ALIEN

Abstraction Layer for Implementation of Extensions in programmable Networks

Collaborative project co-funded by the European Commission within the Seventh Framework Programme

Grant agreement no: 317880
Project acronym: ALIEN
Project full title: Abstraction Layer for Implementation of Extensions in programmable Networks
Project start date: 01/10/12
Project duration: 24 months

Deliverable D2.3: Report on Implementation of the Common Part of an OpenFlow Datapath Element and the Extended FlowVisor

Due date: 31/05/2014
Submission date: 12/06/2014
Editor: Umar Toseef (EICT)
Internal reviewer: Reza Nejabati (UNIVBRIS)
Author list: Umar Toseef, Adel Zaalouk, Kostas Pentikousis (EICT), Artur Binczewski, Bartosz Belter, Łukasz Ogrodowczyk, Iwo Olszewski, Damian Parniewicz (PSNC), Hagen Woesner, Tobias Jungel (BISDN), Jon Matias, Eduardo Jacob, Victor Fuentes (UPV/EHU), Richard G. Clegg (UCL), Roberto Doriguzzi (Create-Net), Marek Michalski, Remigiusz Rajewski, Mariusz Żal (PUT), Tasos Vlachogiannis (UNIVBRIS)

Dissemination level

☑	PU:	Public
☐	PP:	Restricted to other programme participants (including the Commission Services)
☐	RE:	Restricted to a group specified by the consortium (including the Commission Services)
☐	CO:	Confidential, only for members of the consortium (including the Commission Services)

<THIS PAGE IS INTENTIONALLY LEFT BLANK>

Abstract

This document describes the implementation details of the Hardware Abstraction Layer (HAL) based on the design specifications provided by deliverable D2.2. It provides a precise mapping between the HAL specification and its implementation within the overall functional architecture of HAL. It also demonstrates the feasibility of the HAL architecture by elaborating its implementation-level details for various hardware platforms. In addition, this document describes the process of resource virtualization and optical resource reservation and control implementation in ALIEN.

Table of Contents

List of Figures

List of Tables

Excecutive Summary

This document reports on the implementation details of the ALIEN Hardware Abstraction Layer (HAL) based on the design specifications detailed in deliverable D2.2.

The main objective of HAL is to realize OpenFlow capabilities on network elements that do not have native support for OpenFlow and enable their integration in an OpenFlow deployment, such as an SDN experimental facility. In order to achieve this goal, the HAL architecture decouples the hardware-specific control and management logic, which is handled in its Hardware Specific Layer, from the network node-abstraction logic which is implemented through the Cross-Hardware Platform Layer. This decoupling fosters reusability for different HAL components making them readily applicable to a range of hardware platforms as this deliverable documents. In effect, this document demonstrates the feasibility of the purposed HAL architecture by describing the implementation-level details of the aforementioned HAL sublayers for the targeted hardware platforms which include programmable network processors, general purpose packet processors, optical switches, as well as point to multi-point devices.

The document briefly reviews the HAL architecture and its component layers, i.e., Cross-Hardware Platform Layer (CHPL) and Hardware-Specific Layer (HSL). We then proceed to provide a precise mapping between the HAL specification (detailed in deliverable D2.2) and its implementation, pointing out in particular how the software developed in ALIEN contributes to the overall implementation of the functional architecture of HAL.

With respect to the implementation of CHPL, ALIEN has taken advantage of the Revised OpenFlow Library (ROFL) which provides a foundation for the development of OpenFlow controllers and datapath elements, and the eXtensible DataPath daemon (xDPd) which allows the development of platform-specific forwarding modules for a variety of devices. xDPD supports extensions through plug-in modules. Examples of plug-in modules in ALIEN include the virtualization agent, which adds slicing functionality, and NETCONF support for the HAL configuration management interface. CHPL communicates with HSL through a set APIs the implementation details of which are also presented in this document.

The implementation of HSL is, of course, hardware platform dependent. The document explains the process of HSL implementation for the four identified target groups of hardware platforms, namely X86-based packet processing devices, programmable network processors, lightpath devices, and point-to-multipoint devices. In particular, this deliverable reports and illustrates HSL implementation for EZchip NP-3, Cavium Octeon, NetFPGA, ROADM, GEPON and DOCSIS.

Moreover, the document describes how the resource reservation and virtualization mechanisms are implemented in the HAL. Specifically, the HAL virtualization agent implements an OpenFlow-version agnostic slicing mechanism which aims to avoid single points of failure with respect to virtualization as well as to support newer versions of the OpenFlow protocol. Finally, this document explains the implementation of resource reservation and control in optical devices which have different forwarding abstractions than the classic OpenFlow datapath.

This deliverable is public. We hope that it will attract the interest of the wider SDN R&D community working on OpenFlow network implementation.

1 Introduction

OpenFlow is considered the leading control plane standard for Software-Defined Networking (SDN) and has already played a significant role in reshaping network infrastructures. However numerous provider domains are still not equipped with a proper framework that can facilitate the deployment of an OpenFlow-based control plane on legacy network elements. In addition, considering the multitude of network devices and platforms to be supported, some vendors have taken a more cautious approach, thereby indicating a degree of hesitation to add OpenFlow functionality on their own (legacy) equipment. Such issues hinder the migration of today's networks to future SDN-enabled networks. The ALIEN Hardware Abstraction Layer (HAL) is designed specifically addresses these issues. HAL introduces a feasible approach for describing network device capabilities and controlling the forwarding behavior of all OpenFlow and non-OpenFlow capable hardware throughout a network. HAL hides the hardware complexity as well as technology and vendor-specific features, thus presenting a unified abstraction layer to an OpenFlow controller.

Next we provide a brief overview of HAL and its main components, which has been specified in detail in deliverable D2.2 [7] and publications [9], [10].

1.1 HAL Architecture Overview

The main purpose of HAL is to make a legacy network device OpenFlow-compatible through a set of abstractions. This approach allows operators, on the one hand, to extend their OpenFlow-based control plane to legacy (but valuable) infrastructure and, on the other hand, to network modern OpenFlow switches with non-OpenFlow capable devices in a seamless manner.

Considering the large array of devices that can be supported by HAL, the architecture has been based on a modular design which is extensible and compatible with heterogeneous network devices. Moreover by following such a modular design approach the behavior of any platform can be modified and extended without compromising the overall HAL architecture. It also makes HAL's implementation easier and faster for similar network platforms by exploiting module reusability.

A key design choice for HAL is to decouple the hardware-specific control and management logic from the network node abstraction. This decoupling allows HAL to hide the device complexity as well as the technology- and vendor-specific features from the control plane logic. Figure 1.1 illustrates the high-level HAL functional architecture where the decoupling has been achieved through a split into two distinct sub-layers, namely, the Cross-Hardware Platform Layer (CHPL) and the Hardware-Specific Layer (HSL). The former is responsible for node abstraction, virtualization and communication mechanisms. The latter takes care of discovering the particular hardware platform and performing all required configuration using hardware-specific modules. The two sub-layers communicate with each other through one of two interfaces, namely the Abstract Forwarding API and the Hardware Pipeline API depending on the type of the network device.

HAL provides two northbound interfaces to enable the communication between OpenFlow controller(s) and the devices, and to configure the Virtualization Agent via a Network Management System (NMS). The entities represented by "Network Control" and "Network Management" in Figure 1.1 employ the two northbound interfaces.

1.1.1 Cross-Hardware Platform Layer

The Cross-Hardware Platform Layer (CHPL), illustrated in Figure 1.2, is the hardware-agnostic software component which is common across all network devices supported by HAL. It comprises several independent modules responsible for device management (e.g., configuration of underlying device with desired parameters), monitoring (e.g., getting notified about events like status changes of ports on device), and control. The OpenFlow Endpoint in CHPL encapsulates all necessary control plane functionalities, maintains the connections with the OpenFlow controller(s), and manages the forwarding state all the way to the platform drivers.

On the management plane, CHPL presents a unified abstraction of the physical platform (physical ports, virtual ports, tunnels, etc.) to plugin modules hosted by a plug-in manager. This enables various plug-in modules to perform a variety of management-related operations, such as configuration. Examples of plugin modules include a NETCONF or OF-CONFIG

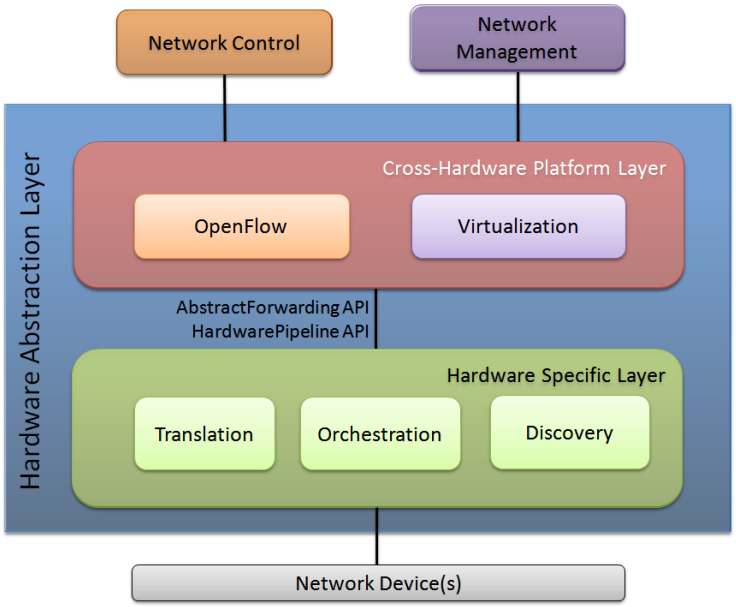

Figure 1.1: High level functional architecture of HAL

agent, a file-based configuration reader, and a Virtualization Agent (VA). The VA, as the name implies, adds resource virtualization features to the platform like a FlowVisor, such as, for instance slicing the device to be shared among multiple users. The main VA objective is to allow multiple users with simultaneous access to the same physical substrate without interference. VA interacts with the OpenFlow endpoint to perform flowspace slicing operations. It applies the slicing policies to the OpenFlow messages sent by the controller to the switch in a protocol-version agnostic way.

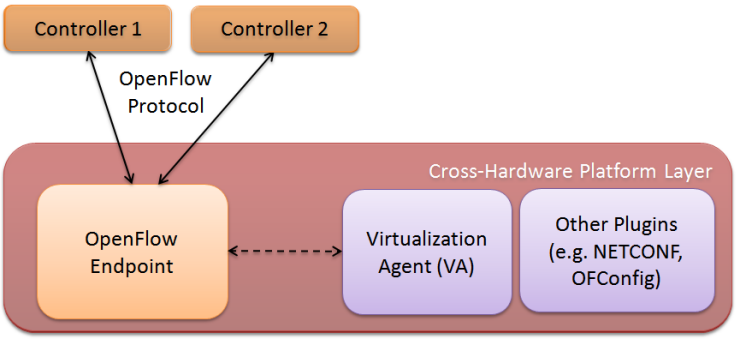

Figure 1.2: High-level schematic of Cross-Hardware Platform Layer

The OpenFlow Pipeline is an optional software component of CHPL that may be employed to implement the OpenFlow table(s) in the sub-layer as illustrated in Figure 1.3. It can also be noticed in this figure that the OpenFlow Endpoint and the OpenFlow Pipeline use the Abstract Forwarding API (AFA) for their communication. The same API is also used by OpenFlow Endpoint to communicate with the Hardware Specific Layer where it provides interfaces for management, configuration and receiving event notifications.

1.1.2 Hardware Specific Layer

The Hardware Specific Layer (HSL) addresses the diversity of network platforms and their communication protocols. Through HSL we can overcome the complexity of implementing the OpenFlow protocol on different hardware platforms. In the real

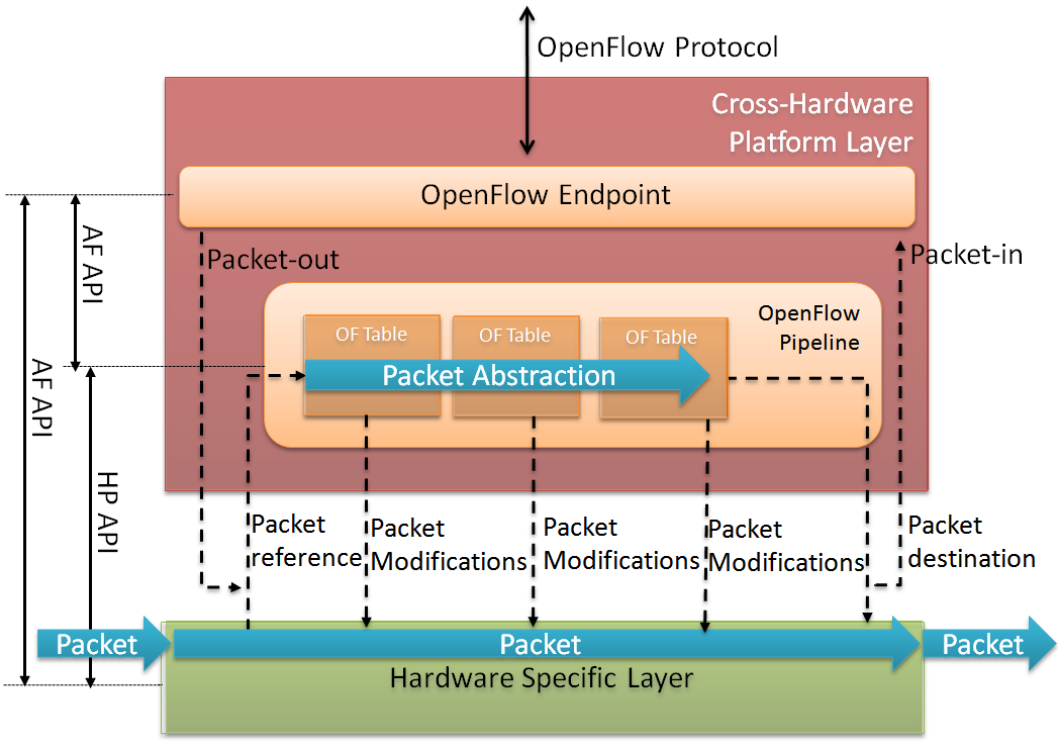

Figure 1.3: OpenFlow entities and interfaces within the Cross-Hardware Platform Layer

world, every network element or platform comes with its own protocol or API for communicating, controlling and managing the underlying system. Such APIs are often proprietary and closed to the research community. In HAL, HSL is responsible for hiding the complexity and heterogeneity of the underlying hardware control for message handling and providing a unified and feature rich interface in its northbound for the upper layer, i.e., the Cross-Hardware Platform Layer. In practice, HSL must deal with different implementations for each hardware platform. This layer has three key modules:

1. Discovery – Collects the information required to initialize CHPL, e.g., a list of devices working together as a single hardware platform instance and controlled by a single OpenFlow agent instance, available network ports and their characteristics such as, for example, transmission technology, transmission speed etc.

1. Orchestration – Sends configuration and control commands to all hardware components of the device that must be engaged in request handling. Orchestration also handles errors such as configuration failures.

1. Translation – Translates data and action models used in CHPL (mostly OpenFlow-based) to the device-specific protocol syntax and semantics, and vice versa.

HSL supports the Hardware Pipeline API (HPA) to interface with CHPL which can be employed, for example, to reuse the CHPL OpenFlow pipeline implementation. This facilitates, for instance, the implementation of the HAL hardware driver for programmable network platforms.

