A L I E N

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 D2.2
Specification of Hardware Abstraction Layer

Version 1.0

Due date:	31/03/2014
Submission date:	29/04/2014
Deliverable leader:	UNIVBRIS
Editor:	Mehdi Rashidi
Internal reviewer:	Eduardo Jacob (UPV/EHU)
Author list:	Artur Binczewski, Bartosz Belter, Łukasz Ogrodowczyk, Iwo Olszewski, Damian Parniewicz (PSNC), Hagen Woesner, Umar Toseef, Adel Zaalouk, Kostas Pentikousis (EICT), Jon Matias, Eduardo Jacob, Victor Fuentes (UPV/EHU), Richard G. Clegg (UCL), Roberto Doriguzzi (Create-Net), Marek Michalski, Remigiusz Rajewski (PUT).

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)

Abstract

This document describes a Hardware Abstraction Layer (HAL) designed to support OpenFlow protocol implementation regardless of protocol's version on non-compatible OpenFlow network devices. This abstraction layer supports various hardware platforms in terms of data plane architecture and closed control plane protocols. The document describes the software modules inside the HAL, their functionality and interfaces. The aim of this document is to present the features of the designed HAL such as extensibility (programmability) and modules reusability for third party users.

Project:	ALIEN (Grant Agr. No. 317880)
Deliverable Number:	D2.2
Date of Issue:	29/04/14

Table of Contents

Executive Summary 6

1. Introduction 7

2. HAL Platform Requirements and Support Challenges 8
    - 2.1 x86-based Packet Processing Devices 8
    - 2.2 Lightpath Devices 10
    - 2.3 Point to Multi-point Devices 10
    - 2.4 Programmable Network Processors 12

3. HAL Logical Architecture 13
    - 3.1 Cross-Hardware Platform Layer 14
        - 3.1.1 Network Management 15
        - 3.1.2 OpenFlow Endpoint 16
        - 3.1.3 Virtualization (slicing) 18
    - 3.2 Hardware Specific Layer 20
    - 3.3 Northbound Interfaces 23
    - 3.4 Cross-Hardware Platform Interfaces 25
        - 3.4.1 Abstract Forwarding API 27
        - 3.4.2 Hardware Pipeline API 30

4. Deploying new functionality and programming network elements 33
        - 4.1.1 Programmable Abstraction of Datapath (PAD) 34
        - 4.1.2 PAD API 36
        - 4.1.3 Forwarding functions and network protocols programming 37
        - 4.1.4 Device capabilities 39
    - 4.2 PAD summary 40

5. Summary and conclusions 42

6. References 43

7. Acronyms 44

Figure Summary

Figure 3.1 HAL architecture block diagram 14

Figure 3.2 Cross-Hardware Platform Layer architecture. Multiple OpenFlow controller handled by the OpenFlow endpoint 15

Figure 3.3 OpenFlow entities and interfaces within Cross-Hardware Platform Layer 16

Figure 3.4 Cross-Hardware Platform packet abstraction object attributes 17

Figure 3.5 Workflow between actual packet and packet abstraction in Cross-Hardware OpenFlow pipeline process 17

Figure 3.6 The slicing process workflow for new packets and FlowMod messages 19

Figure 3.7 HAL's northbound interfaces 23

Figure 3.8 Cross-Hardware Platform interfaces for different ALIEN hardware 26

Figure 3.9 Abstract Forwarding API subsets and invocation model 27

Figure 3.10 Packet processing workflow with Abstract Forwarding API 28

Figure 3.11 The most important objects of the Abstract Forwarding API information model 30

Figure 3.12 Hardware Pipeline API subsets and invocation model 31

Figure 3.13 Packet processing workflow with Hardware Pipeline API 31

Figure 4.1 The Programmable Abstraction of Datapath relation with other components 34

Figure 4.2 PAD forwarding element abstraction 35

Table Summary

Table 3.1 Discovery functionality overview for given ALIEN platforms 21

Table 3.2 Orchestration functionality overview for given ALIEN platforms 21

Table 3.3 Translation functionality overview for given ALIEN platforms 22

Table 3.4 Node Virtualization configuration API functions 25

Table 3.5 Cross-Hardware Platform API overview for different hardware driver based on xDPD/ROFL 26

Table 3.6 Abstract Forwarding API methods 28

Table 3.7 Pipeline Hardware API methods 32

Table 4.1 PAD API methods for ANSI-C 37

Listing Summary

Listing 3.1 Node Virtualization Management request message format 24
Listing 3.2 Node Virtualization Management response message format 24
Listing 4.1 Example use of PAD API in Python 39

Executive Summary

This deliverable specifies the Hardware Abstraction Layer (HAL) which has been developed within the ALIEN project. This includes a comprehensive proposal for a logical architecture of HAL. The development of HAL is the main objective of the ALIEN project in order to realize the networking of OpenFlow capable devices with non-OpenFlow devices. The purposed HAL architecture emphasises on decoupling of hardware-specific control and management logic from the network node-abstraction logic. This enables a network with HAL-enhanced devices to be controlled in the same manner (i.e. compatible) as with network elements that natively support OpenFlow. More importantly, the introduction of HAL goes beyond establishing a unified control plane in legacy networks. Indeed the introduction of HAL brings in additional features, e.g., programmability of network nodes, point-to-multipoint data transport, and optical transport data plane, etc.

The document briefly introduces the motivation behind this work, i.e. the need for developing Hardware Abstraction Layer and then outlines the specific requirements for the design of HAL to make it compatible for a number of popular hardware architectures of network devices including x86-based packet processing devices, lightpath devices, point to multipoint devices, and programmable network processors. Based on the requirements a state-of-the-art architecture is proposed for HAL. Choosing a modular design approach, HAL is decomposed into two sub-layers, i.e., 1) Cross-Hardware Platform Layer to address the issues of node abstraction, virtualization, and communication mechanisms and 2) Hardware Specific Layer which is responsible for discovery and configuration of network resources associated with a particular hardware platform. The HAL logical architecture design was tightly coupled with HAL common parts developments in task 2.3. These activities are contribution towards [xDPD] and [ROFL] projects. Although the HAL architecture in ALIEN project is driven from xDPD and ROFL projects, the architecture presented in this document is a

unique design as a result of xDPD/ROFL analysis. In other words, the HAL design and implementation not only covers xDPD/ROFL projects but also generalize them to support more platforms with more functionality. More information about relation between HAL architecture and practical software implementation will be provided in deliverable D2.3.

The document also describes the process of extending network device capabilities and configurations using HAL. For this purpose "Programmable Abstraction of Datapath" (PAD) architecture is proposed which along with a programming language and a protocol definition language can be used to deploy any desired functionality in the network elements. PAD architecture is at early stage of development and currently out-of-scope of [xDPd] and [ROFL] software projects development.

This deliverable is public and may be followed by all people interested in hardware abstraction issues as well as in SDN concept and OpenFlow environment.

1. Introduction

Software Defined Networking (SDN) and in particular OpenFlow protocol as an SDN enabler is one of the areas which has gained a lot of attentions in recent years. These attentions have led the OpenFlow protocol to see many changes from its first public release. Despite all the attentions from the industry and academia towards the OpenFlow protocol, the speed and the amount of changes in the protocol have made the implementation and support difficult for vendors and third party developers. Moreover, although OpenFlow protocol has a single specification for each version, the diversity of the network hardware and software platforms force vendors and third party users to go into the process of creating new OpenFlow libraries for each and every platform for OpenFlow implementation. This makes the OpenFlow deployment more laborious and time consuming.

Apart from the issues mentioned above, the OpenFlow protocol is under process of standardization but nevertheless the protocol itself has some pitfalls in its current state which are listed below:

- It is designed to address the ASIC and campus switches which rules out the processing capabilities beyond layer-2 layer-3 forwarding
- Lack of support for advanced processing capabilities for Network Processing Units (NPU) and general CPU architectures
- The protocol's processing framework only supports "stateless" operation. That means stateful processing beyond memory-less operation is not available. Moreover, virtual ports are out of the scope of OpenFlow.
- The ports are based on Ethernet abstraction

In this deliverable a Hardware Abstraction Layer is introduced which tries to address these issues. The designed HAL overcomes all the above problems by creating a platform that can be instantiated for specific situations. In the following chapters second chapter, the requirements of the HAL are explained. In chapter three the logical architecture of the HAL and its modules are described and following that in chapter four the programmability feature is explained. Finally in chapter five we conclude and summarise the document.

2. HAL Platform Requirements and Support Challenges

The HAL deals with hardware platforms to provide an abstracted version of them for third party user to implement their OpenFlow applications on the hardware platform. Similar to computer operating systems, HAL has to have the ability to interact with various hardware devices with different architecture. Comparing the HAL design for computer and network devices, one will notice that for network devices in some cases the nature of modular hardware platform makes them reconfigurable and also the data plan in some of network devices are distributed, i.e. the hardware is not placed in one fixed box. These are the fundamental differences between computer and network device hardware architecture which makes the design of the HAL for network devices very different from the HAL for computers. Moreover, in our project, the proposed HAL for network devices should communicate with platforms that do not have standard interface for needed abstractions (i.e. closed or proprietary platforms). To give a better understanding of the facing challenges, in the following, the hardware architecture of all network devices that are going to be used in the project are briefly explained.

2.1 x86-based Packet Processing Devices

As most of the following sections deal with “packet processing devices”, the first section deals with packet handling in software, which is the simplest understanding of programmable network devices. Computing platforms from server boards to mini PCs like Raspberry Pi, Arduino, etc., have seen a shift from pure compute capabilities to I/O performance as an important characteristic figure. The typical computer today has at least two or three independent network interfaces (Ethernet, WiFi, LTE), which makes it potentially a forwarding node.

While the implementation of OpenFlow on these boxes (typically equipped with some Linux or embedded Linux OS) was the first development that took place in the SDN area, the split of control and data plane is not as obvious for x86-based devices as for network devices with an externalized forwarding plane. The main question that has to be answered is how to build a fast packet handling pipeline with minimum interventions from the standard OS kernel.

Kernel-based implementations of the pipeline

The Open vSwitch kernel module has replaced the traditional Linux bridge from Kernel 3.3 onwards. This real impact doesn't lie on the fact that there could be an OpenFlow-switch, but in the fact that the OVS kernel module is actually exposing itself as a pipeline and that there is an API providing basic control functions to the daemon. This can even been seen as a similar approach to HAL, although OpenFlow is not addressed at this level.

User-space implementations of the pipeline

Kernel space implementations have the disadvantage of being tied to the kernel. In fact, new kernel versions may change the internal APIs more frequently than other system level (ie: POSIX) interfaces. Any overload or crash in the implementation is affecting the overall system performance to the extent that the system. Therefore user-space implementations have been investigated that allow starting/stopping datapaths at runtime, and that isolate potentially multiple instances running concurrently.

Speeding up User space implementations

Handing over packets from kernel to user space in Linux implies multiple copies of the frame: From the NIC to the kernel, from the kernel to a ring buffer, then after processing the frame gets copied into another ring buffer associated with another port. As this copy operations are prohibitively expensive when looking for a fast datapath implementation, there are several ways explored at the moment that allow a speed-up of packet processing:

- Packet MMAP [PACKETMMAP]: This is a Linux kernel patch designed to improve packet processing through a zero-copy mechanism. It creates a circular buffer of a configurable size in kernel-space which is mapped in the user-space to expedite the packet processing and forwarding. The shared buffer helps avoid the system calls for packet reading and also saves memory which is otherwise required to make a copy of a packet in order to pass it from kernel- to user-space. Moreover, the patch also improves packet transmission efficiency by allowing multiple packets to be sent using one system call. These enhancements enable a Linux box to handle network links carrying a data rate in the order of Gbps.
- Netmap [NETMAP]: Netmap is a framework implemented in Linux/FreeBSD for fast packet I/O processing and is capable of handling up to 10 Gbps links. It improves packet processing through pre-allocation of memory resources, processing of large packet batches with single system call and offering a shared buffer space between kernel- and user-space to avoid memory copies. This way it provides user applications with very fast access to network packets, both on the receiver and the transmitter side, and including those from/to the host stack. The distinct features of Netmap include safety of operation and hardware agnostic approach.
- Intel DPDK [IDPDK]: The Intel Data Plane Development Kit (DPDK) is an open source set of data-plane libraries and controller drivers for Intel Architecture based platforms. It creates an Environment Abstraction Layer (EAL) for a specific hardware/software environment that has been optimized for the Intel Architecture mode, available hardware accelerators, and other hardware and operating system elements. This brings fast packet processing capabilities to the Intel Architecture based platforms which enables faster development of high speed data packet networking applications. The performance of DPDK scales with the underlying hardware from ultra-low-power Atom processor to new generation Xeon processors.

The latter two are similar in that the port is logically detached from the kernel, and attached to a datapath element. This allows making either a single copy or no copy of the packet when passing it through the pipeline.

2.2 Lightpath Devices

Traditionally IP and transport network are managed and operated by separate groups. Typically both networks do not interact with each other. IP network regards the underlying transport links as a static pipeline. Although IP (packet) is the dominant domain in SDN literature and particularly in OpenFlow protocol, the emergence of converged transport networks could have a significant impact on how multi-layer core networks are built by leveraging the SDN centralized network view concept.

