## Document Properties

| | |
|---|---|
| Document Number: | D 2.1 |
| Document Title: | **NetIDE Core concepts and Architecture v1** |
| Document Responsible: | Pedro A. Aranda (TID), Diego López (TID) |
| Document Editor: | Pedro A. Aranda (TID), Diego López (TID) |
| Authors: | Pedro A. Aranda (TID), Diego López (TID), Holger Karl (UPB), Roberto Doriguzzi (CreateNet), Georgios Katsikas (IMDEA), Carmen Guerrero (IMDEA), Ladislav Lhotka (CZ.NIC), Bernhard Schrder (FTS), Elisa Rojas (IMDEA) |
| Target Dissemination Level: | PU |
| Status of the Document: | Final |
| Version: | 1.0 |

## Production Properties:

| | |
|---|---|
| Reviewers: | Elio Salvadori (CreateNet), Christian Stritzke(UPB), Kevin Phemius(Thales) |

## Document History:

## Disclaimer:

**Abstract:**

This deliverable defines the core technical concepts of NetIDE project, collects requirements, analyses them and provides blueprints of the architecture that will be the reference for RTD and integration activities in the project.
We explore the state of the art and provide a first draft of the NetIDE architecture. This architecture v.0 is based on the PoC implementations of the Data-Centre use case, UC1, sketched in Deliverable D 5.1, "Use Case requirements". These implementations use off-the-shelf components and help us identify missing feature needed for the NetIDE framework. The architecture postulates the use of an external domain-specific language (DSL) as a way of mixing and matching Network Applications (Network Apps) developed with different programming languages and paradigms to different SDN controllers.
We propose to translate the Network Applications into this domain-specific language (DSL) and feed them into an intermediate translator that generates the code for the different SDN controllers we target. We have derived this concept from an initial implementation of one of the use cases described in the Description of Work (DoW) using state-of-the art environments like the Pyretic framework, the POX and Ryu OpenFlow controllers and YANG as a means to describe network equipment configurations.
Finally, we describe the next steps in our way to the NetIDE architecture. Along with projects we had planned to follow, we have enlarged the scopes of our plans with regards to OpenDayLight and included architectural work of the Open Networking Foundation as a result of the first technical review of the project, which was held in Bologna at the end of June 2014.
The emerging of the OpenDayLight controller platform has had an impact on the SDN landscape and on our project in particular. We have incorporated ODL into our next steps and will explore how far the project can leverage on this platform when describing its architecture. Additionally, there are other efforts in the Open Networking Foundation - more precisely the North-bound Interface in the architecture group - we will start following.

**Keywords:**

software defined networks, networking, future internet, openflow, integrated development environment

# Contents

# List of Figures

# Acronyms

| | |
|---|---|
| **AD-SAL** | Application-Driven Service Abstraction Layer (SAL) |
| **API** | Application Programming Interface |
| **ARP** | Address Resolution Protocol |
| **DC** | Data-Centre |
| **DMZ** | De-Militarised Zone |
| **DNS** | Domain Name Service |
| **DDoS** | Distributed Denial of Service |
| **DoW** | Description of Work |
| **DSL** | domain-specific language |
| **ECA** | Event-Condition-Action |
| **EPL** | Eclipse Public License |
| **FRP** | functional reactive programming |
| **FW** | Firewall |
| **HW** | Hardware |
| **IaaS** | Infrastructure as a Service |
| **IDE** | Integrated Development Environment |
| **IP** | Internet Protocol |
| **IRF** | Intermediate Representation Format |
| **LLDP** | Link Layer Discovery Protocol |
| **MAC** | Medium Access Control |
| **MD-SAL** | Model-Driven SAL |
| **NAT** | Network Address Translation |
| **NBI** | North-bound Interface |
| **NFV** | Network Function Virtualisation |
| **ODL** | OpenDayLight |
| **OF** | OpenFlow |

| | |
|---|---|
| **OFSID** | OpenFlow Software Implementation Distribution |
| **ONF** | Open Networking Foundation |
| **OS** | Operating System |
| **OSGi** | Open Services Gateway Initiative |
| **OVS** | Open Virtual Switch |
| **PaaS** | Platform as a Service |
| **PoC** | Proof-of-Concept |
| **RTT** | Round Trip Time |
| **SAL** | Service Abstraction Layer |
| **SDN** | Software Defined Networking |
| **SDNw** | Software Defined Network |
| **SLA** | Service Level Agreement |
| **SotA** | State-of-the-Art |
| **SW** | Software |
| **TCP** | Transmission Control Protocol |
| **UC** | use case |
| **UDP** | User Datagram Protocol |
| **URI** | Universal Resource Locator |
| **VM** | Virtual Machine |
| **VRRP** | Virtual Router Redundancy Protocol |
| **WP** | work-package |

# List of Corrections

# Executive Summary

This deliverable defines the core technical concepts of NetIDE project, collects requirements, analyses them and provides blueprints of the architecture that will be the reference for RTD and integration activities in the project.

We explore the state of the art and provide a first draft of the NetIDE architecture. This architecture v.0 is based on the Proof-of-Concept (PoC) implementations of the Data-Centre use case, UC1, sketched in Deliverable D 5.1, "Use Case requirements". These implementations use off-the-shelf components and help us identify missing feature needed for the NetIDE framework. The architecture postulates the use of an external domain-specific language (DSL) as a way of mixing and matching Network Apps developed with different programming languages and paradigms to different Software Defined Networking (SDN) controllers.

We propose to translate the Network Applications into this domain-specific language (DSL) and feed them into an intermediate translator that generates the code for the different SDN controllers we target. We have derived this concept from an initial implementation of one of the use cases described in the Description of Work (DoW) using state-of-the art environments like the Pyretic framework, the POX and Ryu OpenFlow controllers and YANG as a means to describe network equipment configurations.

Finally, we describe the next steps in our way to the NetIDE architecture. Along with projects we had planned to follow, we have enlarged the scopes of our plans with regards to OpenDayLight and included architectural work of the Open Networking Foundation as a result of the first technical review of the project, which was held in Bologna at the end of June 2014.

The emerging of the OpenDayLight controller platform has had an impact on the SDN landscape and on our project in particular. We have incorporated OpenDayLight (ODL) into our next steps and will explore how far the project can leverage on this platform when describing its architecture. Additionally, there are other efforts in the Open Networking Foundation - more precisely the Northbound Interface in the architecture group - we will start following.

We also provide a view into the next steps to create the NetIDE architecture and a long-term view of the Intermediate Representation Format as a way of representing Network Applications in a tool-friendly way.

This deliverable is tightly coupled with Deliverable D 3.1,"Developer Toolkit Specification". The outcome of the architecture work has been fed into the concept for the Integrated Development Environment that will be used by the project.

The present version of this deliverable includes suggestions and comments resulting from the First Technical Review that took place in Bologna in the 23rd of June, 2013.

# 1 Introduction

NetIDE has set out to provide an Integrated Development Environment (IDE) for SDN development that unifies different controller technologies and allows a developer to write applications that are independent from the underlying Software Defined Networking (SDN) technology. To this avail, NetIDE proposes to use an Intermediate Representation Format (IRF) to store SDN applications that is handled by an IDE and transformed into platform-specific code.

During the lapse of time between proposal submission and project start, important advances in the State-of-the-Art (SotA) have occurred. The introduction of the ODL controller has stirred the SDN landscape with its possibility of integrating proprietary SDN network equipment with OpenFlow (OF) equipment under one controller. However, OpenDayLight is still in its early phases and some unknowns regarding the real level of platform independence persist.

To start both the conceptual and the development work, we have resorted to Pyretic. This higher-level construct, based on the POX controller, provides modular and compositional applications based on high level abstractions of the underlying network infrastructure and a well-defined north-bound interface for SDN applications. However, the original Pyretic distribution is not platform-independent. Therefore, one of the efforts in the project boot phase has been to detach the Pyretic infrastructure from POX (the OF controller it talks to) and make it interact with another OF controller (in our case, Ryu).

Additionally, we have explored whether we can apply state-of-the-art network configuration protocols (in our case YANG) in the framework of NetIDE.

This document is tightly coupled with Deliverable D 3.1 [1]. Both are released at the same time and extensive cross-referencing will be used, in order to keep the documents at a reasonable size.

The rest of the document is structured as follows:

- Chapter 2 briefly presents the state of the art,
- Chapter 3 describes initial steps to identify the architecture components
- Chapter 4 discusses a proof-of-concept implementation for a restricted architecture
- Chapter 5 outlines next steps towards the NetIDE architecture.

# 2 State of the Art

Software Defined Networking implies a paradigm shift in the way networks are designed and operated. They are the result of a general trend, where Hardware (HW) functionality is being replaced by Software (SW) implementations. This evolution has been fostered not only by a continuous increase in computing power and competitiveness of SW based solutions, but also by an evolution in the SW development world towards conceptually sound procedures and easy to use tools. In this chapter, we analyse the state of the art of SDN environments. We examine different controllers for OF based and proprietary SDN environments.

This deliverable concentrates on a selected subset of controllers, namely i) POX, ii) Ryu, iii) Floodlight, iv) Beacon, and v) OpenDayLight. For a high level analysis of a more broad range of open source SDN controllers available today including details on the northbound/southbound interfaces available and documentation, please refer to Deliverable D 6.1 [2].

## 2.1 Open SDN environments based on OpenFlow

A basic OF environment requires some essential components in both data and control planes. A forwarding node is the basic building element of the OF network, it may be a commercial HW-based node or a SW-based implementation. On the other hand, the control plane consists of OF controllers and higher-level language constructs that ease the development of Network Apps for a specific framework.

### 2.1.1 OF switches, controller frameworks and controllers

Many open source projects have emerged to help building OpenFlow networks, both controllers and switches are freely available under different licenses. One of the most popular SW-based OF switch implementation is the Open Virtual Switch (OVS) [3] which is integrated in the Linux kernel 3.3, firmware such as Pica8 [4] and Indigo [5] are also available.

In addition, there are also many open source OF **controllers** and **controller frameworks** for almost every development environment. NOX [6] written in C, Pox [7] in Python, floodlight [8] in Java and Trema [9] for Ruby developers are some of the available controllers. Most of these controller frameworks are part of more or less ambitious, wider-ranging OF development frameworks.

**We emphasize the difference between the controller framework and the controller itself.** For the purpose of the ensuing architecture discussion, this difference is crucial and we will adopt the following terminology. The controller is the actually running piece of software in an operational SDN network. It understands and processes the SDN control protocol (e.g., OpenFlow) and is customized to the needs of the particular network. But since implementing a controller from scratch encompasses considerable amounts of repetitive work, *controller frameworks* have emerged that collect together common basic functionality and that can be customized by *network applications* running in conjunction with the controller (the actual implementations differ widely, ranging from binding libraries together, plugins into a process, or separate processes, but that is not a crucial aspect).

The remaining text in this section gives a brief overview of popular controller frameworks.

### C/C++ OpenFlow controller frameworks

#### NOX

NOX [6] was the first reference implementation of an OF controller. This implementation, referred to as NOX Classic, was programmed in Python and C++ by Nicira Networks side-by-side with OpenFlow. The current implementation is C++ only and, as stated in the NOX website, aims at implementing a C++ Application Programming Interface (API) for OF 1.0 that provides fast, asynchronous I/O. It includes sample components that implement topology discovery, a learning switch and a network-wide switch that is distributed over all the OF switches controlled by an instance of NOX.

#### Trema

Trema [10] is an OF controller framework. The OF controller accepts applications written in C and Ruby as well. In addition to the OF controller, Trema is shipped with an integrated OF network emulator and OF traffic analysis tools based on Wireshark [11]. Trema also provides a test framework (Ruby). The integration of network emulation and test framework using Ruby enables developers to apply “well-known” testing tecnhiques such as mocks, stubs and expectations to OF programming. There is a branch in the Trema repository to support OF 1.3.

Trema is the base for the *ProgrammableFlow* [12] networking framework provided by NEC.

### Python-based OpenFlow controller frameworks

#### POX

POX [7] is an evolution of the Python OF interface that has been dropped from NOX and that has been a very fast and convenient entry path to OF networking and SDN in general. At its core, POX is a platform for the rapid development and prototyping of network control software using Python. In includes a number sample components: l2-learning, l3-learning, pretty-log, etc. to help new users in the development of their applications. It runs on multiple platforms including Linux, Mac OS, and Windows, and supports the same GUI and visualisation tools as NOX. Benchmarks shown in the website confirm that POX provides an improvement over NOX applications written in Python. Nowadays it supports OF version 1.0 and the Nicira OF extensions. Although lots of OF 1.3 feautures are derived from the Nicira extensions, OF v1.3 is not supported.

#### RYU

Ryu [13] is a component-based SDN framework written in Python. Ryu developers described it as a *Network Operating System*. It supports OF version 1.0, 1.1, 1.2, 1.3 and 1.4, as well as the Nicira extensions and the standard OF configuration protocol OF-Config [14]. Ryu can also control and monitor legacy platforms through the support of configuration protocols such as: Netconf [15], NetFlow [16], SNMP [17] and OvSDB [18].

One of the components of the Ryu framework is the *Packet Library*, that is used to parse and build packets for several protocols in the same way the Python-based *dpkt* [19] library does. Ryu’s *Packet Library* includes the implementation of stacked packet formats that allows it supporting standards like VLAN, MPLS, GRE etc. This library provides the framework with the possibility of interacting with network signalling and routing protocols. In this sense, it implements a speaker for the BGP-4 routing protocol [20] and the documentation explains how to implement an application to respond to ICMP packets. Additionally, the Ryu platform includes integrated test applications for OFconfig support and Virtual Router Redundancy Protocol (VRRP) [21] configurations, which are well documented.

Another feature of Ryu is that it can run in virtualisation environments based on cloud orchestrators like the OpenStack framework [22]. Ryu cooperates with OpenStack by means of the Quantum Ryu plugin which is available in the official Quantum releases.

The northbound interfaces exposed by Ryu comprise a RESTful management API and a REST API for OpenStack. Additionally, it gives the developer the possibility of detaching his/her application from the core of Ryu by defining specific application-defined REST APIs.

Ryu is licensed under the Apache 2.0 license.

### Java-based OpenFlow controller frameworks

#### Beacon

Beacon [23] is an OpenFlow controller implementation written in Java that that uses an implementation of the OSGi specification, Equinox. OSGi gives Beacon the capability to not only start and stop applications while it is running, but to also add and remove them (runtime modularity), without shutting down the Beacon process.

Beacon includes the OpenFlowJ library for working with OpenFlow messages. OpenFlowJ is an object-oriented Java implementation of the OpenFlow 1.0 specification.

#### Floodlight

FloodLight [8] is a fork of Beacon. Like its "parent" it supports Java and Python modules but also added Jython support. It allows synchronous and asynchronous messages to the switch and OF messages are converted in Java events, making it simple to assimilate. Floodlight was redesigned without the Open Services Gateway Initiative (OSGi) framework making it easier to develop without OSGi experience. Floodlight is multi-threaded allowing it to have great speed compared to other, mono-threaded controller (like NOX).

The choice of Java as its primary language was justified by three main reasons :

- Java is a very popular programming language.
- There are many IDEs and tools to develop in Java.
- Java is multi-platform, making it easier to export code from any kind of system (Beacon has been deployed on Android and Floodlight on iOS).

Floodlight is actively developed and have the support of Big Switch Networks' engineers who are actively testing and fixing bugs and building additional tools, plugins, and features. The Open Source community is well developed and many plug-ins and fixes were added by external developers in the project's repository. It can easily be modified and is able to supports any size of network with consistent performances. Major improvement can be possible by the use of a more powerful and optimised hardware.

As many OF Controllers, Floodlight has a modular architecture : a Core module (responsible for implementing the OF protocol), associated with "main" modules (which handle the basics such as link and host discovery) and the applications either above (using the modules REST APIs) or embedded in the controller (using Java interfaces).

The latest stable version of Floodlight only support OF up to the 1.1 protocol version. A beta version supporting OF 1.3 is also available but without full support of the capabilities of this version (e.g. rate limiter, meter tables, . . . ).

### OpenDaylight

OpenDayLight (ODL) is an open platform for the development and execution of general purpose SDN network applications and telecommunication-oriented Network Function Virtualisation (NFV) applications. The founding of OpenDayLight was announced in April 2013, after the initial conception of NetIDE. Since then, the ODL architecture and implementation have grown rapidly.

### OpenDaylight: Open Source Programmable Networking Platform

The OpenDayLight consortium describes ODL as follows: *OpenDayLight is an open platform for network programmability to enable SDN and NFV for networks at any size and scale. Enterprises, service providers, equipment providers and academia can download Hydrogen today and begin to evaluate, commercialize and deploy SDN and NFV.*

*OpenDayLight software is a combination of components including a fully pluggable controller, interfaces, protocol plug-ins and applications. The Northbound (programmatic) and Southbound (implementation) interfaces are clearly defined and documented APIs. This combination allows vendors and customers alike the ability to utilize a standards-based and widely supported platform without compromising technical creativity and solution innovation. With this common platform both customers and vendors can innovate and collaborate in order to commercialize SDN- and NFV-based solutions.* [1]

OpenDayLight is supported by leading server and networking industry players (39 members). Cisco is a major contributor which leads to recurring questions about the neutrality of ODL and a potentially hidden agenda.

Unlike other SDN controller projects the ODL scope is more than just the controller framework and also includes:

- SDN applications running on top of the controller. SDN applications implement network services such as firewalls or load balancing. Business applications build on top of these network services,
- Integration with other higher level frameworks such as OpenStack which enables cloud application development, and
- A Service Abstraction Layer (SAL) for multiple lower level frameworks supporting many network equipment programming models.