HAL has been implemented and is in active use over a variety of programmable and closed-box hardware as illustrated in Figure 1.4. The Figure also indicates the demarcation points for AFA and HPA. In the following section, a detailed account of HAL implementation particulars for various types of hardware platforms is provided.

1.2 Software Development in ALIEN

We conclude this short overview of the ALIEN with a few pointers to online repositories for software that was developed in ALIEN and relates to this report.

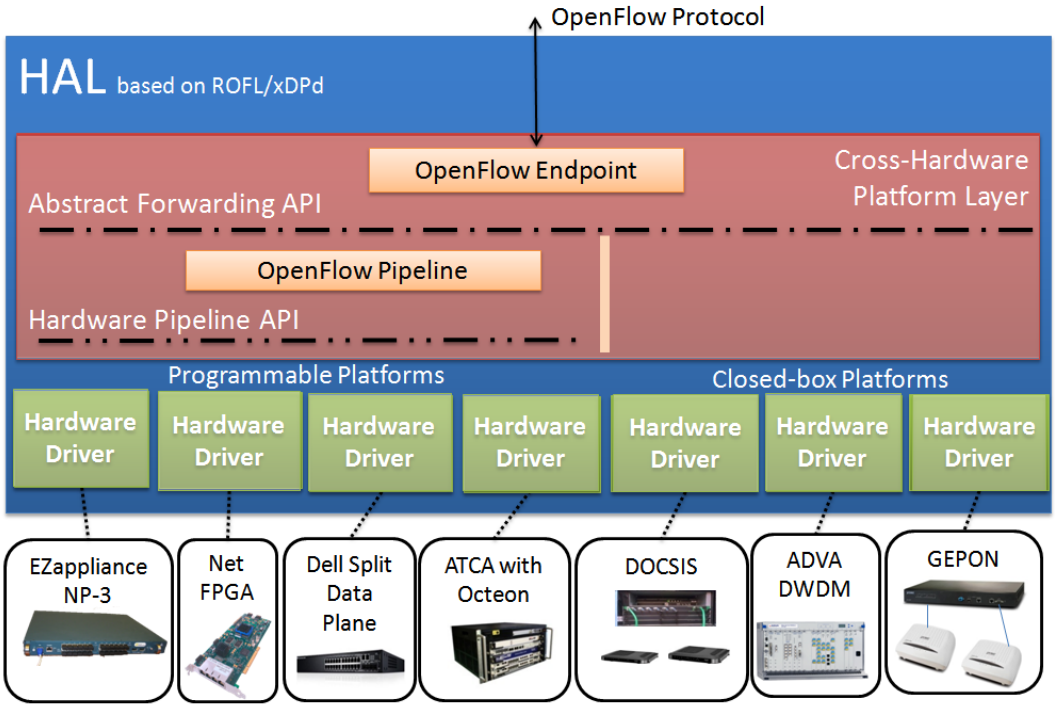

Figure 1.4: HAL implementation over different hardware platforms

Most of the implementation work described in this document has been made available publicly in the form of open source software packages available to the research community. For example, the list of software provided under the Mozilla Public License 2.0 includes:

1. **ROFL**, the Revised OpenFlow Library, which is a set of libraries for building multi-version OpenFlow Controllers and Datapath elements. Information and software repository for ROLF can be found at `http://www.roflibs.org`.
2. **xDPd**, the eXtensible DataPath daemon, a framework built on ROFL for developing OpenFlow/SDN datapath elements and designed to be easily extended with the support of new forwarding devices and platforms, new OpenFlow versions and extensions. Further information as well as the software repository for xDPD is available at `http://www.xdpd.org`.
3. The **xDPd-Virtualization plugin** is a module that adds virtualization capabilities to xDPd. The github repository for this module is available at `https://github.com/fp7-alien/xDPd-Virtualization`.
4. **xCPd**, the eXtensible Control Path daemon, a framework that allows interception of OpenFlow control messages to allow access networks to masquerade as distributed switches using tagging. Source code available at `https://github.com/richardclegg/xcpd`

1.3 Deliverable Outline

The remainder of this deliverable is organized as follows. Section 2 presents CHPL, starting with a brief summary of the ROFL and xDPd implementations. Next, we summarize the implementation of slicer implemented in ALIEN as an xDPd plugin followed by a presentation of the HAL APIs. Section 3 presents implementation details for HSL on four categories of devices, namely, (a) X86-based packet processing devices, (b) programmable network processors, (c) lightpath devices, and (d) point to multi-point devices. Section 4 presents implementation details about the reservation and virtualization of resources for ALIEN devices. Finally, Section 5 summarizes and concludes the deliverable.

2 Cross-Hardware Platform Layer Implementation

As mentioned earlier, the Cross-Hardware Platform Layer (CHPL) is the hardware-agnostic software component which is common across all network devices supported by HAL. In this section we detail the CHPL implementation starting the foundation elements and concluding with the CHPL application programming interfaces (APIs).

2.1 ROFL

The Revised OpenFlow Library [3] can be used to build OpenFlow control applications, controller frameworks, and data path elements. In short, ROFL provides a tool box to build OpenFlow-enabled software. Two of the most valuable tools in ROFL are the OpenFlow Endpoints and the OpenFlow Pipeline described next.

2.1.1 OpenFlow Endpoints

ROFL OpenFlow Endpoints provide basic support for the OpenFlow protocol, which includes protocol parsers, message mangling, and so on. In addition, they map the OpenFlow protocol wire representation to a set of C++ classes. Each OpenFlow Endpoint can be used on the data or on the control plane. That is, a ROFL OpenFlow Endpoint can be incorporated either in a datapath element or in an OpenFlow controller. Respectively, the endpoint can handle the OpenFlow control connection to any controller or datapath element.

In practice, an OpenFlow Endpoint hides the details of the respective protocol version and provides a clean and easy-to-use API to software developers. Currently, ROFL supports three types of Endpoints, namely for OpenFlow 1.0, OpenFlow 1.2, and OpenFlow 1.3.

2.1.2 OpenFlow Pipeline

ROFL has been enhanced during the ALIEN project with building blocks for creating datapath elements, most notably an OpenFlow pipeline, that can be integrated into any hardware platform supporting ANSI C. The OpenFlow pipeline can be used in different ways:

- as a data model of the forwarding plane of an OpenFlow switch
- as a data model and state manager library to maintain the state of the installed flowMod and groupMods entries, associated timers, statistics, and so on. This allows us to let the platform-specific code capture events (e.g. flow_mod insertion, flow_mod removal), APIs to mangle ASIC or other device configuration
- as a data model, state manager, and a software OpenFlow packet processing library, using packet processing APIs to process packets in software or hybrid (i.e. hardware-cum-software) OpenFlow datapath elements.

Furthermore, the ROFL OpenFlow Pipeline supports multiple logical switches on a single OpenFlow switch instance, each running its own OpenFlow version (e.g. OpenFlow 1.0, 1.2 or 1.3). In the case of software switches, in particular, specific matching algorithms (e.g. flowMod look-up) can be defined on a per table and per logical switch basis, such as, for instance, L3 optimized matching.

2.2 xDPd

xDPd has been further enhanced during the ALIEN project duration as a user-space implementation of an OpenFlow datapath element. It currently supports OpenFlow 1.0, 1.2, and 1.3 [8] and it is designed to run on multiple hardware platforms. Arguably, xDPd has a somewhat cleaner software architecture than the OpenFlow Virtual switch (OVS). xDPd implements an internal interface, namely the Abstract Forwarding API (AFA). In xDPd nomenclature, AFA is the API between the hardware-independent Control and Management Module (CMM) and the hardware-dependent Platform Driver (see Figure 2.1).

With respect to implementation and operational experience, xDPd is available on several hardware platforms, including: User-space GNU/Linux (x86-gnu-linux), GNU/Linux Intel DPDK (x86-dpdk), Cavium Octeon, Broadcom, EazyChip (EZchip

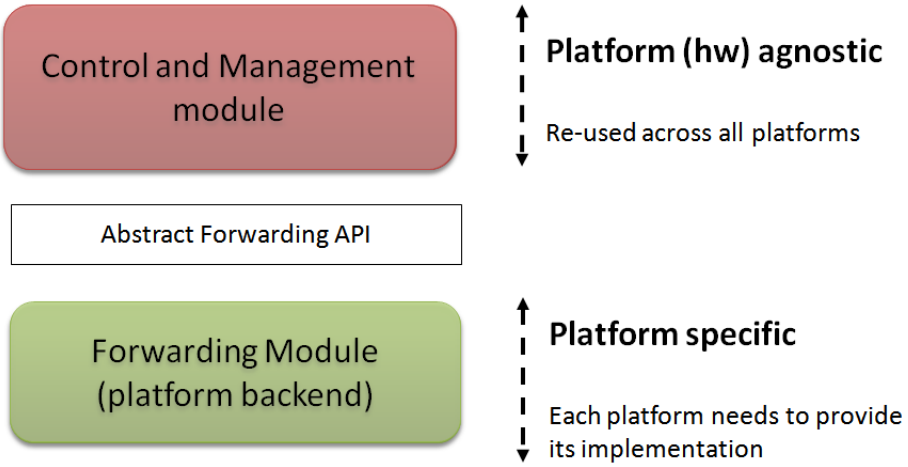

Figure 2.1: xDPd general architecture

NP-3), and NetFPGA-10G; an up-to-date list is available online as well at `http://xdpd.org/#platforms`. Source code availability for each of these platforms may be subject to hardware vendor license and, as such, not all of platform drivers can be open-sourced by the ALIEN partners.

As mentioned earlier, OpenFlow pipeline implementation for different hardware platforms is greatly facilitated by the availability of ROFL. One of the features of xDPd is the creation of multiple Logical Switch Instances (LSIs). LSIs are created either through a configuration file which is processed at start up time, or dynamically through a configuration interface. Each LSI is bound to network interfaces. In the case of multiple LSIs, network interfaces have to be exclusively assigned to one LSI only. This is a simple way of slicing and a first realization of a virtualization.

2.2.1 Control and Management Module

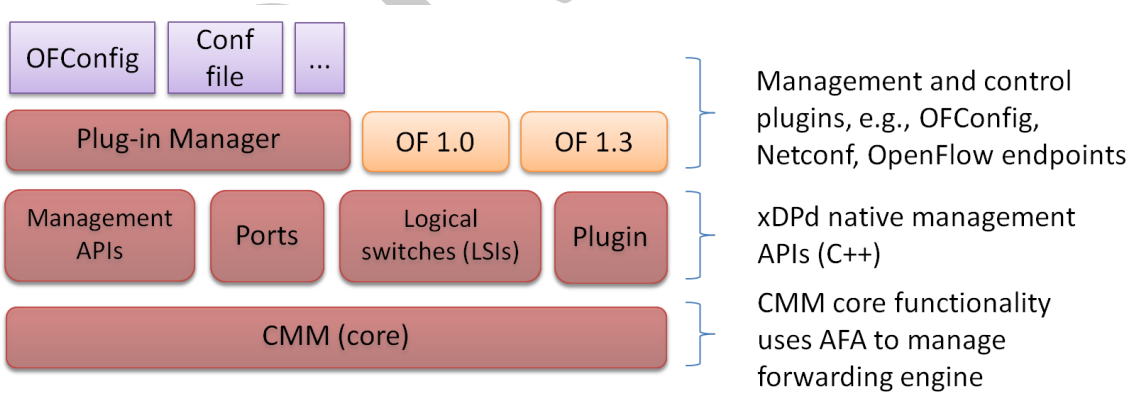

Figure 2.2: Control and Management Module of xDPd

The Control and Management Module (CMM) is the hardware-independent part of xDPd. CMM consists of a core module that implements an OpenFlow-like API in C++, which abstracts over the pure OpenFlow in that it allows LSIs of multiple protocol versions to run in parallel. In order to do so, CMM needs to bind the proper OpenFlow Endpoint version to the LSI, perform sanity checks on the flowMods sent, and in general be prepared to handle messages of multiple versions. As an example, message numbers, protocol fields, and counter formats (32-bit to 64-bit) differ between versions, so there needs to be a proper translation where possible, and marking of messages where necessary.

2.2.2 Plug-in Manager

The xDPd configuration and management interfaces are exposed through the Plug-in Manager. As a result, xDPd can be extended to provide further interfaces to configuration and management entities. Table A.4 summarizes the interfaces implemented by each plug-in.

2.2.3 Slicer

Slicing functionalities within xDPd are provided with a set of methods that form the so-called Virtualization Agent (VA). The main objective of the Virtualization Agent is to enable multiple controllers (which are likely to correspond to different experimenters/tenants) to control the same physical substrate, which is composed of xDPd-enabled switches, without interfering with each other. In its current implementation, the flowspace slicing mechanism is OpenFlow protocol version agnostic and, in principle, works with every field of the packet's header.

Below we summarize the most important operations that are performed by the Virtualization Agent:

- The VA checks the header of the packets against the slice configurations and configures the destination controller for the OpenFlow Endpoints.
- The VA intersects the matches of the flowMod messages coming from the controllers with the corresponding flowspace definition. In the current implementation, the VA sets the VLAN_ID to the value assigned to the slice.
- The VA checks if the actions contained in the flowMod and packetOut messages violate the slice's definition (e.g. sending packets out to a port that is not part of the slice).
- The VA checks if the actions contained in the buckets of the groupMod messages violate the slice's definition (e.g. sending packets out to a port that is not part of the slice).

Figure 2.3 illustrates the implementation of the VA within the HAL architecture. In particular, one instance of the VA is created during the device start-up and is responsible for the correct flowspace slicing. The VA does not inspect the OpenFlow protocol but leverages on the protocol-agnostic xDPd's internal structures to both select the correct controller for switch-to-controller messages and to filter out the controller-to-switch messages that violate the slice definitions.

The slicing process is performed within the OpenFlow Endpoints by calling the methods exposed by the VA. These are the labels MESSAGE ANALYSIS FILTER and SELECT SLICE in Figure 2.3. The process does not involve OpenFlow messages, but operates on protocol-agnostic structures such as of1x_action_group_t and of1x_flow_entry_t; see also Table A.1. However, the "new flow" messages are analyzed outside the endpoints. In particular, this type of processing occurs in the xDPd Control and Management Module, implemented in xdpd/cmm.cc, in order to avoid the OpenFlow protocol inspection (see SELECT CONTROLLER label in Figure 2.3).

The aforementioned functions are implemented within the xDPd code tree and are defined in the header files listed in Table A.5. The files that are added in the configuration plugin for reading and updating the Virtualization Agent database are listed in Table A.6. Moreover, the xDPd code files that have been enhanced during the ALIEN project duration in order to implement the VA functions can be found in Table A.7.

Finally, Table A.1 reports the most relevant functions used to implement the Virtualization Agent.

2.3 APIs

The HAL architecture is comprised of four main layers, (1) the Control and Management Layer, (2) the Cross-Hardware Platform Layer (CHPL), (3) the Hardware Specfic Layer (HSL), and (4) the forwarding / network devices layer. These layers communicate with one another using a set of well-defined interfaces. In this section, the different interfaces of the HAL architecture (i.e., NETCONF, AF API, and the Hardware Pipeline API) are described.

