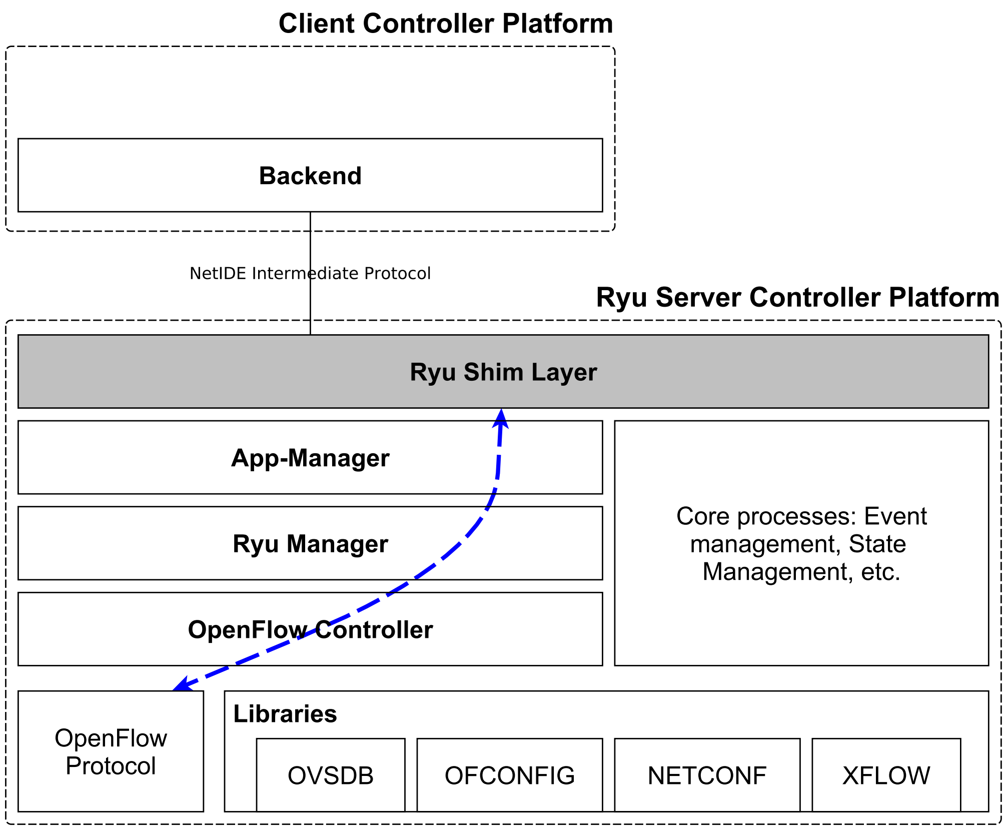
Client Controller Platform
Backend
NetIDE Intermediate Protocol
Ryu Server Controller Platform
Ryu Shim Layer
App-Manager
Ryu Manager
OpenFlow Controller
Core processes: Event management, State Management, etc.
OpenFlow Protocol
Libraries
OVSDB
OFCONFIG
NETCONF
XFLOW

Figure 3.4: The Shim Layer within the Ryu architecture.

Internally, the Ryu Shim Layer includes a set of handlers for the events generated by the devices (e.g. `packet_in, feature_reply, flow_stats, etc.`). On the other side, it implements the communications with the Client Controller Backend through the Intermediate protocol described in Section 2.1 and provides a set of methods for the translation between the OpenFlow messages to the Intermediate protocol messages and vice-versa.

[2]Improved section in response to review requirement R#11

3.2.1 Installation

To use this Shim Layer, first clone the Ryu code (from here [19]) on a local machine and install Ryu by following the procedure described in the README document that accompanies the code. After that, add the Ryu's code path to the PYTHONPATH variable in your `~/.profile` or `~/.bashrc` file (e.g. `export PYTHONPATH="$HOME/ryu"`).
Additional python packages may be required in order to succefully complete the installation procedure. On a Ubuntu 14.04 Linux OS the following must be installed:

- `apt-get install python-pip python-dev python-repoze.lru libxml2-dev libxslt1-dev zlib1g-dev`
- `pip install ecdsa`
- `pip install stevedore`
- `pip install greenlet`

Clone the ryu-shim (read the instructions in the introduction of this Chapter) and enter the `ryu-shim` folder.