Despite the switching paradigm in electrical domain which can be at frame granularity, in optical domain the switching architecture can be constructed to switch fibres or ports that may contain multiple channels or wavelengths. Fibre switching usually is applied in data centre or small to medium range city network with large amount of fibre resources since the switching granularity is in fibre. Although the fibre switching happens by switching between ports, wavelength switching is typically implemented by a reconfigurable add/drop multiplexer (ROADM).

In optical network, traditionally the control plane uses a centralized Network Management System (NMS) that controls the underlying data plan. Although in principle it is similar to SDN concept, in fact it isn’t. In practice the NMS is not able to react to on-demand to changes in data plane to establish an end-to-end path between two ports or nodes: The control plane has to calculate the path and configure ports or nodes before any actual transmission happens. This is because the control plane has no visibility on the data passing on the data plane. This is in contradiction to what happens in IP or packet networks where the control plane can monitor the data on the data plane and react (forward, drop, manipulate) to them based on the application or user policy.

By applying the SDN concept and following the OpenFlow specification, i.e. abstracting the data plane into a flow-table, transport circuits could be represented as a flow-table too. The addendum v0.3 [OFADD] to OpenFlow v1.0 specification is the very first attempt to apply the SDN concept to circuit switching domain. Although the SDN principles are applied in circuit switched domain by implementing OpenFlow protocol, however since the protocol has been created for the IP domain, this does not mean that every OpenFlow functionality in IP (packet) domain is also available in circuit domain. All the modifications, abstractions and functions definitionare defined in this addendum.

The ALIEN project, by applying the OpenFlow specification addendum in its architecture, creates a platform for third party users to abstract the underlying transport data plane into flow-table which ultimately holds the information for cross connections on the device.

2.3 Point to Multi-point Devices

In the context of Access Networks, one of the main challenges is the last mile. This part of the network is considered as the bottleneck in terms of bandwidth and also one of the most expensive segments. The main goal of the Access Network is to connect the customers with the operator's premises in the most cost-effective manner. There are different technologies, such as xDSL, DOCSIS, GPON or GEPON, which rely on different transmission media, such as twisted pair, coaxial cable or optical fiber, respectively. Most of these technologies are based on a shared media to reduce costs by sharing part of the physical link among multiple customers. This implies that the same physical port at the operator's equipment (i.e. the head-end) is shared by multiple customers (i.e. the leaves). As a result, a point to multi-point topology (i.e. a tree) is obtained.

The whole system, including the customer's side equipment, the shared media and the operator's side equipment, can be abstracted as a point to multi-point device. The asymmetry between the head-end and the leaves is one characteristic of these systems, in which the head-end is the "intelligent" part that determines how the access network media is shared.

The last mile solves the problem of connecting the customers' equipment with the operator's network. As a consequence, the technologies developed for access networks focus on moving the traffic up (i.e. from the customer to the operator) and down (i.e. form the operator to the customer). With respect to this issue, the traffic between customers sharing the same physical media could be out of the scope of some of these technologies. In these cases, external equipment should be used for this type of connection between customers.

In the ALIEN project we are dealing with two different access network technologies: DOCSIS and GEPON. Although both technologies can be abstracted as a point to multipoint device, there are some particularities that must be considered.

In the DOCSIS system, the "intelligent" head-end, the CMTS equipment, can be configured basically in two different operation modes, as a router and as a Layer 2 bridge. Typically, the CMTS is configured as a router, however, the operation mode is selected depending on the target deployment. In the context of the ALIEN project, the DOCSIS system must be abstracted to behave as an OpenFlow switch. As a consequence, the CMTS is configured as a Layer 2 bridge, and VLAN/L2VPNs are configured to bridge the traffic from the leaves, i.e. the cable modems (CMs), to the head-end, i.e. the CMTS, and vice versa. One relevant aspect to consider is that the traffic bridged from the CMs to the CMTS is tagged (i.e. VLAN tag is added) at the egress port. Moreover, when acting as a Layer 2 bridge, the CMTS does not implement the forwarding between two CMs. It only works in the up/down directions, which means that an external box in the aggregation network is needed to perform the forwarding between the CMs. Typically, in real deployments, attached to the CM (or even in the same box) a residential/home gateway is also provided to perform some processing at the customer's side.

The GEPON box is a closed source box with limited information available about the main control chip Teknovus TK3721. The standard working mode for the device is that control is through a proprietary interface either through a GUI program running on windows and connected to the management Ethernet port or through a CLI which connects via the serial port. Even if it were possible to install a version of OpenFlow directly on the control chip this would limit the applicability of the approach to only running on those models of GEPON which have a TK3721 chip. A second problem is the speed at which control actions can be implemented. There is no guaranteed speed at which control actions input to the chip occur and some of them are slow. The intended operation mode of the GEPON is to provision links, QoS etc. very rarely and hence it is not expected to respond quickly to control inputs. On occasions, control inputs have response times greater than one second or require a reboot of the machine. In that case, slow responses are inadequate for responding to OpenFlow requests in a reasonable manner and a different mechanism is needed.

When dealing with a closed box from vendors (e.g. a CMTS from Cisco), the actual behaviour of this equipment cannot be changed or reprogrammed, it can only be configured to behave as expected in the target scenario. In this case, this means that the equipment cannot be modified to behave as a full functional OpenFlow switch. The interfaces (which may conform to published standards, as is the case of the DOCSIS, or use proprietary protocols, as is the case of the GEPON) exposed by the vendor are the only entry point to modify the device’s behaviour (based on the set of alternatives approved by the standard). As a consequence, the underlying technology could impose some restrictions to be fully compatible with the OpenFlow specification. For instance, the DOCSIS technology defines the service flows as a unidirectional stream of packets between the CMTS and the CM. All the data packets need to be associated to one service flow, and these service flows must be provisioned in advance. This characteristic of the DOCSIS specification is not compatible with the abstractions made by OpenFlow model, which are based on a switch model that pre-provisioning of packet streams are not needed or even considered. In the case of the GEPON similarly, flows can be provisioned in advance but it is not the intended mode to provision flows quickly “on the fly” in response to events.

2.4 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 that can be implemented on a single hardware unit.

Currently, there are many programmable network platforms available in the market differentiated by programming technologies and processor architectures:

- Programmable silicon gateways (e.g. NetFPGA)
- Traditional NPUs (e.g. EZchip NP and NPA families, Marvel Xelerated, PMC-Sierra WinPath),
- Multicore CPUs with network enhancements (e.g.: Cavium Octeon family, Broadcom XLS/XLR/XLP, LSI Axxia, Freescale QorIQ, Tilera),
- Novel hybrid multicore NPUs (e.g.: Netronome NFP-6xxx, EZchip NPS).

From those available platforms, EZchip NP-3, Cavium Octeon-I and NetFPGA network programmable platforms are used in the ALIEN project. Some common, on the paper, characteristics between all these platforms such as traffic performance, logic implementation and programming flexibility must be treated differently.

To have the optimum traffic performance, the different sets of programmable platforms use the following techniques:

- Integrated ASIC circuits to boost the calculation tasks (protocols headers parsing, pattern/regex matching, checksums, cryptography, security, packet classification and queuing),
- TCAM memory structures for fast and flexible database lookup especially for IP routing,
- Multicore, super-scalar processor architectures for parallel processing of network packets (also multithreading),
- Specialized micro-coded engines (i.e. task optimized cores) which are faster than RISC cores,
- Core pipelining where each core (i.e. task optimized core) performs one specific task.

However, some of these listed techniques (e.g. pipelining, task optimized cores, integrated ASIC circuits) reduce flexibility as they are mostly statically built in silicon chips. Any limitation in programming flexibility can have an influence on full support of OpenFlow for a programmable platform.

The key factor for programming flexibility is the programming language used for a particular platform. Silicon description languages (like Verilog for NetFPGA) and assemblers (for traditional NPUs) are hard for programming and usually higher-level languages are preferred. Currently, most of the multicore CPUs with network enhancements and novel hybrid multicore NPUs can be programmed with C/C++ but nevertheless development for network processors is still a challenging, difficult and time consuming task.

Programmable Network Platforms, besides network processor chip, also contain standard x86 CPU that usually used to deploy control and management software and is the only common hardware block for all platforms. This allows introducing cross-hardware software logics for programmable network devices. However implementation of a certain packet processing functionalities (i.e. MPLS labels, IPv6 addresses, VXLAN tunnelling) are completely different for each network processor platform

Programmable network processors are ideal hardware platforms to introduce and validate new networking concepts. To take advantage of this possibility, in the ALIEN project, dynamic adaptation of network node capabilities is investigated in order to "learn" new protocols (e.g. Content-Centric Network protocol stack) to a data path element with new processing actions which later could be added to the OpenFlow protocol action set.

3. HAL Logical Architecture

Hardware Abstraction Layer (HAL) provides a modular solution for different types of network devices to make an abstraction of the device which is compatible with OpenFlow protocol. To do that, it decouples hardware-specific control and management logic from the network-node abstraction logic (i.e. OpenFlow). Decoupling in the HAL hides the device complexity as well as technology and vendor specific features from the Control Plane logic. The decoupling is done by splitting the HAL (see Figure 3.1) into two layers:

1) Cross-Hardware Platform Layer which is in charge of node abstraction, virtualization and communication mechanisms and

2) Hardware Specific Layer which is in fact a collection of the hardware specific software modules, collectively called driver, responsible for discovering hardware platform resources and configuring network devices.

These two layers are connected to each other with two interfaces (which will be described in section 3.4) :

1) Abstract Forwarding API as an interface to communicate with hardware driver,.

2) Hardware Pipeline API for hardware platforms that use the OpenFlow datapath implementation provided by the Cross-Hardware Platform Layer.

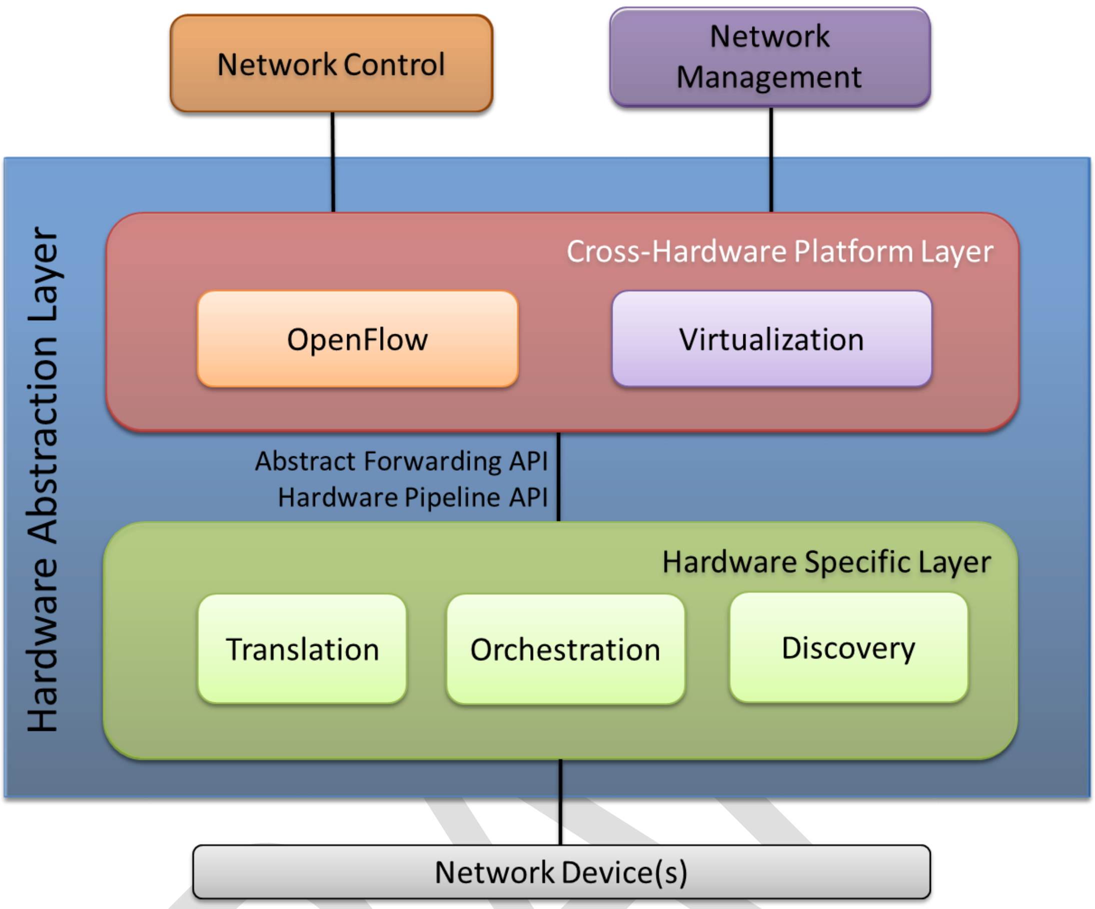

Figure 3.1 HAL architecture block diagram

Although the designed HAL can be applied for compatible OpenFlow devices (i.e. devices with ability to be operated by OpenFlow out of the box), it also targets non-compatible OpenFlow network devices (like those described in [D3.2]) that have additional capabilities such as programmability, point-to-multipoint data transport, optical transport data plane, etc.

The gradual and modular abstraction in the HAL architecture gives the possibility of changing and extending any platform without compromising the whole HAL architecture. It also makes HAL's implementations easier for similar network platforms by module reusability in common components (i.e. OpenFlow pipeline implementation). In the following, each HAL sub-layer structure and their functionalities and interfaces are explained.