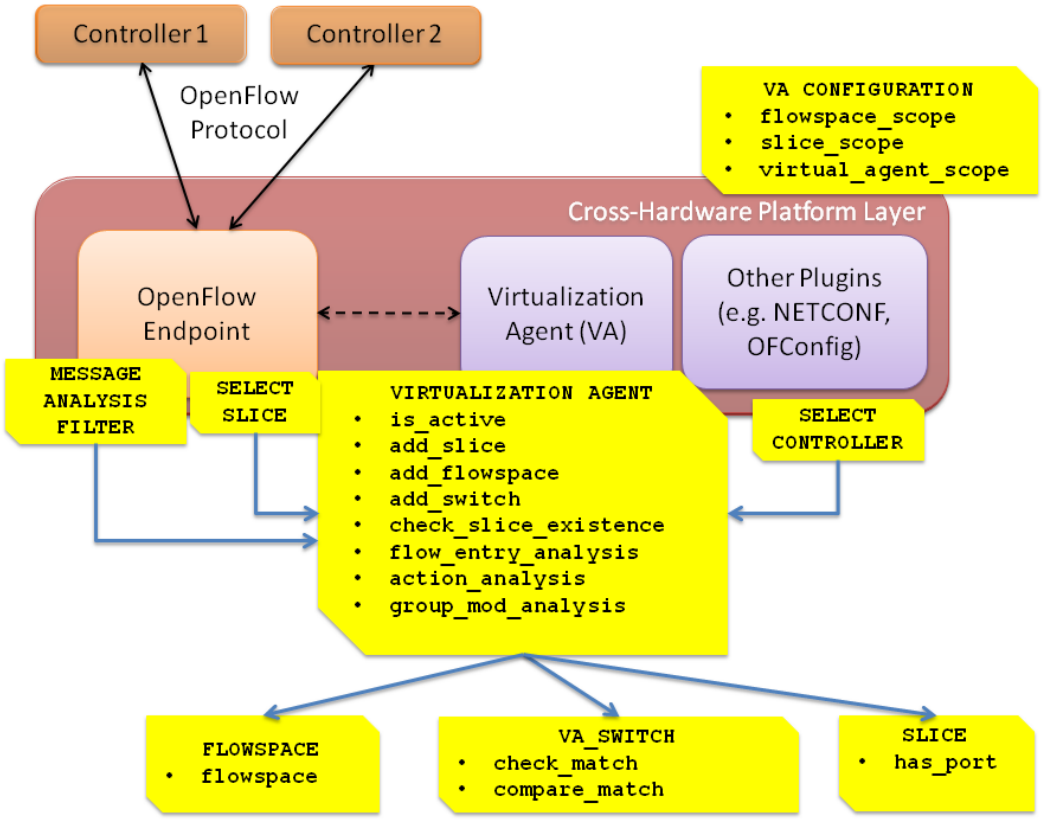

Figure 2.3: The Virtualization Agent implementation within the HAL architecture.

2.3.1 NETCONF

In the OpenFlow protocol specification(s), several configuration and management requirements are included either explicitly or implicitly as described in [OF-CONFIG]. These requirements include:

- connection setup to the controller (e.g., the IP address of the controller, the port number, the transport protocol used, either TLS or plain TCP)
- support for multiple controllers
- connection interruption handling (i.e., fail-over modes in case one of the controllers malfunctions)
- switch and controller certificate configuration for each controller that is configured to use TLS
- queue parameters configuration such as min-rate, max-rate for queue traffic
- switch port configuration
- capability discovery to describe the capabilities of the OpenFlow logical switch, and
- configuration of the switch datapath ID.

Using (static) configuration files to configure each device with the above configuration parameters can be cumbersome and has operational limitations. NETCONF [13] can be employed to automate this process and therefore it is seen as a reasonable alternative to use for managing ALIEN devices at the same time and installing the above configuration parameters. As such, in order to reduce the complexity of the management tasks, a NETCONF extension/plugin is introduced in the HAL architecture implementation, as illustrated in Figure 2.4.

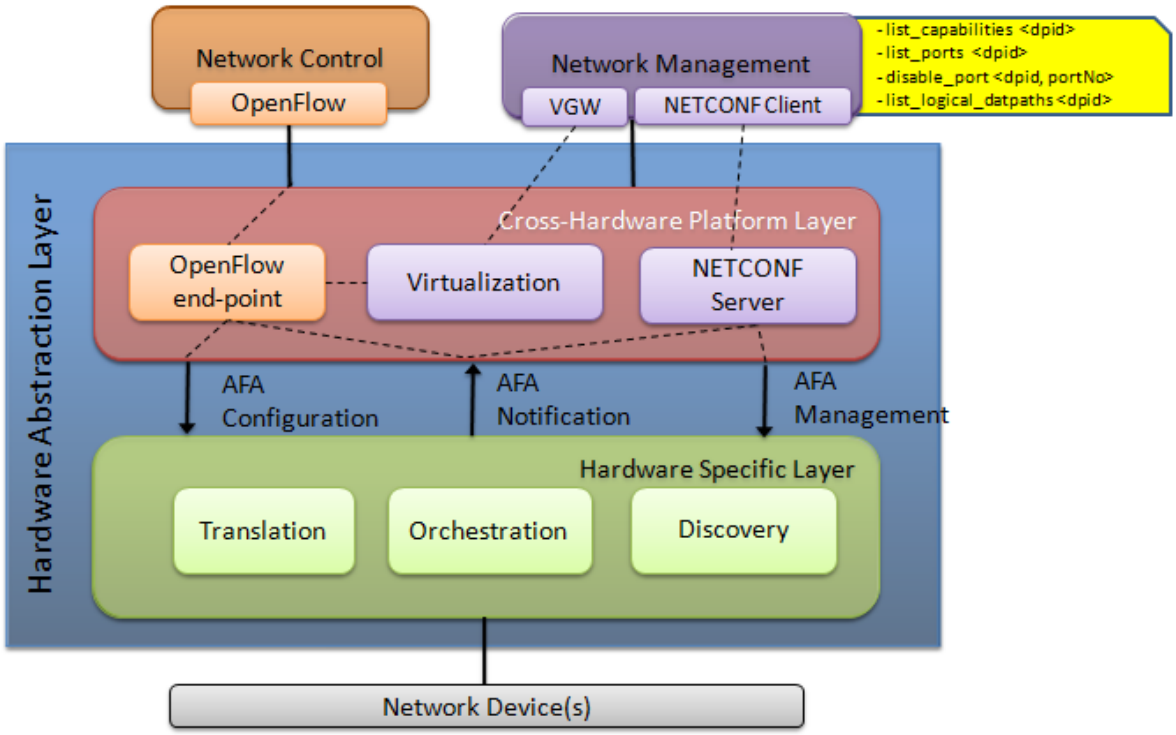

Figure 2.4: NETCONF plugin within the HAL architecture

The main purpose of the NETCONF plugin is to provide ALIEN island administrators and users with a management interface to configure the underlying ALIEN devices with several parameters, such as the OpenFlow controller IP address and switch datapath IDs. For administrators to have management access over network devices, they should be provided with a list of management commands that they can use to configure the underlying devices. For instance, administrators can use a command line interface (CLI) to list the commands that are available. Taking the the OpenFlow management and configuration requirements mentioned above as a baseline, the following commands could be included:

- list_capabilities <dpid>
- list_ports <dpid>
- disable_port <dpid, portNo>
- list_logical_datpaths <dpid>

2.3.2 Abstract Forwarding API

The Abstract Forwarding API (AFA) provides all the interfaces for management, configuration and events notification of the Hardware Specific Layer instance for the associated hardware platform. The management and configuration parts of the AFA interface must be implemented by a hardware driver and called by the Cross-Hardware Platform Layer instance (see Figure 2.5). The Notification part is provided by Cross-Hardware Platform Layer instance and invoked by a hardware driver.

AFA is implemented within ROFL as a set of C header files containing AFA API function declarations as listed in Table A.8. Functions declared in these files must be used by hardware driver subproject created within the xDPd implementation. Table A.2 contains a list of AFA abstract methods declared in the HAL specification [7] and corresponding function(s) implementations in ROFL. Table A.2 also updates information from the HSL specifications document [6] containing the first version of HAL AFA implementation required for HSL specification and development. More information about AFA functions and required parameters could be found in [7] and in the ROFL source code repository.

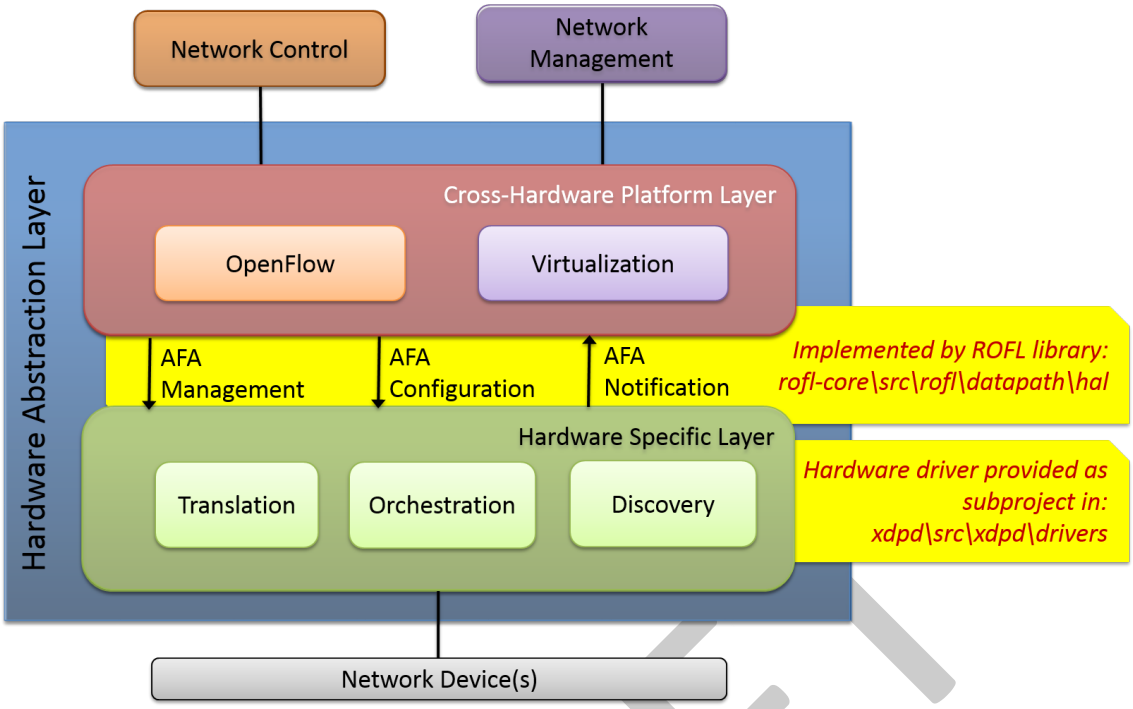

Figure 2.5: Abstract Forwarding API within HAL architecture and implementation details

2.3.3 Hardware Pipeline API

Hardware Pipeline API (HPA) is a low-level interface providing access to network packet operations, memory management, mutex and counter operations which are realized in different ways on different programmable platforms (see Figure 2.6). The main benefit of using the HPA interface is that the hardware driver does not need to implement the OpenFlow Pipeline per se. Rather the hardware driver can reuse the CHPL OpenFlow Pipeline implementation presented earlier in this section. This approach reduces significantly the overall development effort required to implement the HAL hardware driver on programmable network platforms such as Cavium Octeon, Broadcom Triumph2, Intel DPKK, and EZchip NPS processors.

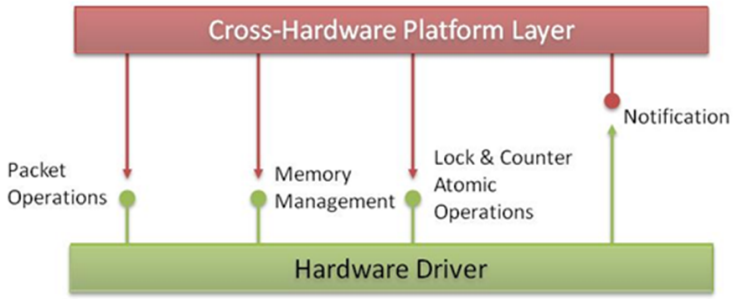

Figure 2.6: Hardware Pipeline API subsets and invocation model

HPA is implemented using ROFL as a set of C header files. The HPA function declarations are listed in Table A.9. Moreover, the list of HPA abstract methods introduced in the HAL specification [7] and corresponding function(s) implementations in ROFL can be found in Table A.2.

Table A.3 updates the information found in the earlier published HSL specification document [6], which presented the first version of HAL Pipeline implementation required for HSL specification and development. More information about HPA functions details and required parameters could be found in [7] and in the ROFL source code repository.

3 Hardware-Specific Layer Implementation

In this section, the HAL Hardware-Specific Layer is presented for four groups of network devices:

- **X86-based packet processing devices** -- This group comprises general purpose network devices that perform packet handling in software. This includes computing platforms from server boards to mini PCs like Raspberry Pi, Arduino, and so on, which typically have at least two or more independent network interfaces to turn them into potential forwarding nodes.
- **Programmable network processors** -- This group refers to network devices which allow their data plane to be programmed to perform packet processing. For some network processors (e.g.: EZchip NP-3, Cavium Octeon, NetFPGA), it is possible to implement the OpenFlow Pipeline directly into hardware.
- **Lightpath devices** -- Since the OpenFlow protocol is limited to an Ethernet-like abstraction, in the case of optical devices, such as reconfigurable optical add-drop multiplexer (ROADM) systems, the abstraction layer must be adapted to meet the OpenFlow extension requirements for supporting circuit-switched networking.
- **Point to multi-point access networks** -- For devices such as those based on standards like Gigabit Passive Optical Network (GEPON) and Data Over Cable Service Interface Specification (DOCSIS), with deployments based on "head" and "tails" topologies, some kind of orchestration is necessary for exposing several devices as a single OpenFlow-enabled "device" through HAL.

Each of these four groups group has different constraints and imposes various implementation challenges which have been explained in detail in earlier deliverables [5] and [6]. As a reminder, deliverable [5] has distinguished five types of network hardware themes, which are used in this section in order to present HAL architecture implementations for these types of hardware platforms. The *Physically Reconfigurable Systems* theme has no impact on HAL design and its implementation thus is not presented in this section.

3.1 Packet Switching Devices

This subsection discusses two types of packet switching devices that have been considered in ALIEN, namely network devices based on the Intel x86 architecture and devices employing programmable network processors.

3.1.1 X86-based Packet Processing Devices

SDN has been earlier associated with datapath forwarding using software switches typically running some Linux OS on a commodity server or PC. In general, this sort of hardware features a small number of network interface cards (NICs) that are attached via the PCI bus to the south bridge on a server mainboard. As most of these servers during the last decade incorporated Intel or AMD (x86) CPUs, the implementation of packet forwarding became essential for this architecture. As the frequency of an individual CPU core reached a limit of approximately 3.7 GHz, some years ago the x86 architecture moved to multi-core CPUs based on replications of older core layouts on a smaller chip surface, benefiting from the tic-toc of large CPU manufacturers (shrinking the masks before moving to a new architecture). At the same time, the PCI bus speed and the south bridge itself became increasingly more of a bottleneck for fast packet forwarding. Recent architectures therefore connect directly PCI lanes to certain CPU cores. This is complemented by the Intel's DPDK, the Data Plane Development Kit. This software development kit replaces Linux kernel drivers for Ethernet cards with libraries that allow direct memory access to the ring buffers on the NIC.

