ALIEN

Abstraction Layer for Implementation of Extensions in programmable Networks

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

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

Deliverable D2.1
Report on Hardware Abstraction Models

Version 4.4

Due date:	31/12/2012
Submission date:	23/01/2013 (re-submitted 04/10/2013)
Deliverable leader:	PUT
Editor:	Janusz Kleban
Author list:	Marc Bruyere (FORCE), Richard G. Clegg, Raul Landa (UCL), Grzegorz Danilewicz, Janusz Kleban, Marek Michalski, Remigiusz Rajewski (PUT), Krzysztof Dombek, Artur Juszczyk, Łukasz Ogrodowczyk, Iwo Olszewski, Damian Parniewicz (PSNC), Roberto Doriguzzi, Matteo Gerola, Elio Salvadori (CREATE-NET), Eduardo Jacob, Jon Matias (UPV/EHU), Andreas Köpsel (EICT), Reza Najebati, Mehdi Rashidi (UoB),

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 report provides the state of the art concerning technologies and important aspects related to hardware abstraction models. The ALIEN project aims at delivering innovative network abstraction layer to connect non-OpenFlow capable equipment to OpenFlow environment, therefore all issues relevant to hardware abstraction mechanism should be considered. The report starts with short description of the main ideas of Software Defined Networking (SDN), where the OpenFlow protocol is used. This is the “hot topic” regarding modern and future communication networks. Next, issues related to hardware abstraction layer and the most important network and protocol description languages related to the goal of the ALIEN project are presented. To hide the complexity of underlying hardware it is possible to use intermediate representation languages and platforms described in this deliverable. The issues related to network virtualization as well as selected aspects of security in SDN are also discussed. To show the practical implementation of the OpenFlow environment the first OpenFlow based experimental facility at Europe is depicted in the report. The network was created within OFELIA collaborative project and allows researchers not only to experiment by using the network but to control the network itself.

Table of Contents

Figure Summary

Table Summary

Executive Summary

This report presents the review of state of the art for different technologies and important aspects related to hardware abstraction models, which are relevant to achieve the main goal of the ALIEN project. The ALIEN project will deliver an innovative hardware abstraction layer which allows connecting non-OpenFlow equipment to OpenFlow networks. The use of any specific network equipment with centralized control software is not an easy and trivial problem. Therefore, the abstraction mechanism is targeting the control and management convergence and interoperability of heterogeneous network elements used for OpenFlow networking. The following equipment is considered in ALIEN: NetFPGA cards, EZChip NP-3 network processors (in EZappliance platform), Cavium OCTEON Plus AMC network processor module in an ATCA systems and DELL switch, Optical switches, GEPON OLT and ONU units, DOCSIS hardware. All this hardware is not yet OpenFlow capable. For that reason ALIEN wants to hide the specific and very diverse hardware from the central control. The central control should be able to contact and control any kind of hardware. This will be impossible without an intermediate layer which resides between the hardware and the control software.

In Section 1 (**Software Defined Networking (SDN)**) a short introduction to SDN technology is provided. Basic ideas of SDN networking are explained. Next, the OpenFlow switch, as the most important component of the SDN network, is presented. An alternative to OpenFlow, namely ForCES (Forwarding and Control Element Separation) is also characterized in this section.

In Section 2 (**Hardware Abstraction Layer (HAL)**) the need for hardware abstraction layer is discussed. The abstraction layer is presented to address needs to cope with the heterogeneity of the existing hardware architectures and associated APIs while not limiting control plane designers to write high performance control plane applications due to a high level and inflexible data path model. The analysis is focused on the implementation of data path in the OpenFlow-based environment.

Section 3 (**Network and Protocol Description Languages**) is devoted to the most relevant network and protocol description languages related to the work defined in the ALIEN project. All these languages are summarized and compared.

Section 4 (**Intermediate Representation Languages**) presents languages which are called "intermediate representation languages". The term comes from computer science, where it means a language or data structure used during internal steps of computer program compilation. The intermediate representation allows for computer program analysis, optimization and transformations to another shape (e.g.: machine code).

Section 5 (**Network Virtualization**) deals with different aspects of network virtualisation. A survey of different approaches to virtualization in selected projects as well as a short description of tools enabling virtualization is provided.