3.1 Cross-Hardware Platform Layer

The Cross-Hardware Platform Layer is shared layer between all different platforms and composed of independent modules dealing with device or system management, monitoring and control plane (OpenFlow). On the management side, this layer presents a unified abstraction of the physical platform (fundamentally physical ports, virtual ports, tunnels, etc.) to plugin modules. The plugin modules can steer the configuration of the OpenFlow endpoints, for instance, defining the OpenFlow controller. Examples of management plugins are NetConf/OFConfig agents (see Section

3.1.1) or a file-based configuration reader. Another example of plugin is the Virtualization Agent (VA) described in Section 3.1.3.

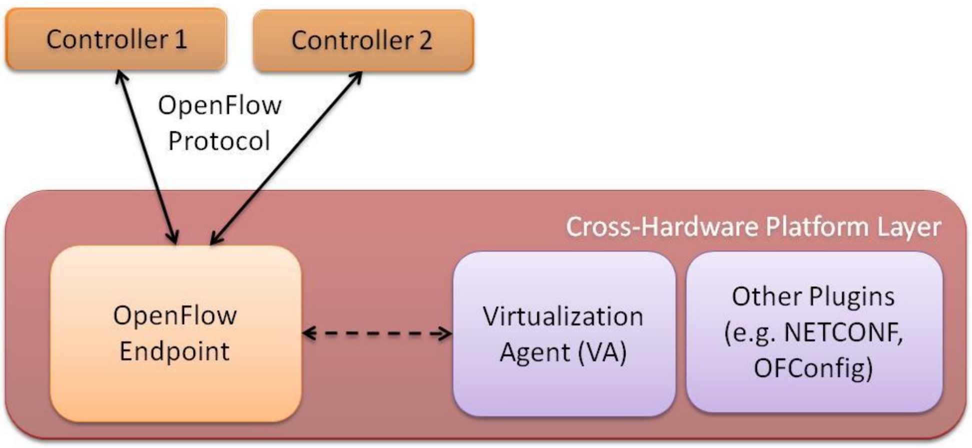
Controller 1
Controller 2
OpenFlow Protocol
Cross-Hardware Platform Layer
OpenFlow Endpoint
Virtualization Agent (VA)
Other Plugins (e.g. NETCONF, OFConfig)

Figure 3.2 Cross-Hardware Platform Layer architecture. Multiple OpenFlow controller handled by the OpenFlow endpoint

In the control plane part, the OpenFlow endpoint is responsible for maintaining connection with the OpenFlow controller. The endpoint encapsulates all the necessary control plane functionalities, as well as a handle to manage the forwarding state down to the platform driver.

The VA is the component in charge of slicing the device to be shared among multiple users. VA interacts with the OpenFlow endpoint to perform the flowspace slicing operations. To make these operations agnostic to OpenFlow version, the VA is queried by the endpoint before creating the OpenFlow messages directed to the controller and after the payload is de-capsulated from the OpenFlow messages coming from the controller.

3.1.1 Network Management

SDN defines the separation of the control plane from the data-plane of a network device [MLA]. Using the OpenFlow protocol, the control plane can communicate with the data plane to perform several functionalities such as adding or removing flow-rules and collecting per-flow, per-table statistics. However, when using the OpenFlow protocol it is falsely assumed that the forwarding devices (i.e., the OpenFlow-enabled switches) are already configured with various parameters such as the IP address(es) of the controller(s). Therefore, it is important to distinguish time-sensitive control-functionalities for which the OpenFlow protocol was designed for (e.g., modifying forwarding tables, matching flows) from non-time-sensitive management and configuration functionalities which are essential for the operation of the OpenFlow-enabled device (e.g., controller IP assignment, changing ports status, etc.) [OF-CONFIG]. Consequently, a standard protocol is required for performing these configuration management functionalities.

In 2002, the Internet Architecture Board (IAB) organized a workshop for guiding the development of the future network management standardization activities. The output of this workshop was a set of requirements to be met by the future management protocol [RFC3535]. In 2003, a working group was formed to produce a protocol meeting these requirements. The protocol produced from this working group was NETCONF [RFC6241]. NETCONF provides several features including, but not limited to, the distinction between configuration and operational states, concurrency support and transactions across multiple network elements, which lacked in management protocols introduced earlier, such as, for example, SNMP [RFC3410]. Each network device that supports the NETCONF protocol should provide a data model that specifies the parameters available for configuration and management. For this purpose, YANG [RFC6020], a highly readable and compact domain specific language, is used for defining NETCONF data models.

In the original work plan for the ALIEN project, management aspects were not taken into consideration. However, as project execution progressed, adding management plane functionality to HAL became an important technical consideration. In the process, it was decided to consider the technical feasibility to add management plane functionality to HAL during the project execution, and it appears that this is indeed desirable. As such, ALIEN will work towards this implementation, a decision which is in line with recent work in SDNRG [SDNLAT]. Although, no decision has been made at this stage about the exact implementation details of the HAL management plane, it appears that NETCONF and OF-CONFIG [OF-CONFIG] are both good candidates for the HAL architecture. Using NETCONF for network management together with YANG for data-modeling in the ALIEN HAL architecture provides a firm base for simpler, more effective and robust configuration management. More progress in this front will be reported in the upcoming deliverable D2.3.

3.1.2 OpenFlow Endpoint

The OpenFlow endpoint component (Figure 3.3) establishes a connection channel to the controller to send/receive encode/decode OpenFlow protocol messages, implements OpenFlow-specific session negotiation and manages state maintenance. It converts the abstraction of any OpenFlow protocol version to common data model with a superset of all OpenFlow versions features. The OpenFlow endpoint is an entity that must be initially configured with a specific OpenFlow version number and can cooperate with OpenFlow controller supporting that version of the OpenFlow. The OpenFlow endpoint communicates with other HAL entities via internal HAL interface (i.e. Abstract Forwarding API). The Cross-Hardware Platform OpenFlow endpoint common data model is part of the Abstract Forwarding API. The data model can be easily extended to handle new header matches, new matching algorithms and new packet processing actions. For more information about Abstract Forwarding API and its data model please see section 3.4.1.

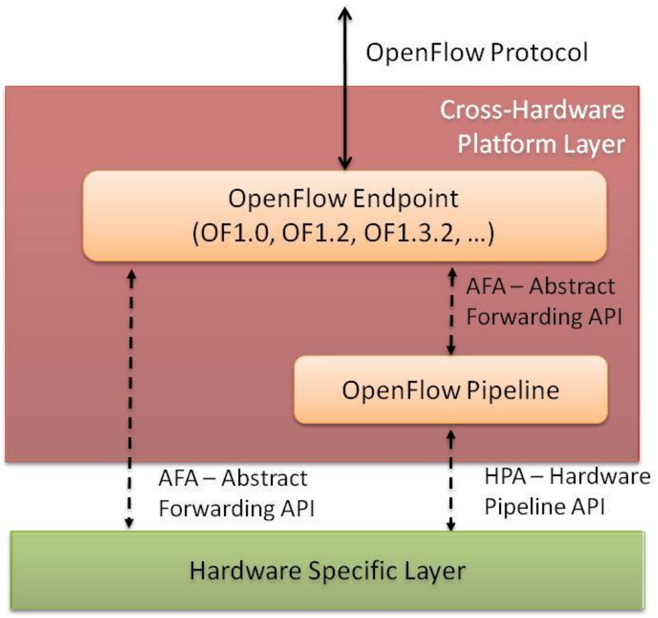
OpenFlow Protocol
Cross-Hardware Platform Layer
OpenFlow Endpoint (OF1.0, OF1.2, OF1.3.2, ...)
AFA – Abstract Forwarding API
OpenFlow Pipeline
AFA – Abstract Forwarding API
HPA – Hardware Pipeline API
Hardware Specific Layer

Figure 3.3 OpenFlow entities and interfaces within Cross-Hardware Platform Layer

The OpenFlow pipeline component implements any number of OpenFlow tables. However, the Cross-Hardware Platform OpenFlow pipeline processes a packet abstraction as a real network packet. The packet abstraction object attributes are presented in Figure 3.4. The packet abstraction is composed of hardware platform reference to a real network packet (i.e. a pointer to memory where packet is stored, automatically generated reference id, etc.), OpenFlow action-set which is passed from one OpenFlow table to next one and modified according to matched flow instructions. Packet abstraction object could also store all successful match entries for diagnostic purposes.

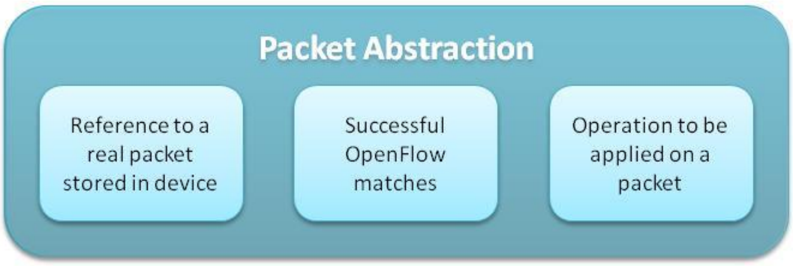
Packet Abstraction
Reference to a real packet stored in device
Successful OpenFlow matches
Operation to be applied on a packet

Figure 3.4 Cross-Hardware Platform packet abstraction object attributes

Figure 3.5 presents how the Cross-Hardware Platform software pipeline processes a network packet. Upon coming a packet into the Hardware Specific Layer, a packet reference is created and is sent to Cross-Hardware Platform Layer which triggers packet abstraction object creation and its processing by the OpenFlow pipeline. Each OpenFlow table can immediately apply changes (modify values of header fields, add/remove tags to the packet located in Hardware Specific Layer (packet reference is required) or modify OpenFlow action-set. Each table can also decide about final destination of the packet and request for sending it to a network port on the device, forward it to the controller (the pipeline is requesting packet-in event generation by Hardware Specific Layer which must send the whole or part of the real packet to the OpenFlow endpoint) or just drop it. The packet-out event, which contains a packet from OpenFlow controller, is also processed via the pipeline, however, the packet bytes must be stored in the Hardware Specific Layer before pipeline processing could start.

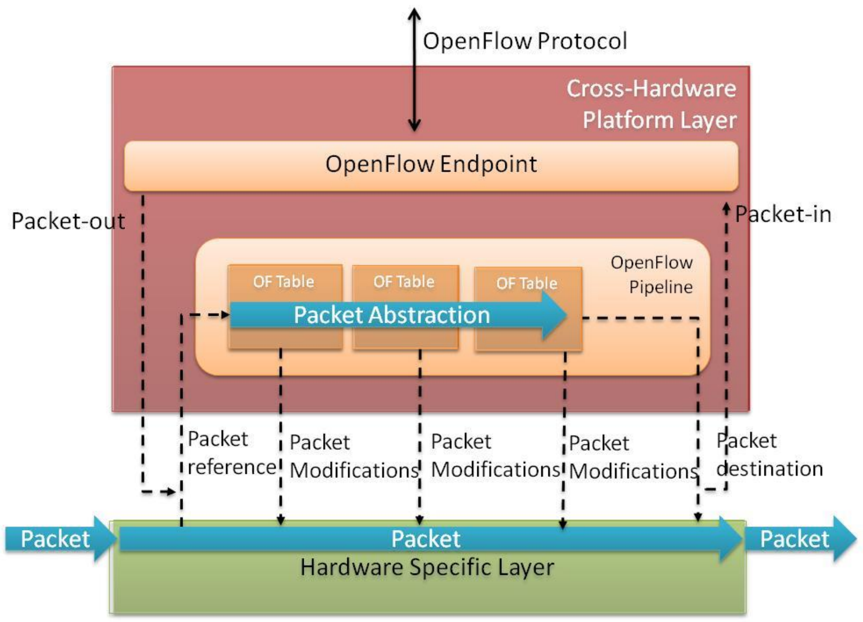

Figure 3.5 Workflow between actual packet and packet abstraction in Cross-Hardware OpenFlow pipeline process

Besides the multiple OpenFlow tables and action-set mentioned above, the Cross-Hardware Platform OpenFlow pipeline supports all other OpenFlow features like priority matching, flow entries expiration, group table, meter table and counter objects.

The Cross-Hardware Platform OpenFlow pipeline must be a pure software and performance efficient implementation of the OpenFlow pipeline which should be deployable on broad spectrum of available CPUs (for software switch solutions) or even modern NPUs (those platforms provide a good traffic throughput for any ANSI-C programming language based packet processing implementations).

3.1.3 Virtualization (slicing)

The Virtualization Agent (VA) is an internal to HAL which aims at providing a distributed slicing mechanism for the ALIEN devices. Like other virtualization approaches ([FlowVisor] and [VeRTIGO]), the VA's main objective is to allow multiple parallel experiments to be executed on the same physical substrate without interfering each other. The VA has been designed with the following goals:

(i) Avoid Single Point of Failures (SPoF) through a distributed slicing architecture.

(ii) Provide an OpenFlow version agnostic slicing mechanism .

(iii) Minimize the latency overhead caused by the slicing operations.

<u>**Distributed slicing**</u>

The VA architecture is designed to avoid SPoFs. In fact, differently from other approaches with a central proxy like FlowVisor and VeRTIGO, the virtualization operations are performed directly on the nodes. A failure of the Virtualization Gateway (see section 3), the only centralized element in the architecture, can only prevent the instantiation of new slices without affecting the ones already in operation. However, a failure of FlowVisor or VeRTIGO would bring down all the running slices.