Figure 2.1 [2] provides an overview of the current scope and architecture of ODL. The analysis presented in this deliverable refers to ODL release “Hydrogen” (February 2014, provided under Eclipse Public License (EPL))[3]. It provides three flavours dedicated to different areas of use cases. These flavours contain only some of the modules shown in the whole architecture in Figure 2.1:

- **Base edition:** Basically the controller core with the standard fundamental features such as topology management, switch manager and Internet address resolution. This is the feature set which is also typically provided by the other SDN controllers.
- **Virtualization edition:** This edition additionally contains SDN applications implementing network overlays (DOVE), virtual networks (VTN, Virtual Tenant Network) and integration into OpenStack. This edition contains many features required to implement a virtualized data centre and to provide cloud services.

[1] source: `http://opendaylight.org`
[2] source: `http://www.opendaylight.org/project/technical-overview`
[3] As of the 29th of September, 2014, the OpenDayLight project released a second version, nicknamed “Helium”

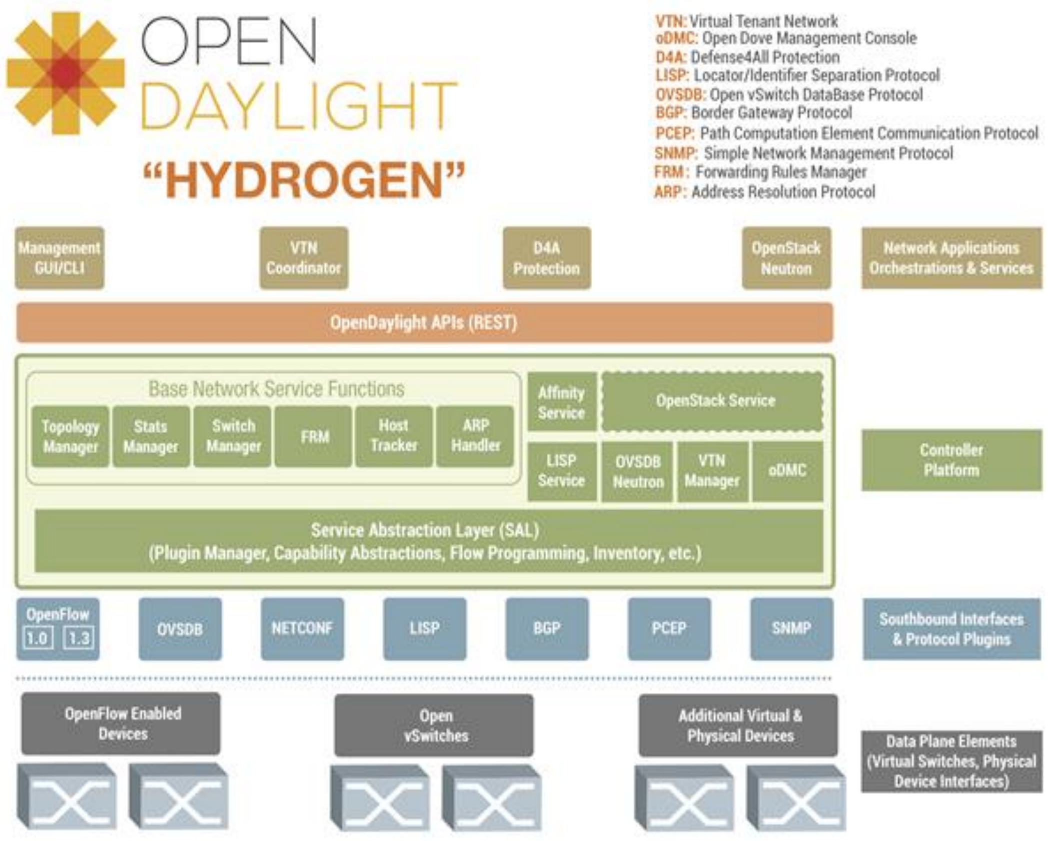

OPEN DAYLIGHT
"HYDROGEN"
VTN: Virtual Tenant Network
oDMC: Open Dove Management Console
D4A: Defense4All Protection
LISP: Locator/Identifier Separation Protocol
OVSDB: Open vSwitch DataBase Protocol
BGP: Border Gateway Protocol
PCEP: Path Computation Element Communication Protocol
SNMP: Simple Network Management Protocol
FRM: Forwarding Rules Manager
ARP: Address Resolution Protocol
Management GUI/CLI
VTN Coordinator
D4A Protection
OpenStack Neutron
Network Applications Orchestrations & Services
OpenDaylight APIs (REST)
Base Network Service Functions
Topology Manager
Stats Manager
Switch Manager
FRM
Host Tracker
ARP Handler
Affinity Service
OpenStack Service
LISP Service
OVSDB Neutron
VTN Manager
oDMC
Controller Platform
Service Abstraction Layer (SAL)
(Plugin Manager, Capability Abstractions, Flow Programming, Inventory, etc.)
OpenFlow 1.0 1.3
OVSDB
NETCONF
LISP
BGP
PCEP
SNMP
Southbound Interfaces & Protocol Plugins
OpenFlow Enabled Devices
Open vSwitches
Additional Virtual & Physical Devices
Data Plane Elements (Virtual Switches, Physical Device Interfaces)

Figure 2.1: An overview of the architecture of ODL rel. "Hydrogen"

- **Service provider edition:** This edition contains services such as the Affinity Metadata Service which defines workload relationships and their Service Level Agreement (SLA) requirements. This edition introduces a higher level of network programming by focusing on the needs of network service provider applications and their connectivity requirements. It supports high-level routing protocols such as BGP and LISP. It also addresses security requirements such as the detection and mitigation of Distributed Denial of Service (DDoS) attacks.

OpenDaylight follows the following architectural principles [4]:

1. **Runtime Modularity and Extensibility** supporting installation, removal and update of service implementations within a running controller, also known as "In-Service Software Upgrade" (ISSU)
2. **Multiprotocol Southbound**: Allow for more than one southbound protocol interface from the controller to support network elements with diverse capabilities. Examples of such protocols are OpenFlow, Netconf, BGP and LISP.
3. **Service Abstraction Layer**: Utilizing the OSGi framework allowing multiple southbound protocols to present the same northbound service interfaces.
4. **Open Extensible Northbound API**: To allow for an extensible set of application-facing

[4] based on `https://wiki.opendaylight.org/view/OpenDaylight_Controller:Architectural_Principles#Overview`

APIs both across runtimes via REST (level 3 API) and within the same runtime, e.g. via function calls (level 2 API). The set of accessible functions should be the same.

5. **Support for Multi-tenancy/Slicing**: This includes allowing the controller to present different views of the controller itself depending on which slice the caller is from.

6. **Consistent Clustering**: Providing redundancy and scale out while ensuring network consistency.

## The Service Abstraction Layer provided by OpenDayLight

The OpenDayLight controller within the general ODL project exposes two sets of northbound APIs known as Service Abstraction Layers, which comes in two flavours: the Model-Driven SAL (MD-SAL) and the Application-Driven SAL (AD-SAL). The task of the SAL is to create a layer to develop applications against that is agnostic with regards to the underlying SDN protocol or protocols. It allows to programmatically access network elements from different SDN platforms in a uniform way.

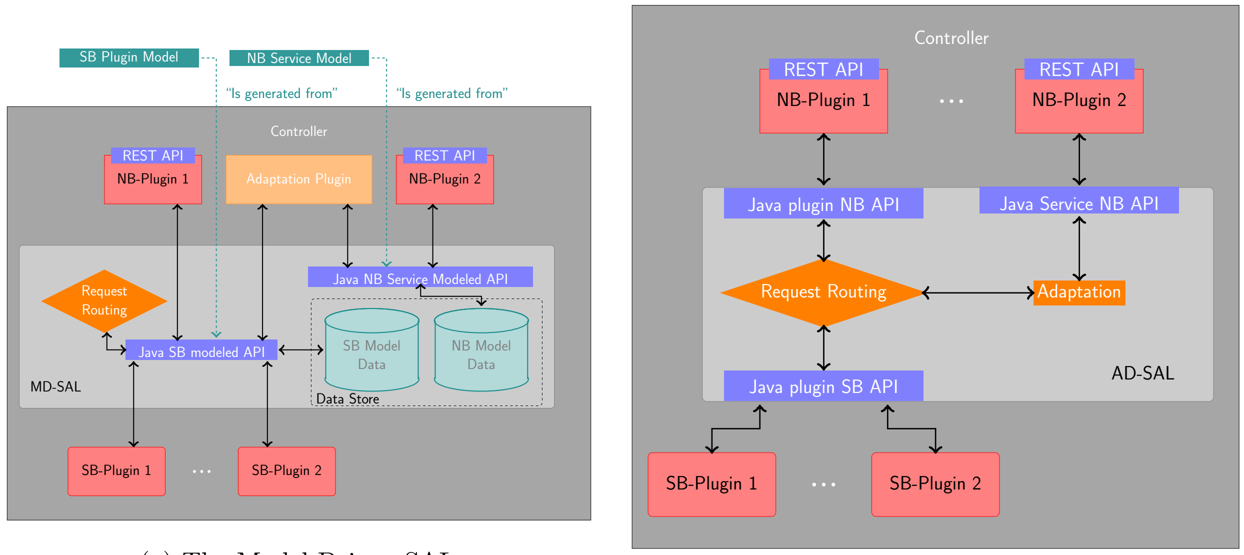
In the case of the AD-SAL, this is done by means of a **Java Contract**, which is usually a set of Java interfaces and supporting objects that represent the data.

In contrast, the MD-SAL is a set of infrastructure services within the ODL framework that is aimed at providing common and generic support to application and plugin development.

The MD-SAL currently provides infrastructure services for:

- Data Store
- RPC / Service routing
- Notification subscription and publish services

This common model-driven infrastructure allows developers of applications and plugins to develop against different APIs (e.g. Java APIs, DOM APIs, and REST APIs) derived from a single model.



(a) The Model-Driven SAL

(b) The Application-Driven SAL

Figure 2.2: The two Service Abstraction Layer models in OpenDayLight
Source: `https://wiki.opendaylight.org/view/OpenDaylight_Controller:MD-SAL:FAQ`

### Comparison between ODL and NetIDE

In a first approach, ODL can be considered another controller in the list of controllers targeted by NetIDE. However, ODL is a controller framework divided into three clear parts: northbound modules, southbound modules and the so-called Service Abstraction Layer module. In the ODL framework, network applications written in YANG and Java, later compiled into jar files, would be at the same level of the Network Application Engine in the architecture of NetIDE, which unifies the network applications written for different controllers into a single application in IRF language. The main added value of ODL for the NetIDE proposed architecture is that ODL is a controller framework capable of supporting multiple protocols, e.g. OpenFlow 1.0, OpenFlow 1.3, BGP-LS, etc. on its southbound interface. The set of southbound protocols of ODL supported should ease the integration of different SDN-based and legacy networks[5]. All of these modules are dynamically linked into the Service Abstraction Layer.

Some of the features that NetIDE could benefit from ODL:

- Active collaborative development from the community (open source project)
- Different southbound protocols (OpenFlow 1.0, BGP-LS, SNMP, NETCONF, etc) allowed for deployment
- Web interface for managing network applications

Some of the added values that NetIDE could bring to ODL:

- Ability of programming the network application with different languages (including legacy)
- Debugging and monitoring capabilities

Figure 2.3 shows a summary comparison of the architectures of NetIDE and ODL. The similarity between the IRF language for NetIDE and the YANG language for ODL is apparent, since they both are located at a similar level in the architecture, as language unifiers for the network applications that will later be deployed on the final controller framework. That is, both the IRF and YANG language allow to define the network application, although in NetIDE IRF is created after a translation from another language, while in ODL YANG is directly used for writing that application. Thus, an initial idea would be transforming the network application for NetIDE into YANG, so that it is later used in ODL. However, the YANG development module is still in an early phase and applications still need to be written or finally modified in Java.

Another relevant difference is that ODL allows the creation of independent southbound applications, also written in Java, as southbound modules for management and interconnection with the physical network. These applications grant high flexibility to the framework.

[5]This assessment is based on the scarce documentation available and needs to be confirmed with new applications

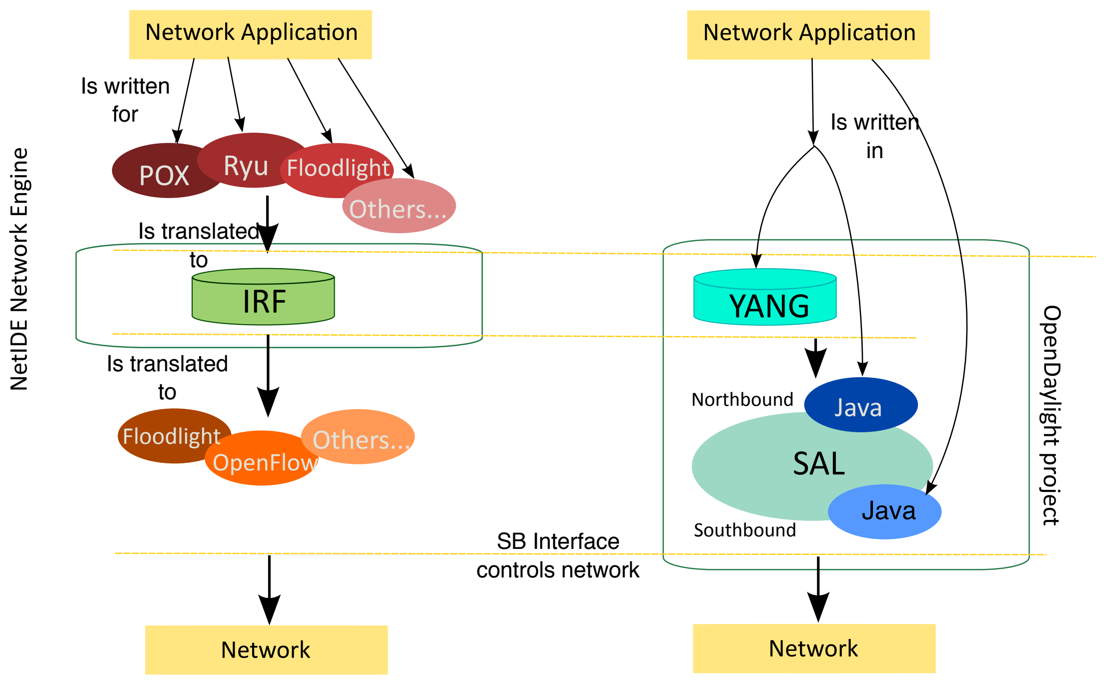


Figure 2.3: Architectural comparison of NetIDE and ODL

### 2.1.2 Northbound Interface for selected OF controllers

According to [24], the northbound API is used to communicate between the SDN controller and the services and applications running over the network. A survey that includes most of our target SDN controllers can be found in [25]. We are not covering the REST API, despite it being a way of overcoming the "programming language barrier". Any application using this API can be made multi-controller by rewriting the parsing code for each of the controllers it wants to support, and anyhow, the REST API will not be able to expose features which are not present in the controller. Instead we are looking at the APIs exposed to applications running on the controller itself.

There is a first fundamental grouping, which is forced by the programming environment selected by the development teams, shown in Table 2.1.

| Programming Language | SDN controller framework |
|---|---|
| Python | POX<br>Ryu |
| Java | Beacon<br>Floodlight<br>ODL |

Table 2.1: Classification by language

#### The Python controllers

Although POX and Ryu are both programmed in Python, their programming model is different.

Ryu provides implementations of a switching hub, the Spanning Tree Protocol, a Firewall, a Router and an OF switch tester. All these applications can be controlled through a REST API. Regarding the programming model, all Ryu applications inherit the `ryu.base.app_manager.RyuApp` class. It is in these applications where the user logic is implemented. The user logic is a set of event handlers. Functions implementing an event handler are marked by a specific decorator.

The scope of POX is more modest. It provides components for a set of layer-2 and layer-3 switches, and an implementation of the Spanning Tree Protocol. POX provides the components to implement a Web service. When writing applications for POX, the programming model is different. Applications are not derived from any specific Python class that implements default behaviour. The programmer has to implement all the registration logic, as well as the event handling loops.

The two models are so different that porting applications between both controllers is not an automatic task.

#### Java based controllers

When developing in the Java SDN world, the first big difference is the whether the controller uses OSGi, like for example Beacon, or not like in the case of Floodlight. A side-by-side comparison of the paradigm implemented in these two controllers can be made by comparing [26] and [27]. The additional effort due to the bundle-support code and the difference in naming conventions make the task difficult, although possible.

Code snippets shown in a blog [28] suggest a certain level of similarity between Beacon and ODL. This suggests that code portability is not completely out of scope when going this direction. The opposite direction seems not to be possible, due to features offered by the OpenDayLight framework only. Just as an example, Beacon does not provide the routing protocols provided by the ODL framework.

## 2.2 OpenFlow network emulation/simulation environments

Since the beginning of OpenFlow, it has been bundled with the Mininet emulation environment in order to allow people to get hands–on experience without the need of a physical OpenFlow network.

### Mininet

Mininet [29] is a Network Emulation platform for quick network test-bed set-up; it allows the creation of large (up to 1024 nodes) OF networks in a single machine. Nodes are created as processes in separate network namespaces using the Linux network namespaces [30], a lightweight virtualization feature that provides individual processes with separate network interfaces, routing tables, and ARP tables. Nodes are connected via virtual Ethernet pairs. Mininet supports binding to external controllers and provides a front end for OF soft switches such as the Stanford reference implementation and OpenVSwitch. Mininet has a full support of OpenFlow versions 1.0 and 1.3 .

### MaxiNet

MaxiNet[6] is an extension of Mininet to clustered environments, developed by one of our partners (UPB). It provides the same interface to an experimenter but allows to run much larger experiments by distributing the execution of several Mininet instances over multiple host machines. Semantic correctness is achieved by proper time dilation of instance execution.