3.2.2 Running

In the current implementation of the NetIDE API Interceptor, the Shim Layers are clients of the TCP connection while the Backends are the servers (listening by default on port 41414). Therefore, to test the Ryu Shim Layer, run the Backend first (see sections 4.1 and 4.2) and then start the Shim Layer with the following command:

```
ryu-manager ryu-shim.py
```

The Ryu Shim Layer listens for connections from the switches on the port 6633. Therefore, when using mininet for testing purposes, start mininet by specifying the controller information as follows:

```
sudo mn --custom netenv.py --topo netenv --controller remote,port=6633
```

3.3 The POX Shim Layer

The POX Shim Layer was originally developed as part of the Pyretic framework [9] (the so-called POX client). In order to facilitate the implementation of NetIDE API Interceptor, it has been slightly adapted to run in stand-alone mode (i.e. without the need of running the whole Pyretic's framework).
The POX Shim Layer is provided along with the Ryu Backend's code (see Section 4.1) for testing purposes as `pox-client.py`.

4 The Backend Layer

The Backend is the module that communicates with the Server controller through the Shim Layer (gray box in Fig. 4.1) and that allows the above applications to run within their native environment as there were no additional layers below (the NetIDE API Interceptor and the Server Platform). In particular, the Backend completes the handshake session with the controller (*hello* and *config* messages) by using the information received from the Shim Layer (see also Fig. 2.1). After the handshake session is completed, the Backend (like the Shim Layer) forwards the control messages (`packet_ins`, `flow_mods`, `packet_outs` etc.) coming from the Shim Layer to the applications and viceversa.
The Backend is launched along with the applications and the Client SDN platform and is in charge of listening for Transport Control Protocol (TCP) connection coming from the underlying Shim Layer.

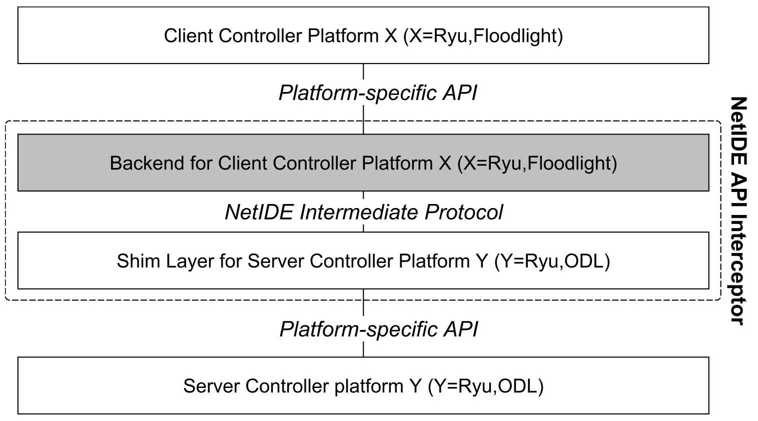
Client Controller Platform X (X=Ryu,Floodlight)
Platform-specific API
Backend for Client Controller Platform X (X=Ryu,Floodlight)
NetIDE Intermediate Protocol
Shim Layer for Server Controller Platform Y (Y=Ryu,ODL)
Platform-specific API
Server Controller platform Y (Y=Ryu,ODL)
NetIDE API Interceptor

Figure 4.1: The Backend.

The source code of the software modules presented in this Chapter and released with this Deliverable can be downloaded from the NetIDE Github repository (the links are listed in the Appendix A). Alternatively, to get the latest version of the code, clone the repository by using the following command:

```
git clone https://github.com/fp7-netide/Engine/
```

4.1 The Ryu Backend[1]

As shown in Fig. 4.2, the Ryu Backend has been implemented as an application for the Ryu platform and is be loaded at Ryu boot-time along with the user's applications. The Backend is basically a modified version of the Ryu's internal OpenFlow Controller. The Backend handles a TCP

[1]Improved section in response to review requirement R#11

connection with the Shim Layer through which it exchanges control messages with the switches. Like the internal OpenFlow Controller, the Backend creates one instance of the Datapath class (called BackendDatapath) for each switch connected. In case of the Backend, there is no direct connection with the switches, but it uses the Intermediate Protocol `switch join BEGIN/END` messages received during the handshake session (see Section 2) to instantiate the BackendDatapath objects, one for each switch.