Recently, a number of software switch implementations added DPDK support (i.e. Open vSwitch, xDPd) and reported significant speed-up of forwarding rates to reach line rates of 10 Gbit/s on low-cost CPU equipment like the Intel Atom platform.

The drawback of DPDK, however, is that it works practically only on Intel CPUs and NICs, limiting its applicability. Netmap [15] increases the number of supported network interface cards while practically allowing the same memory access without the duplicated copy from the NIC to kernel space and then again to user space.

MMAP (memory map) is the general version that circumvents one copy operation between the kernel and user space by allowing direct access from user space processes to the Rx/Tx ring buffers. The measured speed difference between the DPDK and MMAP versions for xDPd during a recent comparison resulted in a factor of more than 5. Detailed documentation of these results will be included in the forthcoming deliverable D5.3.

3.1.2 Programmable Network Processors

Programmable Network Platforms represent a set of network equipment containing a re-programmable hardware unit (NPU or FPGA) that can be adapted to a wide range of network processing tasks (i.e. packet switching, routing, network monitoring, firewall protection, deep packet inspection, load balancing, etc.). These platforms allow for expressing packet processing control/service logic, using a programming language, in form of compiled source code which can be implemented indefinitely on a single hardware unit.

Programmable processors are ideal hardware platforms for introducing and validating new networking concepts. To take advantage of this possibility, in the ALIEN project, dynamic adaptation of network node capabilities has been investigated in order to introduce new protocols to a datapath element with new processing actions which later could be added to the OpenFlow protocol action set.

Currently, there are many programmable network platforms available in the market produced by several vendors such as EZchip, Marvell, Cavium, Broadcom, Freescale, PMC-Sierra, and Tilera. Each vendor provides programmable processors using quite different processor architectures in terms of microcore types (i.e., general core like in CPU, task optimized core); organization (e.g., homogenous cores loosely assigned to tasks, strict pipelines of heterogeneous cores); add-ons (i.e., hardware accelerators for parsing, pattern matching, cryptography, packet classification, querying, among others); and memory accessibility (e.g., standard CPU cores with ASIC network enhancements, task optimized NPU cores). This heterogeneity of network processors is a challenge when establishing common implementation assumptions based on the HAL specification design.

EZappliance Platform

The heart of the EZappliance platform [1] is the EZchip NP-3 network processor (see Figure 3.1), a fully programmable chip which enables flexible parsing, classification, packet header manipulation and switching of pass through packets. It is the part of the implementation stack where packet processing through the OpenFlow Pipeline should occur in order to take advantage of the full performance of processor. Unfortunately, the CHPL pipeline for handling packet abstractions cannot be reused as-is in this platform because the NP-3 processor has very strict time constrains for packet processing and cannot store the packets anywhere inside the platform. For this reason, a new implementation of the OpenFlow Pipeline for NP-3 task-optimized cores was developed from scratch using the EZchip assembly language.

The NP-3 processor is accompanied with a standard CPU foreseen for the deployment of control and management plane functionalities. The standard CPU was used to deploy both CHPL and HSL. Since the CHPL pipeline is not used, HSL could be controlled by CHPL through the AFA interface only.

The HSL for EZappliance devices supports discovery and translator functionalities. The discovery functionality is based on automatic retrieval of information about all data plane ports, along with the corresponding attributes and status. In the case of EZappliance, which is a standalone device, topology discovery is not required (for the same reason, HSL for EZappliance does not include the orchestrator functionality).

The most complex part of HSL is the implementation of translation functionality which transforms OpenFlow-based AFA messages into memory structures located within the NP-3 network processor. The NP-3 memory structures are accessed via the EZdriver provided by EZchip. The semantics used for the EZappliance memory structures is quite similar to OpenFlow, i.e., the memory contains a structure with flow entries but the syntax is mostly different: proprietary binary encoding of packet matching and actions. Translation in the HSL is stateless.

NetFPGA Cards

Similarly to NP-3, NetFPGA cards [2] can be treated as programmable packet processors. They have four 1 Gb/s Ethernet interfaces (or 10Gb/s in a newer versions). Both card versions can work as separate network nodes, however, typically they

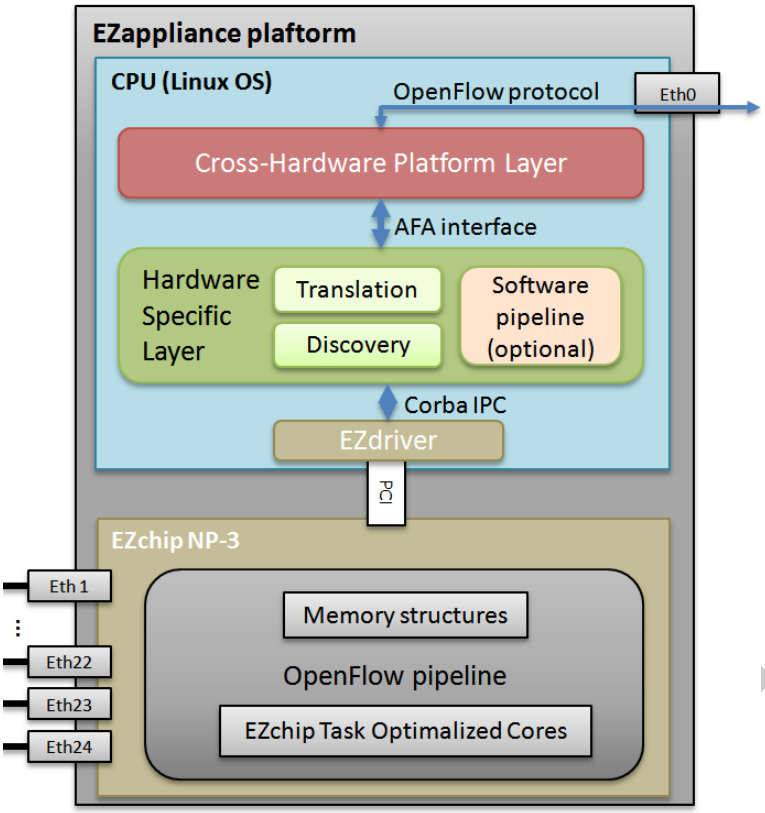

Figure 3.1: HAL adaptation for EZappliance network processor platform

are mounted to a PC and they are integrated with the operational system via PCI or PCIe bus. The program, which is running in the FPGA chip, has to be prepared in Verilog or VHDL. Due to the specific architecture and technology, its performance is very high and it is widely used by designers of different prototypes.

In ALIEN NetFPGA cards are used as a hardware platform. The OpenFlow Pipeline is almost fully implemented in the NetFPGA logic which offers much better performance characteristics compared to using the CHPL pipeline that has to be deployed in the PC operating system as part of HSL. In the NetFGPA HAL realization (see Figure 3.2), the CHPL pipeline is used as a full featured OpenFlow albeit slower implementation which processes only the packets that cannot be handled by the hardware pipeline due to OpenFlow missing features in the current hardware pipeline implementation.

The CHPL for NetFPGA cards is placed in the PC operating system and has a connection with the network controller using the OpenFlow protocol via the NIC of PC. The HSL for NetFPGA also realizes discovery and translation functions. The translation functionality is responsible for recoding of OpenFlow flow entries into a binary representation recognized by the hardware OpenFlow pipeline in the NetFGPA card.

Proper control information (flowMods) are stored in the hardware chip of the NetFPGA card and all possible flow actions (packet forwarding, dropping, etc.) are realized by the hardware chip. It is only the first few packets per each flow (or flows) which cannot be served by the hardware pipeline that are handled in the software realization of HSL.

Cavium Octeon

The Cavium OCTEON family offers a variety of Multi-Core MIPS64 processor boards especially targeted for network processing duties. With 1 to 48 cnMIPS cores on a single chip, depending on the model, and other hardware acceleration units (port I/O, cryptography, DFA, etc.), they are a highly versatile software programmable network platform.

The architecture of the implementation is as follows:

- There is a single MIPS core, called the management core, running a standard CAVIUM Linux OS. The management

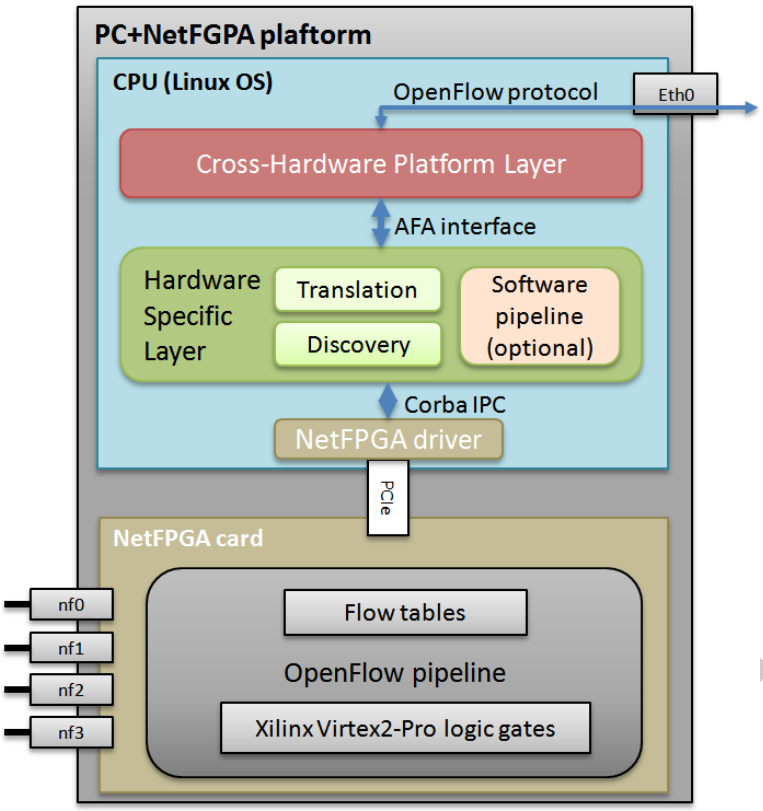

Figure 3.2: HAL adaptation for NetFPGA cards

core is employed to run xDPd (CMM, HSL) and the specific OCTEON driver

- There is shared memory, allocated at boot time in the so-called bootmem area shared across the management core and the rest of the I/O cores.
- Finally, the N-1 remaining cores are devoted to process packets. They run on bare-metal, that is, that is, in standalone mode (Single Executive Standalone, SE-S) in OCTEON's terminology, which means that they run a specific compiled binary program in single-thread mode, without any kind of operating system or thread context swapping whatsoever.

The management core is in charge of dealing with the particular configuration of the fast path rules, so the OpenFlow pipeline, while the remaining cores use this state to process packets continuously (See Figure 3.3). Actual packet flow is going through the SE-S cores exclusively, except in the case when there is no match in the FlowTable.

The interaction of the controller with the device is taking place via OpenFlow. The OpenFlow Endpoint is the one implemented as part of ROFL in the CMM. Inside the OCTEON processor itself, another API is used to access the specific functions and registers of hardware accelerators. This API is called Simple Executive API (SE-API) or HAL in the OCTEON Users' Manual (not to be confused with the ALIEN-specified HAL). The Linux core implements a pipeline that is a logical representation of the SE-S cores, and no packet actually passes through this one, except Packet-outs for convenience.

3.2 Lightpath Devices

OpenFlow, as a control protocol, promotes the use of flows instead of packets as the most vital unit of control alongside the separation between control and data planes. The optical domain, however, has long followed this approach since there is a clear separation of the control from the data plane. In addition, the notion of packets does not even exist in this domain. Instead, a lightpath, which can be considered as a flow, is the fundamental unit of information when establishing a connection from one optical node to another. However, OpenFlow focuses mostly on packet switching and, originally

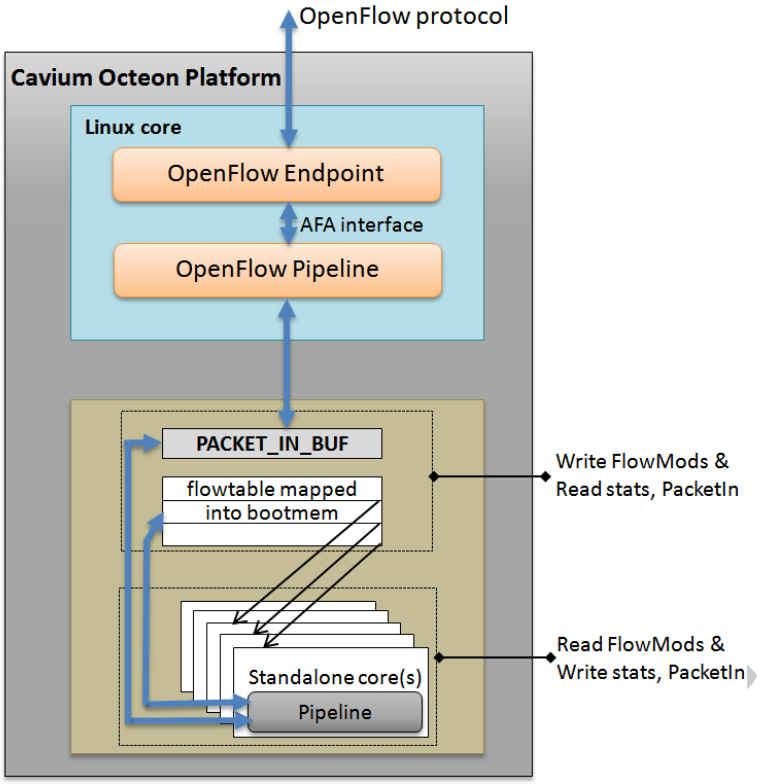

Figure 3.3: Relation between Linux core and SE-S cores in the OCTEON Plus implementation

did not offer any support for circuit switches, though this has been addressed to some degree later through a number of extensions proposed [4].

Lightpath devices are used mainly, but not limited to, in the core of the network to provide high-speed links between transit nodes of the network. Optical switch devices offer a number of benefits to the operators such as scalability (since and they can switch large amounts of data with very low latency) and energy efficiency (compared to the power consumed by an electronic switch device). The emerging convergence of optical packet domains fostered by OpenFlow can enable operators to satisfy the growing demands for reduced latency and large amount of bandwidth from the current and evolving applications (e.g. 4K streaming, video on demand, etc.).

The ADVA FSP 3000 is a high-performance Wavelength-Division Multiplexing (WDM) networking system for bidirectional transmission of optical signals. The system uses a modular structure which enables a flexible upgrade of capacity and functionality according to network requirements. The transmission between the modules is optical and passive, which means that the device control is completely separated from the data plane.

As opposed to a packet switch, an OpenFlow-enabled circuit switch consists of a cross-connect table and an OpenFlow channel to the controller. The cross-connect table maintains a list of entries with all connections between the ports inside the switch. The OpenFlow Endpoint is handled using the ROFL library, which has been enhanced to support the optical extensions to the protocol. the OpenFlow Pipeline functionality is not supported by lightpath devices since there is no notion of packets in the optical domain and no packets can be buffered or forwarded to the controller.

As illustrated in Figure 3.4, in order to get the OpenFlow abstraction of the device the Simple Network Management Protocol (SNMP) management interface is used. However, this interface first needs to be configured manually with a valid IP address to enable remote access. SNMP communication (traps, get/set messages) provides all the information that can be extracted from the network element, while the layer above is responsible for receiving and translating from this pool of resources the those that are required for the OpenFlow abstraction (OpenFlow Resources). The layers described above compose the HSL of the ADVA network elements. On top of that the functionality and the facilities supplied by ROFL are

employed to maintain the OpenFlow channel and handling messages received from the extended OpenFlow controller.