### OpenFlow on the NetKit network emulation environment

NetKit [31] is a network emulation environment for the Linux Operating System (OS). It is mainly used to recreate networking laboratories that help understand the user how routing protocols work in real networks, using Virtual Machines (VMs) that run on using User-Mode Linux. The VMs uses Quagga [32] routing daemon and other mainstream Linux networking utilities to provide the user with a look and feel of commercial eqipment. An installation of the SW reference implementation of the OF switch OVS on a Netkit VM is reported in [33]. It also explains how to install the NOX OF controller to complete the OF emulation environment on NetKit.

### Simulating OpenFlow on NS-3

NS-3 [34] includes support for OF switching. It relies on building the OpenFlow Software Implementation Distribution (OFSID) [35], and integrates into the simulation environment using NS-3 wrappers. At the point of this writing, OF support status in NS-3 is as follows: (1) version 1.0 of the OF protocol, (2) there is a version of OFSID that supports MPLS. This version was created by Ericsson. However, the integration with NS-3 is limited, because there is no way to pass MPLS packets to the NS-3 OF switch implementation.

## 2.3 Applications extending the basic OF control plane

Additionally, other OF related projects mainly focus on additional use cases. Flowvisor [36] provides OF specific network virtualization allowing multiple OF controllers to act on a subset of the elements deployed in an OF network; and RouteFlow [37] investigates how to integrate IP routing features in the OF control plane.

[6] `https://www.cs.upb.de/?id=maxinet`

## 2.4 High-level SDN Programming Languages

When controlling the network becomes a software matter, the natural consequence is to look for high-level languages able to provide a level of abstraction powerful enough to simplify programming and achieve more complex behaviours. In this respect, we can consider OF similar to a processor instruction set in normal IT, while several network programming languages have already being proposed.

FML [38] is a policy language for the NOX controller that allows operators to declaratively specify network policies. It provides a high-level declarative policy language based on logic programming. An FML program consists of a collection of inference rules that collectively defines a function that assigns a set of flow constraints to each packet.

Procera [39] provides a controller architecture and high-level network control language that allows operators to express network behaviour policies, without resorting to general-purpose programming of a network controller. Procera is designed to be reactive, supporting how most common network policies react to dynamic changes in network conditions. Procera is able to incorporate events that originate from sources other than OpenFlow switches, allowing it to express policy that reacts to conditions such as user authentications, time of day, bandwidth use, or server load.

The Frenetic language [40] is intended to support to following essential features in the context of OF networks:

- High-level abstractions that give programmers direct control over the network, allowing them to specify what they want the network to do.
- Modular constructs that facilitate compositional reasoning about programs.
- Portability, allowing programs written for one platform to be reused with different devices.
- Rigorous semantic foundations that precisely document the meaning of the language and provide a solid platform for mechanical program analysis tools.

The current Frenetic framework prototype combines a streaming declarative query sub-language and a functional reactive sub-language that, together, provide many of the features listed above.

Pyretic [41] is one member of the Frenetic family of SDN programming languages. It is both a programmer-friendly domain-specific language embedded in Python and the runtime system that implements programs written in the Pyretic language on network switches.

Pyretic will play a significant role in later chapters, hence a few more details are in order. Figure 2.4 shows the internal architecture of the Pyretic environment. Applications take advantage of the Pyretic runtime and drive the OF network elements through a chain of elements which include the Pyretic backend and the Pyretic client that runs on the OF controller platform. The backend and the client communicate over a TCP stream using a well defined API.

Nettle [42] allows networks of OpenFlow switches to be controlled using a high-level, declarative and expressive language. At the lowest layer, it includes a Haskell library for working with the OpenFlow protocol. The next layer provides a programming model for observing and controlling networks based on functional reactive programming (FRP).

## 2.5 Tools for SDN and legacy networks

Traditionally, the main method for specifying the desired behaviour of network devices has been *configuration*. Most devices provide a command-line or graphical (web) configuration interface, and some also support the NETCONF configuration protocol [15], or its RESTful cousin RESTCONF [43].

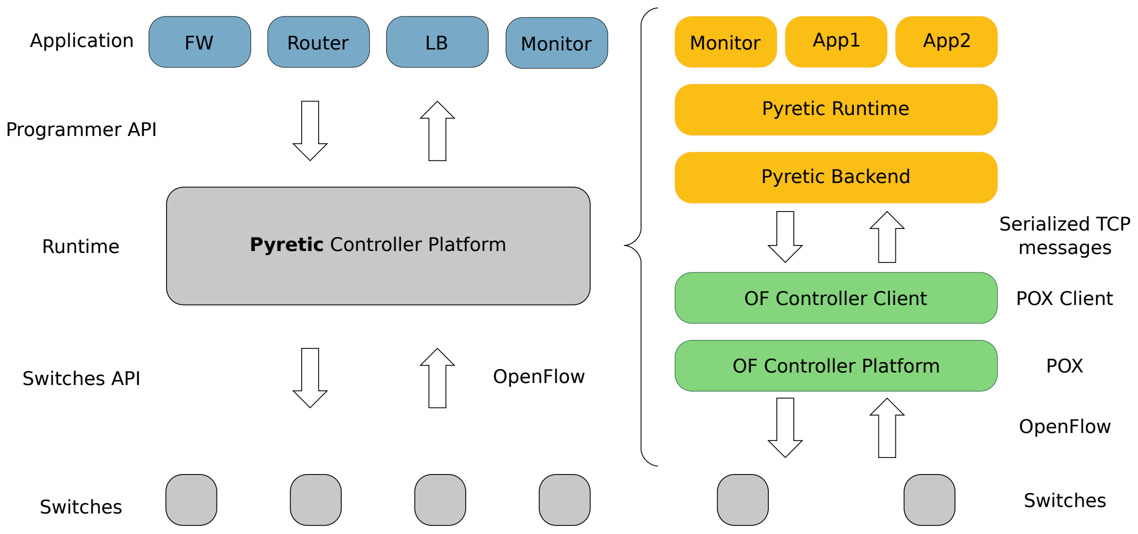


Figure 2.4: The Pyretic framework

For the NetIDE project, configuration data are potentially important for the following three reasons:

- The OF-Config protocol [14] uses the NETCONF protocol for a remote configuration of OF switches. This includes essential IP addressing and DNS information as well as the address(es) of OF controller(s).
- Most real-world networks are not pure SDN. Traditional devices, such as routers or firewalls, often have crucial roles in the overall network behaviour. It is therefore important to be able to interact with such devices and configure them accordingly.
- A configuration of a network device can be viewed as a formal specification of the device's desired behaviour. As such, it may be used as input, translated to IRF, and further processed by the NetIDE tool chain.

### 2.5.1 YANG

YANG [44] is a standard language for modelling configurations and state data. Its development in the IETF was closely connected to the NETCONF protocol but recently YANG has been used in other contexts as well.

The primary syntax of YANG resembles the C programming language or BSD-style configuration syntax. An alternative XML syntax named YIN is also defined as a part of the standard.

YANG is designed to model not only configuration and state data but also user-defined remote procedure calls (RPC) and asynchronous notifications. The data may be encoded in XML as well as in JSON [45].

YANG data models are structured into *modules*. Every YANG module declares a namespace to which all data nodes defined by the module belong. Modules may also be further subdivided into *submodules* that share the main module's namespace.

Basic building blocks of a YANG data hierarchy are similar to XML schema languages such as RELAX NG [46]:

- A *leaf* node is a single configuration or state parameter. Every leaf has a type and it may also have a default value.

- A *container* node contains leafs, other containers, lists or leaf-lists or nodes. YANG also distinguishes containers that are used only as folders for organising data from those whose presence has a semantic meaning, such as turning on a certain function.
- A *leaf-list* node represents a sequence of leafs. A minimum and maximum number of entries in the sequence may be specified.
- A *list* node is similar to a leaf-list but represents a sequence of containers with the same structure. Every list declares one or more child leafs as the list key that uniquely identifies each list entry.
- A *choice* node represents multiple alternative content models that are allowed at a given place.

The selection of YANG built-in types is roughly comparable to that of XSD Datatype Library [47]. However, YANG also allows for deriving new named datatypes from existing ones by specifying additional restrictions.

Semantic constraints may be specified in YANG using the **must** statement. Its argument is an XPath expression that has to evaluate to true for a valid data instance.

Reusable collections of data nodes may be defined in YANG as *groupings* that may be used repeatedly, also in other modules.

An important element of modularity in YANG data models is the **augment** statement. Its main purpose is to extend an existing module from outside by adding new data nodes at any location.

### 2.5.2 Standard Data Models

After the YANG language specification [44] was published as a Proposed Standard, work started in the IETF on the development of core data models that are needed for most network devices:

- *ietf-interfaces* [48]: configuration and management of network interfaces;
- *ietf-ip* [49]: configuration of IP parameters and management of the IP layer;
- *ietf-system* [50]: basic system configuration and management;
- *ietf-routing* [51]: framework for configuration and management of the routing subsystem

More specialised data models, e.g. for routing protocols, are expected to be developed by different IETF Working Groups.

## 2.6 Tools supporting the software development cycle in Software Defined Networks

### 2.6.1 OpenFlow Debuggers

Another component for an OF network application development cycle is the debugger. The task of Controlling how packets flow in an OF based network is performed by an application running in the OF controller. Tools to control that this application works as expected and to isolate malfunctioning parts of it are needed. These tools fall into the same category of tools as debuggers in the SW development cycle.

Handigol et.al. have proposed an OF debugger called *ndb* [52]. This tools mimics the GNU debugger *gdb* [53] in the SW development cycle in some aspects. It is a command-line tool that loads an OF application and allows it to run natively in the OF controller. It uses breakpoints and back-traces to inspect how an application is behaving. The intended use of *ndb* is as follows: (i) The debugger loads the network application. (ii) The programmer sets breakpoints at specific points in the network application, where the state wants to be observed. This normally happens when the network application was not able to handle a specific network packet. (iii) The network application is launched and specific network debugging traffic is launched. (iv) The debugger halts the application at a breakpoint. (v) Controller state and test packets that led to the network application hitting the breakpoint are stored (vi) The debugger hands control to the programmer who has access to the state and the packets that were being processed when the application hit the breakpoint.

As mentioned above, Trema [9] and other OF controllers go beyond the mere implementation of an OF controller and provide an OF development framework. In the specific case of Trema, it includes a Wireshark [11] plugin to dissect OF packets [54] and a bridge to display OF events on Wireshark, known as Tremashark [55].The next step in this development is provided by OFRewind [56]. This tools collects network traces, that can afterwards be replayed in order to detect configuration or SW errors in a network. Additionally, tools to check the different components in the OF eco-system. FlowChecker [57] provides a method to check the flow-tables within an OF switch.

A very comprehensive approach to OF testing should be provided by the final version of NICE [58]. This tool, which is available in a preliminary version [59], is designed to test network applications on an OF controller that control several switches. Techniques like code analysis will allow the programmer to determine classes of packets that receive the same treatment in the controller and generate stimuli that cover the whole OF application under test. This tool will also create a network model using an iterative approach to refine the model and cover the application fully.

### 2.6.2 LoxiGen

LoxiGen [60] is a tool that generates OpenFlow protocol libraries in multiple languages. It is composed of a frontend that parses wire protocol descriptions and a backend for each supported language (currently C, Python, and Java).

LoxiGen currently supports OpenFlow versions 1.0, 1.1, 1.2, and 1.3.1. Versions 1.0 and 1.3.1 are actively used in production. Support for versions 1.1 and 1.2 is considered experimental.

LoxiGen is licensed under the EPL version 1.0. The LoxiGen compiler license contains an exception giving the user permission to license the generated code under any license that they choose, provided they maintain the same copyright headers contained in LoxiGen.

# 3 The NetIDE architecture – A first draft

In this chapter, we document the bootstrap work undertaken by the project. The purpose was to, on one hand, identify those components of the proposed NetIDE architecture that we can implement using state-of-the-art components and technologies directly and, on the other hand, to find gaps and shortcomings these technologies have compared to the scope of NetIDE.

We recall that the Description of Work (DoW) [61] stipulates support for both development and execution of network control applications. The discussion here focuses on the development aspects as they form the underpinning for an execution/runtime support. In later architecture deliverables, support for runtime aspect will become more prominent. First ideas on that have already been published [62].

We start this discussion by recalling, mostly from the DoW with some additional considerations, the requirements we see for the development, deployment, and execution phase (Section 3.1). Based on that, we consider a broad range of shortcoming of the existing controller frameworks (Section 3.2). From that, we derive a first draft of the core architectural abstraction for the NetIDE Intermediate Representation Format (IRF) (Section 3.3). To support the credibility of this approach, we have undertaken a first proof-of-concept implementation, discussed in Section 3.4.

## 3.1 Requirements in different phases

We envision an Integrated Development Environment for Network Application Development that is closely related to equivalent systems in the Software delvelopment world. Deliverable D 5.1 [63] provides information regarding requirement handling in the different phases in the lifetime of a Network Application. In this section, we briefly recall how requirements guide the work of the different kinds of Network Application users.

### 3.1.1 Development phase

During the development phase, requirements guide the developer produce a useful Network Application:

- **Requirements collection:** the developer defines the scope of the Network Application to be developed.
- **Analysis and Design**: following the results of the requirements collection phase, the developer specifies the actual behaviour of the Network Application and models relevant aspects that will be the variables of and constraints for it.
- **Development**: the developer actually codes the network program using his favorite network programming language, translates it to the IRF, and develops the APIs exposed by the Network App.
- **Testing and Validation**: With support of Unit tests and simulators, the developer will be able to assure that the behaviour of the Network Application complies with the specification before actual deployment in the production environment.

### 3.1.2 Deployment phase

After the application reaches production grade, it can be deployed in the production environment. Debug and testing tools are still available to developers during this phase in order to assess performance, detect and debug malfunctions, etc. In case of need, the application can be fed back into the development cycle for a new iteration.

### 3.1.3 Runtime phase

Once the Network Application is deployed, it needs to be monitored to check that it complies with the expected behaviour. In case the observed behaviour deviates from it, there will be a need for a debugger to understand what conditions cause such a misbehaviour and how it needs to be modified to meet the requirements.

Additionally, the system has to allow to run several Network Applications in parallel. We cannot anticipate that the IDE will produce optimised Network Applications at compile time. Hence, we will need to implement *optimisers* that deploy the applications onto the SDN platform minimising redundant use of resources. With the time resource usage in the SDN infrastructure might degrade. Therefore, the Network App needs to provide enough information to implement *garbage collection* to free up resources that were allocated to a Network Application but remain unused in the SDN infrastructure.

### 3.1.4 Requirements conclusions

As stipulated by the DoW, our approach is to create a unifying language that covers orthogonal aspects of the deployment models of different SDN approaches (for example, some SDN platforms handle the network on a packet-level basis, while other take decisions based on the information of flows) and that is executable and translatable across different SDN flavours. We call this unifying language the NetIDE IRF and aim at making it deploy-able on actual SDN substrates in the same way as OpenFlow or vendor-specific applications are deployed today. Section 3.3 summarizes our insights so far and links the architecture discussion to Deliverable D 3.1 as well.

## 3.2 Shortcomings of existing solutions

When pondering the brief survey of the state of the art in Chapter 2 and reflecting that with the DoW, there are two aspects where current solutions have *conceptual* shortcomings: 1. the interaction between control applications and the actual controller itself, 2. the interaction of the controller with actual switches, where the first aspect is certainly the more challenging one. There are also some *practical* or convenience shortcomings; they pertain mostly to the development environment and are discussed in other deliverables. However, overcoming the conceptual shortcomings will necessitate additional functionality also from the development environment, making this more than a mere implementation exercise. Let us first consider these two conceptual aspects in more detail.

### 3.2.1 The control application/controller aspect

When selecting the SDN platform, the developer is confronted with a number of controller frameworks with different programming languages, component models, etc. Additionally, the interaction between the developer of a control application and the controller framework or, respectively, between the control application and the ready control is complicated by a number of issues discussed below. While not all of these issues hold or are relevant for all frameworks described in Chapter 2, many of them recur across the board.

**General-purpose programming languages** With very few exceptions, controller frameworks assume that the actual control applications are developed using one specific, general-purpose programming language (with Java and Python being popular examples). While this probably lowers the entry barrier to novice programmers, it is at the core of some of the issues discussed next. It is also a considerable hindrance to executing applications written in different programming languages inside a single controller framework (of whatever ilk).

As an example, we concentrated in creating applications for controllers written in Python and Java to see the extend to which Jython [64] can help bridge the gap between both programming languages. However, the results were quite discouraging and we were not able to produce even a simple Network Application written in Python that could be executed on a Java based controller using Python.

Only some of the frameworks (along with a wider development model, then) use explicitly crafted languages.

**No separation of phases** There commonly is no explicit separation of the three crucial phases development, deployment, and execution foreseen in the controller frameworks as such or in their development environments (if any exist at all).

Two examples of problems we encountered were:

- **checking topologies at development time and runtime:** we were not able to specify the class of topologies in which the developed control application can run at development time in our test environments. In other words, it is not possible to check at deployment time whether a particular application is suitable for a given network and the developer has to know the topology a Network App can run on before launching it. At runtime, this can lead to strange, difficult to find misbehaviour or outright failure.
- **debugging an application:** we found no way to carry information available at development time into the execution phase to support, e.g., a network debugger. There is no commonly agreed upon format for such information across different platforms.

**Various interaction models** The way a controller and its control applications interact can be multifaceted. While there is a certain prevalence for so-called *event-condition-action* patterns, this is not necessarily the only possible pattern. Pyretic is one of the very few of the controllers and their associated programming paradigms that make this model explicit.