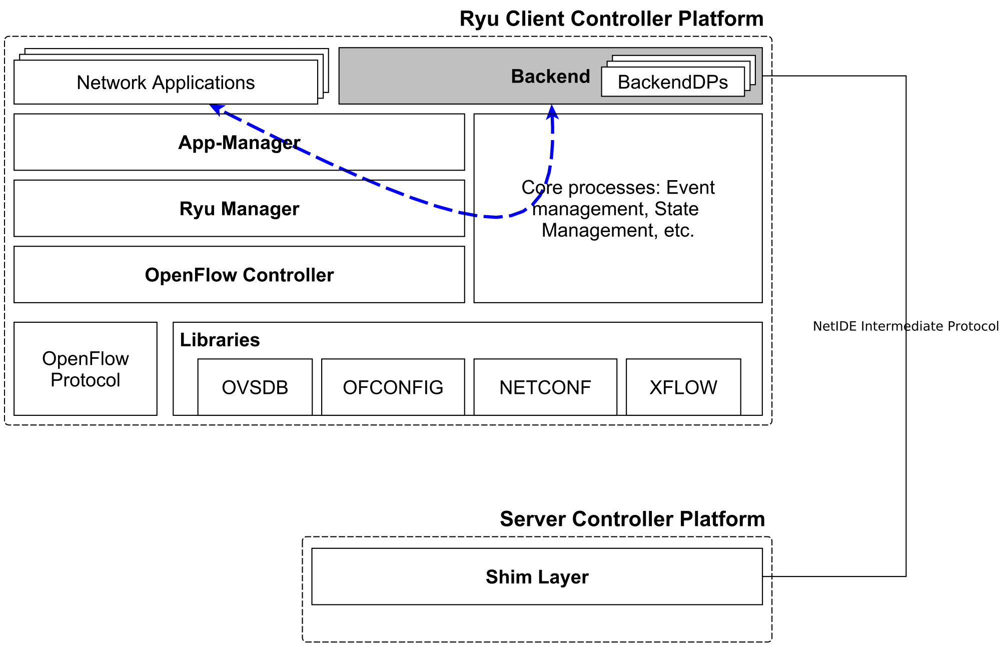
Ryu Client Controller Platform
Network Applications
Backend
BackendDPs
App-Manager
Ryu Manager
OpenFlow Controller
Core processes: Event management, State Management, etc.
NetIDE Intermediate Protocol
OpenFlow Protocol
Libraries
OVSDB
OFCONFIG
NETCONF
XFLOW
Server Controller Platform
Shim Layer

Figure 4.2: The Backend within the Ryu architecture.

In order to communicate with the user's applications, the Backend leverages on the Ryu's Event Management System that allows the BackendDatapath objects to send events such as `features_reply`, `packet_in`, etc. to all the applications (also called *observers*) which have implemented the *handlers* for those events.
When the Backend is running, the Ryu's internal OpenFlow controller remains inactive since no devices are connecting through the usual OpenFlow control channel. Therefore, the user's applications only detect and control the devices discovered through the Backend.

4.1.1 Installation

The Ryu Backend is provided as an additional module for the Ryu controller. In order to use it, download the Ryu's source code first (`git clone https://github.com/osrg/ryu.git`). After that, clone the `ryu-backend` code (read the instructions in the introduction of this Chapter) and copy the `backend` sub-folder into the `ryu/ryu` folder of the Ryu's code tree and add the Ryu's code path to the PYTHONPATH variable in your ∼`/.profile` or ∼`/.bashrc` (e.g. `export PYTHONPATH="$HOME/ryu"`).
Finally, install the Ryu controller by entering the ryu folder and by running the command:

```
python ./setup.py install
```

Additional python packages may be required in order to succefully complete the installation procedure. On a Ubuntu 14.04 Linux OS the following must be installed:

- `apt-get install python-pip python-dev python-repoze.lru libxml2-dev libxslt1-dev zlib1g-dev`
- `pip install ecdsa`
- `pip install stevedore`
- `pip install greenlet`

4.1.2 Running

From the Ryu's code folder, run the following command to use the Ryu Backend with the, e.g., `simple_switch` application on top of it:

```
ryu-manager --ofp-tcp-listen-port 7733 ryu/backend/backend.py ryu/app/simple_switch.py
```

While the Backend listens for connections on pothe TCP port 41414, we have also to prevent Ryu from binding the default OpenFlow TCP port 6633 (that the Backend does not use in any case) by using the option `--ofp-tcp-listen-port` in order to avoid conflicts with the underlying Server Controller platform which uses the TCP port 6633 to communicate with the switches.
Finally start the Shim Layer (as described in Chapter 3).