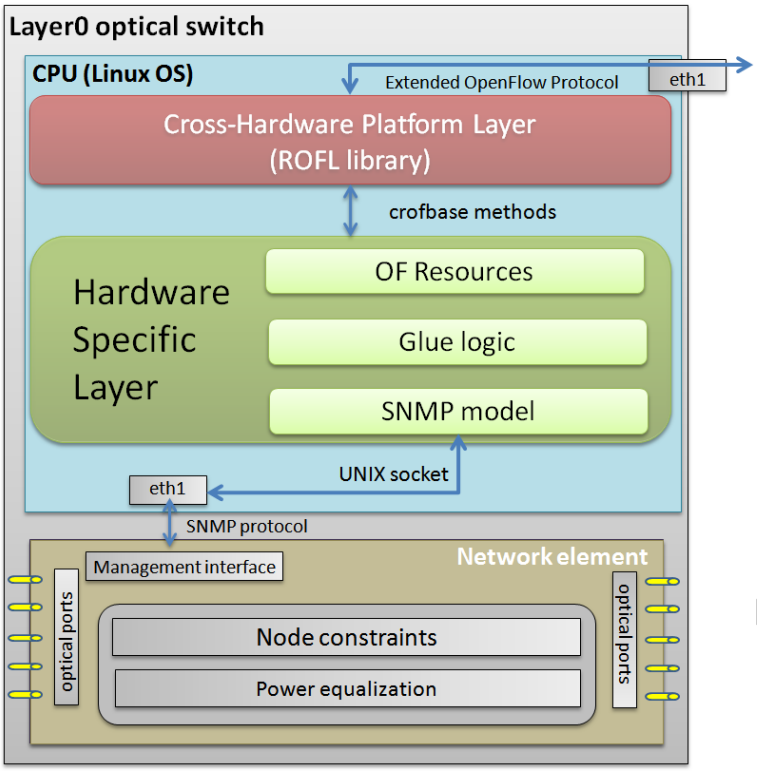

Figure 3.4: HAL adaptation for ADVA FSP 3000 switch

3.3 Point-to-MultiPoint Networks

In general, point-to-multipoint devices consist of a "head end" which communicates with several "tail end" devices, usually through broadcast means with some form of multiplexing which allows the devices to know which traffic is intended for them. This approach is very common in access technologies. Within the ALIEN project, two types of hardware in this category are used, namely the Gigabit Ethernet Passive Optical Network (GEPON) and the Data Over Cable Service Interface Specification (DOCSIS). Descriptions of both devices can be found in [5].

The Access Network (AN) provides the connectivity between the home/business customer's location (i.e. subscribers) and the operator's premises. This part of the network is known as the last mile and it is considered as the bottleneck in terms of bandwidth. It is also often the most expensive part of an operator's network. There are several technologies currently used in commercial deployments depending on the available media, such as xDLS (copper), DOCSIS (cable) or GPON (fiber). In order to deploy the system in the most cost-effective manner, this media is shared by a set of subscribers. As a result, bandwidth sharing is one of the goals of any of those AN technologies. Regardless of the specific technology used, all these systems can be abstracted as a point-to-multipoint (i.e. operator-to-subscribers) device.

One of the main challenges of these systems is that they are so specific in nature (i.e. focus on the Access Network) and technology that it is hard to integrate their control and management planes in a more generic framework, such as an application-oriented and multi-access technology solution. In this context, the SDN paradigm and OpenFlow are the tools that enable this integration by introducing a common abstraction for networking devices, i.e. the ALIEN-specified HAL. This layer deals with specific interfaces and hides the dependence on the technology. In the end, a HAL-based AN is agnostic with respect to the actual technology deployed.

In the following subsection, we present an example of this proposal for a DOCSIS system, which exposes OpenFlow as its northbound interface. By doing this, the whole system can be abstracted as a single OpenFlow device.

3.3.1 DOCSIS Access Network

The DOCSIS platform, illustrated in Figure 3.5, comprises three main elements: the CMTS, the cable, and the cable-modems (CMs). The CMTS is the head-end and 'intelligent' part of the system, which determines the use of the shared media by the CMs. The CMTS must be configured in the bridge mode (i.e. TLAN or L2VPN) to be compatible with OpenFlow abstractions. The cable is the shared media (coaxial) between the CMTS and several CMs. Finally, the CMs are the tails of the system located at the subscriber's location. Collocated with the CMs, it is customary to deploy a managed OpenFlow User Instance (OUI) to implement some service related networking logic. In order to implement connectivity between any CMs in bridge mode, an external device (i.e. aggregation switch -AGS-) is needed adjacent to the CMTS.

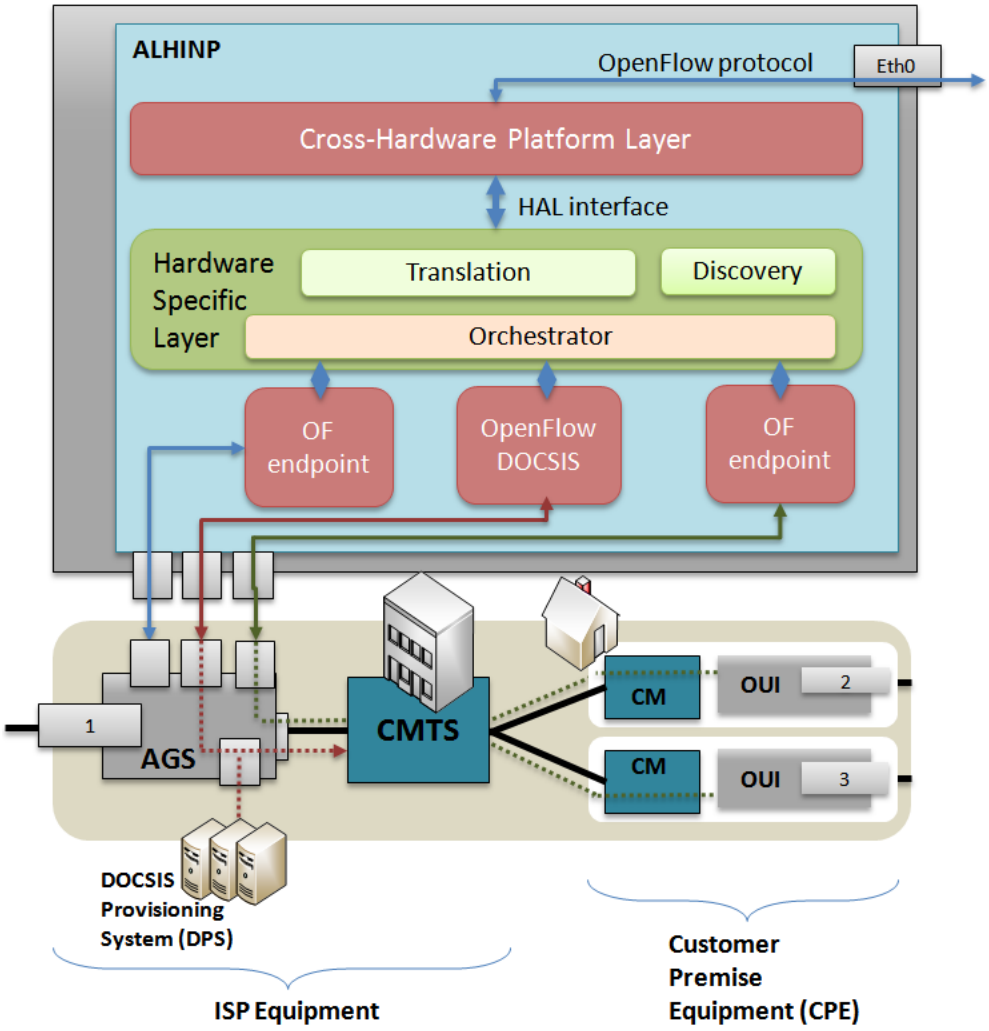

Figure 3.5: HAL adaptation for DOCSIS Access Network

Since the DOCSIS platform is closed we cannot reprogram the devices, control is only possible through vendor-supported standard interfaces. In principle, this limits the integration of DOCSIS under an OpenFlow interface. However, by adding the OUI and aggregation switch in the picture (i.e. as helper boxes), we can orchestrate the whole system to overcome these limitations and implement a fully compatible solution. As a result, the ALHINP (ALien HAL Integrating Network Proxy) performs the proper abstraction from the whole system by sitting (in the control plane) between the set of network devices and the OpenFlow controller. This proxy is based on AFA, since the actual data plane remains outside the DOCSIS proxy.

As previously mentioned, the ALHINP resides on an external box logically located between the platform and the controller and implements both layers of HAL: CHPL and HSL. In this implementation, CHPL consists of the OpenFlow Endpoint

using the AFA as its southbound interface. Therefore, CHPL interacts with the HSL through AFA.

HSL for DOCSIS platform implements the discovery, orchestration and translator functionalities. The discovery component provides information each time a new CM is connected to the system. As a consequence, ALHINP dynamically updates the virtual ports exposed to the controller, since each CM is abstracted as a new virtual port of the virtual OpenFlow switch. The orchestration component enables the coordination of multiple hardware components (i.e. OUI, CMs, CMTS and AGS) so they act as a single device.

All control plane interactions between the controller and the proxy must be handled to achieve a similar/emulated response from the set of devices. In order to improve the modularity of the system the orchestrator assumes three domains: OUI, AN and AGS. Each domain implements its own driver to interact with the target device via the available interface. Moreover, by doing this separation the proxy can be easily developed and the AN technology can be changed just by developing a new driver for it. Finally, the translation component implements the logic to map the virtual ports (from a single virtual DataPath Identifier) to real ports and physical DataPath Identifiers, and vice versa. This functionality is implemented in coordination with the orchestration module and gets input from the discovery module.

ALHINP can be used as an example of how any other AN technology can be abstracted following a similar approach and be exposed though an OpenFlow interface. By doing so, the AN can be controlled as any other OpenFlow resource, and even more, any previously developed OpenFlow application can run without any adaptation. As next steps we are investigating how the management of the shared media (e.g. the bandwidth assigned to each subscriber) can be exposed though the OpenFlow interface of the proxy. The appropriate extensions are currently under development.

3.3.2 GEPON Access Network

Similarly to DOCSIS, GEPON has three main elements which can be considered analogous: the OLT (Optical Line Terminal), the splitter and the ONUs (Optical Network Units); see Figure 3.6. OLT is the head end device that is the most intelligent part of the system and is responsible for orchestrating the ONUs. The ONUs are usually situated in customer premises. Data transfer between ONUs and OLTs is optical. In a typical deployment, data between ONUs goes via a head-end switch outside the OLT. The optical part of the network is passive and all data from the OLT goes to all ONUs which share their time using time division multiplexing.

As with DOCSIS, GEPON is proprietary/closed source equipment. A different approach was taken to that taken by the DOCSIS implementation although the two approaches are complementary. Instead of having the OUI boxes collocated with the tail-end equipment, the GEPON HSL works with changes only at the head-end device. The key change is to enhance the switch outside the OLT to be OpenFlow-enabled and to add a proxy device, known as eXtensible Control Path daemon (xCPd). xCPd speaks OpenFlow northbound and southbound and pretends to be a large virtual switch with one port for the OLT and one port for every ONU. XCPd translates these virtual ports to either the appropriate real port and, if appropriate, a VLAN tag that is associated with the appropriate ONU. Where OpenFlow requirements cannot be met using just VLAN tags, then xCPd communicates directly with the OLT via its management port. xCPd orchestrates the changes to the OLT and the translations of matches and actions to the lower-level OpenFlow switch. The downside is that because VLAN tags are used between the OpenFlow switch and the OLT then those tags cannot be used elsewhere unless the device supports QinQ (stacked VLAN tags) which the model at UCL does not.

xCPd is a generic framework that could be used to control any access network with the following requirements:

- all traffic between tail-end devices goes via a switch upstream from the head-end.
- the head-end device can route traffic via tags and untag them.
- the head-end device can tag packets from tail end devices.

For some OpenFlow functionality then hardware-specific sections must be written that is particular to the hardware in question. This is the control path labelled MGMT in Figure 3.6. Porting to new access devices meeting the above requirements automatically will achieve the majority of OpenFlow 1.0 functionality. Port statistics will not map correctly without hardware-specific code being written for the hardware to be ported.

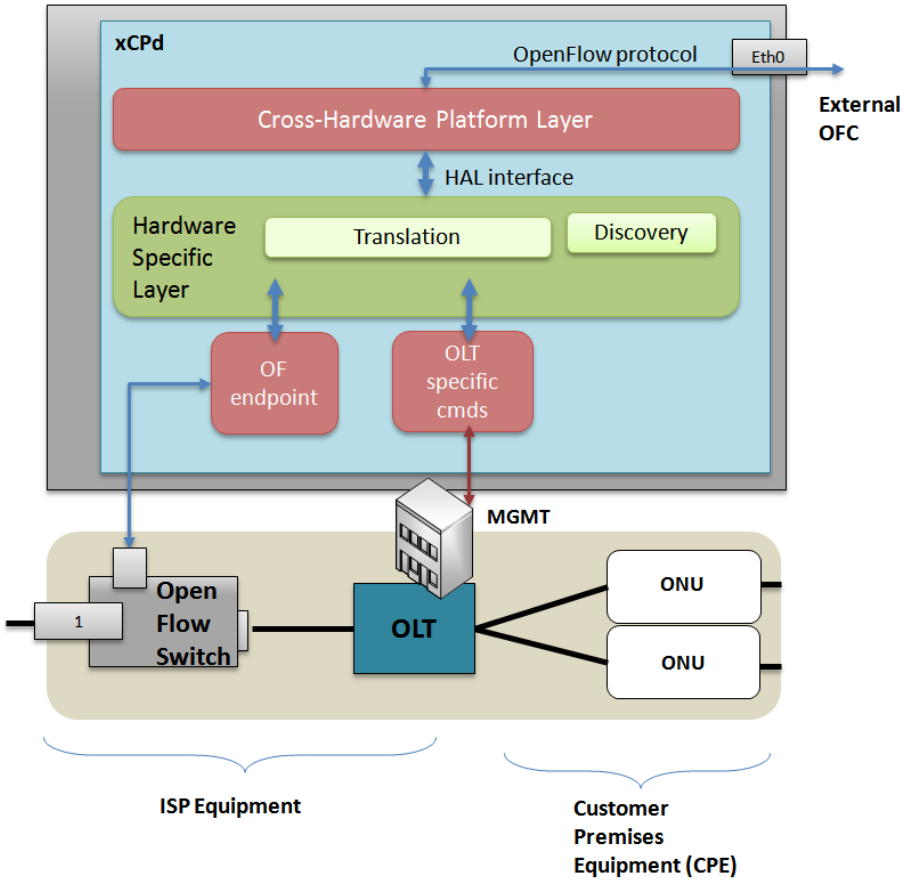

Figure 3.6: HAL adaptation for GEPON Access Network

4 Resource Reservation and Virtualization

This section summarizes the outcomes of the Task 2.4 development activities focused on Reservation and Virtualization of Resources.

Resource reservation refers to the characterization of resource types including interfaces, processing units, and the flowspace. As a result, the extension of the OpenFlow data model is provided to expose such resources through the control channel.