**Implicit interactions** Even if the interaction model is well specified, it is not necessarily easy to unearth such interactions from a concrete piece of code realizing a control applications. This is perhaps the point where different programming paradigms differ the most: Some approaches make it very explicit, going to the point of introducing explicit language constructs. Other approaches are very implicit here, relying on proper function calls and observance of programming rules that are perhaps online specified in natural language, but are not amenable to automatic processing.

Extracting interactions between controller and a single control application then becomes a considerable challenge; it can turn practically impossible if multiple control applications start implicitly interacting with the controller. This is in particular the case if each control application stores its own view of the current state of the network (e.g., switch states, which data flows consuming what data rate, . . . ). This can easily lead to solutions that are hard to understand, tune, or debug.

**Imprecise concurrency semantics** Essentially all controller frameworks have some more or less well developed notion of being able to support multiple control applications. It is, however, not always fully clear how the precise concurrency semantics of these applications looks like: are they run in parallel (trigger by an event?), in sequence, can the execution sequence be aborted, etc.

And even if that semantics is well defined, it can differ significantly between different controller frameworks. An application developed for one framework might easily have some implicit expectations how it is executed with respect to other applications in the same controller. Transplanting such an application into a controller with a different semantics is again likely to fail.

**Different component models** Only some of the controller frameworks explicitly support a component model. For example, OSGi is supported by Beacon and OpenDayLight, two controllers written using the Java programming language and integrated into the Eclipse IDE. However, they differ in programming model and API [65]. Hence reusing control applications as components across these two controllers is not possible without a source code porting (i.e. rewriting) process.

We have experienced this in the proof-of-concept work reported in Chapter 4. In our case, the choice was POX and Ryu, two OpenFlow controllers written in the same programming language (Python), but with completely different models.

**Multiple controllers** Most controller frameworks think in terms of a single controller that runs inside a given network. While there is some work on replicating a controller for fault-tolerance reasons (e.g., ONIX [66]) or performance reasons (e.g., Kandoo) [67], this is still a notion that is not deeply embedded in typical frameworks. In particular, the coordination effort necessary to achieve consistent behavior is both complex and run-time intensive and necessitates severe changes in a controller's data structures (if said controller is not prepared for coordinated execution).

And even if that effort has been completed for one controller framework, that does not mean it is now possible to mix different controllers inside one network. The difference in concurrency semantics, execution patterns, etc. will make this a very brittle exercise at best.

Running multiple *uncoordinated* controllers in a single network is obviously an absurd notion.

In short, we see substantial difference between controller frameworks with respect to their programming languages and component models, the level of explicitness of application/controller interaction and concurrency model. All these differences contribute to the hardness of combining control applications from multiple frameworks inside a single controller.

### 3.2.2 The controller/switch aspect

Once an SDN solution is deployed in an actual controller, this controller has to talk to the network switches using the SDN control protocol. OpenFlow's claim to fame is in large part the attempt to harmonize and standardize this control protocol. Its success not withstanding, it is still the case that there are switches that use legacy or closed-source, vendor-defined protocols; examples are Juniper Networks' Junos XML API [68] or Cisco's OnePK framework [69].

As long as the semantics of these protocols is similar or identical, mere syntactic differences are of little concern and can be dealt with by a common driver architecture: translating requests/replies from one protocol to another is a mere mechanic exercise.

A more interesting case happens when there are indeed semantic differences between these control protocols or between the assumed capabilities of a switch. This will be the focus of the corresponding task.

From a practical purpose, however, we should note that the work of the OpenDayLight consortium has already had some considerable impact here. Their attempts to standardize these very protocols seems promising and might succeed in defining de-facto standards, turning a conceptually interesting research question into one that is less relevant for the real world. We shall follow the activities of OpenDayLight on this point very closely and will decide how and whether to pursue this line of work further or whether to rely on OpenDayLight here.

## 3.3 Core concept: The IRF and its aspects

As described in the Section 3.2, we are confronted with a set of very diverse controllers with diverging models, different programming languages, etc. Since NetIDE tries to provide a means to make them collaborate and allow for developments to work across the board, we argue in this sectionthat the introduction of a domain-specific language is beneficial towards this goal.

### 3.3.1 Internal and external domain-specific languages (DSLs): IRF behavioural aspects

The discussion in Section 3.2 has highlighted the different expressive power levels of different controller frameworks and their associated languages. In particular, it is useful to categorise them loosely according to their level of formality:

- Some frameworks use an explicit, non-general-purpose language to define control applications (example PI-Calculus).
- Some frameworks use a general-purpose language without any easily recognizable structure how the control application interacts with the controller; at best, specific function calls could be detected (example Beacon).
- The middle ground between these two extremes is taken by approaches that use a general-purpose language, but endow it with enough structure such that the execution semantics of the controller is reflected in that structure. The typical example is Pyretic: control programs written in Pyretic essentially express Event-Condition-Action (ECA) patterns that are then interpreted and executed by the corresponding POX controller.

This last, middle-ground case is particularly interesting as we believe this to be the right balance between structure and entry barrier. In fact, such approaches are well known in software engineering: into a general-purpose language, recognizable elements are embedded for “programming” in a very specific domain. These approaches are called *internal domain-specific language (DSL)*. We shall concentrate, as a first hypothesis, for the remainder of this architecture discussion, on controller frameworks that have a more or less well developed internal DSL.

But while internal DSLs are highly appropriate for a programmer, they are not necessarily useful to be used across different controllers. For example, the “host language” into which they are embedded differs in our examples. Executing that is in practice not feasible even though perhaps conceivable in principle (imagine a controller that has to execute ECA patterns specified in both Java and Python).

We have hence decided to follow a different path for the NetIDE architecture: Assuming that a controller application is written using an internal DSL (and our editor and tool support will make that a very attractive choice), we will compile them into code of a common, *external DSL*. This external DSL has to be expressive enough to encompass all the (reasonable) functionality present in the internal DSLs that we will consider. In particular, it has to be able to express a suitable superset of Event-Condition-Action patterns along with the data basis on which a controller application operates. It has to maintain sufficient meta-data (e.g., variable names) to later allow simple debugging.

Structuring such an external DSL is a challenging task. It is simplified here by the fact that the controller applications we are considering here are, after all, restricted to a fairly narrow domain. It hence makes sense to get inspiration from similar fields, and autonomous computing did develop a suitable structure for such a task: the MAPE-K reference model. This is a general pattern how to structure languages and execution paradigms with a control loop that includes components that 1. Monitor a system, 2. Analyse a system, 3. Plan and Execute actions, 4. and include an explicit representation of a Knowledge base that holds information about the controlled system. In addition, there is the notion of a decision component that can coalesce planning actions coming from different planners.

*Our job is hence to analyze a control application, written in an internal DSL like Pyretic, and extract the monitoring, analyzing, and planning components out of it and represent them explicitly.* The execution of the plans will be actually quite simple: the only real actions control applications can take are actions that modify switching tables. This takes place via the SDN control protocol (e.g., OpenFlow) in a uniform, well standardized manner and can hence be thought of as the main task for the controller. In this sense, the controller takes on a different role: it turns into an interpreter for the MAPE-K data and code. [1]

The details of how we leverage the MAPE-K scheme will be discussed in detail in Deliverable D 3.1. Here, let us just point out one advantage and one challenge:

**Flexible choice of concurrency model** Section 3.2 has discussed the difficulties of different concurrency models in different controller frameworks. These different semantics still persist, but the decision component of a MAPE-K system is the natural place to include a flexible resolution logic to arbitrate between the different models.

Consider the simple case when indeed all control applications come from the same controller framework. Then, the decision component only has to mimic that concurrency model (sequential, parallel execution) – such a functionality can be provided in general.

In the more complicated case, when controller applications from different frameworks is to be mixed, no easy resolution is likely. Such a logic will have to be provided manually, either at development or at deployment time (when the decision is made to mix applications from different frameworks). But at least, there is a single, well understood place where these resolutions can take place, rather than having to distribute this logic over a complex code base.

**Code "blobs"** In a ideal scenario, the various internal DSLs are expressive enough to support all kinds of monitoring, analyzing and planning needs, and all these functions can than be directly represented in the external DSL.

In practice, this is unlikely to be always feasible. More likely, there will be some functionality that is invoked by a piece of code in an internal DSL that is not immediately representable in any form of generalized format. We are hence faced with the need to execute black-box type of code that can be considered a "binary" code (from the perspective of the MAPE-K interpreter) – we shall use the common term of a "blob" (binary large object') to refer to such pieces of code. As long as the interfaces to such a code are clearly defined, this should be doable; the main advantage here is the structuring of the programs as well as the Knowledge base against which all these binary code blocks operate.

[1] TODO: explain difference to DoW. Driver turned into a compiler. We have another driver further down, but that's a different thing.

Nevertheless, will we see that as something that is doable, we also acknowledge that support for such blobs holds considerable challenges and does require further analysis.

In summary, the external DSL constitutes the behavioral aspects of NetIDE's IRF.

### 3.3.2 Topology patterns: IRF structural aspects

In addition to the behaviour of the IRF, it is necessary to express structural aspects as well. In particular, the topology of the network that is to be controlled by SDN is relevant here.

Looking more closely, one has to distinguish between topology aspects in the different phases.

**Topology during development** An application developer typically has some sort of network topology in mind when developing a control application. That can be a trivial one when a simple "in/out" application like a firewall is developed. It can be a very complex topology like a Clos network [70] when thinking about routing in a data centre.

Usually, an application developer will try to develop an application that is fairly generic and usable in many contexts. Hence, developing for a *specific* topology is not useful. Rather, one should think in terms of *topology patterns*: a (typically) parameterized family (or set of families) of topologies to which this control application is applicable.

**Topology during deployment** When deploying a (set of) applications in a real network, the parameterized topology should be matched with the real network topology (if known at deployment time). Moreover, if multiple applications are deployed, their respective topology patterns should be checked whether they are mutually compatible.

**Topology during executing** Once the network operates and control applications run, the topology description of the applications is replaced by the actual topology of the network (possibly statically provided, possibly dynamically configured). In terms of the MAPE-K approach, topology data forms part of the knowledge base.

In summary, the topology description is the data model part of the IRF. It expresses the state of the system. Compared to the behavioural description, it is much simpler, yet still requires further investigation. Our initial attempts using YANG are reported below (Section 4.3), but have no yet achieved the level of flexibility and parameterizability we are looking for.

### 3.3.3 Transformation flow: DoW vs. now

With its intended ability to express both structural and behaviour aspects, the IRF is a centrepiece for NetIDE work. In the DoW, we had envisioned transformations into the IRF and using "drivers" to pass IRF artefacts in a suitable form to a specific controller. Based on our current understanding so far, we view these transformation steps somewhat differently by now:

1. We still foresee the transformation from a controller program into IRF. We currently assume that this will only be possible if these programs were written for a controller framework that exposes some sort of internal DSL as a structuring means.

2. We are hopeful that the extraction of blobs can succeed and that this can be embedded into an IRF document.

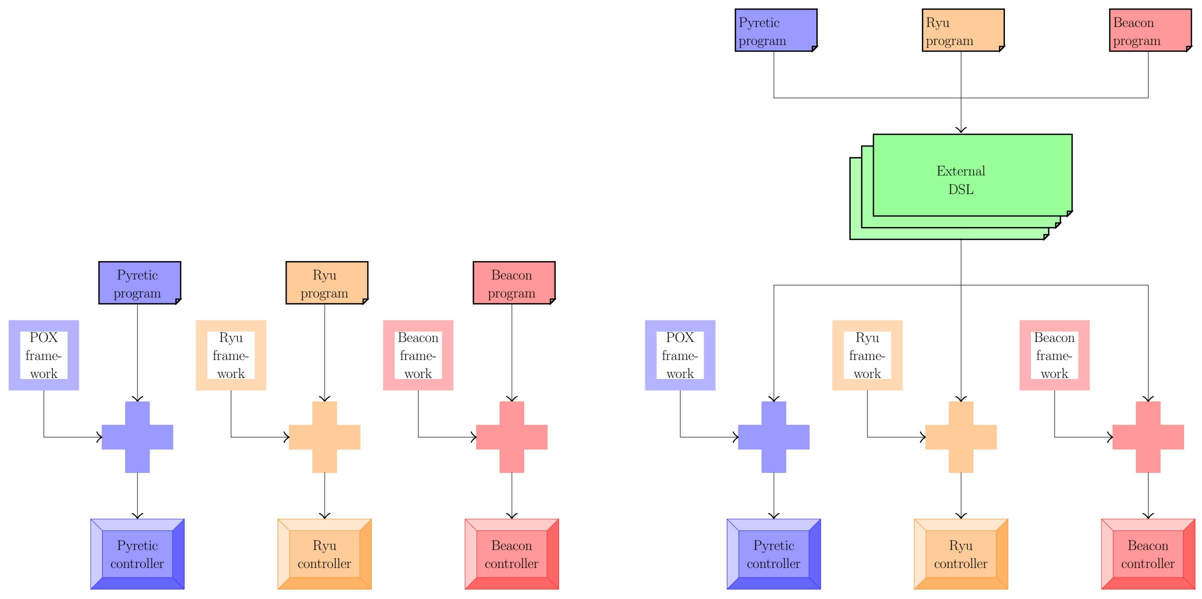
3. The notion of a "driver" seems no longer appropriate to describe the connection between an IRF document and a controller. A driver carries the connotation of a program that runs continuously and translates data between different formats (in the broadest sense of the word).

Rather, we currently imagine to translate the IRF into specific code that matches a particular controller; this happens once at deployment time, rather than continuously at runtime.

Nonetheless, there are aspects that take place continuously at runtime and that involve the IRF; the prime example is the modification of the topology representation in the MAPE-K knowledge base. Obviously, this process is based on IRF formats.

4. There is, however, another notion of a driver: A piece of software that can translate between different formats of SDN control protocols. As discussed above, this driver can range from fairly simple to highly complex.

Figure 3.1 compares our current understanding of the NetIDE flow of transformation with our understanding at the time of preparing the Description of Work.



(a) Before NetIDE: only directly mapping from program to controller possible

(b) With NetIDE: translation into External DSL allows flexible mixing

Figure 3.1: Transformation flow between different software artefacts

## 3.4 A simplified architecture: NetIDE v.0

The architecture we have outlined above has a rather high ambition level. In particular, it seems challenging to mix control applications coming from very different frameworks. To get a first understanding on feasibility, we started our practical work (described in the following Chapter 4) with a very restricted version of the architecture.

Specifically, we made the following main assumptions:

1. We only considered two controller frameworks, namely Pyretic/POX and Ryu.
2. We only consider Pyretic as an internal DSL for SDN control applications.
3. We only consider static topology aspects (no patterns) and looked at the opportunities offered by YANG.

Because of these limitations, one could look at this version 0 of our architecture (along with its proof-of-concept implementations) as a sort of Pyretic++. We definitely intend to go beyond these simplifying restrictions in the course of the project; for now, it seemed advisable to allow us to test our ideas with quick and early test implementations.

The work on Pyretic helps us understand the challenges of the proposed network API: currently Pyretic is attached to the POX OpenFlow controller. Detaching it from the underlying controller would yield a platform-independent implementation. The glue between Pyretic and the underlying controller is provided by means of an API that is implemented by means of serialised TCP messages. These messages help us understand the requirements for the Intermediate Representation Format, especially for the way we want to express the dynamic behaviour of the NetIDE network applications, which is the basis for the *External DSL* depicted in Figure 3.1b. This work is linked with the IRF work within WP 3. We target a detached Pyretic that provides a run-time environment to applications and drives a backend that communicates with different Pyretic clients through an intermediate process (the NetIDE interpreter) that provides adaptation for different controllers.

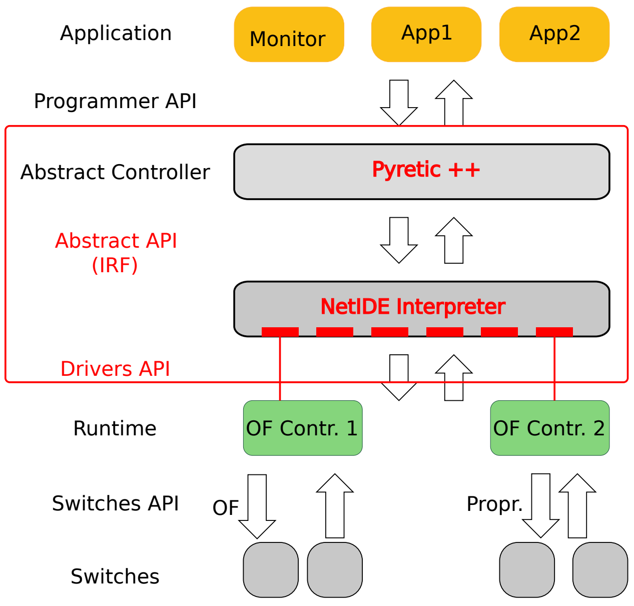


Figure 3.2: The NetIDE v.0 framework as an evolution of Pyretic

This detached Pyretic, shown as Pyretic++ in Figure 3.2, is intended to be generic and will speak Pyretic on its northbound interface, i.e. towards the different Network Applications. Its southbound interface provides an abstract API which exposes all functionality the Network Applications will be able to use. This functionality is abstracted from the different SDN models we envisage in NetIDE and is expressed by means of the Intermediate Representation Format (IRF).

The NetIDE interpreter takes the application expressed in the Intermediate Representation Format and transforms it into controller-specific code. We envisage one such transformer per controller type supported by the NetIDE architecture. The driver API limits the set of packet types and actions the controller code can contain and sends the code to the controller.

We envisage the drivers API to be capable of driving pure OpenFlow controllers and SDN controllers that drive proprietary equipment. In the initial tests, we are targeting the OpenFlow controllers POX and Ryu.