4.2 The Floodlight Backend[2]

The NetIDE Backend is loaded as a Floodlight Module when Floodlight starts up, implementing the `IFloodlight` interface. By accessing the native Floodlight application interface, this gives the module access to events that are not available from the RESTful interface that standard SDN Applications consume.

By default, Floodlight listens on port `6633` for all connecting OpenFlow switches. By changing this to a different port `6634`, no physical/virtual switch will now connect to Floodlight. For each Switch connected event that is received by the NetIDE module from the Shim Layer, a new *Dummy Switch* class is created that connects to Floodlight on this separate port, `6634`. The *Dummy Switch* will replay and respond to OpenFlow messages (`hello`, `echo_reply`, `config_reply`, `stats_reply`, etc). Floodlight will now add this switch to its list of Managed Switches and trigger the `addedSwitch` event to other `IOFSwitchListener` listeners. This is where SDN applications developers traditionally push `flow_mod` messages proactively to the switch.

As Network Applications and other Floodlight modules send OpenFlow messages to switches, each Dummy Switch will now receive this message. After inspecting the message, it then creates the relevant *NetIDE Intermediate Protocol* message and sends it to the Shim Layer of the Server SDN Controller. When the response is received from the Shim Layer, the equivalent OpenFlow Reply message is created and the relevant properties set. The message is then sent back to Floodlight through the correct Dummy Switch. Similarly, all switch initiated messages (eg, `packet_in`) are sent by the Shim Layer to the NetIDE Module of Floodlight, which in turn creates the relevant OpenFlow message and sends it out through the correct Dummy Switch.

[2]Improved section in response to review requirement R#11

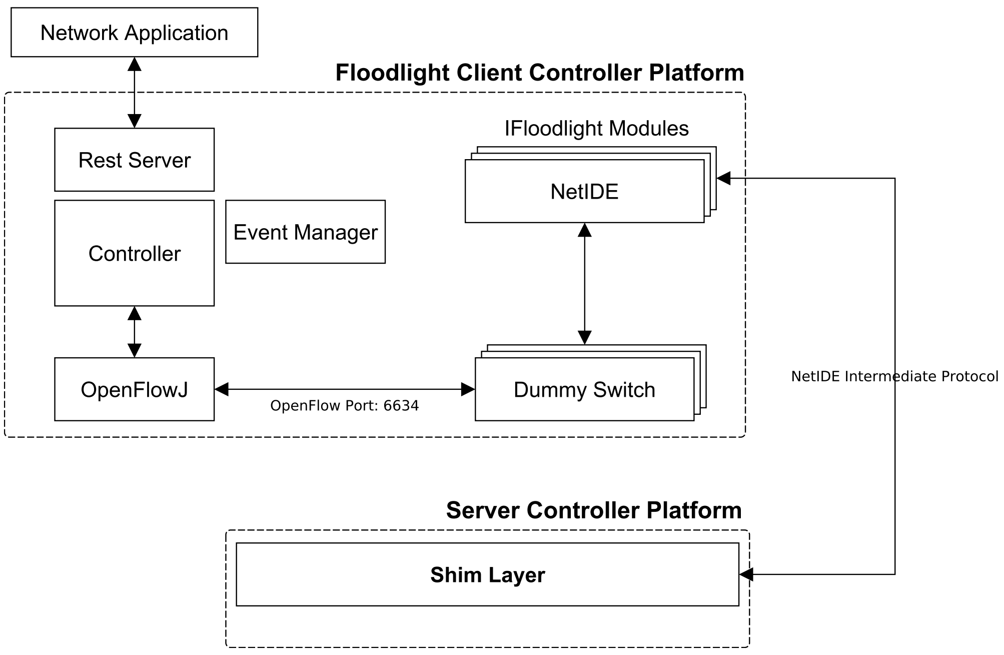

Figure 4.3: The Floodlight Client Controller Backend

Note: Floodlight tracks a channel per connected switch, so it is important that each response from the Shim Layer is relayed through the correct Dummy Switch's connection. This is tracked through the Datapath Identifier (DPID) and provides uniqueness.

It is possible to view the list of switches that Floodlight is managing by using the Floodlight Management Web interface available at http://localhost:8080/. What is important is that the list of switches and topology being managed are abstracted from Floodlight through the NetIDE intermediate layer. These switches are actually physically connected to the Server SDN Controller.