Resource Virtualization refers to the segmentation of the available resources such as forwarding nodes and flowspace. The outcome of the Resource Virtualization activities within WP2 is the implementation of a distributed and OpenFlow version agnostic slicing mechanism as a component of the HAL. This component, called Virtualization Agent, tries to overcome two of the main limitations of other approaches like FlowVisor [14] and VeRTIGO [11] that are: Single Point of Failures (SPoF) and lack of support for versions >= 1.1 of the OpenFlow protocol.

4.1 Optical Resource Reservation and Control

The OpenFlow protocol was initially introduced to allow programmability in networks; however the initial proposal took into consideration only the packet switches of the network. As opposed to the packet switch, an OpenFlow enabled circuit switch consists of a cross-connect table instead of a matching packets table. This subsection describes the changes needed to be applied to the original Stanford implementation in order to control the ADVA optical ROADM network elements. Amendments in the protocol, as expected, need to be applied both on the OpenFlow agent as well as the controller used to control the device itself.

The implementation of the OpenFlow datapath for the ROADM network element is based on the circuit switching extensions ver. 0.3 [4]. These extensions are merged with the OpenFlow specification v1.0 and have been developed in a way that it does not break the protocol structure size and fields. Additionally to these extensions we have developed some further extensions that are required for the ADVA ROADM network element to be controlled by OpenFlow.

In the following a short outline of the most important changes included in the OpenFlow addendum is provided though readers interested to know more about the OpenFlow circuit extensions should refer to the original document [4]. First of all, additional capabilities have been added in the features reply message to accommodate the extra features of the optical switch. Also in the features reply message a new structure has been added to describe the physical ports (ofp_phy_cport) of a circuit switch and some of the existing padding bytes have been used to specify the number of circuit ports in the optical switch.

Moreover, there are some additional messages that have been defined to enable control of the optical devices. Optical cross connections are setup and torn down by the controller using the CFLOW_MOD message and some errors message types have been added to inform the controller if something goes wrong. The CFLOW_MOD message contains the so called logical equivalent of ofp_match structure, the ofp_connect structure which describes the cross connection inside the switch. Also a CPORT_STATUS message has been added to allow the switch to inform the controller about changes in the state of the physical circuit port.

In addition to these extensions, we have utilized the flexibility OpenFlow provides by defining a number of extensions using the OFPT_VENDOR (4) message type which is used as a stage is foreseen as a staging area for new protocol (experimenter) features. Vendor extension feature allows for extending the protocol without breaking the compatibility with the base protocol specification.

The vendor OpenFlow message contains a field vendor after the OpenFlow header which is the vendor id for the device/vendor that this message has been implemented. A vendor code has also been defined for the ADVA ROADMs, OOE_VENDOR_ID (0x41445641) to identify a set of messages that are specific for this device. Furthermore, a new header was developed for this type of message:

```
struct ooe_header {
    struct ofp_header  header;    // openflow header
    uint32_t           vendor;    // vendor id
    uint32_t           type;      // message type (OOE_ message type)
    uint8_t            data(0);   // message payload
}
```

In addition, a number of device specific messages and respective codes were defined in order to identify them. The purpose of these types will be explained in the following paragraphs.

```
enum ooe_type {
    OOE_SWITCH_CONSTRAINTS_REQUEST,// switching constraints
    OOE_SWITCH_CONSTRAINTS_REPLY,  // switching constraints
    OOE_POWER_EQ_REQUEST,          // power equalization
    OOE_POWER_EQ_REPLY,            // power equalization
}
```

Switching constraints describe how the physical ports are connected with each other inside the ROADM. This relationship between ports comes from the internal network element configuration. The device comprises a number of physical cards connected with each other through fiber jumpers. The switching constraints map informs whether the optical signal can flow between particular ports. However, it should be noted that switching constraints do not tell whether a setup is really possible for a lightpath, even if these ports are physically connected. In order to be able to determine whether it is

indeed able to do that, the features of the port must be consulted to check that the specified wavelength λ is supported by both ports.

The ADVA ROADM cards require that the power equalization procedure to be triggered after a cross-connection is created in the Wavelength Selective Switch (WSS). Without power equalization, the ROADM card will be blocking the signal flow. The extended OpenFlow controller can send a power equalization request to the OpenFlow switch and therefore instruct the switch to equalize the optical signal power on modules that require such procedure. Such equalization is triggered by specifying ports and a wavelength. Equalization is triggered on modules that are located along the internal signal path between these ports; the request is unidirectional.

4.2 Resource Virtualization

Resource virtualization in HAL-enabled devices is achieved with the virtualization mechanisms implemented through the Virtulization Agent (VA) and the Logical Switch Instances (LSIs). Other approaches like [14] or [11] do not provide the support for OpenFlow protocol versions beyond 1.0 and, moreover, introduce an additional layer on the control channel to obtain the virtualization of the network resources which represents a SPoF. The implemented framework aims at providing a distributed virtualization architecture (no SPoFs) which is able to run on multi-version OpenFlow switch network scenarios.

The VA, whose implementation has been described in detail earlier in this deliverable 2.2.3, aims at virtualizing the forwading plane with flowspace slicing techniques; see also Deliverable D2.2 [7] for more details.

LSIs allows the partitions of the physical devices into several virtual switches. Each virtual switch is configured as a subset of ports of the physical device and includes an endpoint that supports a given version of the OpenFlow protocol and connects to a single OpenFlow controller.

While the objective of the VA is to allow the forwarding plane to be shared among multiple controllers, each with distinct forwarding logic, the objective of the LSIs is to logically extend the forwarding plane with more nodes than the number actually available in the physical infrastructure. The combination of the two mechanisms permits each HAL-enabled physical node to be split into multiple logical nodes and each logical node to be shared among different OpenFlow controllers.

The LSI management is located in the Cross-Hardware Platform Layer and implemented with the class named "switch_manager" in xdpd/management/switch_manager.h whose main methods are listed in Figure 4.1.

4.3 Resource Description

Datapath resource descriptions are required both for network management and control systems in order to distinguish different types of network nodes and their capabilities. The knowledge about a resource is presented to the network users or applications and used for the reservation and control of some parts of the resource (or the whole resource). Resource description can be also used internally in HAL, for example, for the translation of OpenFlow requests into device-specific configuration. In ALIEN we employ different approaches for resource description as appropriate by the device type, as explained in the remainder of this section.

4.3.1 Resources in Programmable Packet Switching Devices

Programmable packet switching devices, such as network processors and CPU-based switches, have a basic device structure (i.e. flow-tables, ports) which is coherent with the OpenFlow data model. However, the device programmability features specific to each platform open a new aspect of datapath management. The node management system (could be performed by the OpenFlow controller) may provide knowledge about the required data plane protocols.

Protocol description contains the header format and header placement within the packet and allows a device to locate, parse, modify or remove such a protocol header during packet processing. An example of such description, using the P4 language [12] is presented below. The network headers description contains fields labels, position from the beginning of the header and bit length meaning are presented here:

```
header ethernet {
```

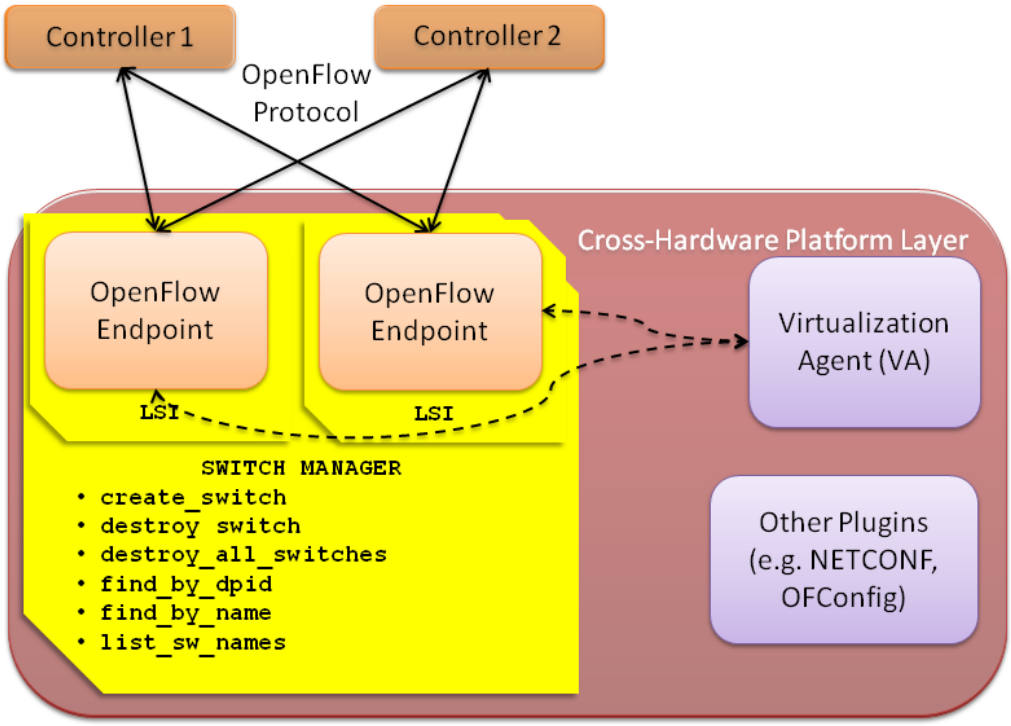

Figure 4.1: Implementation of the Logical Switch Instances management within the HAL architecture.

```
  fields {
    dst_addr : 48; // width in bits
    src_addr : 48;
    ethertype : 16;
  }
}

header ipv4 {
  fields {
    __skip__ : 8; // not interpreted bits
    dscp : 6;
    ecn : 2;
    __skip__ : 56;
    src_ip : 32;
    dst_ip : 32;
    __skip__ : 16;
    ip_proto : 8;
  }
}
header udp {
  field {
    src_port : 16;
    dst_port : 16;
    __skip__ : 32;
```

```
  }
}
header vxlan {
  field {
    __skip__ : 32;
    segment_id : 24;
    __skip__ : 8;
  }
}
```

Some header fields, which are not required during packet processing in the datapath, and are not used by flow entries, are to be interpreted by the special directive *__skip__*. The sequence of headers parsing within the packet is specified in the following description:

```
parser ethernet {
  switch(ethertype) {  \\ header field based lookup
    case 0x800: ipv4;  \\ what is next header
  }
}
parser ipv4 {
  switch(ip_proto) {
    case 0x11: udp;
  }
}
parser udp {
  switch(dst_port) {
    case 0x12B5: vxlan;
  }
}
```

As listed above, you can distinguish the next-parsed header by a specific value of any proper header field.

When these descriptions are applied over a network node, then the node forwarding engine is capable of parsing and processing based only on Ethernet, IPv4, UDP and VXLAN headers. Then, other header fields passed in flow entries are not recognized and flow entries are skipped.

During the whole device life-time, one set of protocols may be replaced with a new set of protocols, depending on the actual network device role in the network. The current implementation of HAL does not support data plane protocol knowledge management. Deliverable [D2.2] describes a proposition of datapath architecture that could provide protocol knowledge management capabilities which may be implemented in the form of the early prototype till the end of the ALIEN project.

4.3.2 Resources in Lightpath Devices

As described earlier in this deliverable, circuit switches are functioning in a completely different way compared to the packet processing devices, and thus require a new set of resource description. The ALIEN project has decided to use the proposed extension to the OpenFlow protocol for circuit switched devices [4] that have been already included in the succeeding versions of the protocol (i.e. v1.4). This extension was implemented in the HAL prototype for L0 switch (ADVA DWDM system).

The structure used to describe a physical port in packet processing devices has been modified in order to be able to describe a circuit switch port. At this point we should note the existence of *peer_datapath_id* and *peer_port_no* fields since it is not possible in optical networks to discover neighbors using LLDP frames.

```
/*
 * Circuit switch physical port description
 */
struct ofp_phy_cport {
    uint16_t port_no;
    uint8_t hw_addr[OFP_ETH_ALEN];       /* Ethernet address - 6-byte */
    uint8_t name[OFP_MAX_PORT_NAME_LEN]; /* Null-terminated - 16 byte */

    uint32_t config;           /* Bitmap of OFPPC_* flags. */
    uint32_t state;            /* Bitmap of OFPPS_* flags. */

    /* Bitmaps of OFPPF_* that describe features.  All bits zeroed if
     * unsupported or unavailable. */
    uint32_t curr;             /* Current features. */
    uint32_t advertised;       /* Features being advertised by the port. */
    uint32_t supported;        /* Features supported by the port. */
    uint32_t peer;             /* Features advertised by peer. */

    uint32_t supp_sw_tdm_gran; /* TDM switching granularity OFPTSG_* flags */
    uint16_t supp_swtype;      /* Bitmap of switching type OFPST_* flags */
    uint16_t peer_port_no;     /* Discovered peer's switchport number */
    uint64_t peer_datapath_id; /* Discovered peer's datapath id */
    uint16_t num_bandwidth;    /* Identifies number of bandwidth array elements */
    uint8_t  pad[6];           /* Align to 64 bits */
    uint64_t bandwidth[0];     /* Bitmap of OFPCBL_* or OFPCBT_* flags */
};
```

The *cfow_mod* message contains the so-called logical equivalent of the *ofp_match* structure and it is the message sent to the circuit switch in order to modify its cross-connect table.

```
/* Circuit flow setup, modification and teardown (controller -> datapath) */
struct ofp_cflow_mod {
   struct ofp_header header;              /* Openflow header */
   uint16_t command;                      /* one of OFPFC_* commands */
   uint16_t hard_timeout;                 /* max time to connection tear down,
                                             if 0 then explicit tear-down required */
   uint8_t pad[4];                        /* Align to 64 bits */
   struct ofp_connect_ocs connect;        /* 8B followed by variable length arrays */
   struct ofp_action_header actions[0];   /* variable number of action */
};
```

The *ofp_connect_ocs* structure describes the cross-connected ports inside the switch:

```
/* Description of a cross-connection */

struct ofp_connect_ocs {
   uint16_t wildcards;                /* identifies ports to use below */
   uint16_t num_components;           /* identifies number of cross-connects
                                         to be made - num array elements */
```

```
   uint8_t pad[4];                      /* Align to 64 bits */

   uint16_t in_port[0];                 /* OFPP_* ports - real or virtual */
   uint16_t out_port[0];                /* OFPP_* ports - real or virtual */

   struct ofp_tdm_port in_tport[0];     /* Description of a TDM channel */
   struct ofp_tdm_port out_tport[0];

   struct ofp_wave_port in_wport[0];    /* Description of a Lambda channel */
   struct ofp_wave_port out_wport[0];
};
```

Finally, a *cport_status* message has been added to allow the switch to inform the controller about changes in the state of the physical circuit port:

```
struct ofp_cport_status {
   struct ofp_header header;
   uint8_t reason;                /* One of OFPPR_* */
   uint8_t pad[7];                /* Align to 64 bits */
   struct ofp_phy_cport desc;     /* Circuit port description */
};
```

4.3.3 Resources in Point-to-Multipoint Devices

Point-to-multipoint devices (GEPON and DOCSIS devices in the ALIEN project) are exposed to the network management (and network control) as a OpenFlow network node, abstracting the whole access network.