# 4 A proof of concept: Pyretic and Ryu

## 4.1 Goals of the proof of concept

We have defined the simplified architecture to allow us to do a reality check: is that concept feasible even under simplifying assumptions? Specifically, we wondered how difficult it would be to cross-use programs written for two similar controllers: POX and Ryu. Both Python-based, yet they do have differences with respect to their execution semantics.

In order to advance towards a controller-independent framework, we have focused on the API between the backend and the client, and communicated the Pyretic infrastructure to a Ryu controller. The API between backend and client has provided a preliminary model for the Intermediate Representation Format, which is being used in WP 3 for designing the development environment architecture.

The Pyretic implementations in Sections 4.4 and 4.5 explore possible evolution paths from the POX-centric Pyretic environment towards the NetIDE architecture. We use the Pyretic on POX implementation as a reference. Pyretic offers a client-backend interface between the framework and the controller. This interface provides an API that we have used to extract an initial concept for the behavioural models in the IRF.

Additionally, Section 4.3 gives more details on how we attempt to leverage YANG as a topology description technique and how to make it more dynamic than it is today. Both discussions are routed in UC1, the Data-Centre (DC) use case, described in detail in another deliverable and briefly recapitulated in Section 4.2.

## 4.2 Brief reminder: The Data-Centre use case

In order to be profitable, DC operators need an attractive service offering that maximises the number of users the infrastructure serves. To achieve this objective, providers may choose to offer *Infrastructure as a Service* (IaaS) [71], *Platform as a Service* (PaaS) [72] or similar service strategies, collectively known as "xaaS" [73]. Each service proposal depends on client needs, alliances, provider portfolio and other factors that are out of the scope of this document.

The DC use case in NetIDE concentrates on an SDN-based IaaS provider. In this case, the provider sells virtual DC infrastructure to their clients who get a DC infrastructure on which to operate their services. Figure 4.1 shows one possible virtual DC offering to a client. The figure highlights areas or patterns that can be used as "components": 1. the Internet connection, including some functionalities (like, for example, the advertisement of the client's IP prefix to the Internet); 2. a De-Militarised Zone (DMZ), where the fortified Web and Domain Name Service (DNS) servers are located; 3. the protected zone, hosting the back-end servers that serve sensitive data to the web server. It also shows another important feature of pattern-based network design: the specific client depicted in Figure 4.1 has chosen to have a DMZ-like zone directly connected to the Internet. However, another client might choose to protect this zone with a Firewall. Additionally, different clients might choose different kinds of Firewalls: one client might need a stateful Firewall (FW) in order to protect the DMZ while a stateless solution might suffice for another. The use case also demonstrates that one of the corner-stones for a successful SDN-based strategy is to have *reusable components* to build the configurations offered to clients in an automated way. Reusable

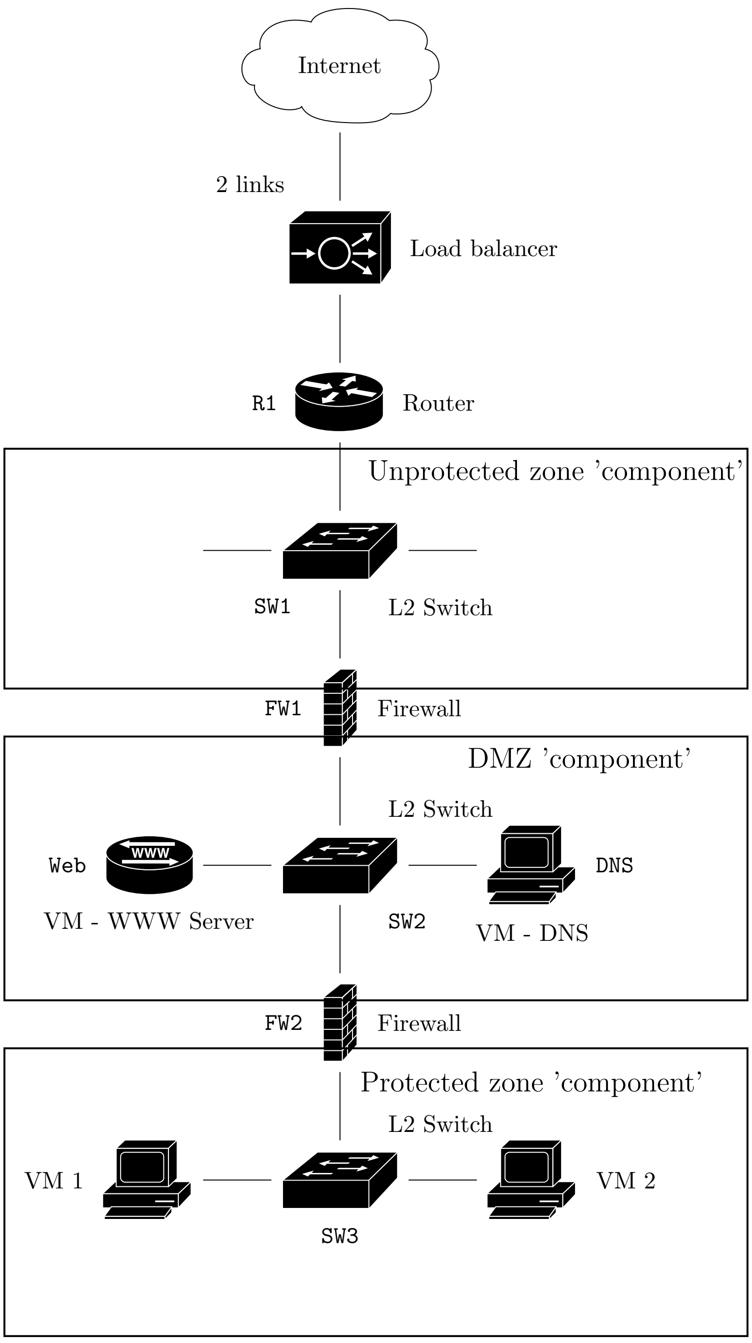


Figure 4.1: Logical view of a virtual data-centre offering

components let the provider leverage on the development costs: once a component is developed and debugged, it can be instantiated in all client deployments. Additionally, all clients will benefit from enhancements on the components used in their infrastructures.

## 4.3 Implementation in YANG

In traditional statically configured networks, two pieces of information completely determine the network structure and behaviour:

1. *topology*, i.e. essentially a graph with nodes representing network devices and edges representing communication links;
2. *configurations* of individual network devices.

SDN networks add a highly dynamic component – the controller – that doesn't fit this picture very well: its behaviour is typically specified using a program written in a high-level programming language (see Sec. 2.1.1).

The purpose of the YANG implementation was to investigate an alternative approach to the Intermediate Representation Format (IRF) definition based on the YANG data modelling language, and test it on the Data-Centre use case. The requirements were as follows:

- The format must be reasonably general, i.e. not limited or tuned to the particular use case.
- To the maximum extend possible, the format must reuse existing data models for device configuration.
- An instance of this format must be able to capture the topology and behaviour of the Data-Centre uses case as specified above.

After a series of experiments, the best layout of the YANG-based IRF turned out to be the simplest one: a coordinated set of configurations, each in a separate file. One configuration is central: it describes the network topology and provides links to the other configuration files.

Then, for every physical network device (switch, router, firewall, host etc.) there is one additional configuration file. It contains a standard configuration in that it could be directly installed in devices that support NETCONF and corresponding data models. However, it is not required that all devices support NETCONF: for those that don't, the configuration may need to be translated to a device-specific configuration language.

Each configuration can be serialised in either XML or JSON format.

The exact organisation of the set of configuration files is somewhat arbitrary. For the Data-Centre use case, we used the following layout with an extra directory for device configurations[1]:

```
data-centre-topo.json
node-conf/
  DNS-config.json
  FW1-config.json
  FW2-config.json
  R1-config.json
  SW1-config.json
  SW2-config.json
  SW3-config.json
  Web-config.json
```

Optionally, all configuration files may be collected in a single archive such as TAR or ZIP.

### 4.3.1 Implementation

It is possible to affirm that there is no single notion of network topology, which can mean different things in different contexts. The common denominator should always be the physical topology but various virtual topology abstractions can be built on top of it.

Monsanto et al. [74] distinguish two orthogonal approaches for creating topology abstractions:

[1]the device names correspond with Figure 4.1

1. many-to-one mapping, in which several physical components are coalesced into a virtual component ("big switch");

2. one-to-many mapping, in which one physical component gives rise to multiple virtual components, such as VLANs or virtual hosts.

Monsanto et al. also note that the two approaches for topology abstractions are incompatible. Therefore, in order to keep the possibility of creating arbitrary abstractions, a specification of network topology must include all details and avoid introducing any abstractions on its own.

This consideration prevented us from using the existing topology data model [75] that was developed by the I2RS Working Group in the IETF and also adopted for the OpenDaylight project. This data model is designed for one-to-many abstractions but does not support many-to-one abstractions.

We therefore developed a new data model for representing physical network topologies and other necessary information. It is based on the assumption that the physical topology is the fundamental substrate and all abstractions built on top of it must be specified in the configuration files.

Figure 4.2 shows the entities appearing in the data model together with their relationships. The complete YANG module appears in Appendix A.

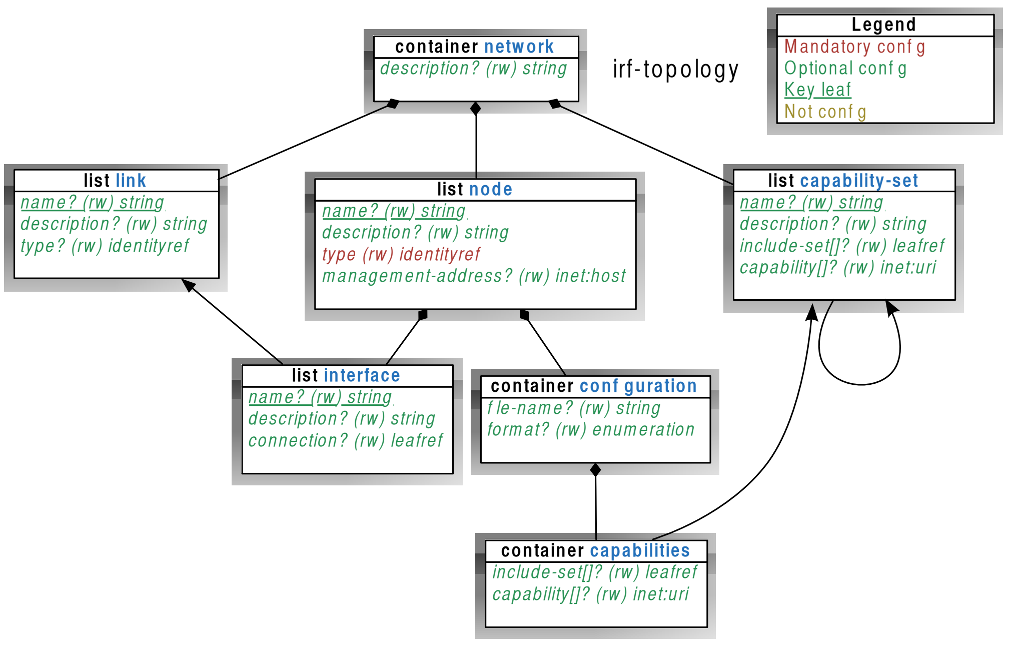


Figure 4.2: UML diagram of the topology data model

Basically, a network is represented as a set of nodes (devices or "cloud-like" objects) and a set of links. Each node has one or more interfaces that may be connected to links – this is how the network graph is defined.

A node representing a physical device also contains a reference to the device's configuration file, declaration of its format (XML or JSON), and also a specification of the data model implemented by the device in the form of capability Universal Resource Locators (URIs) (see [44], sec. 5.6.4).

In order to avoid repeating the long capability URIs over and over again, the data model allows for defining named *capability sets* that may be used for any number of nodes.

### 4.3.2 Gap analysis

For the Data-Centre use case, the YANG-based IRF could capture most of the required functionality. The data models of individual devices appearing in the Data-Center use case (as formulated above) can be built from existing data models. However, after extending and refining the services of the Data-Centre network, several potential gaps of the YANG-based approach may be encountered:

1. The design of topologies can be efortlessly done in YANG. Specifically, we use the "container network" in our use case to define a network topology and the nodes that shape it. However, that network topology might be hard to visualise from its instantiation in YANG.
2. Regarding static configuration, many parameters might have to be taken into account, which are defined by following different standards. So only a few parameters, the most common for all the components in the network, can be specific while the rest need to be generalised. For example, IP addresses can be easily assigned and, in our particular use case, the "grouping address-and-port" was defined in order to set the domain name or the IP address of the device and its associated port number.
3. Truly dynamic behaviour of SDN switches that requires complex controller programs may be difficult or impossible to capture in a static configuration, since YANG is not able to save dynamic configuration as it is a static language. Therefore, YANG language should be extended to provide that functionality.
4. Modularity support of functions can not be easily demarcated in YANG since it is directly related to the dynamic configuration. Although the YANG language defines the so-called YANG modules, these modules are static in nature and do not allow support for functionality modules, which are dynamically defined by the SDN software in the end.

The first gap is not critical for the development, but YANG could be extended to allow that visualisation if needed. The second deficiency should be tackled by clearly defining what items of the static configuration are fundamental for the system and what ones are not crucial or could be omitted for the system to work properly. Finally, the third and fourth items are quite incompatible with the static definition of YANG. While static configurations can, to some extent, describe dynamic behaviour (e.g. in a form of match-action rules), they are certainly no Turing-complete languages.

In conclusion, it is quite safe to assume that YANG-modelled configuration data can be used to represent two parts of the IRF, namely the network topology and static configuration of devices. However, for SDN controller programs and the different modules that shape them, a more capable domain-specific language has to be developed.

## 4.4 Implementation of the use case using Pyretic (standard POX-based client)

In this section, we summarize the technical details of the use case introduced in section 4.2 using the Pyretic programming language and backend. The target of this section is to 1. showcase the simplicity of this language for composing SDN modular applications, 2. give us a credit for considering Pyretic as a strong reference framework for our NetIDE architecture and 3. reveal any implementation drawbacks of the current Pyretic release which should be addressed by NetIDE towards a promising, unified, platform independent SDN platform.

Pyretic, as part of the Frenetic project, targets at providing simple, reusable and high-level abstractions for programming Software Defined Networks. Specifically, Pyretic is a runtime system that operates upon the OF protocol for SDN and the Python-based POX controller, serving a two-fold purpose.

First of all Pyretic is a programming language for packet processing that facilitates the composition and deployment of SDN applications. The key idea behind this is that network programmers do not need to deal with low-level implementation details such as message/event exchange between controller and switches or rule installation/modification/deletion actions. Instead, network programmers need a set of tools and primitives that can simply and intuitively describe the objectives (e.g. drop/forward/flood packets) of network administrators, so called policies, by acting upon the network dataplane. The most important part of this domain specific language is the provision of a set of operators that undertake the policy composition either sequentially or in parallel. This key feature enables reusing and integrating small, independent SDN modules to compose big policies; a critical contribution towards realistic SDN deployments.

Secondly, Pyretic is also a runtime system that acts as an intermediate between the high-level programmers interface and the underlying topology. In that sense, on the northbound side, the runtime speaks Pyretic while on the southbound side, a client is responsible to communicate with the SDN controller and the switches. The interface between Pyretic and the client (summarized in Section 4.5) have been designed to translate Pyretic instructions into OpenFlow messages and vice-versa. As already introduced in Section 3.4, an improved/extended version of this interface will provide the Intermediate Representation Format (IRF). Currently this client is a POX client that speaks OpenFlow but it can be extended to interact with any SDN controller as also presented in section Section 4.5.

The intermediate runtime system keeps track of the topology, the message and event exchange among all the entities as well as the dynamic policies specified by the application and can proactively or reactively update the dataplane.

The next two subsections give a technical, Pyretic-view of the first NetIDE use case and a gap analysis describing Pyretics strong points and limitations.

### 4.4.1 Implementation

The Data-Centre use case, as presented in Section 4.2, is a realistic first step to implement key SDN applications and identify which are the strong and weak points of Pyretic both as a programming language and a backend.

Topology-wise, in order to emulate the devices of this use case, we use mininet [29], a network emulator described in Section 2. This tool facilitates the deployment and interconnection of virtual devices, managed by SDN controllers and provides the infrastructure to Pyretic and other runtimes to act upon. Thus, the topology implementation is a unique piece of python code (i.e. file UC1_DataCenter.py), used by all the NetIDE SDN implementations as a common dataplane.

From the application point of view, the modular approach of Pyretic helps to implement different functionality of Figure 4.1 as small, independent modules and compose them accordingly to achieve the use case purposes. The core modules of this scenario take advantage of Pyretics class called DynamicPolicy. This is a key class, inherited by all the deployed network functions, which is dynamically enforced to the dataplane once the programmer instantiates it. A DynamicPolicy instance exposes self.policy object which is applying the programmers decisions once its value is overridden. If not modified by the programmer, this object drops the received packets by default.

Besides the implemented modules from our side, we extensively use a key module already provided by the Pyretic community, the Medium Access Control (MAC) learner, which enables the basic L2 connectivity of all the devices of the topology by learning the Address Resolution Protocol (ARP) addresses and forwarding accordingly. The following paragraph introduces the modules that compose this use case and the Pyretic programming style details. The modules are available in [76]:

- **Commons.py.** A python script that contains global variables which define the Internet Protocol (IP) prefixes of the network, as specified by the network operator, the protocols and ports used by the firewalls as well several other variables that parameterize the Pyretic modules such as the monitoring interval of the network Monitor. This script acts as glue between the topology and the application and is a single point of parameterization for the whole use case functionality.