<u>**Protocol agnostic**</u>

The VA does not inspect the control protocol (here OpenFlow) to perform the slicing process therefore it can, in principle, support any control protocol (even different from OpenFlow).

<u>**Latency overhead**</u>

The resource virtualization operations have a cost in terms of additional latency on actions that cross between the control and the forwarding plane. The overhead depends on how the virtualization mechanism is implemented but other elements can contribute to the total latency too. In particular, contrary to FlowVisor and VeRTIGO, the Virtualization Agent neither inspects the OpenFlow protocol nor needs to establish additional TLS connections.

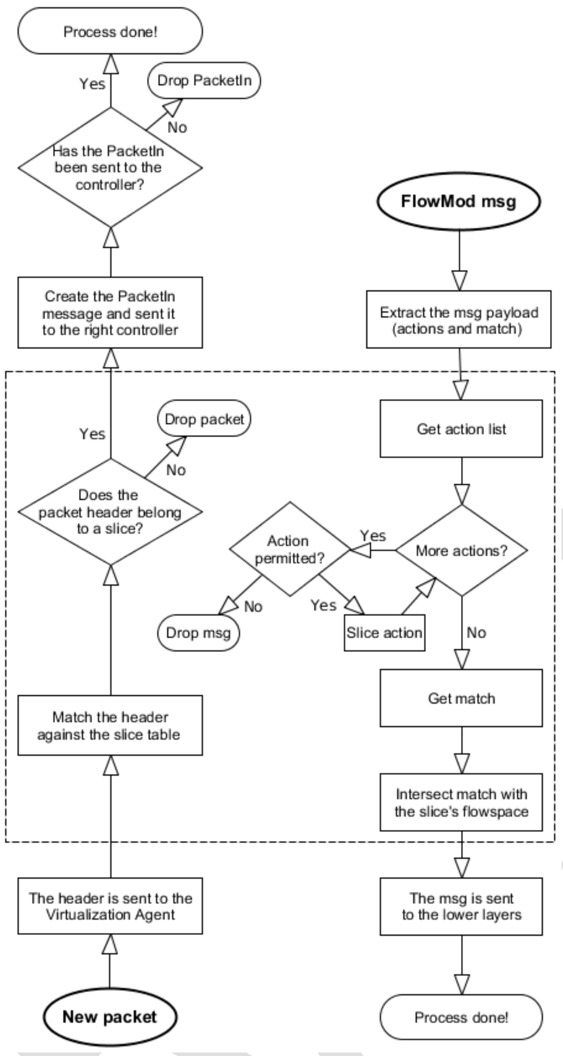
Process done!
Yes
Drop PacketIn
No
Has the PacketIn been sent to the controller?
FlowMod msg
Create the PacketIn message and sent it to the right controller
Extract the msg payload (actions and match)
Yes
Drop packet
No
Get action list
Does the packet header belong to a slice?
Action permitted?
Yes
More actions?
No
Yes
Drop msg
Slice action
No
Get match
Match the header against the slice table
Intersect match with the slice's flowspace
The header is sent to the Virtualization Agent
The msg is sent to the lower layers
New packet
Process done!

Figure 3.6 The slicing process workflow for new packets and FlowMod messages

The slicing mechanism

In OpenFlow protocol, for each incoming packet that does not have an entry in the switch flow table, a `PacketIn` event is generated and sent to the controller through the control channel. The controller, in turn, can answer with a `FlowMod` message to modify the flow tables of the switch.

Referring to the flowchart depicted in Figure 3.6, for each new packet, the fields of its header are matched against the flowspaces assigned to the configured slices. If the VA finds a match, the header is sent to the related OpenFlow endpoint which builds the `packet-in` message by using the protocol version used for the communication with the controller. Vice-versa, if no correspondence is found, the VA tells the lower layers to drop the packet.

On the other side, the VA applies the slicing policies to the OpenFlow messages sent by the controller to the switch. In order to keep the VA internal processes agnostic to protocol version, the VA intercepts the actions and the related flow match after they are de-capsulated from the OpenFlow message and before they are inserted into the switch's flow table. The actions are checked against the controller's flowspace (i.e. the VA checks if the controller is trying to control traffic outside its flowspace) and the match is intersected with the flowspace. The latter operation ensures that the actions are only applied to the flows matching the flowspace assigned to the controller, i.e. the VA prevents interference among different slices.

3.2 Hardware Specific Layer

The idea behind the Hardware Specific Layer is to deal with diversity of the network platforms and their communication protocols to overcome the complexity of implementing OpenFlow protocol on different hardware. In real world, every network equipment or platform has its own protocol or API for communicating, controlling and managing the underlying system. In the proposed HAL, the Hardware Specific Layer is responsible to hide the complexity and heterogeneity of underlying hardware control for message handling and provide a unified and feature rich interface in its northbound for the upper layer i.e. Cross-Hardware Platform Layer. Although the Hardware Specific Layer on its northbound has a unified interface, on the southbound, it is in direct contact with the underlying hardware, which makes it dependent to the hardware in terms of communicating protocol and programming language. This results the layer to have different implementation method for each platform.

Following the modularity principle and also in order to make the HAL flexible enough to support different hardware platform, different modules in Hardware Specific Layer take care of supporting hardware platforms heterogeneity. The layer has been designed in a way that the changes inside its modules do not affect the upper layer (hardware independent) functionality and in most cases there is no need to manipulate the architecture. In the following the modules inside the Hardware Specific Layer and their functionalities are explained.

Discovery

In order to initialize Cross-Hardware Platform Layer, a set of information about network device(s) must be provided from Hardware Specific Layer. The information needed are:

(i) A list of devices working together as a single hardware platform instance and controlled by a single OpenFlow agent instance. For each device, the access information is also required.

(ii) A list of all network ports and their characteristics (e.g. transmission technology, transmission speed, operational status, etc.) from every device.

(iii) The internal hardware platform topology (e.g. how all devices within a hardware platform instance are interconnected) must be recognized. This is required for orchestration functionality to work properly in Hardware Specific Part (HSP) in HAL.

There are various design options to implement discovery functionality. The discovery can be manual (e.g. platform administrator creates static configuration files containing some part of required information and HSP loads that configuration file during initialization) or automatic (e.q. Hardware Specific Layer queries each device for all information and reacts to new notifications coming from the device) or combination of both approaches. Depending on the implementation and also the platform, the discovery process could be active just only during Hardware Specific Layer initialization or executed continuously (e.g. periodical queries in order to discover changes in the hardware). Table 3.1 presents discovery functionality implementation in Hardware Specific Layer for given Alien hardware platforms.

Table 3.1 Discovery functionality overview for given ALIEN platforms

Device	Discovery functionality overview
EZappliance	EZappliance discovery functionality is mostly automatic through Corba interface established to the device and only device access information is stored statically within the configuration file. EZappliance is a hardware platform represented by a single device so topology discovery is not required.
NetFGPA	Basic information comes from default set of settings or configuration file. NetFPGA is a single device; there is no "internal topology of the device".
ATCA	Functionality will be configured via a config file of xDPd, and discovered by the controller via OpenFlow messages (features_reply and table_features_reply for v.1.2)
DOCSIS	Proxy based solution deployed over DOCSIS access network has a known topology model. However, users' CPEs can join to the network dynamically and they are detected and exposed as a new interface at the virtualized model of the network.
GEPON	The discovery can be provided by a configuration file or by polling the GEPON device via the management terminal for new ONU being plugged in.
Layer0 switch	Discovery functionality is done in Layer0 switch using the management interface of the device. The hardware platform provides one interface per device. The details of this interface are given using a configuration file. Topology discovery is not required since each device is managed independently.

<u>**Orchestration**</u>

In some cases, the hardware platform is composed of multiple hardware components acting independently but controlled centrally (e.g. DOCSIS, GEPON). The orchestration procedure goal is to send configuration commands to all hardware components that must be engaged in the request handling in a synchronized, ordered and atomic fashion. The orchestration process must identify if coming request from Cross-Hardware Platform layer was successfully applied to all hardware components. Also, the orchestration process should be able to recover from configuration failures on a single hardware component and restore initial state of all the hardware components. The orchestration process is initialized by a request (e.g. Add-flow method of AFA interface) from Cross-hardware Platform interface.

Table 3.2 presents orchestration functionality implementation in Hardware Specific Layer for given Alien hardware platforms.

Table 3.2 Orchestration functionality overview for given ALIEN platforms

Device	Orchestration functionality overview
EZappliance	Orchestration not required because EZappliance platform is composed of a single device.
NetFGPA	Orchestration not required because NetFPGA card is a single device.
ATCA	For the OCTEON NPU, no orchestration is required. Between different blades of the ATCA chassis, for now, a vendor-specific Broadcom configuration tool (FastPath) is used to guide traffic in and out the OCTEON.
DOCSIS	Orchestration is the base of the solution to expose a set or resources as a unique DPID (Datapath

	Identifier). As an example, in order to provide a virtual model, messages incoming from the controller may require to be splitted into several particular messages over the devices of the network to perform the requested action (stats report, set a configuration, set a new flow...) by the controller. Orchestration also is responsible of hiding network internals which are not exposed to the controller (each user DPID connection implies a message exchange between proxy and clients' DPID)
GEPON	Orchestration not required. The interaction is only with a head end device (the OLT) which takes care of connections to other devices.
Layer0 switch	Orchestration not required. The switching hardware is composed of a single device.

Translation

The Translator module in Hardware Specific Layer is responsible for the translation of data and action models used in Cross-Hardware Platform Interfaces (mostly OpenFlow-based) to device's protocol syntax and semantics and vice versa. Translator acts as a middleware between OpenFlow switch model and underlying physical device. Due to heterogeneity of the network devices, translation specification and implementation is different for each network device. Generally the module is responsible for translating all port numbering, flow entries and packet related actions from OpenFlow switch model into platform specific interface commands and processor instructions or configuration modifications D3.2. In most cases of hardware platforms, the translation functionality will be stateful and requires storing of information about all handled OpenFlow entries and its translation to specific device commands. It allows to modify or delete a device's applied re-configuration which strictly refers to a given flow entry.

Table 3.3 presents translation functionality implementation in Hardware Specific Layer for given Alien hardware platforms.

Table 3.3 Translation functionality overview for given ALIEN platforms

Device	Translation functionality overview
EZappliance	OpenFlow actions are translated into memory structure entries located within NP-3 network processor chip. The semantics used for EZappliance memory structures is quite similar to OpenFlow (e.g. in NP-3, there is defined flow memory structure containing flow entries) but syntax is mostly different (i.e.: property binary coding of packet matching and actions). The translation is stateless because of very close semantics to OpenFlow.
NetFGPA	All OpenFlow actions realized in hardware are stored in hardware part of NetFPGA card. Their definitions have to be translated into form accepted by hardware implementation of modules for particular actions due to fast realization.
ATCA	OpenFlow actions are directly executed by the MIPS cores of the OCTEON network processor using the C-pipeline implementation of xDPd.
DOCSIS	Required to expose the real ports of the network as a unique port identifier in the virtualized model. The main functionality is translated a virtual port identifier into a certain DPID (Datapath Identifier) and the real port associated, and vice versa.
GEPON	The translation is on the control plane and this involves translating, for example, an OpenFlow message directed to a virtual port *N* to another OpenFlow message to send a packet to physical port 2 tagged with an specific VLAN tag *N*.
Layer0 switch	OpenFlow messages are translated into device's specific management commands. There is an

	intermediate layer of abstraction between the resource model of the device and the OpenFlow abstraction. Translation is performed by the lower layer of the datapath. The translated commands are sent to the device using SNMP protocol. Also the OpenFlow protocol itself has been extended to accommodate actions related to optical switches (e.g. optical cross-connect) according to the Circuit Switch Addendum v0.3.

3.3 Northbound Interfaces

The northbound interface is an interface that allows a particular component of a network device to communicate with higher-level components. The HAL provides two northbound interfaces: OpenFlow and JSON-RPC. The former enables the communication between one or more OpenFlow controllers and the ALIEN devices and the latter is used to configure the Virtualization Agent from a Network Management System (NMS).