In Section 6 (**OFELIA**) the presentation of the OFELIA collaborative project and an experimental facility created within the project is included. A brief summary of how ALIEN may be integrated with OFELIA facilities is provided.

Section 7 (**Security Aspect**) discusses the state of the art in security as it applies to the ALIEN project. In specific it covers security aspects related to Software Defined Networking and intermediate representation languages.

This report will be used in Task T2.2 to design both a novel hardware description language and a functional abstraction mechanism for uniform representation of any type of alien hardware and their functionalities.

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 Software Defined Networking (SDN)

Software Defined Networking (SDN) were created by researchers because of their frustration with the actual state of networks. Many computer and network scientists wanted to test their new networking ideas on real life networks. Unfortunately, it was impossible. Typical university networks were not enough – insufficient bandwidth, expensive hardware (switches and routers) made it impossible to test new ideas on a large enough scale. In turn, the switches and routers at the core of the Internet are locked down – their software constitutes intellectual property of companies such as Cisco and Hewlett-Packard. Therefore, in some cases it was impossible to program networks even in case when it was allowed to.

Researchers response to that situation was an OpenFlow which opens up the Internet to researchers [1], [2]. OpenFlow is an open standard which allows centralized way of programming flow tables in heterogeneous switches and routers [3], [4]. Such tables include flow entries with actions to determine instructions for routing and packet processing e.g. actions decide how packets should be modified and which packet should be send through which port. This idea allows to manage flow tables (adding, removing or editing table entries to switches or routers) via a central controller. Strong point of this solution is that researchers can experiment with new kind of flows which is not interfering with existing ones. It makes also easiest to test a new kind of routing and switching protocols.

Of course, OpenFlow is not the only one method supporting the SDN concept. Cisco introduced its own SDN idea called Cisco ONE [5], [6]. This solution, unlike OpenFlow, allows to program layers in the network both above and below the data and control plane/layer [1]. Another solution for SDN – as an alternative to OpenFlow and Cisco ONE – is ForCES introduced by IETF [7], [8], [9], [10], [11], [12], [13], [14], [15]. More information about OpenFlow and ForCES can be found in sections 1.3 and 1.4, respectively.

The current networks (in many cases the core networks) are inability to scale and extent because of greater requirements from day to day. Device vendors offers however, their solutions which are closed and have their own policies which both makes unable to extend functionality of such a network devices. If some device (switch or router) suddenly brakes down network administrators have to replace it by a new one and, what's more important, they have also to reconfigure other devices to bring previous network functionality. It takes many hours, days or sometimes even weeks. Nowadays network reached almost critical level from the society point of view that's why many network operators and administrators plan to modify their network infrastructure using innovative technology.

1.1 Overall Information

Software Defined Networking (SDN) is an idea of network which allows network administrators or operators to flexibly manage network devices like routers or switches using software running on special dedicated to this purpose external servers. The main idea of this solution is to separate the control plane/layer from the packet forwarding plane/layer [2], [16]. The SDN architecture can be seen in Figure 1.1. The control plane is realized as a "network operating system" (more simply "network OS") which manages the entire current network state just from one central point in such a network. To make possible to realize network-state abstractions and forwarding, few SDNs logically centralized control has been proposed. Some OpenFlow controllers are:

- NOX – the first OpenFlow controller [17], [18],
- POX – high-level SDN API including a queriable topology graph and support for virtualization [19],
- Beacon – Java-based controller that supports both event-based and threaded operation [20], [21],
- FloodLight – Java-based OpenFlow controller [22],
- Trema – full-stack framework for developing OpenFlow controllers in Ruby and C [23],
- Ryu – open-source Network Operating System (NOS) that supports OpenFlow [24],
- BigSwitch [25], [26].

The network OS possessed all control plane functions [27]. In addition it can interact with many network devices (switches, routers, virtual switches) and presents a new abstract view of the network state. To configure flow tables, forwarding rules, etc., a network OS uses forwarding plane abstraction. Network-function oriented applications, such a VLAN provisioning and load-balancing, can use these abstractions to achieve the desired network behaviour without knowledge of detailed physical configuration. This is a huge advantage of using SDN with the current network.