- **Monitor.py.** The first pyretic module that comes aside to the UC functionality acts as a logger of the various network events. This module takes advantage of the three, basic Pyretic event handlers, namely:

  1. *packets()*: Handler that captures packet in events coming to the controller. This handler can be configured to return certain or all the packet header fields.
  2. *count_bytes()*: Handler that counts the received bytes in the controller. Based on the same philosophy as the first one, the programmer can specify the fields to be appeared, enabling a categorization of the counters based on several fields (e.g. Transmission Control Protocol (TCP) byte counters, port 80 byte counters).
  3. *count_packets()*: Similar to the previous counter, but returns the number of counted packets per category. Using a parallel composition (operator +) of the three modules above, Monitor gives useful information to the network administrator by printing those packet/byte counters and potentially the content of each packet received. In this module, policy object is a parallel addition of the three query objects:

     ```
     policy = PktQuery + PktCapture + ByteQuery
     ```

- **Firewall.py.** Following the same philosophy, Firewall module inherits DynamicPolicy and for any captured packet-in, it applies a static, spatially-oriented set of firewall rules. The first set defines the external firewall functionality that sits between Internet and DMZ and allows external requests (Internet-side) to access only DNS (User Datagram Protocol (UDP) at port 53) and Web (TCP at port 80) services at the DMZ. The rest of the traffic is blocked. The second set, specifies the internal firewall device functionality that sits between DMZ and Protected zone and allow only outgoing traffic. The most straightforward way to enforce the above rules is to create an object (i.e. Blocked) that filters all the blocked traffic by composing unions of rules (operator —) and then using if_() condition of Pyretic we can specify:

  ```
  if_(Blocked, drop, mac_learner()), where:
  Blocked = DNSToInternet | DNSToInternet_Reply | HTTPToInternet |
      HTTPToInternet_Reply
  ```

  This command applies drop action to the packets filtered by Blocked object and uses MAC learner to forward the remaining traffic.

- **LoadBalancer.py** This is the most complete and representative Pyretic module since it uses the most important Pyretic tools. First of all the Load Balancer needs to filter the received packet-in events keeping only the ones that need to be balanced (e.g. the outgoing direction). This is implemented by instantiating a packet() handler similar to the monitoring module, that gives the source IP addresses of the incoming packets. Then, for those packets, it applies a round-robin policy that splits the flows across to internal load balancing servers based on the source IP addresses. Packets from the same IP are redirected to the same server so as to avoid TCP retransmissions caused by multipath latencies. The modifications applied to the packets affect the source and destination IP fields only. Finally, the policy object is updated with the load balancing rules.

- **UC1_DC_NetManager.py** The last but not least module of this use case is the one that mostly highlights Pyretics strong points. In particular, this is the main module that acts as a composer of the Big UC Policy by calling appropriately all the above modules. Technically speaking, this piece of code specifies how pyretic behaves upon the different classes of packets. The following expression gives you the main idea on the way that SDN composition is handled by this language:

```
ARPPkt >> mac_learner() +
LB >> mac_learner() >> AccessControl +
Monitor(MonitoringInterval)
```

Specifically, for layer 2, ARP packets filtered by ARPPkt are sequentially composed with the MAC learning module. In layer 3, LB packets are passed through the same module to learn the next hops and then filtered by the firewall rules. Moreover, Monitor is constantly capturing all packets. Using the operator + we compose the three critical parts in parallel to achieve the goals of this use case.

In this section we achieved to implement a key use case (UC) using a modular, pure SDN language like Pyretic. Taking a look at the Pyretic code repository in [77], it is easy to understand that the major achievement is that all OF terminology is abstracted from the programmer and replaced by intuitive high-level set of expressions that specify the way that packets are handled. Together with the powerful composition operators that allow effectively reusing existing modules, these are the most prevalent contributions of Pyretic to the SDN world.

### 4.4.2 Gap analysis

From the above description of Pyretics details, we can easily capture the strong points of this domain language and runtime system. However, delving into the details of Pyretic we also gathered a very useful set of gaps that pose restrictions to the SDN developers. Towards a holistic and unified network management approach, NetIDE flags those gaps and takes the explicit stance to provide solutions that overcome these limitations. The following list briefly presents and discusses the aforementioned gaps:

1. Pyretic is a domain specific language that focuses on the design and implementation of efficient packet processing tools. Pyretic does not have the means to statically configure (e.g. assign IPs, masks, routes, etc.) the nodes of the topology before starting the application. This should is a task that should exist before starting any Pyretic application.

2. Pyretic does not provide the tools to easily program stateful behaviors in the network, such as a stateful firewall that keeps track of the open connections/sessions and applies different rules according to those connections (e.g. simply allow one-way communication between 2 nodes, h1 to h2 but not vice-versa). If we need such kind of functionality, we should build extra logic on top of Pyretic. The only construct currently available to approach this functionality is if_() but the programmer should take care of the way to use this, since the policy given as a 3rd argument is going to be applied to all the rest packets that fall out the 1st argument. This may cause inconsistencies and conflicts with other modules.

3. The basic version of Pyretic currently available cannot handle proactive rule installation that can speed up network performance in the case of the static firewalls of this use case. In that sense, initially, all packets go to the controller which then re-actively installs the rules to the switches, increasing the Round Trip Time (RTT) of the first packets per flow.

4. The Network Address Translation (NAT) functionality for the gateway router of this use case is something that also falls out of the scope of Pyretic since NAT devices do many more things than just L2 or L3 forwarding. For this reason, Pyretic can be used to implement basic, static NAT functionality such as IP and port modifications. Even in this simple case however, there is a prominent need to handle stateful connections, an issue already highlighted above.

## 4.5 Implementation of the use case on Pyretic (using a novel Ryu-based client)

Section 4.4 reports the results of an initial evaluation of Pyretic as programming language for the implementation of the Data-Centre use case (Section 4.2). However the current implementation of Pyretic it tightly coupled to POX OF controller. By implementing a Pyretic client for Ryu, we implement a proof-of-concept for an IRF based on the API exposed by the interface between Pyretic and the Pyretic backend running in the OF controller. We confirm that an OF controller with a different design concept can be used as a backend for Pyretic and thereby show that the coupling between Pyretic and the underlying OF is not as close and initially suspected.

The Ryu controller has two main aspects that make it attractive for the NetIDE framework. First, like POX, Ryu is written in Python language. Although both POX and Ryu use different implementations of the OF library, the methods used to build most of the common OF messages are very similar. This affinity between the two approaches can be leveraged to define the IRF.

Secondly, since Ryu natively supports multiple versions of the OF protocol[2], the implementation of a Ryu-based OF client provides a way to understand the requirements for NetIDE to support different versions of the OF protocol. In particular, this aspect will facilitate and improve the definition of the IRF as well as the implementation of the upper layers of the NetIDE architecture like the Network Application Engine (e.g. components like the interpreter and the resource manager).

### 4.5.1 Implementation

The implementation of the Pyretic client for Ryu can be summarized in the following steps: (i) implementation of a socket client that interfaces with the Pyretic backend, (ii) implementation of the handlers that intercept the messages coming from Pyretic and that must be forwarded to the switches and (iii) overriding of the handlers in Ryu for the common OF messages coming from the switches that must be forwarded to Pyretic.

However, this is not sufficient to start the Ryu client from Pyretic (or any other client different from POX). In fact, Pyretic is very tight to POX and neither the CLI arguments nor the initialization processes contemplate methods to specify a different controller as envisioned in Figure 4.3.

In the current version of Pyretic[3], the OpenFlow messages supported by the backend and by the OpenFlow client can be inferred by the method called "found_terminator" which is in charge of parsing the Pyretic's protocol.
On the Pyretic's backend side, we can find the parser for the messages coming from the network through the OF client and that should be passed to the Pyretic's upper layers. Messages are formed

[2]up to v1.4 at the time of writing
[3]commit 6d3fd862feb0b47255d3bd4081a960b70e057676 of May 2, 2014

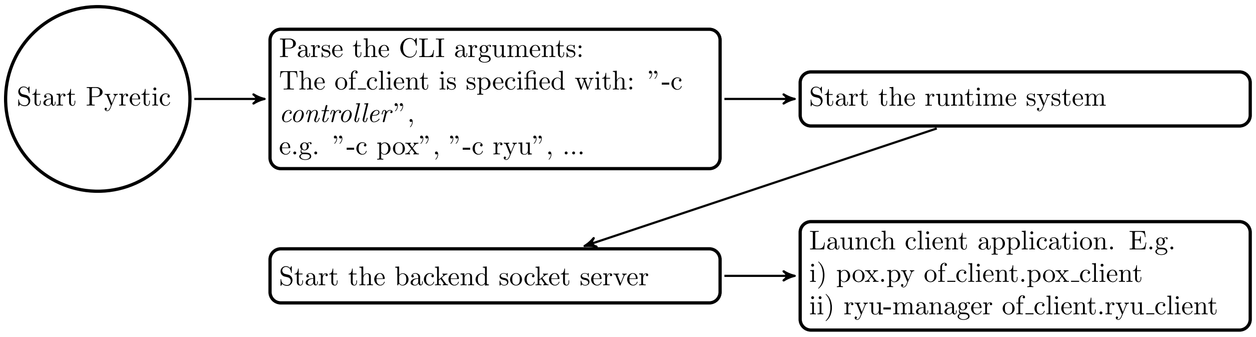

Start Pyretic
Parse the CLI arguments:
The of_client is specified with: "-c *controller*",
e.g. "-c pox", "-c ryu", ...
Start the runtime system
Start the backend socket server
Launch client application. E.g.
i) pox.py of_client.pox_client
ii) ryu-manager of_client.ryu_client

Figure 4.3: Pyretic's bootstrap operations.

tuples which first element is a string containing the name of the message. The supported OpenFlow messages are the following:

- **PacketIn.** The tuple sent by the OF client to the Pyretic's backend is ("packet", datapath_id, in_port, ethernet frame)
- **LLDP PacketIn.** The tuple is ("link", originator datapath_id, originator port, datapath_id, in_port)
- **FeaturesReply.** This message is used to pass the list of the available ports of each switch to Pyretic. This is achieved with a first tuple ("switch", "join", datapath_id, "BEGIN"), then the OF client sends one ("port", "join", datapath_id, port_no, port_config, port_state) for each port and finally ("switch", "join", datapath_id, "END")
- **Switch disconnected**: The client notifies Pyretic the disconnection of a switch from the controller with the tuple ("switch", "part", datapath_id)
- **FlowStatsReply.** The tuple is ("flow stats reply", datapath_id, flow_stat_dict). Where *flow_stat_dict* is a dictionary containing the flow statistics.
- **PortStatus.** The client notifies Pyretic a port status event with the tuple ("port", "join" / "mod" / "part", datapath_id, port_no, port_config, port_state) for ADD, MODIFY and REMOVE events respectively.

On the other direction, from Pyretic to the OF client, the following messages are implemented:

- **PacketOut.** The tuple received from Pyretic is ("packet", packet). Where *packet* is a dictionary that contains the datapath_id of the switch, the in_port and the out_port.
- **LLDP PacketOut messages.** LLDP packet are sent to a switch after receiving the following tuple ("inject_discovery_packet", datapath_id, port)
- **FlowMod.** FlowMod messages are handled with three different tuples, depending on the operations to be performed on the flow tables of the switch: add, delete and clear. To add an entry in the flow table the tuple ("install", pred, priority, action_list) is used, where pred is a dictionary containing the datapath_id, the match and the in_port. To remove an entry from the tables the tuple is ("delete", pred, priority). Finally, to clear the tables the tuple is ("clear", datapath_id). If the datapath_id is not specified, the flow tables of the whole network are emptied.
- **FlowStatsRequest.** Flow statistics requests are sent by Pyretic through the OF client with the tuple ("flow_stats_request", datapath_id).

### 4.5.2 Gap analysis

The analysis of the Pyretic's southbound APIs highlighted several gaps of their current implementation. These limitations are mainly due to the conceptual definition of the Pyretic's highlevel language that obviously influence the definition of the interface with the underlying OpenFlow clients. The following list summarizes the main gaps of the Pyretic's southbound APIs that NetIDE should fill for the implementation of the Intermediate Representation Format (IRF) as envisioned in Section 3.4:

1. The current version of the APIs do not allow the specification of neither idle nor hard timeouts for the installed flow entries. As a consequence, all the FlowMod messages are sent by the OF clients to the switches with the timeouts set to zero, which means that the flow entries never expire. Referring to the Data Center use case, the L2 switches cannot leverage on an aging mechanism for the MAC tables entries based on the idle timeout. In this case the only solution would be to use a timer at application level and leverage on the flow statistics to understand when a MAC address is no longer used.

2. Some OF messages are not implemented in the Pyretic's APIs. The list of non-supported OpenFlow messages is the following: aggregate flow, table, queue and port statistics, flow removed, port modification, queue configuration, read state and error message. For instance, the load balancer implementation of the Data center use case could have leveraged on the port statistics to apply certain optimization policies. Moreover, port statistics are often used to detect network congestions and re-route part of the traffic to alternative paths. Additionally, support for flow removed messages could also be used to detect when flows are no longer active in the network and, consequently, to update the state of MAC tables of learning switches or to adapt the firewall policies.

3. Further extensions could be necessary in case of clients supporting OF versions beyond 1.0 to either support the new functions or to embody those functions within code "blobs" as envisioned in Section 3.3.1.

## 4.6 Conclusions on Proof-of-Concept work

The initial practical work looked at the use case from three different points of view: 1. how well the implementation is suited to handle information to set up the scenario (i.e. *static behaviour* like the Layer-2 interconnection between elements in scenario), 2. how well the implementation is suited to express the *dynamic* behaviour of the different elements in the scenario (e.g. how a firewall implements state-full filtering), 3. how well the implementation is able to isolate information that will have the same meaning but will change from user to user (e.g. the IP prefix advertised by a router will change from user to user, but the router is expected to advertise an IP prefix always).

Our proof-of-concept implementations show that the base components of the NetIDE architecture cannot be completely implemented using off-the-shelf code and need NetIDE-specific extensions:

1. YANG is well suited for static configurations but can only be applied with limitations to express the dynamic behaviour we expect in Network Apps.

2. Pyretic as a high-level language needs some refinement regarding module support, static configurations and optimisation of the interaction between the controller and the network elements.

3. The interface between Pyretic and POX provides only a subset of the OF specification 1.0, which is valid as a starting point. However, this interface should be extended to become independent of the underlying protocol, be it proprietary or OpenFlow.

NetIDE will benefit from its easy portability to controllers different from POX, even written in other programming languages like Java (FloodLight, OpenDayLight, Beacon), C (Nox, Trema) or Ruby (Trema). A different analysis should be produced for non-OF control protocols (like Cisco's OnePK), whose APIs cannot easily be mapped to the Pyretic API.

Implementing the TCP-based Pyretic API on the Ryu controller yields following requirements for the IRF:

- **OpenFlow version agnosticity:** if we want to support as many controllers as possible, we will need to support (ideally) all versions of the OF protocol and handling of the different versions has to be done in a protocol agnostic way.
- **Implementation of support protocols:** Link Layer Discovery Protocol (LLDP) is an essential part of the Pyretic API. The IRF will need to provide means for including support/signalling protocols needed by high-level SDN languages (for example, to implement topology discovery or similar functionality).

With the implementations described in this chapter we have an initial proof of concept for the NetIDE architecture:

- we show that we are working on a concept that is controller-agnostic
- we show that YANG can be used as a possible base for the IRF, provided we define a way of including
- we provide work-package (WP) 3 with an example of an API that is used to handle the dynamic behaviour of an SDN application for their work on the IDE.

# 5 Next steps

In this final chapter, we discuss the technological alternatives we will be exploring, as well as some initial thoughts about where the discussion on the Intermediate Representation Format should lead us. With the work presented in this Deliverable and Deliverable D 3.1, we have laid the foundations for the NetIDE framework and the Intermediate Representation Format. Up to this point in time, we have concentrated on the Data-Centre use case. While we will continue using it as one of the guiding lines of our work, we will start implementing the other use cases described in the Description of Work, both to test the developed concepts and to incorporate new aspects they unveil.

## 5.1 Technological alternatives

In the immediate future, we will conclude the implementation of the concept shown in Figure 3.1b.We will implement the transformers from the IRF into Ryu and POX programs and test them against the use case. We also plan to investigate how to integrate the more static information provided by the YANG implementation with the behavioural descriptions provided by the Python programs into a unity. Additionally, we will explore how to enhance the support for modularity in the IRF in order to start the work on the IRF code repository.

In order to live up to the claim included in the DoW that NetIDE will provide a controller-agnostic framework, we will continue to integrate new SDN controllers using the Pyretic++ paradigm following the paradigm shown in Figure 3.2.

We have contacted the Pyretic development team and are in the way of establishing a good working relationship with them. As a result, we envisage to contribute the Ryu back-end to the Pyretic project. In this line of work, the project has also started to work on a similar back-end for the OpenDayLight controller framework. In addition, we also plan to evaluate other high-level SDN languages and constructs like Frenetic, in order to extract concepts that could be useful for the NetIDE architecture. With regards to Pyretic, our action plan includes exploring the implementation of back-end client API for other OF controllers and evaluating how far it can be used in proprietary SDN environments.

As one of our first targets, we envisage to work on step up our work on OpenDayLight for the following reasons:

- as already expressed in the SotA overview of ODL, it has grown to be a significant player in the SDN landscape the project targets,
- in contrast to our initial PoCs, which have been based on Python-based OpenFlow controllers, OpenDayLight is written in Java and will help us highlight programming language independent features and find ways around programming language dependencies,
- as Ryu, OpenDayLight supports more than one OF flavour, as well as managing legacy equipment.
- in addition to OpenFlow, OpenDayLight supports a proprietary SDN platform (Cisco OnePK), and
- ODL has become a major player in the SDN landscape, with a large pool of contributors; it is therefore an external effort the project might leverage upon.