4.2.1 Importing the code into Floodlight

This Section helps the user to run the Floodlight Backend module (`fl-odl` module in the NetIDE Github repository) that connects to the underlying Shim Layer.

When developing a new module for Floodlight, you must download the Floodlight source code first and add your module as a new namespace.
In particular, follow these steps:

- **Setup Eclipse**: download Floodlight (version 0.90) from their Github repo [20] and import the project into Eclipse.
- **Import NetIDE code**: copy the net.floodlightcontroller.interceptor namespace (and *.java files) from the `fl-odl` folder into the Floodlight project and copy the java classes (ElementsConfigurator.java and IElementsConfiguratorService.java) in the new folder.
- **Edit config files**: modify the Floodlight properties files:
  - /src/main/resources/META-INF/services/net.floodlightcontroller.core.module.IFloodlightModule: add the line:

`net.floodlightcontroller.interceptor.BackendChannel`
at the end of Module section, before the port setting

– /src/main/resources/floodlightdefault.properties: add the line:
  `net.floodlightcontroller.interceptor`
  at the end of the file.

4.2.2 Compilation and running

Floodlight uses *ant* as its build tool, you can build by issuing the ant command within a shell/-command window. A Project Object Model (POM) file is provided so that Maven can be used for compiling the code:

```
mvn clean install
```

Run the Backend by launching Floodlight with the following command:

```
jar - java -jar /target/floodlight.jar
```

5 Conclusions and Future Work

In this Deliverable we have presented the implementation details of the current version of the NetIDE Network Engine and of its core module called NetIDE API Interceptor.
The NetIDE API Interceptor represents the glue between the SDN applications written for different SDN frameworks (currently Ryu and Floodlight) and the Server SDN Platform (currently OpenDaylight and Ryu) that is directly connected to the physical network. The API Interceptor can be divided in two different layers called *Backend* and *Shim Layer* that communicate through the so-called Intermediate API (see Chapter 2).
In order to speed up the development process, in this first implementation of the API Interceptor, Backend and Shim Layer communicate by using the same API proposed by the Pyretic team for the interaction between the runtime system and the OpenFlow clients of the Pyretic framework.

5.1 Future Work

In the next releases of the Network Engine we will provide an enhanced version of this API by overcoming some of the current limitations. In particular, only the version 1.0 of the OpenFlow protocol is supported (this means that neither the applications nor the Server Platform can use v1.1 or higher). Moreover, even the support of many important features of OF v1.0 are missing, such as idle and hard timeouts, buffer identifiers, etc.
Beside the enhancement of the Intermediate Protocol and related API, the short-term plan includes the integration of several additional tools into the API Interceptor such as the debugger, the profiler, the resource manager and others.
Another aspect that we will consider in the future activities is the estimation of the latency overhead introduced on the control channel by the Network Engine and, in particular, by the different implementations of the Backend and Shim Layer. Understanding the performance of the Network Engine is particularly important as the control plane must be responsive to flow-level changes in order to avoid reducing responsiveness within the network (especially in production environments)[1].

5.2 Roadmap[2]

A Roadmap of the NetIDE App Engine for the next two years of the project is the following:

Tools: The set of debugging tools is composed of: Logger, Debugger, Resource Manager, Garbage Collector, Simulator, Model Checker and Profiler. These tools are going to be connected to the API Interceptor. A initial pub/sub mechanism for some of the tools have been investigated. The 1st release of the Tools reported in the D4.2 includes a first version of the Logger that uses this mechanism for communicating the Logger and the Shim Layer. Next steps are the study of the requirements of the rest of the tools and the definition and implementation of the API between the

[1]In response to review comment C#5
[2]In response to review requirement R#12

tools and the API Interceptor. On M24 there will be an intermediate release of the tools with the definition of this API. On M36 there will be a full definition of this API jointly with the full release of all the tools.

Intermediate Protocol: The first release of the Engine included the Pyretic protocol as initial protocol for communicating the Backend and the Shim. This Intermediate Protocol needs to be extended to manage the current Pyretic API limitations. A new Intermediate Protocol definition has been provided at M15 and initial Ryu-based prototypes of both Backend and Shim Layer supporting the new protocol will be provided by M20. These prototypes will help the implementation of the new Intermediate Protocol for the other platforms supported by the Network Engine.