Normally, the network OS can control and oversee a whole network. In addition it can manage:

- routing protocols,
- access control,
- network virtualization,
- energy management,
- new prototype features.

The strongest argument to use SDN is that such a network can be extended by a new features added in software. In addition, network operators and administrators can programmatically configure this simplified network abstraction rather than having to hand-code tens of thousands of lines of configuration scattered among thousands of devices. In other words, an SDN controller's centralized intelligence allows configuration of every device in real-time and extension of network services for a new application in a matter of days or even hours rather months or weeks as is needed today.

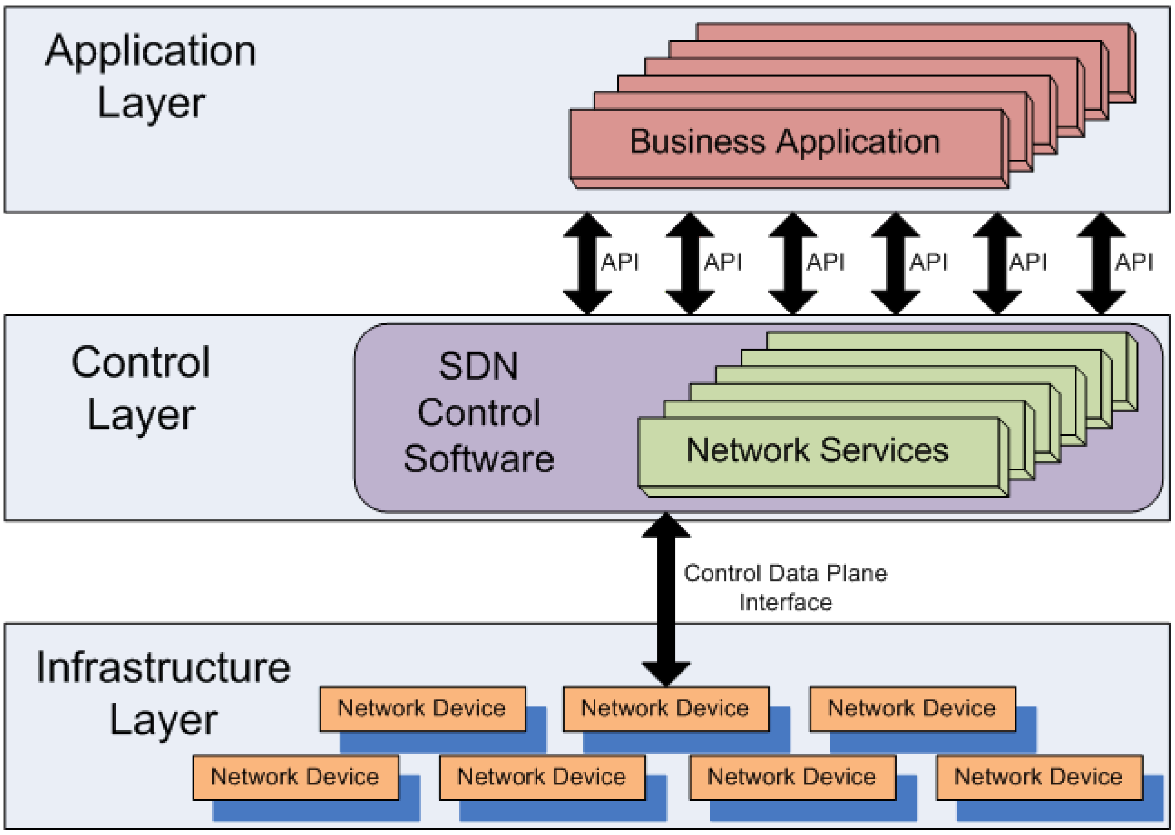


Figure 1.1: The SDN architecture [28]

1.2 The Future of the Network

SDN and OpenFlow or ForCES-based SDN technologies enable IT administrators to address the high-bandwidth, dynamic nature of today's applications, and adapt the network to ever-changing business needs. More importantly, this solution allows significant reduction to operational and management complexity.

The following benefits of SDN architecture can be distinguished [28]:

- **Centralized control of multi-vendor environments** – SDN control software can manage and configure any OpenFlow-enabled network device (switches, routers, virtual switches) from any vendor. Rather than manage small groups of devices (switches, routers, virtual switches) from individual vendors, IT administrators can use SDN-based tools and management ideas to quickly deploy, configure, and update devices across the entire network.