4.3.3.1 Resources in DOCSIS Architecture

DEVICE MAP

Certain structures are required in order to maintain the coherence between all devices connected to the ALIEN-Hardware INtegration Proxy (ALHINP) and the virtual model exposed to the OpenFlow controller. As soon as a cable modem is detected in the network, ALHINP creates the corresponding structure for it. After assigning the corresponding VLAN_VID, the rest of the structure is filled with the parameters as soon as they are discovered (no relationship between OUI and CM is required in advance as they are dynamically detected by ALHINP).

```
struct device {
    uint64_t MAC_OUI;    /*MAC of the OUI*/
    uint64_t DPID;       /*DPID of the OUI*/
    uint16_t vlan;       /*Vlan provisioned over CMTS*/
};

std::map< uint64_t mac_CM, struct device> devicemap;
```

The *devicemap* stores for each cablemodem the corresponding OpenFlow Parameters and VLAN assigned by ALHINP proxy. This map is dynamically filled when a connection from OUI or CM is detected.

PORT STRUCTURES

```
struct realport {
    uint64_t DPID;           /*DPID of the OUI*/
    uint32_t realport_id;    /*Real port ID at DPID*/
};

std::map< uint32_t virtualport, struct realport> portmap;
```

The *portmap* stores the ports enabled by the proxy, which are virtually exposed to the controller. For each virtualport its corresponding realport and OUI_DPID is stored.

ALHINP CONFIGURATION

The configuration of ALHINP,which is the component that orchestrates all the devices of the architecture,is stored in the next structure, where network user-defined parameters are defined.

```
struct ALHINP {
    uint64_t ALHINP_DPID;        /*DPID of ALHINP exposed to the controller*/

    std::string CTRL_IP;         /*Controller IP*/
    std::string CTRL_OF_VERSION; /*OF version of the controller */
    std::string CTRL_PORT;       /*OF Port */

    std::string LISTEN_IP_AGS;   /*OF AGS LISTENING IP */
    std::string LISTEN_PORT_AGS; /*OF AGS LISTENING PORT */
    uint32_t    CMTS_PORT;       /*Port where CMTS is attached*/
    uint32_t    DPS_PORT;        /*Port where Provisioning system is attached*/
    uint32_t    ALHINP_PORT;     /*Port for the ALHINP OUI connections*/

    std::string LISTEN_IP_OUI;   /*OF OUI listening IP */
    std::string LISTEN_PORT_OUI; /*OF OUI listening Port */
    uint32_t    NETPORT;         /*Port connected to the Cablemodem*/

    std::string DPS_IP;          /*DOCSIS Provisioning server IP */
    std::string CMTS_IP;         /*CMTS IP */
};
```

This structure describes the overall *ALHINP* configuration parameters, given by the user, according to the architecture setup.

5 Summary

This document provides the implementation details of Hardware Abstraction Layer (HAL). The HAL provides a platform for OpenFlow protocol implementation on non-OpenFlow capable network devices. Two sub-layers which comprise HAL are Cross-Hardware Platform layer and Hardware-Specific layer. Based on the design specifications developed within the ALIEN project, this document provides an account of implementing the two aforementioned layers. The goal of Cross-Hardware Platform layer is to provide a device abstraction using the services of Hardware-Specific layer which has to deal with the underlying hardware platform peculiarities. The Cross-Hardware Platform layer implementation is unanimous for all hardware platforms which makes it an ideal place to implement functionalities like network management and virtualization. Moreover, it also helps achieve an OpenFlow version agnostic device abstraction. The implementation of Hardware-Specific layer has to be carried out for each underlying hardware platform and sometimes it is even different from device to device within the same network platform category. Therefore, depending on the hardware architecture of the underlying device, the hardware-specific layer has to be implemented adaptively to offer the functionality described in the specifications.

The document describes the implementation details of Cross-Hardware Platform layer and its plug-ins, i.e., NETCONF and Virtualization Agent. In addition, the experiences are shared for implementing Hardware-Specific layer on most widely used network device platforms such as programmable hardware, transport network devices (optical or circuit switch) and closed platform with proprietary communication protocols such as GEPON. By achieving functional implementations of Hardware-Specific layer on the aforementioned devices and its integration with the Cross-Hardware Platform layer to realize OpenFlow capabilities validates the feasibility of HAL architecture design and its specifications.

References

[1] EZappliance: NP-3 Network Processor. http://www.ezchip.com/p_np3.htm.

[2] NetFPGA. http://netfpga.org/.

[3] Revised OpenFlow Library. http://www.roflibs.org/.

[4] Extension to the OpenFlow Protocol in support of Circuit Switching. http://archive.openflow.org/wk/images/8/81/OpenFlow_Circuit_Switch_Specification_v0.3.pdf, 2010.

[5] Deliverable D3.1: Hardware platforms and switching constraints. http://www.fp7-alien.eu/files/deliverables/D3.1-ALIEN-final.pdf, 2013.

[6] Deliverable D3.2: Specification of hardware specific parts. http://www.fp7-alien.eu/files/deliverables/D3.2-ALIEN-final.pdf, 2013.

[7] Deliverable D2.2: Specification of Hardware Abstraction Layer. http://www.fp7-alien.eu/files/deliverables/D2.2-ALIEN-final.pdf, 2014.

[8] OpenFlow Specifications. https://www.opennetworking.org/sdn-resources/onf-specifications/openflow, 2014.

[9] Ł. Ogrodowczyk, et al. Hardware Abstraction Layer for Non-OpenFlow Capable Devices. In *Proceedings of the TERENA Networking Conference (TNC)*, May 2014.

[10] D. Parniewicz, et al. Design and Implementation of an OpenFlow Hardware Abstraction Layer. In *Proceedings of the ACM SIGCOMM Workshop on Distributed Cloud Computing (DCC)*, August 2014. Accepted for publication.

[11] R. Doriguzzi Corin et al. VeRTIGO: Network Virtualization and Beyond. In *Proceedings of the European Workshop on Software Defined Networking (EWSDN)*, pages 24--29, Oct 2012.

[12] D. Heimbigner. P4: A Logic Language for Process Programming. In *Proceedings of the 5th International Software Process Workshop on Experience with Software Process Models*, ISPW '90, pages 67--70, Los Alamitos, CA, USA, 1990.

[13] R. Enns, et al. Network Configuration Protocol (NETCONF). https://tools.ietf.org/html/rfc6241, 2011.

[14] R. Sherwood, et al. Carving Research Slices out of Your Production Networks with OpenFlow. *SIGCOMM Comput. Commun. Rev.*, 40(1):129--130, January 2010.

[15] L. Rizzo. netmap: A Novel Framework for Fast Packet I/O. In *Proceedings of the USENIX Annual Technical Conference*, pages 101--112, Boston, MA, 2012.

Acronyms

AFA: Abstracted Forwarding API
ALHINP: ALIEN Hardware INtegration Proxy
API: Application Programming Interface
CM: Cable Modem
CMM: Control and Management Module
CHPL: Cross-Hardware Platform Layer
DOCSIS: Data Over Cable Service Interface Specification
FPGA: Field Programmable Fate Gate Array
HAL: Hardware Abstraction Layer
HSL: Hardware Specific Layer
HSP: Hardware Specific Part
HPA: Hardware Pipeline API
LTE: Long Term Evolution
NMS: Network Management System
NPU: Network Processing Unit
PAD: Programmable Abstraction for Datapath
ROADM: Reconfigurable Optical Add drop Module
ROFL: Revised OpenFlow Library
SDN: Software Defined Networking
OF: OpenFlow
TCAM: Ternary Content-Addressable Memory
TCP: Transmission Control Protocol
TLS: Transport Layer Security
VA: Virtualization Agent
VG: Virtual Gateway
VoD: Video on-Demand
xDPd: Extensible Data Path Daemon

Appendix A

VA subset	VA abstract method	VA method implementation in ROFL
Slice	Port presence	**Function declaration:** bool has_port(string port_name); **located in:** virtual-agent/slice.h
Virtualization Agent	Is active	**Function declaration:** bool is_active(); **located in:** virtual-agent/virtualagent.h
	Add slice	**Function declaration:** void add_slice(slice∗ slice_to_add, bool connect); **located in:** virtual-agent/virtualagent.h
	Add flowspace	**Function declaration:** void add_flowspace(flowspace∗ flowspace_to_add); **located in:** virtual-agent/virtualagent.h
	Add switch	**Function declaration:** void add_switch(va_switch∗ switch_to_add); **located in:** virtual-agent/virtualagent.h
	Check slice existence	**Function declaration:** bool check_slice_existance(string slice_name, uint64_t dpid); **located in:** virtual-agent/virtualagent.h
	Flow Entry Analysis and modification	**Function declaration:** of1x_flow_entry_t∗ flow_entry_analysis(cofctl ∗ctl, of1x_flow_entry_t ∗entry, openflow_switch∗ sw); **located in:** virtual-agent/virtualagent.h
	Actions analysis and modification	**Function declaration:** of1x_action_group_t∗ action_analysis(cofctl ∗ctl, of1x_action_group_t ∗action_group, openflow_switch∗ sw); **located in:** virtual-agent/virtualagent.h
	Group analysis and modification	**Function declaration:** cofmsg_group_mod∗ group_mod_analysis(cofctl ∗ctl, cofmsg_group_mod ∗msg, openflow_switch∗ sw); **located in:** virtual-agent/virtualagent.h
Flowspace	Stores the slices flowspaces	**Function declaration:** struct flowspace; **located in:** virtual-agent/flowspace.h
VA Virtual Switch	Check Flowspace match	**Function declaration:** bool check_match(const of1x_packet_matches_t pkt, std::list<flowspace_match_t∗> it); **located in:** virtual-agent/va_switch.h
VA Virtual Switch	Compare match	**Function declaration:** bool compare_match_flow(const of1x_packet_matches_t∗ pkt, flowspace_match_t∗ it); **located in:** virtual-agent/va_switch.h

Table A.1: Virtualization Agent implementation within the xDPd's code

AFA subset	AFA abstract method	AFA method implementation in ROFL
Datapath Management	Init-driver	**Function declaration:** hal_result_t hal_driver_init (const char∗ extra_params); **located in:** rofl-core\src\rofl\datapath\hal\driver.h
	Destroy-driver	**Function declaration:** hal_result_t hal_driver_destroy (void); **located in:** rofl-core\src\rofl\datapath\hal\driver.h
	Create-switch	**Function declaration:** hal_result_t hal_driver_create_switch (char∗ name, uint64_t dpid, of_version_t of_version, unsigned int num_of_tables, int∗ ma_list); **located in:** rofl-core\src\rofl\datapath\hal\driver.h
	Get-switch	**Function declaration:** of_switch_snapshot_t∗ hal_driver_get_switch_snapshot_by_dpid(uint64_t dpid); **located in:** rofl-core\src\rofl\datapath\hal\driver.h
	Destroy-switch	**Function declaration:** hal_result_t hal_driver_destroy_switch_by_dpid (uint64_t dpid); **located in:** rofl-core\src\rofl\datapath\hal\driver.h
	Get-ports	**Function declaration:** switch_port_name_list_t∗ hal_driver_get_all_port_names (void); **located in:** rofl-core\src\rofl\datapath\hal\driver.h
	Get-port	**Function declarations:** switch_port_snapshot_t∗ hal_driver_get_port_snapshot_by_name (const char ∗name); switch_port_snapshot_t∗ hal_driver_get_port_snapshot_by_num (uint64_t dpid, unsigned int port_num); **located in:** rofl-core\src\rofl\datapath\hal\driver.h
	Enable-port	**Function declarations:** hal_result_t hal_driver_bring_port_up (const char∗ name); hal_result_t hal_driver_bring_port_down_by_num (uint64_t dpid, unsigned int port_num); **located in:** rofl-core\src\rofl\datapath\hal\driver.h
Datapath Management	Disable-port	**Function declarations:** hal_result_t hal_driver_bring_port_down (const char∗ name); hal_result_t hal_driver_bring_port_down_by_num (uint64_t dpid, unsigned int port_num); **located in:** rofl-core\src\rofl\datapath\hal\driver.h
	Attach-port-to-switch	**Function declaration:** hal_result_t hal_driver_attach_port_to_switch (uint64_t dpid, const char∗ name, unsigned int∗ port_num); **located in:** rofl-core\src\rofl\datapath\hal\driver.h
	Detach-port-from-switch	'***Function declarations:***' hal_result_t hal_driver_detach_port_from_switch (uint64_t dpid, const char∗ name); hal_result_t hal_driver_detach_port_from_switch_at_port_num (uint64_t dpid, const unsigned int port_num); **located in:** rofl-core\src\rofl\datapath\hal\driver.h

Datapath Configuration	Set-port-drop	**Function declaration:** hal_result_t hal_driver_of1x_set_port_drop_received_config (uint64_t dpid, unsigned int port_num, bool drop_received); **located in:** rofl-core\src\rofl\datapath\hal\openflow\openflow1x\of1x_driver.h
	Set-port-forward	**Function declaration:** hal_result_t hal_driver_of1x_set_port_forward_config (uint64_t dpid, unsigned int port_num, bool forward); **located in:** rofl-core\src\rofl\datapath\hal\openflow\openflow1x\of1x_driver.h
	Set-port-packet-in	**Function declaration:** hal_result_t hal_driver_of1x_set_port_generate_packet_in_config (uint64_t dpid, unsigned int port_num, bool generate_packet_in); **located in:** rofl-core\src\rofl\datapath\hal\openflow\openflow1x\of1x_driver.h
	Set-port-advertise	**Function declaration:** hal_result_t hal_driver_of1x_set_port_advertise_config (uint64_t dpid, unsigned int port_num, uint32_t advertise); **located in:** rofl-core\src\rofl\datapath\hal\openflow\openflow1x\of1x_driver.h
	Set-pipeline-config	**Function declaration:** hal_result_t hal_driver_of1x_set_pipeline_config (uint64_t dpid, unsigned int flags, uint16_t miss_send_len); **located in:** rofl-core\src\rofl\datapath\hal\openflow\openflow1x\of1x_driver.h
	Set-table-config	**Function declaration:** hal_result_t hal_driver_of1x_set_table_config (uint64_t dpid, unsigned int table_id, of1x_flow_table_miss_config_t config); **located in:** rofl-core\src\rofl\datapath\hal\openflow\openflow1x\of1x_driver.h
Datapath Configuration	Packet-out	**Function declaration:** hal_result_t hal_driver_of1x_process_packet_out (uint64_t dpid, uint32_t buffer_id, uint32_t in_port, of1x_action_group_t* action_group, uint8_t* buffer, uint32_t buffer_size); **located in:** rofl-core\src\rofl\datapath\hal\openflow\openflow1x\of1x_driver.h
	Add-flow	**Function declaration:** hal_result_t hal_driver_of1x_process_flow_mod_add (uint64_t dpid, uint8_t table_id, of1x_flow_entry_t** flow_entry, uint32_t buffer_id, bool check_overlap, bool reset_counts); **located in:** rofl-core\src\rofl\datapath\hal\openflow\openflow1x\of1x_driver.h
	Modify-flow	**Function declaration:** hal_result_t hal_driver_of1x_process_flow_mod_modify (uint64_t dpid, uint8_t table_id, of1x_flow_entry_t** flow_entry, uint32_t buffer_id, of1x_flow_removal_strictness_t strictness, bool reset_counts); **located in:** rofl-core\src\rofl\datapath\hal\openflow\openflow1x\of1x_driver.h
	Delete-flow	**Function declaration:** hal_result_t hal_driver_of1x_process_flow_mod_delete (uint64_t dpid, uint8_t table_id, of1x_flow_entry_t* flow_entry, uint32_t out_port, uint32_t out_group, of1x_flow_removal_strictness_t strictness); **located in:** rofl-core\src\rofl\datapath\hal\openflow\openflow1x\of1x_driver.h