In the short term, we plan to evaluate the Service Abstraction Layer it provides by trying to implement a Pyretic client for it. This Proof-of-Concept implementation will enhance our understanding of the external DSL we propose for NetIDE.

The SDN landscape is evolving so rapidly that we cannot completely anticipate new developments. However, we will continue to follow the evolution of OpenStack, because any integration with it should improve the general acceptance of the results of the project and because we do not rule out additional valuable inputs from it.

From an architectural point of view, we are starting to follow the discussions of the Open Networking Foundation (ONF) regarding their proposed architecture in general and their proposed North-bound Interface (NBI), which we will evaluate and assess the extent to which this NBI may find a place in the NetIDE architecture.

We also plan to integrate other SW tools that provide abstractions for different SDN solutions. As pointed out in Section 4.6, we need the IRF to become agnostic to OF flavours. The component in the NetIDE framework we target in out efforts to become SDN flavour-agnostic is the NetIDE interpreter (see Figure 3.2). In this line of work we are studying the way ODL claims to provide controller independence and following the evolution of tools like LoxiGen [60] that generate OF protocol libraries for different programming languages that are OpenFlow protocol version agnostic.

All these efforts target an architecture that is as independent of the underlying SDN networking layer as possible within the constraints of the project. We are presenting our work to different related communities, like we have done already at the Pyretic summer school 2014. We aim at a total adoption of the architecture that ensures its existence beyond the life time of the NetIDE project. This strategy is also applied to components developed by the project.

# A Topology YANG Module

```
module irf-topology {

  namespace "http://www.netide.eu/ns/irf-topology";

  prefix "irt";

  import ietf-inet-types {
    prefix "inet";
  }

  organization
    "NetIDE Project.";

  contact
    "Project Web: <http://www.netide.eu>

     Editor: Ladislav Lhotka <lhotka@nic.cz>";

  description
    "This module is a part of the Intermediate Representation Format
     (IRF) data model. It contains YANG definitions for specifying
     network topology.

     Copyright  2014 NetIDE Project Consortium.

     TBD: licence text";

  revision 2014-05-13 {
    description
      "Initial revision.";
  }

  /* Identities */

  identity node-type {
    description
      "Base identity form which specific node types are derived.";
  }

  identity host {
    base node-type;
    description
      "This identity represents a host.";
```

```
}

identity router {
  base node-type;
  description
    "This identity represents a router.";
}

identity switch {
  base node-type;
  description
    "This identity represents a switch.";
}

identity firewall {
  base node-type;
  description
    "This identity represents a firewall.";
}

identity dummy-node {
  base node-type;
  description
    "This identity represents an unspecified node or network.";
}

identity link-type {
  description
    "Base identity form which specific link types are derived.";
}

identity multi-access {
  base link-type;
  description
    "This identity represents a multi-access link.";
}

identity point-to-point {
  base link-type;
  description
    "This identity represents a point-to-point link.";
}

/* Groupings */

grouping description-leaf {
  leaf description {
    type string;
    description
```

```
        "Textual description of an object.";
    }
  }

  grouping capa-set {
    description
      "This grouping defines a recursive specification of a set of
       capabilities.";
    leaf-list include-set {
      type leafref {
        path "/irt:network/irt:capability-set/irt:name";
      }
    }
    leaf-list capability {
      description
        "List of URIs representing node's capabilities including

         - NETCONF capabilities,

         - supported YANG modules, using the URI format specified in
           RFC6020, sec. 5.6.4.";
      type inet:uri;
    }
  }

  grouping common-data {
    description
      "Leafs that are common for all objects (nodes, links, etc.).";
    leaf name {
      type string {
        length "1..max";
      }
      description
        "Name of the object, also serves as a list key.";
    }
    uses description-leaf;
  }

  grouping address-and-port {
    description
      "Device address and optional port.";
    leaf address {
      mandatory "true";
      type inet:host;
      description
        "Domain name or IP address of the device.";
    }
    leaf port {
      type inet:port-number;
      description
        "Port on which the device listens.";
    }
  }

  /* Data */

  container network {
    description
      "Specification of a network consisting of a list of nodes and a
       list of links.";
    uses description-leaf;
    list capability-set {
      key "name";
      description
        "Each entry of this list defines a set of capabilities that
         can be referred to from nodes and thus become part of their
         data model.";
      uses common-data;
      uses capa-set;
    }
    list node {
      key "name";
      description
        "Each entry of this list describes a network node (host,
         router, switch etc.).";
      uses common-data;
      leaf type {
        mandatory "true";
        type identityref {
          base node-type;
        }
        description
          "Type of the node.";
      }
      leaf brand {
        type string;
        description
          "Identification of the manufacturer.";
      }
      leaf model {
        type string;
        description
          "Identification of the model.";
      }
      container operating-system {
        description
          "Identification of the operating system.";
        leaf name {
          type string;
          description
            "Name of the operating system.";
        }
        leaf version {
          type string;
          description
            "Operating system version.";
        }
      }
      container configuration {
        when "../type != 'irt:dummy-node'";
        description
          "Parameters of node configuration.";
        container methods {
          description
            "Specification of configuration methods supported by the
             device.";
          container netconf {
            description
              "NETCONF parameters.";
            container ssh {
              presence "Support for NETCONF over SSH";
              description
                "Indicates support and provides parameters for
                 NETCONF over SSH.";
              uses address-and-port {
                refine "port" {
                  default "830";
                }
              }
            }
            container tls {
              presence "Support for NETCONF over TLS";
              description
                "Indicates support and provides parameters for
                 NETCONF over TLS.";
              uses address-and-port {
                refine "port" {
                  default "6513";
                }
              }
            }
          }
          container openflow {
            presence "Support for OpenFlow";
            description
              "Indicates support and provides parameters for
               OpenFlow.";
```

```
    uses address-and-port {
      refine "port" {
        default "6653";
      }
    }
  }
  container cli {
    description
      "CLI parameters.";
    container ssh {
      presence "CLI via SSH";
      description
        "Indicates that the device's command-line interface
         is accessible through SSH.";
      uses address-and-port {
        refine "port" {
          default "22";
        }
      }
    }
  }
}
container capabilities {
  description
    "Collection of capabilities supported by the node.

     It can be defined as a union of individual capabilities
     and/or predefined capability-sets.";
  uses capa-set;
}
leaf file-name {
  type string;
  description
    "Name of the config file (inside the 'node-conf'
     subdirectory).

     By default, the file name is the same as the name of the
     node with an extension '.xml' or '.json', depending on
     the 'format' parameter.";
}
leaf format {
  type enumeration {
    enum XML;
    enum JSON;
  }
  default "XML";
  description
    "Format of the config file.";
}
```

```
      }
      list interface {
        key "name";
        min-elements "1";
        description
          "Every node has one or more interfaces.

           The 'name' key must be the same as the name that is used
           for the interface in the config file (this doesn't apply
           to dummy nodes).";
        uses common-data;
        leaf connection {
          type leafref {
            path "../../../link/name";
          }
          description
            "The link to which the interface is connected.";
        }
      }
    }
    list link {
      key "name";
      description
        "Each network has zero or more links.";
      uses common-data;
      leaf type {
        type identityref {
          base link-type;
        }
        default "irt:point-to-point";
        description
          "Type of the link.";
      }
    }
  }
}
```

# B Pyretic Module

## B.1 Main

```
#!/usr/bin/python

####################################################################################################
###       Name: UC1_DC_NetManager.py
###     Author: Georgios Katsikas - katsikas@imdea.org
### Description: Pyretic Implementation of NetIDE UC1 - Main Module
####################################################################################################

import os

# Pyretic libraries
from pyretic.lib.std   import *
from pyretic.lib.corelib import *

# Generic Pyretic Modules
from pyretic.modules.Commons      import *
from pyretic.modules.Monitor      import Monitor
from pyretic.modules.Firewall     import Firewall
from pyretic.modules.mac_learner  import mac_learner
from pyretic.modules.LoadBalancer import LoadBalancer

### Main class for UC1 DataCenter Implementation
class UC1_DC_NetManager(DynamicPolicy):
       def __init__(self):
              super(UC1_DC_NetManager, self).__init__()
              self.FW = Firewall()
              self.policy = None

              # Initialize and Start
              self.SetInitialState()

       # Initial configuration of DC Application
       def SetInitialState(self):
              # LB configuration
              self.LB_Device = LB_Device
              self.PublicIP = PublicIP
              self.ServerIPs = [LB_Server_1, LB_Server_2]
              self.ClientIPs = [ipp2, ipp3, ipp4]

              # Firewall configuration
              self.FWDevices = [Firewall_1, Firewall_2]

              return self.Start()

       # Dynamically update enforced policy based on the last values of all the modules
```

```
        def Start(self):
                # Handle ARP
                ARPPkt = match(ethtype=ARP_TYPE)

                # Instantiate Firewalls
                AccessControl = self.FW.ApplyFirewall()

                # Instantiate Load Balancer
                LB  = LoadBalancer(self.LB_Device, self.ClientIPs, self.ServerIPs,
                    self.PublicIP)

                self.policy =  (
                                        ( ARPPkt >> mac_learner() ) + # L2 Learning SWs
                                        ( LB >> mac_learner() >> AccessControl ) + # LB + FWs
                                        Monitor(MonitoringInterval)  # Monitoring
                               )

                return self.policy

################################################################################
### Bootstrap Use Case
################################################################################
def main():
        return UC1_DC_NetManager()
```

## B.2 Load Balancer

```
#!/usr/bin/python

################################################################################################
###       Name: LoadBalancer.py
###     Author: Omid Alipourfard - omida@cs.princeton.edu
###     Editor: Georgios Katsikas - katsikas@imdea.org
### Description: Round Robin Load Balancer
################################################################################################

# Pyretic libraries
from pyretic.lib.std   import *
from pyretic.lib.query import *
from pyretic.lib.corelib import *

################################################
# Translate from
#   client -> public address : client -> server
#   server -> client : public address -> client
################################################
def Translate(c, s, p):
        cp = match(srcip=c, dstip=p)
        sc = match(srcip=s, dstip=c)

        return ((cp >> modify(dstip=s)) +
                (sc >> modify(srcip=p)) +
                (~cp & ~sc))
```

```
#################################################################
# Simple round-robin load balancing policy                     #
#                                                              #
# This implementation will drop the first packet of each flow. #
# An easy fix would be to use network.inject_packet to send the #
# packet to its final destination.                             #
#################################################################
class LoadBalancer(DynamicPolicy):
       def __init__(self, Device, Clients, Servers, PublicIP):
              super(LoadBalancer, self).__init__()
              #print("[Load Balancer]: Device ID: %s" %(Device))
              #print("[Load Balancer]: Server addresses: %s %s" %(Servers[0],
                  Servers[1]))

              self.Device   = Device
              self.Clients  = Clients
              self.Servers  = Servers
              self.PublicIP = PublicIP
              self.Index    = 0

              # Start a packet query
              self.Query    = packets(1, ['srcip'])
              # Handle events using callback function
              self.Query.register_callback(self.LoadBalancingPolicy)

              # Capture packets that arrive at LB and go to Internet
              self.Public_to_Controller = (match(dstip=self.PublicIP,
                  switch=self.Device)>> self.Query)
              self.LB_Policy = None
              self.policy   = self.Public_to_Controller

       def UpdatePolicy(self):
              self.policy = self.LB_Policy + self.Public_to_Controller

       def LoadBalancingPolicy(self, pkt):
              Client = pkt['srcip']

              # Be careful not to redirect servers on themselves
              if Client in self.Servers: return

              # Round-robin, per-flow load balancing
              Server = self.NextServer()
              p = Translate(Client, Server, self.PublicIP)
              print("[Load Balancer]: Mapping c:%s to s:%s" % (Client, Server))

              # Apply the modifications
              if self.LB_Policy:
                     self.LB_Policy = self.LB_Policy >> p
              else:
                     self.LB_Policy = p

              # Update LB policy object
              self.UpdatePolicy()

       # Round-robin
```

```
        def NextServer(self):
                Server = self.Servers[self.Index % len(self.Servers)]
                self.Index += 1
                return Server
```

## B.3 Firewalls

```
#!/usr/bin/python

################################################################################################
###       Name: Firewall.py
###     Author: Georgios Katsikas - katsikas@imdea.org
### Description: Firewall configuration for the two firewalls of UC1
################################################################################################

import os

# Pyretic libraries
from pyretic.lib.std   import *
from pyretic.lib.corelib import *

# Pyretic modules
from pyretic.modules.Commons  import *
from pyretic.modules.mac_learner import mac_learner

class Firewall(DynamicPolicy):
        def __init__(self):
                super(Firewall, self).__init__()

                # Initial policy objects are empty
                self.Blocked = None
                self.Allowed = None
                self.policy = None

        def ApplyFirewall(self):
                # Set rules to devices
                ac1 = self.ConfigureFW1()
                ac2 = self.ConfigureFW2()

                # Update block object
                if self.Blocked:
                        self.Blocked = self.Blocked | ac1 | ac2
                else:
                        self.Blocked = ac1 | ac2

                # The packets tha match Block object are dropped. The rest are forwarded.
                self.Allowed = if_(self.Blocked, drop, mac_learner())

                # Update policy object
                return self.UpdatePolicy()

        def UpdatePolicy(self):
                self.policy = self.Allowed
        return self.policy

    def ConfigureFW1(self):
        # Only UDP at port 53
        ICMPtoDNS      =      (
                                    match(ethtype=IP, protocol=ICMP,
                                        switch=Firewall_1, srcip=ipp1, dstip=ipp2)
                              )
        TCPtoDNS       =      (
                                    match(ethtype=IP, protocol=TCP,
                                        switch=Firewall_1, srcip=ipp1, dstip=ipp2)
                              )
        DNSToInternet  =      (
                                    match(ethtype=IP, protocol=UDP,
                                        switch=Firewall_1, srcip=ipp1,
                                        dstip=ipp2) & ~match(dstport=DNS_PORT)
                              )

        DNSToInternet_Reply =  (
                                    match(ethtype=IP, protocol=UDP,
                                        switch=Firewall_1, srcip=ipp2,
                                        dstip=ipp1) & ~match(srcport=DNS_PORT)
                              )

        # Only TCP at port 80
        ICMPtoWeb      =      (
                                    match(ethtype=IP, protocol=ICMP,
                                        switch=Firewall_1, srcip=ipp1, dstip=ipp3)
                              )

        UDPtoWeb       =      (
                                    match(ethtype=IP, protocol=UDP,
                                        switch=Firewall_1, srcip=ipp1, dstip=ipp3)
                              )

        HTTPToInternet =      (
                                    match(ethtype=IP, protocol=TCP,
                                        switch=Firewall_1, srcip=ipp1,
                                        dstip=ipp3) & ~match(dstport=HTTP_PORT)
                              )

        HTTPToInternet_Reply = (
                                    match(ethtype=IP, protocol=TCP,
                                        switch=Firewall_1, srcip=ipp3,
                                        dstip=ipp1) & ~match(srcport=HTTP_PORT)
                              )

        # Compose the above rules --> FW1 functionality
        p =     (
                        DNSToInternet | DNSToInternet_Reply | ICMPtoDNS | TCPtoDNS |
                        HTTPToInternet | HTTPToInternet_Reply | ICMPtoWeb | UDPtoWeb
                )

        return p

    def ConfigureFW2(self):
```

```
			# Incoming is blocked --> FW2 functionality
			p =	(
					match(ethtype=IP, dstip=ipp4, switch=Firewall_2)
				)

			return p
```