Server Controller platform: The initial version of the Network Engine considered OpenDaylight and Ryu as the Server Controllers. Next version of the Engine will provide the integration of the ONOS platform as a Server Controller platform. A preliminary study and prototype of this approach will be delivered by M24 and a final version by M36.

Client Controller Platform: The first release of the Network Engine included a single Client Controller scenario. Conflict resolution mechanism will be investigated and a preliminary scenario with several Client Controllers coordinated by the conflict resolution mechanism will be provided by M24. An enriched version of the conflict resolution mechanism will be provided by M36.

Shim Layer: The Shim layer has been implemented for two different platforms: OpenDaylight and Ryu. As anticipated above, a ONOS implementation of the Shim Layer will be released by M24 along with new versions of the OpenDaylight and Ryu Shim Layers supporting the new Intermediate Protocol. A full implementation of the Shim Layer functionality for the three cases will be provided in the next release by M30. A full performance evaluation of the Shim layer will be included in the final results by M36.

Backend: The backend has been implemented in the first release of the Network Engine for Ryu and Floodlight. A new version of both Backend implementations supporting the new Intermediate Protocol will be provided in the next release by M24. A full performance evaluation of the Backend will be included in the final results by M36.

A Source code repositories

Stable releases of the NetIDE Network Engine can be downloaded from the ”Releases” section of the NetIDE Github repository:

```
https://github.com/fp7-netide/Engine/releases
```

In particular, the tag `0.1` has been used to identify the code released along with this deliverable D4.1.

Alternatively, to get the latest version of the code, clone the ”Engine” repository by using the following command:

```
git clone https://github.com/fp7-netide/Engine/
```

The repository is organized as follows:

fl-odl : The Floodlight Backend

odl-shim : The OpenDaylight Shim Layer

ryu-backend : The Ryu Backend (with the POX Shim Layer in the ”tests” folder)

ryu-shim : The Ryu Shim Layer

Bibliography

[1] WineHQ: Run Windows applications on Linux, BSD, Solaris and Mac OS X. `https://www.winehq.org/`.

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

[3] Project Floodlight. `http://www.projectfloodlight.org/floodlight/`.

[4] OpenDaylight - A Linux Foundation Collaborative Project. `http://www.opendaylight.org`.

[5] The NetIDE consortium. D2.2v2 - NetIDE Architecture Redefinition: the NetIDE Runtime Approach. Technical report, The European Commission, 2015.

[6] Open Networking Foundation.

[7] The NetIDE consortium. D5.2 - Use Case Design: 1st release. Technical report, The European Commission, 2014.

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

[9] The Pyretic framework. `http://frenetic-lang.org/pyretic/`.

[10] Wine architecture. `https://www.winehq.org/docs/winedev-guide/x2884/`.

[11] Xin Jin, Jennifer Gossels, Jennifer Rexford, and David Walker. Covisor: A compositional hypervisor for software-defined networks. In *12th USENIX Symposium on Networked Systems Design and Implementation, NSDI 15, Oakland, CA, USA, May 4-6, 2015*, pages 87–101, 2015.

[12] Jeffrey C. Mogul, Alvin AuYoung, Sujata Banerjee, Lucian Popa, Jeongkeun Lee, Jayaram Mudigonda, Puneet Sharma, and Yoshio Turner. Corybantic: towards the modular composition of SDN control programs. In *Twelfth ACM Workshop on Hot Topics in Networks, HotNets-XII, College Park, MD, USA, November 21-22, 2013*, page 1, 2013.

[13] The NetIDE consortium. D5.1 - Use case requirements. Technical report, The European Commission, 2014.

[14] T. Bray. The JavaScript Object Notation (JSON) Data Interchange Format. RFC 7159 (Proposed Standard), March 2014.

[15] NEMO: A Language Style NBI . `https://wiki.opendaylight.org/view/Project_Proposals:NEMO`, may 2015. Accepted 8 May 2015.

[16] David Bainbridge and Raphael Amorim. Network Intent Composition. `https://wiki.opendaylight.org/view/Network_Intent_Composition:Main`, jan 2015.

[17] OpenDaylight Helium. `http://www.opendaylight.org/software/downloads/helium`.

[18] Apache Karaf. `http://karaf.apache.org/`.

[19] Ryu source code. `https://github.com/osrg/ryu`.

[20] Floodlight source code. `https://github.com/floodlight/floodlight`.