- **Reduced complexity through automation** – SDN offers a flexible network automation and management framework. It will solve a problem with manual management tasks by providing proper tools. These tools will reduce operator overhead, decrease network instability introduced by operator error and support emerging IT-as-a-Service and self-service provisioning models. In addition, cloud-based applications can be managed through intelligent orchestration and provisioning systems, further reducing operational overhead while increasing business agility.

- **Higher rate of innovation** – SDN allows faster business innovation. IT network operators or administrators can program and reprogram the whole network in real time to meet user requirements and business needs. By virtualizing the network infrastructure and abstracting it from individual network services SDN and OpenFlow, ForCES, or Cisco ONE give IT (in future maybe even users) the ability to tailor the behaviour of the network and introduce new services and network capabilities in a matter of hours.

- **Increased network reliability and security** – IT administrators or network operators can define high-level configuration and policy statements. Next, they are translated down to the infrastructure via OpenFlow (it could be done also via ForCES or via Cisco ONE). This solution eliminates the need to individually (re)configure network devices each time an end point, service, application is added or moved. This is done also if a policy changes.

- **More granular network control** – OpenFlow's flow-based control model application of policies at a very granular level, including the session, user, device, and application levels, in a highly abstracted, automated fashion. Such control enables cloud operators to support multiple tenantsi while maintaining traffic, traffic isolation, security, and elastic resource management when customers share the same infrastructure.

- **Better user experience** – By centralizing network control and collecting state information, in one point, making it available to high-level application an SDN can better adapt to dynamic business needs.

1.3 OpenFlow

1.3.1 Overall Information

OpenFlow is an open standard supporting communications interface defined between the control and forwarding planes of an SDN architecture. The location of OpenFlow in SDN architecture is shown in Figure 1.1. The OpenFlow ecosystem consists of routers, switches, virtual switches, and access points. The main idea of OpenFlow is to give access to and facilitate manipulation of the forwarding plane of network devices. It provides an open interface to control how data packets are forwarded through the network, and a set of management abstractions used to control topology changes and packet filtering. The behavior of network devices may be modified through a well-defined "forwarding instruction set". In this case network control may be moved out of the networking nodes to logically centralized control software. The OpenFlow protocol specifies a set of instructions that can be used by an external application to program the forwarding plane of network devices.

Currently, OpenFlow is implemented by major vendors in commercial switches, routers and wireless access points to allow researchers to run experimental routing protocols for example in campus networks without needing to reconfigure the internal workings of network devices. Now users can control how traffic flows through a network defining flows and determining what path those flows take through a network, regardless of the underlying hardware. The OpenFlow standard may be used for applications such as virtual machine mobility, high-security networks and next generation IP based mobile networks [29].

OpenFlow consists of three parts (see Figure 1.2) [3]:

- **Flow Tables** installed on switches. The switch is informed how to process the flow by an action associated with each flow entry.

- A **Controller**, which uses the OpenFlow protocol to communicate with switches to impose policies on flows. The OpenFlow protocol provides an open and standard way for the controller to communicate with switches and allows entries in the flow table to be defined externally. Flows are transmitted through the network on paths predefined using the controller and enforced by switches.

- A **Secure Channel** that connects the remote controller (remote control process) to switches and allows secure communication between them. The SSL protocol may be used to securely send commands and packets from the controller to switches using the OpenFlow protocol.

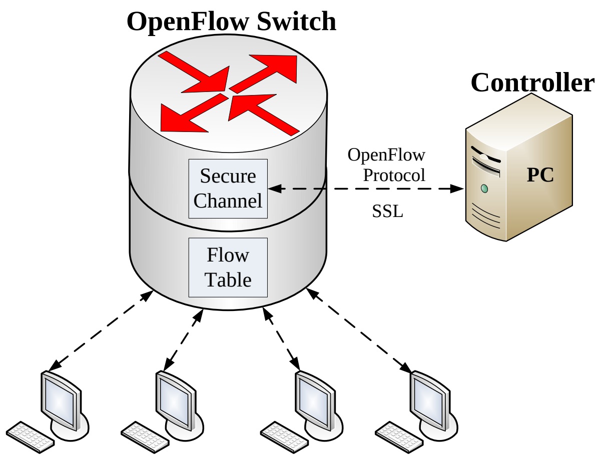


Figure 1.2: The architecture of an OpenFlow switch [28]