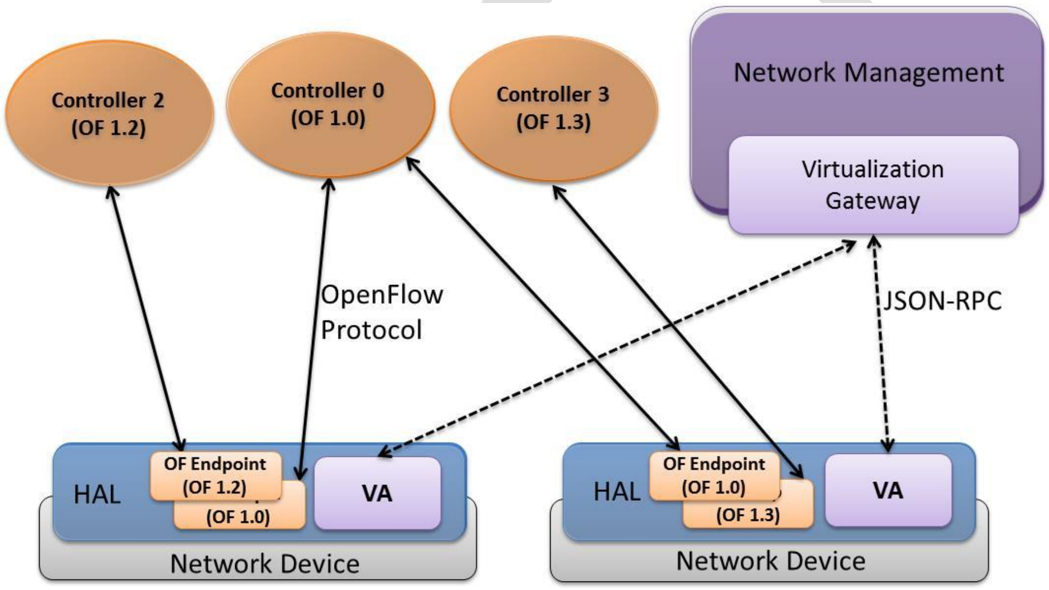

Figure 3.7 HAL's northbound interfaces

OpenFlow protocol

The OpenFlow channel is the interface that connects each ALIEN device (and OpenFlow switches in general) to a controller. Through this interface, the controller configures and manages the device, receives events from the device, and sends packets out of the device. The OpenFlow channel is usually encrypted using TLS, but may be run directly over TCP.

The HAL's OpenFlow interface is provided by OF endpoints instances. Multiple instances are required when different versions of the protocol are used on the same device. On the other hand, multiple controllers using the same version of the protocol are handled by a single OF endpoint.

The current implementation of the HAL supports version v1.0 and v1.2 of the protocol. Support of v1.3.2 is under development. The specification of all versions of the OpenFlow protocol can be found on the Open Networking Foundation web site [OFSPEC].

Node Virtualization Management

The VA slices the overall flowspace among many OF Controllers based on the configuration received from a NMS. The NMS communicates to the VA through the Virtualization Gateway (VG) sub-module.

The management interface between VA and VG is implemented according to the JSON RPC 2.0 Spec found at [JSONRPC] where each request from the VG to the VA will be implemented with the following wire protocol:

```
{
 "id":<string>,
 "method":<command-name>,
 "params":<input>,
 "jsonrpc":"2.0"
}
```

Listing 3.1 Node Virtualization Management request message format

On the opposite direction (from VA to VG), each reply will be implemented with the following wire protocol:

```
{
 "id":<string>,
 "result":<output>,
 "error": {
"code" : <error-code>,
           "msg" : <msg>,
"data" : <data>
           },
 "jsonrpc":"2.0"
}
```

Listing 3.2 Node Virtualization Management response message format

The error field will only be presented when needed. Table 3.4 defines the configuration API composed of set of functions allowing for slicing and flowspace management (see also the FlowVisor CLI tool*fvctl*manual for more details on the single commands).

Table 3.4 Node Virtualization configuration API functions

Command name	Description
add-slice	Creates a new slice
delete-slice	Deletes a slice and removed all of the flowspace corresponding to the slice
list-slices	Displays the current configured slices
list-slice-info	Displays slice's details (e.g. the controller URL)
add-flowspace	Creates a new rule and returns the new rule's ID
remove-flowspace	Removes rule with id=ID
list-flowspace	Lists the VA's configured flow-based slice policy rules
list-datapaths	Displays the devices (e.g., switches) currently connected to the VG.

3.4 Cross-Hardware Platform Interfaces

This section describes common interfaces exposed by Hardware Specific Layer towards Cross-Hardware Platform Layer. Cross-Hardware Platform interfaces must be used both by OpenFlow agent and any hardware driver which would like to cooperate within the HAL framework. Cross-Hardware Platform interfaces are designed with goal for the minimization of efforts required to implement a new hardware driver and achieving OpenFlow control over that hardware platform.

The ALIEN project identified two sets of API that could be used during hardware driver implementation (see Figure 3.8):

- Abstract Forwarding API – more general API (comparing to the next one) which could be used for any hardware platform and it is only available option for close-box platforms
- Pipeline Hardware API – more low-level API that requires availability of the fast access to network packets processed within the platform; it is the preferable option for programmable platforms because hardware driver doesn't have to implement OpenFlow pipeline itself.

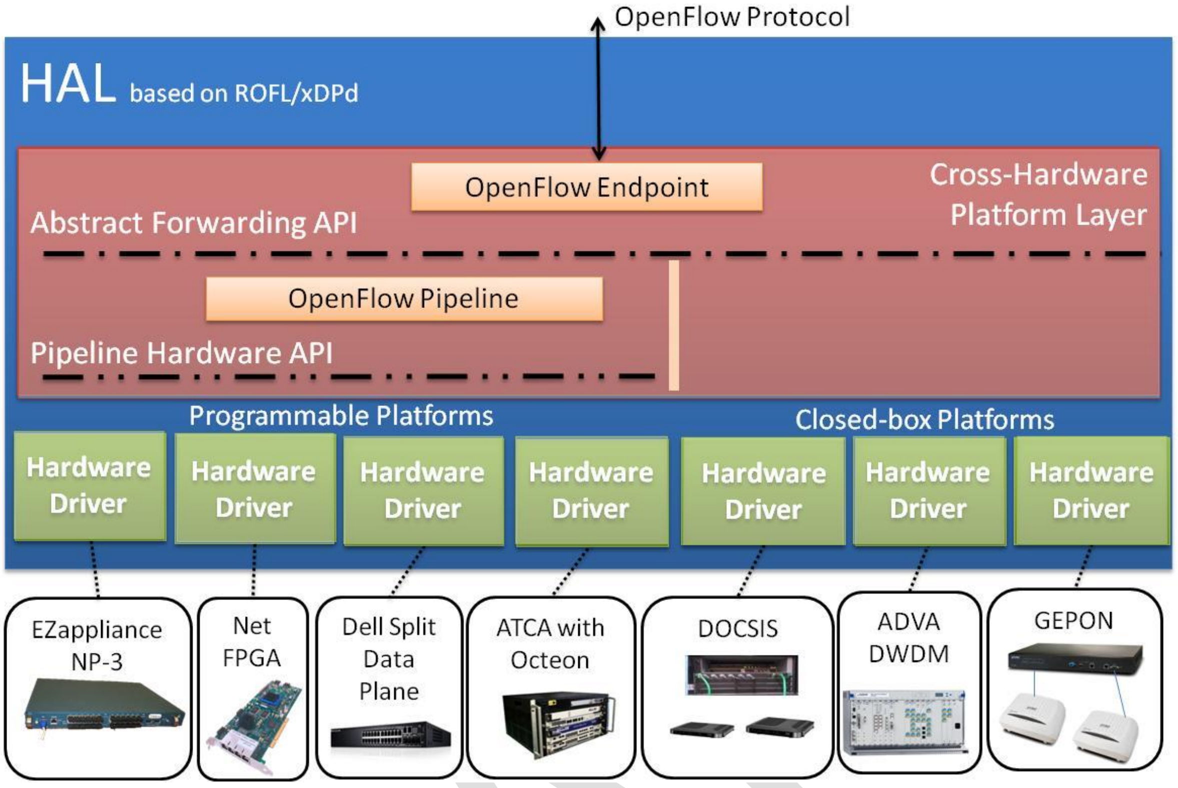

Figure 3.8 Cross-Hardware Platform interfaces for different ALIEN hardware

The ALIEN project has provided the practical development of The Hardware Abstraction Layer in the form of [xDPd] and [ROFL] software projects. The implementation of Abstract Forwarding API and Pipeline Hardware API within xDPd/ROFL is done in the form of C language libraries where functions' declarations are available in a set of C header files provided by ROFL library (in this way the interface definition is common for all platforms). The handlers API implementation is done in C source files that must be provided by each hardware driver. The details of both API implementation (i.e. function names and arguments) are presented in [D3.2].

The summary of Cross-Hardware Platform APIs usage is presented in Table 3.5. More information about HAL API usage can be found in deliverable D3.2, containing the high-level components specification of Hardware Specific Parts for ALIEN hardware platforms.

Table 3.5 Cross-Hardware Platform API overview for different hardware driver based on xDPD/ROFL

Platform	Main hardware driver API	Additional information
DOCSIS	Abstract Forwarding API	Hardware Pipeline API not used because no access to packets.
GEPON	Abstract Forwarding API	Hardware Pipeline API not used because no access to packets.
Layer-0 switch	Abstract Forwarding API	Hardware Pipeline API not used because there is no packet.
EZappliance	Abstract Forwarding API	The Hardware Pipeline API cannot be used natively because NP-3 processor has very strict time constrains regarding packet processing time (in order words: NP-3 cannot store packets which is required by CHPL pipeline). However, Hardware Pipeline API is used as part of

		"slow-path" software switch which is only supportive component (see D3.2 for more details
NetFPGA	Abstract Forwarding API	Full hardware realization of the OF pipeline offers much higher performance thus CHPL pipeline with Hardware Pipeline API is used only as part of "slow-path" software switch which is only supportive component (see D3.2 for more details).
ATCA + Octeon	Hardware Pipeline API	Abstract Forwarding API is used internally within Cross-Hardware Platform Part.
BroadcomTriumph2[1]	Hardware Pipeline API	Abstract Forwarding API is used internally within Cross-Hardware Platform Part.
GNU/Linux x86 software switch[1]	Hardware Pipeline API	Abstract Forwarding API is used internally within Cross-Hardware Platform Part.
GNU/Linux + Intel DPDK software switch[1]	Hardware Pipeline API	Abstract Forwarding API is used internally within Cross-Hardware Platform Part.

3.4.1 Abstract Forwarding API

The Abstract Forwarding API (AFA) provides all the interfaces for management, configuration and events notification of the Hardware Specific Part instance and associated hardware platform. The management and configuration parts of the AFA interface must be implemented by a hardware driver and called by Cross-Hardware Platform Part (see Figure 3.9) but Notification part is provided by Cross-Hardware Platform part and invoked by a hardware driver.

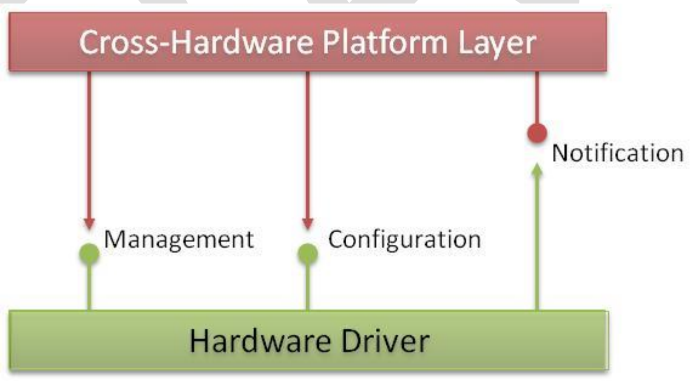

Figure 3.9 Abstract Forwarding API subsets and invocation model

The management part of AFA is in charge of hardware driver initialization, network interface discovery, logical switch creation/destruction of, attaching/detaching of network interfaces to/from logical switches and administratively enabling/disabling network interfaces. Abstract Forwarding API will enable a hardware platform to be logically partitioned into several OpenFlow-controlled data path elements. The logical switches are parallel and independent

[1]Implementation originally made outside of the ALIEN project, but refined in ALIEN.

entities, each controlled by one instance of OpenFlow endpoint. Any switch port (network interface) can be attached to only one logical switch.

The hardware driver, by using AFA interface, performs the entire network packet processing in OpenFlow pipeline within the network device (see 3.10). The AFA interface participates in the configuration of flow entries tables located within the hardware platform and in the configuration of the pipeline and network interfaces. Additionally, AFA allows for OpenFlow packet-in and packet-out events which allow for passing network packet between datapath implementation located in the device and OpenFlow controller. However, packet-in and packet-out operations are mostly not possible in close-box hardware platforms (i.e. DOCSIS, GEPON, Layer0 switch, etc.).

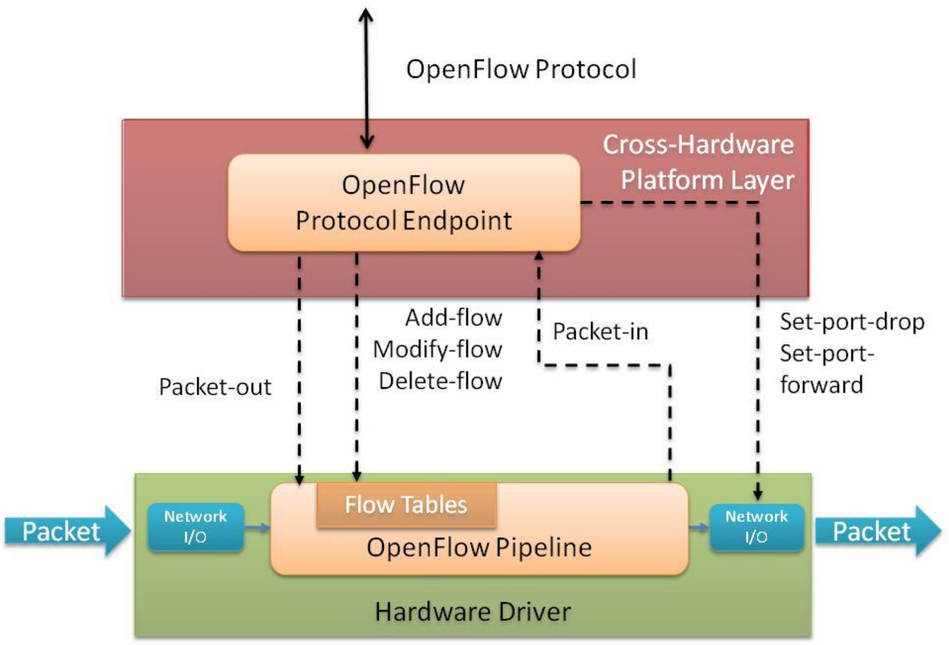

Figure 3.10 Packet processing workflow with Abstract Forwarding API

The detailed list of methods available within AFA interface is presented inTable 3.6.