	Get-flow-stats	**Function declaration:** of1x_stats_flow_msg_t∗ hal_driver_of1x_get_flow_stats (uint64_t dpid, uint8_t table_id, uint32_t cookie, uint32_t cookie_mask, uint32_t out_port, uint32_t out_group, of1x_match_group_t ∗const matches); **located in:** rofl-core\src\rofl\datapath\hal\openflow\openflow1x\of1x_driver.h
	Add-group	**Function declaration:** rofl_of1x_gm_result_t hal_driver_of1x_group_mod_add (uint64_t dpid, of1x_group_type_t type, uint32_t id, of1x_bucket_list_t ∗∗buckets); **located in:** rofl-core\src\rofl\datapath\hal\openflow\openflow1x\of1x_driver.h
	Modify-group	**Function declaration:** rofl_of1x_gm_result_t hal_driver_of1x_group_mod_modify (uint64_t dpid, of1x_group_type_t type, uint32_t id, of1x_bucket_list_t ∗∗buckets); **located in:** rofl-core\src\rofl\datapath\hal\openflow\openflow1x\of1x_driver.h
	Delete-group	**Function declaration:** rofl_of1x_gm_result_t hal_driver_of1x_group_mod_delete (uint64_t dpid, uint32_t id); **located in:** rofl-core\src\rofl\datapath\hal\openflow\openflow1x\of1x_driver.h
Datapath Configuration	Get-group-stats	**Function declarations:** of1x_stats_group_msg_t∗ hal_driver_of1x_get_group_stats (uint64_t dpid, uint32_t id); of1x_stats_group_msg_t∗ hal_driver_of1x_get_group_all_stats (uint64_t dpid, uint32_t id); **located in:** rofl-core\src\rofl\datapath\hal\openflow\openflow1x\of1x_driver.h
Notification	Add-port	**Function declaration:** hal_result_t hal_cmm_notify_port_add (switch_port_snapshot_t∗ port_snapshot); **located in:** rofl-core\src\rofl\datapath\hal\cmm.h
	Modify-port	**Functions declarations**: hal_result_t hal_cmm_notify_port_status_changed (switch_port_snapshot_t∗ port_snapshot); hal_result_t hal_cmm_notify_monitoring_state_changed (monitoring_snapshot_state_t∗ monitoring_snapshot); **located in:** rofl-core\src\rofl\datapath\hal\cmm.h
	Delete-port	**Functions declaration**: hal_result_t hal_cmm_notify_port_delete (switch_port_snapshot_t∗ port_snapshot); **located in:** rofl-core\src\rofl\datapath\hal\cmm.h
	Packet-in	**Functions declaration**: hal_result_t hal_cmm_process_of1x_packet_in (uint64_t dpid, uint8_t table_id, uint8_t reason, uint32_t in_port, uint32_t buffer_id, uint8_t∗ pkt_buffer, uint32_t buf_len, uint16_t total_len, packet_matches_t∗ matches); **located in:** rofl-core\src\rofl\datapath\hal\openflow\openflow1x\of1x_cmm.h
	Flow-removed	**Functions declaration**: hal_result_t hal_cmm_process_of1x_flow_removed (uint64_t dpid, uint8_t reason, of1x_flow_entry_t∗ removed_flow_entry); **located in:** rofl-core\src\rofl\datapath\hal\openflow\openflow1x\of1x_cmm.h

Table A.2: Abstract Forwarding API implementation within the ROFL project

HPA subset	HPA abstract method	HPA method implementation in ROFL
Packet operations	Get-packet-size	**Function declaration:** uint32_t platform_packet_get_size_bytes (datapacket_t ∗const pkt); **located in:** /platform/packet.h
	Get-port-in	**Function declaration:** uint32_t platform_packet_get_port_in (datapacket_t ∗const pkt); **located in:** /platform/packet.h
Packet operations	Get-packet-field	**Function declaration:** uint64_t platform_packet_get_eth_src (datapacket_t ∗const pkt); uint64_t platform_packet_get_eth_dst (datapacket_t ∗const pkt); uint16_t platform_packet_get_eth_type (datapacket_t ∗const pkt); uint16_t platform_packet_get_vlan_vid (datapacket_t ∗const pkt); uint8_t platform_packet_get_vlan_pcp (datapacket_t ∗const pkt); uint32_t platform_packet_get_mpls_label (datapacket_t ∗const pkt); uint8_t platform_packet_get_mpls_tc (datapacket_t ∗const pkt); bool platform_packet_get_mpls_bos (datapacket_t ∗const pkt); uint8_t platform_packet_get_ip_proto (datapacket_t ∗const pkt); uint8_t platform_packet_get_ip_ecn (datapacket_t ∗const pkt); uint8_t platform_packet_get_ip_dscp (datapacket_t ∗const pkt); uint32_t platform_packet_get_ipv4_src (datapacket_t ∗const pkt); uint32_t platform_packet_get_ipv4_dst (datapacket_t ∗const pkt); uint16_t platform_packet_get_tcp_src (datapacket_t ∗const pkt); uint16_t platform_packet_get_tcp_dst (datapacket_t ∗const pkt); uint16_t platform_packet_get_udp_src (datapacket_t ∗const pkt); uint16_t platform_packet_get_udp_dst (datapacket_t ∗const pkt); uint8_t platform_packet_get_icmpv4_type (datapacket_t ∗const pkt); uint8_t platform_packet_get_icmpv4_code (datapacket_t ∗const pkt); uint8_t platform_packet_get_pppoe_code (datapacket_t ∗const pkt); uint8_t platform_packet_get_pppoe_type (datapacket_t ∗const pkt); uint16_t platform_packet_get_pppoe_sid (datapacket_t ∗const pkt); **located in:** /platform/packet.h

	Set-packet-field	**Function declaration:** void platform_packet_set_eth_src (datapacket_t* pkt, uint64_t eth_src); void platform_packet_set_eth_dst (datapacket_t* pkt, uint64_t eth_dst); void platform_packet_set_eth_type (datapacket_t* pkt, uint16_t eth_type); void platform_packet_set_vlan_vid (datapacket_t* pkt, uint16_t vlan_vid); void platform_packet_set_vlan_pcp (datapacket_t* pkt, uint8_t vlan_pcp); void platform_packet_set_mpls_label (datapacket_t* pkt, uint32_t label); void platform_packet_set_mpls_tc (datapacket_t* pkt, uint8_t tc); void platform_packet_set_mpls_bos (datapacket_t* pkt, bool bos); void platform_packet_set_ip_proto (datapacket_t* pkt, uint8_t ip_proto); void platform_packet_set_ip_dscp (datapacket_t* pkt, uint8_t ip_dscp); void platform_packet_set_ip_ecn (datapacket_t* pkt, uint8_t ip_ecn); void platform_packet_set_ipv4_src (datapacket_t* pkt, uint32_t ip_src); void platform_packet_set_ipv4_dst (datapacket_t* pkt, uint32_t ip_dst); void platform_packet_set_tcp_src (datapacket_t* pkt, uint16_t tcp_src); void platform_packet_set_tcp_dst (datapacket_t* pkt, uint16_t tcp_dst); void platform_packet_set_udp_src (datapacket_t* pkt, uint16_t udp_src); void platform_packet_set_udp_dst (datapacket_t* pkt, uint16_t udp_dst); void platform_packet_set_icmpv4_type (datapacket_t* pkt, uint8_t type); void platform_packet_set_icmpv4_code (datapacket_t* pkt, uint8_t code); void platform_packet_set_pppoe_type (datapacket_t* pkt, uint8_t type); void platform_packet_set_pppoe_code (datapacket_t* pkt, uint8_t code); void platform_packet_set_pppoe_sid (datapacket_t* pkt, uint16_t sid); **located in:** /platform/packet.h
	Copy-time-to-live	**Function declaration:** void platform_packet_copy_ttl_out (datapacket_t* pkt); **located in:** /platform/packet.h
	Decrement-time-to-live	**Function declaration:** void platform_packet_dec_nw_ttl (datapacket_t* pkt); **located in:** /platform/packet.h
Packet operations	Pop-tag	**Function declaration:** void platform_packet_pop_vlan (datapacket_t* pkt); void platform_packet_pop_mpls (datapacket_t* pkt, uint16_t ether_type); void platform_packet_pop_pppoe (datapacket_t* pkt, uint16_t ether_type); **located in:** /platform/packet.h
Packet operations	Push-tag	**Function declaration:** void platform_packet_push_vlan (datapacket_t* pkt, uint16_t ether_type); void platform_packet_push_mpls (datapacket_t* pkt, uint16_t ether_type); void platform_packet_push_pppoe (datapacket_t* pkt, uint16_t ether_type); **located in:** /platform/packet.h
	Drop-packet	**Function declaration:** void platform_packet_drop (datapacket_t* pkt); **located in:** /platform/packet.h
	Output-packet	**Function declaration:** void platform_packet_output (datapacket_t* pkt, switch_port_t* port); **located in:** /platform/packet.h
Memory management	Allocate-memory	**Function declaration:** void* platform_malloc(size_t length); **located in:** /platform/memory.h

	Free-memory	**Function declaration:** void platform_free(void∗ data); **located in:** /platform/memory.h
	Copy-memory	**Function declaration:** void∗ platform_memcpy (void∗ dst, const void∗ src, size_t length); **located in:** /platform/memory.h
	Move-memory	**Function declaration:** void∗ platform_memmove (void∗ dst, const void∗ src, size_t lengt); **located in:** /platform/memory.h
	Set-memory	**Function declaration:** void∗ platform_memset (void∗ src, int c, size_t length); **located in:** /platform/memory.h
Mutex & Counter atomic operations	Init-mutex	**Function declaration:** platform_mutex_t∗ platform_mutex_init (void∗ params); **located in:** /platform/lock.h
Mutex & Counter atomic operations	Destroy-mutex	**Function declaration:** void platform_mutex_destroy (platform_mutex_t∗ mutex); **located in:** /platform/lock.h
	Lock-mutex	**Function declaration:** void platform_mutex_lock (platform_mutex_t∗ mutex); **located in:** /platform/lock.h
	Unlock-mutex	**Function declaration:** void platform_mutex_unlock (platform_mutex_t∗ mutex); **located in:** /platform/lock.h
	Increase-counter	**Function declaration:** void platform_atomic_inc32 (uint32_t∗ counter, platform_mutex_t∗ mutex); void platform_atomic_inc64 (uint64_t∗ counter, platform_mutex_t∗ mutex); **located in:** /platform/atomic_operations.h
	Decrease-counter	**Function declaration:** void platform_atomic_dec32 (uint32_t∗ counter, platform_mutex_t∗ mutex); void platform_atomic_dec64 (uint64_t∗ counter, platform_mutex_t∗ mutex); **located in:** /platform/atomic_operations.h
Notification	Process-packet-in-pipeline	**Function declaration:** void __of1x_process_packet_pipeline (const of_switch_t ∗sw, datapacket_t ∗const pkt); void of1x_process_packet_out_pipeline (const of1x_switch_t ∗sw, datapacket_t ∗const pkt, const of1x_action_group_t∗ apply_actions_group); **located in:** /openflow/openflow1x/pipeline/of1x_pipeline_pp.h

Table A.3: Hardware Pipeline API implementation within the ROFL project

Function Declaration	Description
virtual void init(void);	Initialization of the Plug-In
virtual std::string get_name(void);	Retrieval of the Plug-In name
virtual void notify_port_added(const switch_port_snapshot_t* port_snapshot);	Notification of the Plug-In, that a new port was added
virtual void notify_port_attached(const switch_port_snapshot_t* port_snapshot);	Notification of the Plug-In, that a new port was attached to a LSI
virtual void notify_port_status_changed(const switch_port_snapshot_t* port_snapshot);	Notification of the Plug-In, that a port status changed
virtual void notify_port_detached(const switch_port_snapshot_t* port_snapshot);	Notification of the Plug-In, that a port was detached from a LSI
virtual void notify_port_deleted(const switch_port_snapshot_t* port_snapshot);	Notification of the Plug-In, that a new port was deleted
virtual void notify_monitoring_state_changed(const monitoring_snapshot_state_t* monitoring_snapshot);	Notification of the Plug-In, that a monitored state changed.

Table A.4: Interfaces for Plug-in Manager

Header File	Description
xdpd/virtual-agent/va_switch.h	switch settings for Virtualization Agent database
vxdpd/virtual-agent/flowspace.h	flowspace implementation
xdpd/virtual-agent/slice.h	slice implementation
xdpd/virtual-agent/virtualagent.h	VA definitions

Table A.5: Header files defining Slicer functions

Header File	Description
xdpd/management/plugins/config/virtual-agent/flowspace_scope.h	flowspace rules
xdpd/management/plugins/config/virtual-agent/slice_scope.h	slice settings and controllers' details
xdpd/management/plugins/config/virtual-agent/virtual-agent_scope.h	virtualization agent settings
xdpd/management/plugins/config/root_scope.*	registration of the above-described settings to the plugin manager

Table A.6: Header files defining access functions for virtualization agent database

Header File	Description
xdpd/openflow/endpoint.h	added functions that allow the VA to obtain the pointers to the controllers connected to the endpoint
xdpd/openflow/openflow_switch	added a function to retrieve the endpoint instance
xdpd/openflow/openflow1Y/openflow1Y_switch	added a function that creates new virtual switches with no controllers and one that adds controllers to the virtual switches.
xdpd/openflow/openflow1Y/of1Y_endpoint	added the connection to the VA to enable the slicing mechanism
xdpd/cmm.cc	added the mechanism that selects the right controller for the new flows based on the of1X_packet_matches_t data and on the flowspace rules.

Table A.7: List of xDPD code files modified to enable virtualization agent functions

Header File	Description
rofl-core/src/rofl/datapath/hal/driver.h	HAL driver management functions (AFA Management)
rofl-core/src/rofl/datapath/hal/cmm.h	HAL port event callbacks (AFA Notification)
rofl-core/src/rofl/datapath/hal/openflow/openflow1x/of1x/driver.h	HAL datapath configuration functions (AFA Configuration)
rofl-core/src/rofl/datapath/hal/openflow/openflow1x/of1x/cmm.h	HAL datapath event callbacks (AFA Notification)

Table A.8: A set of C header files containing AFA API function declarations

Header File	Description
rofl-core/src/rofl/datapath/pipeline/platform/packet.h	pipeline packet processing functions (HPA Packet Operations)
rofl-core/src/rofl/datapath/pipeline/platform/memory.h	a set of calls used by library to perform dynamic memory allocation/deallocation, as well as other memory operations (like copy or move) (HPA Memory management)
rofl-core/src/rofl/datapath/pipeline/platform/lock.h	a set of calls used by HPA to perform mutual exclusion operations (HPA Lock operation)
rofl-core/src/rofl/datapath/pipeline/platform/atomic/operations.h	a set of calls used by HPA to perform counters increments (HPA Counter Atomic Operations)
rofl-core/src/rofl/datapath/pipeline/openflow/openflow1x/pipeline/of1x/pipeline.h	HPA pipeline packet processing routines (HPA Notification)

Table A.9: A set of C header files containing HPA function declarations