## B.4 Monitor

```
#!/usr/bin/python

################################################################################################
###       Name: Monitor.py
###     Author: Georgios Katsikas - katsikas@imdea.org
### Description: Module that gathers and prints useful statistics (e.g. pkt/byte
    counters)
################################################################################################

import os

# Pyretic libraries
from pyretic.lib.corelib import *
from pyretic.lib.std import *
from pyretic.lib.query import *

# Pyretic modules
from pyretic.modules.Commons import *

### Monitoring Module for DC Topology ###
class Monitor(DynamicPolicy):
	# Monitor Constructor
	def __init__(self, monitoringInterval):
		super(Monitor, self).__init__()
		self.MonitoringInterval = monitoringInterval
		self.SetInitialState()

	# Start monitoring
	def SetInitialState(self):
		#self.ByteQuery = self.ByteCounts()
		self.PktQuery  = self.PacketCounts()
		self.PktCapture = self.PacketInsector()
		self.UpdatePolicy()

	# Dynamically Update Policy
	def UpdatePolicy(self):
		self.policy = self.PktQuery + self.PktCapture # + self.ByteQuery

	# Prints counted packets
	def PacketCountPrinter(self, PktCounts):
		print("------------------------ Packet Counts ------------------------")
		for k, v in sorted(PktCounts.items()):
			print u'{0}: {1} pkts'.format(k, v)
		print("---------------------------------------------------------------")
```

```
# Counts packets every second
def PacketCounts(self):
        q = count_packets(self.MonitoringInterval, ['srcip', 'switch', 'protocol'])
        q.register_callback(self.PacketCountPrinter)
        return q

# Prints counted bytes
def ByteCountPrinter(self, ByteCounts):
        print("------------------------- Packet Bytes ------------------------")
        for k, v in sorted(ByteCounts.items()):
                print u'{0}: {1} bytes'.format(k, v)
        print("---------------------------------------------------------------")

# Counts bytes every second
def ByteCounts(self):
        q = count_bytes(self.MonitoringInterval, ['srcip', 'switch', 'protocol'])
        q.register_callback(self.ByteCountPrinter)
        return q

# Packet capture
def PacketInsector(self):
        q = packets(1, ['srcip', 'dstip', 'switch', 'protocol', 'ethtype'])
        q.register_callback(self.PacketPrinter)
        return q

# Prints captured packet
def PacketPrinter(self, pkt):
        print "------------------------- Packet Content ------------------------"

        # Capture Ethernet IP + ICMP/UDP/TCP
        if pkt['ethtype'] == IP_TYPE:
                print "Ethernet packet"
                raw_bytes = [ord(c) for c in pkt['raw']]
                print "Ethernet payload is %d" % pkt['payload_len']
                eth_payload_bytes = raw_bytes[pkt['header_len']:]
                print "Ethernet payload is %d bytes" % len(eth_payload_bytes)
                ip_version = (eth_payload_bytes[0] & 0b11110000) >> 4
                ihl = (eth_payload_bytes[0] & 0b00001111)
                ip_header_len = ihl * 4
                ip_payload_bytes = eth_payload_bytes[ip_header_len:]
                ip_proto = eth_payload_bytes[9]
                print "IP Version = %d" % ip_version
                print "IP Header_len = %d" % ip_header_len
                print "IP Protocol = %d" % ip_proto
                print "IP Payload is %d bytes" % len(ip_payload_bytes)

                # Number 6 is TCP
                if ip_proto == TCP:
                        print "TCP packet"
                        tcp_data_offset = (ip_payload_bytes[12] & 0b11110000) >> 4
                        tcp_header_len = tcp_data_offset * 4
                        print "TCP Header Length = %d" % tcp_header_len
                        tcp_payload_bytes = ip_payload_bytes[tcp_header_len:]
                        print "TCP Payload is %d bytes" % len(tcp_payload_bytes)
                        if len(tcp_payload_bytes) > 0:
```

```
                                   print "Payload:\t",
                                   print ''.join([chr(d) for d in tcp_payload_bytes])
                     # Number 17 is UDP
                     elif ip_proto == UDP:
                            print "UDP Packet"
                            udp_header_len = 8
                            print "UDP Header Length = %d" % udp_header_len
                            udp_payload_bytes = ip_payload_bytes[udp_header_len:]
                            print "UDP Payload is %d bytes" % len(udp_payload_bytes)
                            if len(udp_payload_bytes) > 0:
                                   print "Payload:\t",
                                   print ''.join([chr(d) for d in udp_payload_bytes])
                     # Number 1 is ICMP
                     elif ip_proto == ICMP:
                            print "ICMP packet"
                            print pkt
                     else:
                            print "Unhandled IP packet type"
              # Capture Ethernet ARP
              elif pkt['ethtype'] == ARP_TYPE:
                     print "ARP packet"
                     print pkt
              else:
                     print "Unhandled packet type"
              print "-----------------------------------------------------------------"
```

## B.5 Commons

```
#############################################################################################
###       Name: Commons.py
###     Author: Georgios Katsikas - katsikas@imdea.org
### Description: Global variables for Pyretic Modules
#############################################################################################

# Pyretic libraries
from pyretic.lib import *
from pyretic.lib.corelib import *
from pyretic.lib.std import *

################### IP Setup ##################
# Internet side
ipp1 = IPPrefix('10.0.0.0/24')
# DNS Server
ipp2 = IPAddr('10.0.1.17')
# Web Server
ipp3 = IPAddr('10.0.1.18')
# Intranet
ipp4 = IPAddr('10.0.1.32')

# Load Balancer configuration (Internet side)
LB_Server_1 = IPAddr('10.0.0.1')
LB_Server_2 = IPAddr('10.0.0.2')
PublicIP   = IPAddr('10.0.0.100')
```

```
################################################

# Middleboxes' IDs
Firewall_1 = 9
Firewall_2 = 10
LB_Device = 4

# Protocols
ICMP = 1
TCP  = 6
UDP  = 17
IP   = 0x0800

# Allowed ports
DNS_PORT = 53
HTTP_PORT = 80

# Messages
ERROR = -1

# Monitoring Interval period (in seconds)
MonitoringInterval = 5
```

# C Bibliography

[1] The NetIDE consortium. D3.1 - Developer Toolkit Specification. Technical report, The European Commission, 2014.

[2] The NetIDE consortium. Preliminary Market Analysis. Technical report, The European Commission, 2014.

[3] OpenVSwitch. `http://openvswitch.org/development/openflow-1-x-plan`.

[4] Pica8 Open Network Fabric. `http://www.pica8.org/solutions/openflow.php`.

[5] Indigo - Open Source OpenFlow Switches. `http://www.openflowhub.org/display/Indigo/Indigo+-+Open+Source+OpenFlow+Switches`.

[6] About NOX. `http://www.noxrepo.org/nox/about-nox/`.

[7] About pox. `http://www.noxrepo.org/pox/about-pox/`. Last visited: Sat, 10 Nov 2012.

[8] Floodlight: a Java-based OpenFlow Controller. `http://floodlight.openflowhub.org/`.

[9] Trema: Full-Stack OpenFlow Framework in Ruby and C. `https://github.com/trema/`.

[10] HIDEyuki Shimonishi and Yasuhito Takamiya and Yasunobu Chiba and Kazushi Sugyo and Youichi Hatano and Kentaro Sonoda and Kazuya Suzuki and Daisuke Kotani and Ippei Akiyoshi. Programmable Network Using OpenFlow for Network Researches and Experiments. In *Proceedings of the Sixth International Conference on Mobile Computing and Ubiquitous Networking*, pages 168–171. Information Processing Society of Japan, May 2012.

[11] Angela Orebaugh, Gilbert Ramirez, Josh Burke, and Larry Pesce. *Wireshark & Ethereal Network Protocol Analyzer Toolkit (Jay Beale's Open Source Security)*. Syngress Publishing, 2006.

[12] ProgrammableFlow Networking. `http://www.necam.com/SDN/`, month 2014.

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

[14] Open Networking Foundation. Of-config 1.2: Openflow management and configuration protocol. Onf specification, ONF, 2014.

[15] R. Enns, M. Bjorklund, J. Schoenwaelder, and A. Bierman. Network Configuration Protocol (NETCONF). RFC 6241 (Proposed Standard), June 2011.

[16] B. Claise. Cisco Systems NetFlow Services Export Version 9. RFC 3954 (Informational), October 2004.

[17] J.D. Case, M. Fedor, M.L. Schoffstall, and J. Davin. Simple Network Management Protocol (SNMP). RFC 1157 (Historic), May 1990.

[18] B. Pfaff and B. Davie. The Open vSwitch Database Management Protocol. RFC 7047 (Informational), December 2013.

[19] Python packet creation / parsing library. https://code.google.com/p/dpkt/, 2013.

[20] Y. Rekhter, T. Li, and S. Hares. A Border Gateway Protocol 4 (BGP-4). RFC 4271 (Draft Standard), January 2006. Updated by RFCs 6286, 6608, 6793.

[21] R. Hinden. Virtual Router Redundancy Protocol (VRRP). RFC 3768 (Draft Standard), April 2004. Obsoleted by RFC 5798.

[22] Using Ryu Network Operating System with OpenStack as Network controller. https://github.com/osrg/ryu/wiki/OpenStack.

[23] David Erickson. Beacon Home. https://openflow.stanford.edu/display/Beacon/Home, 2012. Last visited: Sun, 18 Nov 2012.

[24] What are SDN Northbound APIs. http://www.sdncentral.com/north-bound-interfaces-api, 2014. note.

[25] Sukhveer Kaur, Japinder Singh, and Navtej Singh Ghumman. Network Programmability Using POX Controller. *ICCCS Conference Proceedings*, 2014. Downloaded from http://www.sbsstc.ac.in/icccs2014/Papers/Paper28.pdf.

[26] Kuang-Ching Wang. How to write a module - Floodlight Controller. http://docs.projectfloodlight.org/display/floodlightcontroller/How+to+Write+a+Module, Dec 2013.

[27] Your First Beacon Bundle. https://openflow.stanford.edu/display/Beacon/Your+First+Bundle, Aug 2013.

[28] Frank Drr. Developing OSGi components for OpenDayLight. www.frank-durr.de/?p=84, Jan 2014.

[29] Bob Lantz, Brandon Heller, and Nick McKeown. A network in a laptop: rapid prototyping for software-defined networks. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, Hotnets-IX, pages 19:1–19:6, New York, NY, USA, 2010. ACM.

[30] Mininet at github. https://github.com/mininet/mininet, 2010.

[31] Universit Roma III, Compunet Lab. The poor man's system to experiment computer networking. http://wiki.netkit.org/index.php/Main_Page, May 2011. Last visit: 06 Jun 2014.

[32] Quagga Software Routing Suite. http://www.nongnu.org/quagga/, March 2013. Last visit: 06 Jun 2014.

[33] Sam Burnett. Getting Resonance and OpenFlow to Work with Netkit. http://www.cc.gatech.edu/~sburnett/posts/2010-05-20-resonance-netkit.html, May 2010. Last visit: 06 Jun 2014.

[34] ns3. http://www.nsnam.org. Last visited: Fri, 09 Nov 2012.

[35] Josh Pelkey. Openflow software implementation distribution. http://code.nsnam.org/jpelkey3/openflow, Apr 2011. Last visit: Tue, 18 Dec 2012.

[36] Rob Sherwood, Glen Gibb, Kok-Kiong Yap, Guido Appenzeller, Martin Casado, Nick McKeown, and Guru M. Parulkar. Can the production network be the testbed? In *OSDI*, pages 365–378, 2010.

[37] RouteFlow Project: IP routing on SDN. `https://sites.google.com/site/routeflow/`.

[38] Timothy L. Hinrichs, Natasha S. Gude, Martin Casado, John C. Mitchell, and Scott Shenker. Practical declarative network management. In *Proceedings of the 1st ACM workshop on Research on enterprise networking*, WREN '09, pages 1–10, New York, NY, USA, 2009. ACM.

[39] Andreas Voellmy, Hyojoon Kim, and Nick Feamster. Procera: a language for high-level reactive network control. In *Proceedings of the first workshop on Hot topics in software defined networks*, HotSDN '12, pages 43–48, New York, NY, USA, 2012. ACM.

[40] Nate Foster, Rob Harrison, Michael J. Freedman, Christopher Monsanto, Jennifer Rexford, Alec Story, and David Walker. Frenetic: a network programming language. *SIGPLAN Not.*, 46(9):279–291, September 2011.

[41] Joshua Reich, Christopher Monsanto, Nate Foster, Jennifer Rexford, and David Walker. Modular SDN Programming with Pyretic. *USENIX ;login*, 38(5):128–134, Oct. 2013.

[42] Andreas Voellmy and Paul Hudak. Nettle: Taking the sting out of programming network routers. In Ricardo Rocha and John Launchbury, editors, *PADL*, volume 6539 of *Lecture Notes in Computer Science*, pages 235–249. Springer, 2011.

[43] K. Watsen A. Bierman, M. Bjorklund and R. Fernando. Restconf protocol. Internet-Draft draft-ietf-netconf-restconf-00, IETF, Mar 2014.

[44] M. Bjorklund. YANG - A Data Modeling Language for the Network Configuration Protocol (NETCONF). RFC 6020 (Proposed Standard), October 2010.

[45] L. Lhotka. Json encoding of data modeled with yang. Internet-Draft draft-ietf-netmod-yang-json-00, IETF, Apr 2014.

[46] IEEE. Information technology – document schema definition language (dsdl). part 2: Regular-grammar-based validation – relax ng. second edition. International Standard ISO/IEC 19757-2, IEEE, Dec 2008.

[47] P. V. Biron and A. Malhotra. Xml schema part 2: Datatypes second edition. W3C Recommendation REC-xmlschema-2-20041028, World-Wide Web Consortium, Oct 2004.

[48] M. Bjorklund. A YANG Data Model for Interface Management. RFC 7223 (Proposed Standard), May 2014.

[49] M. Bjorklund. A yang data model for ip management. Internet-Draft draft-ietf-netmod-ip-cfg-14, IETF, Mar 2014.

[50] A. Bierman. A yang data model for system management. Internet-Draft draft-ietf-netmod-system-mgmt-14, IETF, May 2014.

[51] L. Lhotka. A yang data model for routing management. Internet-Draft draft-ietf-netmod-routing-cfg-14, IETF, May 2014.

[52] Nikhil Handigol, Brandon Heller, Vimalkumar Jeyakumar, David Maziéres, and Nick McKeown. Where is the debugger for my software-defined network? In *Proceedings of the first workshop on Hot topics in software defined networks*, HotSDN '12, pages 55–60, New York, NY, USA, 2012. ACM.

[53] Richard Stallman, Roland Pesch, and Stan Shebs. *Debugging with GDB: the source level debugger*. GNU Press, Boston, MA, USA, 2002.

[54] Openflow wireshark disector. `http://www.openflow.org/wk/index.php/OpenFlow_Wireshark_Dissector`, 2011. Last visited: Fri, 09 Nov 2012.

[55] Yasunobu Chiba and Yasunori Nakazawa. tremashark: A bridge for printing various events on Wireshark. `git://github.com/trema/trema.gitmaster/tremashark`, 2011. Last visited: Fri, 09 Nov 2012.

[56] Andreas Wundsam, Dan Levin, Srini Seetharaman, and Anja Feldmann. Ofrewind: enabling record and replay troubleshooting for networks. In *Proceedings of the 2011 USENIX conference on USENIX annual technical conference*, USENIXATC'11, pages 29–29, Berkeley, CA, USA, 2011. USENIX Association.

[57] Ehab Al-Shaer and Saeed Al-Haj. FlowChecker: configuration analysis and verification of federated openflow infrastructures. In *Proceedings of the 3rd ACM workshop on Assurable and usable security configuration*, SafeConfig '10, pages 37–44, New York, NY, USA, 2010. ACM.

[58] Marco Canini, Daniele Venzano, Peter Perešíni, Dejan Kostić, and Jennifer Rexford. A NICE way to test openflow applications. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, NSDI'12, pages 10–10, Berkeley, CA, USA, 2012. USENIX Association.

[59] Marco Canini, Daniele Venzano, Peter Perešíni, Dejan Kostić, and Jennifer Rexford. nice-of - A NICE way to test OpenFlow controller applications - Google Project Hosting . `http://code.google.com/p/nice-of/`.

[60] Dan Talayco, Ed Swierk, Jeffrey Townsend, Ken Chiang, Rich Lane, Rob Sherwood, and Shudong Zhou. LoxiGen. `https://github.com/floodlight/loxigen`, Jun 2013.

[61] The NetIDE Consortium. NetIDE - An integrated development environment for portable network applications. Annex I: Description of work 619543, The European Commission, Nov 2013.

[62] Federico M. Facca and Elio Salvadori and Holger Karl and Diego R. López and Pedro Andrés Aranda Gutiérrez and Dejan Kostic and Roberto Riggio. NetIDE: First Steps towards an Integrated Development Environment for Portable Network Apps. *EWSDN-2013 Second European Workshop on Software Defined Networks*, 0:105–110, 2013.

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

[64] Josh Juneau, Jim Baker, Frank Wierzbicki, Leo Soto, and Victor Ng. *The Definitive Guide to Jython: Python for the Java Platform*. Apress, Berkely, CA, USA, 1st edition, 2010.

[65] Anirudh Ramachandran. App Development Tutorial - GENI. http://groups.geni.net/geni/raw-attachment/wiki/GEC19Agenda/IntroToOFOpenDaylight/OpenDaylight-app-development-tutorial.pdf. Last visit: 06 Jun 2014.

[66] Teemu Koponen, Martin Casado, Natasha Gude, Jeremy Stribling, Leon Poutievski, Min Zhu, Rajiv Ramanathan, Yuichiro Iwata, Hiroaki Inoue, Takayuki Hama, and Scott Shenker. Onix: A distributed control platform for large-scale production networks. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–6, Berkeley, CA, USA, 2010. USENIX Association.

[67] Soheil Hassas Yeganeh and Yashar Ganjali. Kandoo: A framework for efficient and scalable offloading of control applications. In *Proceedings of the First Workshop on Hot Topics in Software Defined Networks*, HotSDN '12, pages 19–24, New York, NY, USA, 2012. ACM.

[68] Junos XML API and Junos XML Management Protocol Overview. `http://www.juniper.net/techpubs/en_US/junos13.3/topics/concept/junos-script-automation-junos-xml-protocol-and-api-overview.html`, Nov 2013. Last visit: 29 Apr 2014.

[69] Cisco's One Platform Kit (onePK). `http://www.cisco.com/c/en/us/products/ios-nx-os-software/onepk.html`, May 2014.

[70] Charles Clos. A study of non-blocking switching networks. In *Bell System Technical Journal*, pages 406–424. March 1953. doi:10.1002/j.1538-7305.1953.tb01433.x.

[71] Interroute. What is Infrastructure as a Service? `http://www.interoute.com/what-iaas`, Jan 2013. Last visit: 14 May 2014.

[72] Interroute. What is Platform as a Service? `http://www.interoute.com/what-paas`, Jan 2013. Last visit: 14 May 2014.

[73] Sridhar Karnam. Anything-as-a-Service (XaaS): Future of cloud computing. `http://sridharkarnam.com/2010/06/17/anything-as-a-service-xaas-future-of-cloud-computing/`, June 2010. Last visit: 14 May 2014.

[74] Christopher Monsanto, Joshua Reich, Nate Foster, Jennifer Rexford, and David Walker. Composing software defined networks. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 1–13, Lombard, IL, 2013. USENIX.

[75] A. Clemm, H. Ananthakrishnan, J. Medved, T. Tkacik, R. Varga, and N. Bahadur. A yang data model for network topologies. Internet-Draft draft-clemm-i2rs-yang-network-topo-00, IETF, Feb 2014.

[76] NetIDE Software Repository. `http://redmine.netide.eu/`.

[77] The Pyretic runtime system. `https://github.com/frenetic-lang/pyretic`, may 2014.