Table 3.6 Abstract Forwarding API methods

AFA subset	AFA method	Description
Driver Management	Init-driver	Initializes hardware platform driver which covers all initial operations required for a particular hardware platform (e.g.: checking hardware accessibility, discovering hardware capabilities and resources, initialize device configuration as well). Only initialized driver can be used.
	Destroy-driver	Destroy driver state. Allows platform to be properly cleaned.
	Create-switch	Instruct driver to create an OpenFlow logical switch.
	Get-switch	Retrieve the detailed OpenFlow logical switch information.
	Destroy-switch	Instructs the driver to destroy the OpenFlow logical switch.

	Get-ports	Retrieve/discover the list of all network ports of the hardware platform.
	Get-port	Get detailed information about a network port.
	Enable-port	Brings up a port.
	Disable-port	Brings down a port.
	Attach-port-to-switch	Attempts to attach a port to OpenFlow logical switch.
	Detach-port-from-switch	Detaches a port from OpenFlow logical switch.
Datapath configuration	Set-port-drop	Instructs driver to drop or not drop all incoming packets on the port.
	Set-port-forward	Instructs driver to send or not send (output) packets on this port to the network link.
	Set-port-packet-in	Instructs driver to generate or not generate packet-in events for the port.
	Set-port-advertise	Instructs driver to modify port advertised flags representing features being advertised by the port.
	Set-pipeline-config	Instructs driver to set pipeline configuration.
	Set-table-config	Instructs driver to set table configuration.
	Packet-out	Instructs driver to send a packet from controller out through the datapath.
	Add-flow	Instructs driver to add new flow entry.
	Modify-flow	Instructs driver to modify the existing flow entry.
	Delete-flow	Instructs driver to delete the existing flow entry.
	Get-flow-stats	Fetch the flow statistics on a given set of matches.
	Add-group	Instructs driver to add a new group.
	Modify-group	Instructs driver to modify group.
	Delete-group	Instructs driver to delete group.
	Get-group-stats	Instructs driver to fetch the group statistics.
Nofitication	Add-port	Notifies that port was added within the platform.
	Modify-port	Notifies that port was modified (status changed) added within the platform.
	Delete-port	Notifies that port was removed within the platform.
	Packet-in	Handle packet from datapath to be send to controller.
	Flow-removed	Process a flow removed event coming from the datapath.

The configuration part of AFA handles OpenFlow messages sent from OpenFlow controller to the datapath element. This interface must allow for setting, updating and removing flow entries within Hardware Specific Part, configuring pipeline, tables and switch ports properties and requesting statistics for flows, tables, ports or queue structures. Additionally, this interface contains packet-out functionality which allows for driver to send a packet from controller out through the datapath.

The notification part of AFA generates events related to adding/removing of network interfaces within the hardware platform, switch port attributes modifications or state, flow entry expiration and incoming of a packet for the controller.

The information model used for Abstract Forwarding API is shown in Figure 3.11. The configuration part of this model is strictly based on OpenFlow specification.

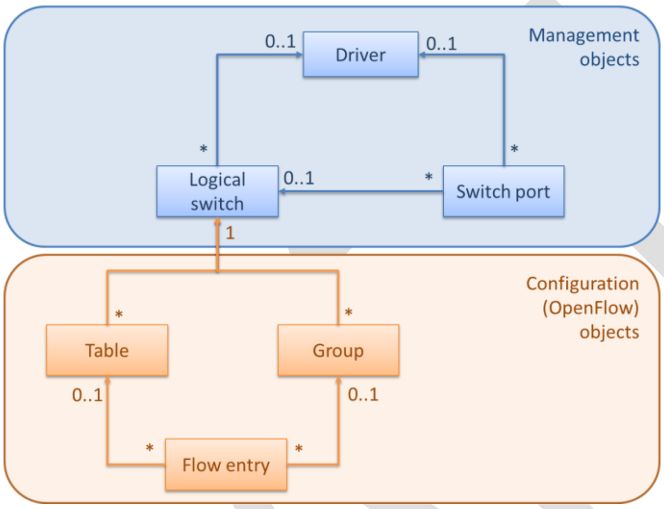

Figure 3.11 The most important objects of the Abstract Forwarding API information model

3.4.2 Hardware Pipeline API

The main goal of Hardware Pipeline API (HPA) is to minimize development efforts required to implement HAL hardware driver on programmable network platforms which allows to deploy and run general C/C++ code (e.g. Cavium Octeon, Broadcom Triumph2, Intel + DPKK, EZchip NPS processors). HPA is a low-level interface giving access to network packets operations, memory management, mutex and counter operations which are differently realized on different programmable platforms (see HPA invocation model in Figure 3.12). The main benefit of HPA interface usage is that hardware driver doesn't have to implement OpenFlow pipeline itself and reuse Cross-Hardware Platform OpenFlow pipeline implementation.

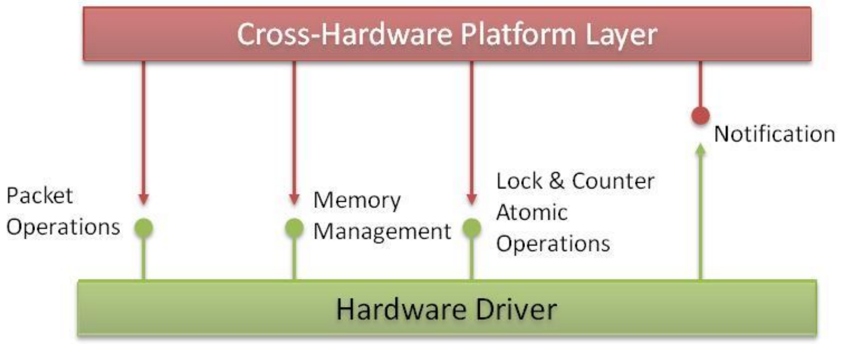
Cross-Hardware Platform Layer
Notification
Packet Operations
Memory Management
Lock & Counter Atomic Operations
Hardware Driver

Figure 3.12 Hardware Pipeline API subsets and invocation model

In general, hardware platforms hardware driver can, at discretion, use HPA when:

- processing packets via software pipeline (e.g. in software switches, certain network processors, or pack out events on ASICs or FPGAs)
- Maintaining the OpenFlow state (e.g.: installed flows, flow expiration) using Cross-Hardware Platform pipeline deployed inside hardware driver implemented using AFA interface (regardless of its nature: hardware, software or hybrid).

The Cross-Hardware Platform pipeline provides an implementation of an OpenFlow forwarding engine. It also includes a runtime sub-API, which is able to process abstract representations of real data packets across an OpenFlow pipeline. The pipeline platform interface exists to synchronize the abstract representation of packet in the pipeline with the actual packet in a network device (mangling the packet, like reading packets fields, modifying packet and apply forwarding decisions). The overview of packet processing with HPA interface is presented in Figure 3.13.

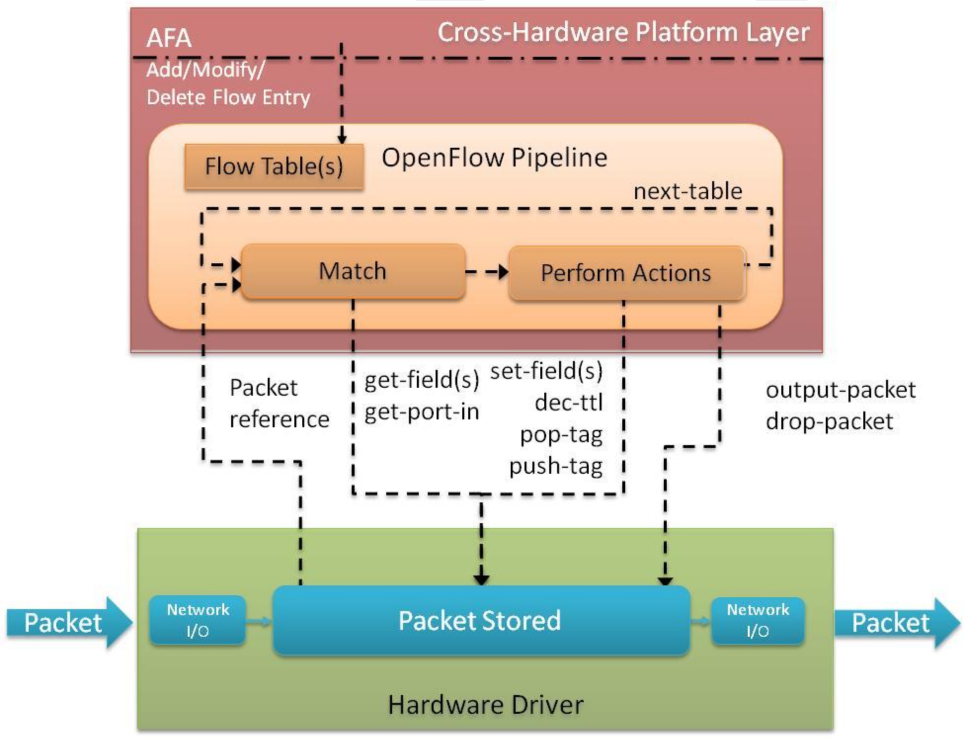

Figure 3.13 Packet processing workflow with Hardware Pipeline API

Additionally, Cross-Hardware Platform pipeline deployment on particular hardware platforms requires specific memory management and synchronization mechanisms. For this reason, the pipeline uses an additional set of methods for memory, mutex and counter operations platform interfaces to abstract these platform specific functions from the hardware-agnostic OpenFlow pipeline implementation. The detailed list of methods available in HPA interfaces is presented inTable 3.7.

Table 3.7 Pipeline Hardware API methods

HPA subset	HPA method	Description
Packet operations	Get-packet-size	Getting the size of the packet processed within the hardware platform.
	Get-port-in	Getting an identifier of the port where packet was received.
	Get-packet-field	Getting specified header field value of OSI layers 2-4 protocols (802.3 Ethernet, ICMPv4, IPv4, MPLS and TCP/UDP) from a packet processed within the hardware platform.
	Set-packet-field	Setting (overwriting) specified header field value of OSI layers 2-4 protocols (802.3 Ethernet, ICMPv4, IPv4, MPLS and TCP/UDP) to a packet processed within the hardware platform.
	Copy-time-to-live	Copy the TTL field between outermost and next-to-outermost header in a packet.
	Decrement-time-to-live	Decrement the TTL field in a packet.
	Pop-tag	Pop the outer-most shim header/tag (VLAN, MPLS and PPPoE) from a packet.
	Push-tag	Push a newshim header/tag (VLAN, MPLS and PPPoE) onto a packet.
	Drop-packet	Drop a packet.
	Output-packet	Output a packet to the port.
Memory management	Allocate-memory	Allocates a block of dynamic memory.
	Free-memory	Frees a block of dynamic memory.
	Copy-memory	Copies a content of memory block to another memory block.
	Move-memory	Moves a content of memory block to another memory block.
	Set-memory	Sets bytes of the block of memory to the specified value.
Mutex & Counter atomic operations	Init-mutex	Allocate the memory within the hardware platform for the mutex and perform the mutex initialization.
	Destroy-mutex	First destroy the mutex and then release the memory allocated.
	Lock-mutex	Lock mutex.
	Unlock-mutex	Unclock mutex.
	Increase-counter	Performs an atomic increment of the counter
	Decrease-counter	Performs an atomic decrement of the counter
Nofitication	Process-packet-in-pipeline	Provide packet reference and processes a packet abstraction through the Openflow pipeline.

Packet operation functions allow the pipeline to get information about the packet and manipulate the processed packet by the network device. Thanks to this part of API, the pipeline doesn't have to pass the entire actual packet between device and pipeline. Instead, the pipeline creates abstracted packet representation to be processed through pipeline tables. Packet operation functions should be implemented for each hardware driver.

Memory allocation, mutex and counter functions provide a common interface for memory management, synchronization mechanisms and atomic counter operations for any programmable network processors. These functions should be implemented for each hardware platform.

The notification interface is implemented by Cross-Hardware Platform pipeline and can be used by hardware driver to activate packet processing within OpenFlow pipeline. In order to active packet processing in the pipeline, a reference to packet stored in the device must be provided by hardware driver.

4. Deploying new functionality and programming network elements

OpenFlow is foreseen as a successful hardware abstraction, but during works in the ALIEN project, we have identified several limitations of current OpenFlow specifications [WHITEPAPER]. We believe capabilities of modern networking hardware are heavily restricted by the OpenFlow abstract device model and there is still a need for more elastic and powerful solution. In this chapter about datapath programmability concept, we would like to address the following limitations of OpenFlow:

- Lack of autonomous capabilities (e.g. MAC learning required for generating ARP responses or PAT port allocation for every new o) with affects the performance of current OpenFlow solutions
- Lack of possibility for generic packet modifications (e.g. adding/removing protocol headers which could be used, i.e., for generating ARP and ICMP responses)
- Tight coupling to current network protocols which make it hard to introduce Future Internet revolutionary solutions (e.g. for Named Data Networking)
- More and more network protocols are foreseen to be covered by OpenFlow control which results in a field explosion in OF specification (due to the fact that big sets of protocol fields are very hard to be packed in the limited sizes of available TCAM memories).

Taking into account all the above mentioned limitations and drawbacks of available technologies, we propose a new approach, which aims to:

i. Expose full capabilities of network processors that are currently the most powerful packet processing hardware.

ii. Create a unified interface for managing and controlling of various types of networking hardware with very diverse capabilities.

We would like to introduce Programmable Abstraction of Datapath (PAD), which is an enhancement of hardware abstraction for data plane elements. The PAD allows for programming of datapath behaviour using generic byte operations, defining protocol headers and providing function definitions. This solution includes also a system for reporting device capabilities in order to provide unified support for diversity of network devices. The PAD is not an extension or generalization of any existing solution, especially OpenFlow. The proposal elaborated in this chapter derives functionalities exposed by the abstraction mechanisms of the data plane hardware, not focusing on requirements exposed by the control plane, which continuously evolves, trying to meet overall objectives of the SDN paradigms.

The ideal packet forwarding hardware characteristics have been introduced and explained in [RPF]. Our solution, as a hardware interface that completely hides details of a real hardware beneath, tries to follow these guidelines. The P4 language [P4] and protocol oblivious forwarding [POF] both try to provide interface for datapath programmability but are focused on specific networking hardware (advanced ASICs and network processors respectively). OF-DPA [OF-PDA] has the narrowest scope implementing the abstraction mechanisms specifically designed for Broadcom Ethernet switches. Figure 4.1 presents possible the relations between PAD concept and solutions described in mentioned papers.

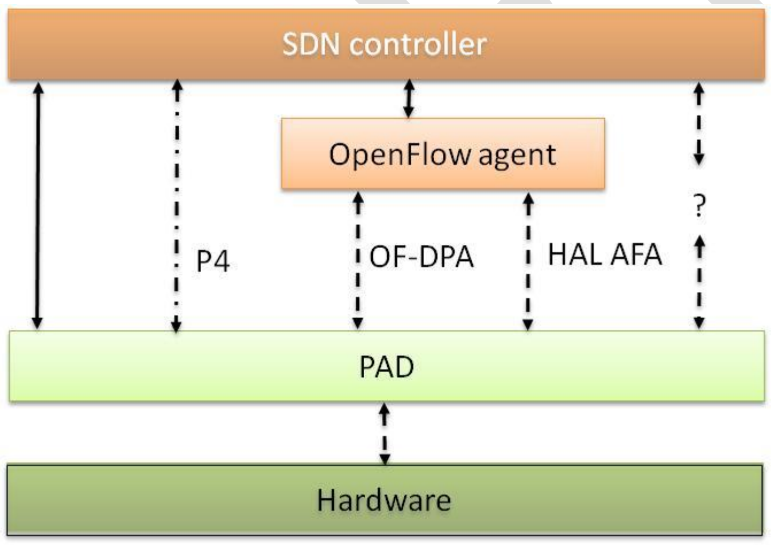

Figure 4.1 The Programmable Abstraction of Datapath relation with other components

The ALIEN Hardware Abstraction Layer (HAL) AFA interface aims to support diverse hardware types but it use OpenFlow based data model only. The AFA interface can utilize PAD solution in order to program OpenFlow model and operations into alien hardware platforms. However, we must emphasise that our PAD solution, presented in this chapter, is currently in very early stage of design and does not contain many functionalities which can be found in quite complex OpenFlow specifications like v1.3 or v1.4.

4.1.1 Programmable Abstraction of Datapath (PAD)

The forwarding abstraction is a model of network device's processing mechanism. Programmable Abstraction of Datapath (PAD) comes from the fact that all network devices perform similar steps during packet processing: reading packet headers (parsing), making forwarding decisions based on a current configuration (searching), performing necessary actions (modifying and forwarding). This observation has been used as an inspiration for the PAD network device abstraction presented on Figure 4.2.

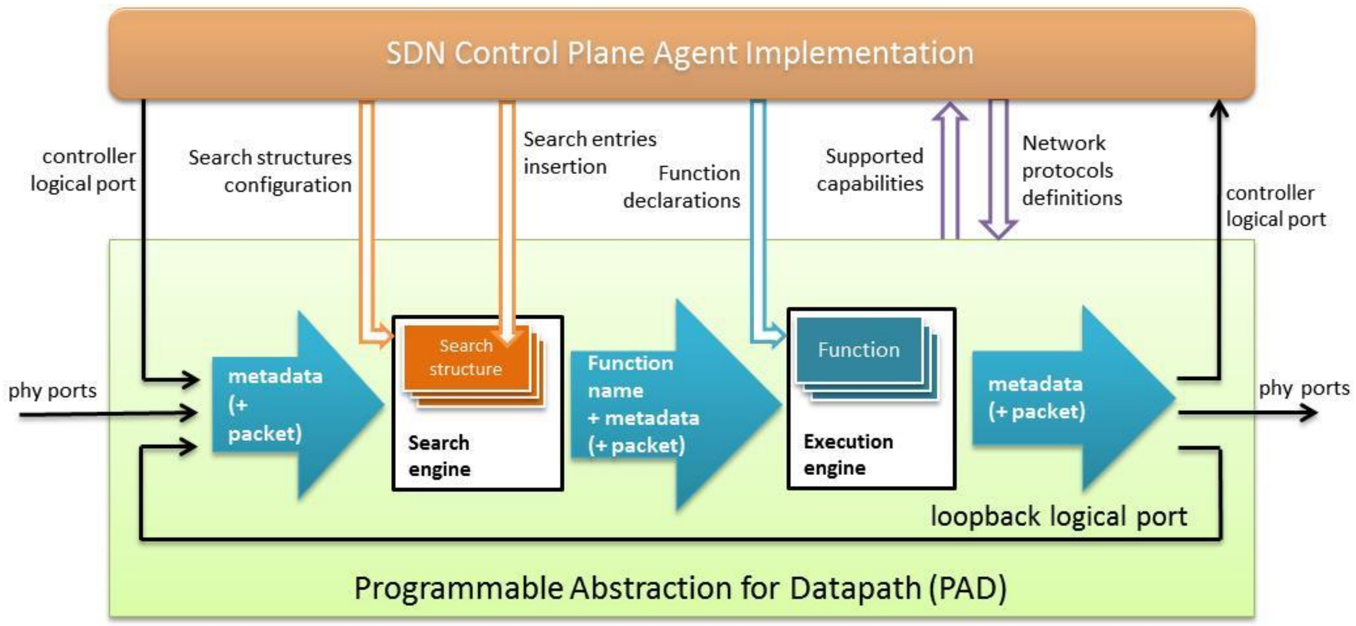

Figure 4.2 PAD forwarding element abstraction

The PAD architecture is composed of several functional components, including ports, search engines, search structures, an execution engine and forwarding functions. Within the PAD, a packet may be processed several times through all these blocks. A packet received from the ingress port is bonded with metadata and passed to the search engine. After the successful search, a packet metadata and search result is passed to the execution engine. A search result contains a function name that will be executed on a packet. At the final stage the packet is passed to the egress port. In most cases, packet processing may run several passes through the PAD using a loopback logical port.

<u>Ports entities</u>

The port in the PAD can be either physical or logical. Physical ports represent physical interfaces of a specific network device. Logical ports are parts of a device abstraction model and are not co-related with any entity on a physical network device (however, particular PAD implementations can use some physical entities to implement this functionality). The loopback logical port is a unidirectional port that interconnects egress and ingress sides of the PAD and allows for multi pass packet processing. A controller's logical port is a bidirectional port that allows for transmission of the processed packet to and from the controller through the northbound interface. Each port (physical or logical) can have a number of sub-ports (e.g. representing different channels on a single optical interface or different SSIDs on a wireless interface).

<u>Search engine</u>

The search engine is a logical module that performs search operations on search structures. The search key is created from parsed fields from a network packet and its metadata (i.e. an ingress port and a sub-port number). The search result is a name of the function defined in the execution engine and packet processing will start with a call of this function.

Each PAD implementation has to support at least one search structure. The number of supported search structures in the PAD model is not restricted. The search structure number 0 is always used for the first pass of the packet processing. Structure numbers for next passes are inserted in a packet metadata and sent together with a packet through the loopback port. In each pass only one search structure can be used. By default, each structure should be a ternary search structure (i.e. support masking of specific bits in a key). Some PAD implementations can additionally support a definition of exact match search structures (i.e. structures without masking possibility). Search structures are defined by their ID number and a key structure. The key can be composed of generic "fields" (e.g. an ingress port number, first 6 bytes and 4 bytes starting from byte 12) or previously defined protocol fields (e.g. an ingress port number, a destination mac address, ethertype and vlan tag, if exists). Both possibilities can be combined in a single structure definition.

Execution engine

The execution engine executes (interprets) functions, which are sets of hardware independent instructions, and translate instructions into hardware-specific actions. Functions declared in execution engine can be called from the body of other functions. The first executed function for a given frame is the one passed from the search structure. A processed frame can be modified here (some devices, such as optical, could not support this feature). In most cases, the processing should finally result in a transmission of a processed data packet to the egress port.

4.1.2 PAD API

A northbound interface of the PAD allows controlling the network device behaviour. The PAD's northbound interface functionalities can be classified into three functional groups:

Datapath Capabilities - this group of operations allows for information retrieving about capabilities which are supported by a specific hardware. Each PAD instance can implement a certain set of the datapath functionalities. This part of the northbound interface will be used for getting information about search structures limitations and available instruction set before search structures are configured and functions declarations are installed. More information about datapath capabilities exposed by the PAD could be found in section 4.1.4.

Datapath Management - this part of the interface allows for managing search structures, functions and defining network protocols. New search structures can be defined with a unique ID and a key description. The management part of this API defines rules of network device operations. The full configuration of a device will require several invocations of appropriate commands. However, the device should not be able to operate with incoherent and incomplete configuration. For this reason, all operations from this part of the API have to be committed in a single, atomic operation to take effect. The PAD implementation accumulates all changes, prepares the new configuration and loads it into the hardware after the commit command is executed with minimal possible interruption on the actual traffic processing. Some management operations, e.g. adding a new function can be theoretically performed without affecting the network device operation. The management part in the API could be extended with additional operations allowing such modifications without committing and stopping device. This additional set of operations is optional and supported only by certain PAD implementations.

Datapath Control - This part of the PAD interface allow for adding and removing of entries in defined search structures. In most cases, first two parts of the API are used before the processing starts. i.e. the PAD user retrieves device capabilities, configures search structures, uploads functions and then starts the processing. The control part of the interface, in opposite, is used during the entire time of the device operation, up to many thousands times per second depending on the application.

The PAD API is designed to transparently handle network protocol and function definitions which decouple the PAD API with protocol and function specifications methods. The PAD API methods (see Table 4.1) use '*char**' parameters containing a string representation of protocol definitions.

Table 4.1 PAD API methods for ANSI-C

API group	RAD API function	Function description
Datapath capabilities	`char* get_all_capablities()`	Returns a string containing list of supported capabilities in the datapath
	`char* get_capability(char* capability_name)`	Returns a string containing a capability value or empty string when no such capability.
Datapath management	`bool replace_protocols(char* protocols_spec)`	Returns true if network protocols specification where successfully installed in the datapath
	`bool add_protocol(char* protocol_spec)`	Returns true if a network protocols where successfully updated with provided protocol specification
	`bool remove_protocol(char* protocol_name)`	Returns true if a given protocol knowledge was successfully removed from the datapath
	`bool add_structure(uint8_t id, char* key, uint32_t size)`	Returns true if a search structure with a given key schema and total number of entries were allocated within the search engine.
	`bool remove_structure(uint id)`	Returns true if a given search structure were deleted in the search engine.
	`bool commit_configuration()`	Return true if the PAD configuration is consistent and the datapath was initialized.
Datapath control	`bool add_entry(uint8_t structure_id, uint64_t key, uint64_t mask, char* result)`	Return true if key, mask and result values were added successfully as a search entry to a given search structure. If key already exist then entry result value is replaced. A result contains both forwarding function name as all parameters values required by the function.
	`bool remove_entry(uint8_t structure_id, uint64_t key, uint64_t mask)`	Return true if a search entry containing given key and mask were removed from a given search structure.

4.1.3 Forwarding functions and network protocols programming

The PAD API requires the use of some kind of forwarding function declarations, protocol definitions programming. These two applications create very different requirements and therefore we propose the use of two different languages.

The programming language is used to define operations performed by the execution engine. Sets of operations are loaded into the program memory as named functions. For each packet, one or more functions are executed.

The primary goal of this language is to allow developers to modify packet in any way including removing existing packet and creating a new one. For this reason the language has to provide wide but specific capabilities that can be easily extended using supported capabilities mechanism.

Key features of this language are:

- Conditional statements and loops
- Fixed-point variables
- Arithmetical and logical operations
- Bitwise operations
- Remove, insert and modify any byte in the packet
- Checksum computation
- Send packet to output port
- Add/delete an entry to/from search structure

In case of implementations that support data plane network protocol definition capability, all defined protocol fields names are also available in the programming language and can be used within search keys and inside functions bodies. All fields that already exist in the packet header can be accessed as regular variables. New empty structure of the protocol header can be inserted into any place in the packet.

The network protocol definition language allows programmers to introduce new data plane network protocols to a network device. Without this feature, the entire packet is seen only as an array of bytes that can be accessed only by its index. The definition of the protocol header consists of field names, order and sizes as well as information about protocol encapsulation. Protocol encapsulation explains how a given protocol header is linked to other headers, e.g. value ‘0x86DD’ of “Type” field in Ethernet header means that the next header will be IPv6.

This feature allows using specific header fields in search structures or functions definitions regardless of their actual position in the packet. For example, the IP destination address can be used in routing table search structure definition regardless of possibility of VLAN header occurrence that will change the location of the IP header in the packet. In implementations that do not support protocol definition, the programmer needs to consider all possible locations of given field in packet and handle them independently. The protocol definition language can be defined as a new solution or can be based on existing solutions like NetPDL [NetPDL] or P4 [P4] languages.

Possible languages for both protocol and function definitions are still intensively considered. However we present an example use of them and the PAD API in Listing 4.1 Example use of PAD API in Python

. This code snippet presents example implementation of a simple label switching router using generic byte operations. Switching is based on a value of 4 byte long field inserted just after the Ethernet header and announced by the Ethertype value of ‘0x88b5’. The function call in the line 4 defines a search structure with the key of the length of 6 bytes composed from ethertype value (2 bytes) and 4 following bytes. The string variable defined in the line 8 contains a definition of a simple function that sends a packet to the output port given as a parameter. A value of port parameter is provided by search result. Line 12 adds the previously defined function to the execution engine. The function defined in the line 14 removes the ether type and the tag value before invoking previously defined function ‘send_to’. The function defined in the line 21 adds a new tag header to the processed packet and sends them to the output port. The whole new configuration is installed on the datapath with commit command in the line 29. Lines 31, 33 and 35 add entries to the defined search structure that respectively switches packets with the tag value ‘0x17’ to the port 7, switches packets with the tag value ‘0x18’ to the port 8 and removes tags from the packet with the tag value ‘0x11’ before sending it to the port 1.

```
 1: from pad import add_structure, add_function,
 2:                 add_entry, commit_configuration
 3:
 4: add_structure(id=0,
 5:     key=“””2 bytes from byte[12].bit[0],
 6:            4 bytes from byte[14].bit[0]””””)
 7:
 8: function = “””
 9:     send_to(port){
10:         send_to_physical(port);
11:     }””””
12: add_function(definition=function)
13:
14: function = “””
15:     decapsulate_and_send(port){
16:         remove(from=byte[12].bit[0], length=6B);
17:         send_to(port);
18:     }””””
19: add_function(definition=function)
20:
21: function = “””
22:     encapsulate_and_send(tag, port){
23:         insert(after=byte[12].bit[0], value=0x88b5);
24:         insert(after=byte[14].bit[0], value=tag);
25:         send_to(port);
26:     }””””
27: add_function(definition=function)
28:
29: commit_configuration()
30:
31: add_entry(structure_id=0, key=0x88b500000017,
32:           mask=0xffffffffffff, result=”send_to(7)“)
33: add_entry(structure_id=0, key=0x88b500000018,
34:           mask=0xffffffffffff, result=”send_to(8)“)
35: add_entry(structure_id=0, key=0x88b500000011,
36:           mask=0xffffffffff, result=”decapsulate_and_send(1)“)
37:
```

Listing 4.1 Example use of PAD API in Python

4.1.4 Device capabilities

Different network devices support different capabilities. Not all of hardware platforms allow manipulating forwarded frames and packets. In particular optical devices allow only circuit switching in the forwarding plane. Narrow set of network devices mostly based on network processors and programmable entities like FPGAs enable an access to the hardware datapath through a standardized API. Heterogeneity in the area of hardware with a diversified nature of the forwarding paradigm causes problems with a definition of the unified abstraction for all types of network devices [D3.1].

The PAD which exposes capabilities of different hardware platforms makes the concept of hardware abstraction more general and unified. The possibility of using only one chosen part of PAD functionality ensures elasticity of the presented solution. Each physical device is expected to support only a part of PAD's functionalities that is appropriate for the device. A well-defined system of supported capabilities will allow implementing the PAD on top of optical switches as well as network processor appliances without limiting their capabilities.

The key parameters included in capabilities definition:

- Maximal number of search structures
- Maximal length of a key in search structure
- Support for exact matches
- Supported instruction sets
- Support for protocol definitions

For example an optical switch can be presented by the following capabilities (because of the nature of the traffic forwarding at the optical level):

- Only metadata in search structure key (without direct access to frame)
- Sending the frame to port (without frame modification)
- Dropping the frame

Programmable Ethernet switches with full access to the frame support:

- Compound keys in search structure enable matching to different frame header fields as well as frame metadata with counters, ingress port identification etc.
- Sending the frame to port
- Modifications of the frame
- Dropping the frame

In the case of closed platforms or platforms with limited access to datapath, it is possible to implement the PAD model which uses an available management interface to the device only (e.g. CLI, SNMP). Because of different capabilities of network devices not all functions in execution engine will be installed in the PAD as well as entries in search structures will be adjusted to supported device capabilities.

4.2 PAD summary

The PAD internal architecture presented 4.1.1 should not be considered as complete a set of software modules for implementation on each hardware platform. Each implementation will provide a functional equivalent of presented abstraction using the software architecture suitable for the given hardware platform. The presented solution can be seen as a primary interface for all interactions with datapath of hardware devices that will implement it. As an open hardware abstraction layer that can be used by local control plane processes as well as by remote network controllers using middleware protocols.

OpenFlow, the most popular SDN protocol, can be implemented on top of the PAD as a middleware for compatibility with existing controllers. Such device configuration will present to the controller only functionalities supported by OF, but by using standardized hardware interface will be open for replacing OpenFlow implementation with a new one, overcoming current limitation of the OF protocol.

The only locally available PAD API requires a middleware protocol to be used by a remote network controller; however it can be also used by local control software (not necessarily an SDN-based solution). In such configuration the PAD can be used as a hardware abstraction layer for the implementation of legacy network protocols like STP or OSPF.

The PAD has been designed with ease of use in mind, but still is relatively easy to implement on most of hardware architectures. The PAD architecture can be directly converted into software modules which makes implementation straight forward on C language programmed network processor platforms like Cavium Octeon, Broadcom XLS/XLR/XLP or x86 supported by Intel DPDK. On EZchip NP based devices [EZchip] the code generated for parsers can be deployed in TOPparse and TOPsearch and can be used for all searches. Compiled actions can be deployed on TOPmodify and also partially in TOPresolve if additional instructions are needed.

Some considerations have been made in this chapter regarding implementation of the PAD prototype model, but there are still open issues that should be solved. Some of them affect the language specification for a description of supported capabilities, the language specification for actions definitions, the PAD metadata definition, the detailed specification of an API to the PAD, prototype implementation and definition of protocol that will expose PAD API functionalities to remote users.

The PAD architecture enables generic abstraction of different kind of network devices. The PAD model is not restricted only to Ethernet-based protocols. An opportunity to change the network device behaviour (also on the fly) based on a well-defined “network program” is a flexible solution which allows changing frames, packets or datagrams handling on the fly. The ability of fine-grained definition of the forwarding behaviour of network equipment gives new opportunities towards the SDN concept.

5. Summary and conclusions

In this document we explained the HAL architecture for network devices, designed to provide a platform for third party users for their OpenFlow implementation. The abstraction architecture has been designed in a way to cover most network device platforms such as programmable hardware, transport network devices (optical or circuit switch) and closed platform with proprietary communication protocols such as GEPON.

The ultimate goal for a hardware abstraction layer is to decouple the underlying hardware from control and programming interfaces on the northbound interfaces and create a cross-platform framework. Also, it should have a standard API which could be extended to support new hardware or features and supply a structured procedure for troubleshooting. Comparing this ideal HAL to the one that has been described in this document, one can see that the diversity of technologies used in network device platforms and various data plane architectures does not allow to implement this ideal HAL. Nevertheless the benefits of such an approach is very evident.

To compensate for these obstacles, the HAL in ALIEN project consists of two sub-layers: Cross-Hardware sub-layer which is in charge of implementing those components that do not rely on underlying hardware platform and Hardware Specific sub-layer which is in charge of implementing the components that are tightly coupled with underlying hardware platform. Dividing the HAL into two sub-layers enables the abstraction to happen gradually which gives the ability for expansion for new features such as on-demand programmability without redesigning the HAL architecture and also backward support for old OpenFlow versions.

Alternatively, an initial specification of Programmable Abstraction of Datapath (PAD) solution has been proposed to support on-demand protocol agnostic programmability of network devices. This solution proposes to use descriptive languages such as [NetPDL] or [P4] to introduce new protocols to the network device.

In conclusion, the HAL proposed in the ALIEN project not only supports the conventional hardware platform switches but also provides a framework for other network hardware platforms which are not designed to support OpenFlow protocol natively.

6. References

[OFADD]	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
[D3.2]	Deliverable D3.2 “Specification of hardware specific parts”, http://www.fp7-alien.eu/files/deliverables/D3.2-ALIEN-final.pdf
[WHITEPAPER]	Hardware Abstraction Layer (HAL) whitepaper, http://www.fp7-alien.eu/files/deliverables/ALIEN-HAL-whitepaper.pdf
[JSONRPC]	JSON-RPC protocol, http://www.jsonrpc.org/specification
[OFSPEC]	OpenFlow specifications,https://www.opennetworking.org/sdn-resources/onf-specifications/openflow
[xDPd]	The OpenFlow eXtensible DataPath daemon project, https://www.codebasin.net/redmine/projects/xdpd
[ROFL]	Revised OpenFlow Library, https://www.codebasin.net/redmine/projects/rofl-core
[FlowVisor]	R. Sherwood et al., “Can the production network be the testbed?” in Proc. of USENIX OSDI, Canada, 4-6 Oct. 2010.
[VeRTIGO]	R. Doriguzzi Corin, M. Gerola, R. Riggio, F. De Pellegrini, and E. Salvadori, “Vertigo: Network virtualization and beyond,” in Software Defined Networking (EWSDN), 2012 European Workshop on, 2012, pp. 24–29.
[P4]	P. Bosshart et al. “Programming Protocol-Independent Packet Processors”, "Programming Protocol-Independent Packet Processors.", arXiv preprint, arXiv:1312.1719, 2013
[NetPDL]	NetPDL Language Specification portal, http://www.nbee.org/doku.php?id=netpdl:index
[PACKETMMAP]	Linux packet mmap, https://www.kernel.org/doc/Documentation/networking/packet_mmap.txt
[NETMAP]	netmap: a novel framework for fast packet I/O, https://www.usenix.org/system/files/conference/atc12/atc12-final186.pdf
[IDPDK]	Intel® DPDK: Data Plane Development Kit, http://dpdk.org/doc
[SDNLAT]	E. Haleplidis, S. Denazis, K. Pentikousis, J. Hadi Salim, D. Meyer, and O. Koufopavlou, "SDN Layers and Architecture Terminology", Internet Draft, draft-haleplidis-sdnrg-layer-terminology-04 (work in progress), March 2014.
[OF-CONFIG]	Open Networking Foundation, "SDN Architecture Overview", December 2013, https://www.opennetworking.org/images/stories/downloads/sdn-resources/technical-reports/SDN-architecture-overview-1.0.pdf
[MLA]	Feamster, Nick, Jennifer Rexford, and Ellen Zegura. "The Road to SDN: An Intellectual History of Programmable Networks." (2013).
[RFC3535]	Schoenwaelder, Jürgen. "Overview of the 2002 IAB network management workshop." (2003).
[RFC6241]	Enns, R., et al.*Network Configuration Protocol (NETCONF): RFC 6241*. RFC Editor [online]. 2011. Available at: http://www. rfc-editor. org/rfc/rfc6241. txt. able at: http://www. eecg. toronto. edu/~ lie/papers/zarek_mscthesis. pdf, 2011.
[RFC3410]	Case, J., et al. "RFC 3410-Introduction and Applicability Statements for Internet Standard Management Framework."*IETF, RFC3410*(2002).
[RFC6020]	Bjorklund, M. "RFC 6020."*YANG-A Data Modelling Language for the Network Configuration Protocol (NETCONF)*(2010).
[RPF]	M. Casado et al., “Rethinking Packet Forwarding Hardware”, ACM SIGCOMM HotNets Workshop, Nov. 2008.
[POF]	H. Song, “Protocol-oblivious forwarding: Unleash the power of SDN through a future-proof forwarding plane”, ACM SIGCOMM HotSDN Workshop, Aug. (2013)
[OF-PDA]	(2014) Broadcom, “OpenFlow Data Plane Abstraction (OF-DPA)” [Online]; http://www.broadcom.com/collateral/pb/OF-DPA-PB100-R.pdf.

7. Acronyms

AFA: Abstracted Forwarding API
API: Application Programming Interface
CM: Cable Modem
FPGA: Field Programmable Fate Gate Array
HAL: Hardware Abstraction Layer
HSP: Hardware Specific Part
HPA: Hardware Pipeline API
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
VA: Virtual Agent
VG: Virtual Gateway
xDPd: Extensible Data Path Daemon