The OpenFlow architecture separates data path and control path functions. The data path functions are still implemented on the switch, while routing decisions are moved to the controller, which is a standard server. This solution is different from a classical router or switch, where a data and control planes occur in the same device. Now, the controller communicates via the OpenFlow protocol with switches using such messages as packet-received, send-packet-out, modify-forwarding-table, and get-stats. The data plane is based on the flow table, where each entry contains a set of packet fields to match, and an action e.g. send-out-port, modify-field, drop. If there is no matching flow entry for a received packet the OpenFlow switch sends this packet to the controller, which decides how to handle the packet. The packet may be dropped or a flow entry may be added to the flow table to inform the switch how to forward similar packets in the future.

1.3.2 OpenFlow Switch

The main components of an OpenFlow switch are shown in Figure 1.3 [30]. It consists of one or more flow tables and an OpenFlow channel to an external controller. The flow tables are used to manage flows and perform packet lookups, and forwarding. The OpenFlow channel is used by the controller to manage the switch via the OpenFlow protocol. Using this communication channel the controller can add, update, and delete flow entries in flow tables. Each flow table contains a set of flow entries with match fields, counters, and a set of instructions to apply to matching packets. Matching starts at the first flow table and continues to additional flow tables until a matching entry is found. In the case of matching, the instructions associated with the matching entry are executed. If no match is found in the flow table the packet may be forwarded to the controller over the OpenFlow channel, may be dropped or may continue to the next flow table. Actions performed on packets and pipeline processing are defined by a set of instructions associated with each flow entry. Packet forwarding, packet modification and group table processing are described by actions. Pipeline processing instructions define how subsequent tables process packets and what kind of information, in the form of metadata, is sent between tables. If a next table is not specified by the instruction set associated with a matching flow entry the table pipeline processing stops and the actions in the pre-defined action set of the packet are executed.

Packets may be forwarded to a physical or logical port. The logical port may be defined by the switch or by the OpenFlow switch specification. Ports defined by the OpenFlow switch specification are called reserved ports. These ports may specify generic forwarding actions such as: sending to the controller, flooding, or forwarding using non-OpenFlow methods. The switch-defined logical ports may specify link aggregation groups, tunnels or loopback interfaces.

Additional processing is specified by a group table. The group table consists of group entries and represents sets of actions for flooding, as well as more complex forwarding semantics (e.g. multipath, fast reroute, and link aggregation). Each group entry contains a list of actions buckets with specific semantics dependent on group type. The actions in one or more action buckets are applied to packets sent to the group.

The detailed information about each component of the OpenFlow switch as well as the OpenFlow protocol may be found in [30].

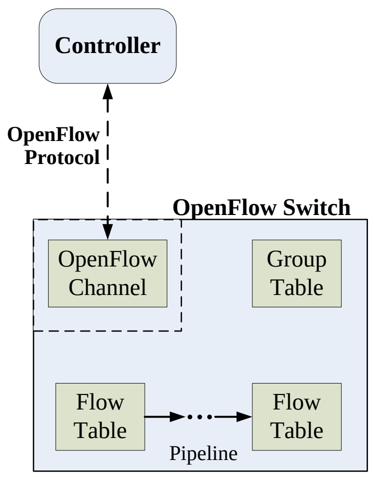


Figure 1.3: Main components of the OpenFlow switch [30]

1.3.2.1 *OpenFlow Ports*

OpenFlow ports are the logical network interfaces made by an OpenFlow switch for OpenFlow processing. Using these ports OpenFlow switches connect logically to each other. The number of OpenFlow ports may not be identical to the number of network interfaces provided by the switch hardware. Some physical network interfaces may be disabled for OpenFlow, and some additional OpenFlow ports may be defined by the switch. OpenFlow packets arrive to an ingress port, next they are processed by the OpenFlow pipeline, and finally they may be forwarded to an output port. An OpenFlow switch must support three types of OpenFlow ports: physical ports, logical ports and reserved ports.

1.3.2.2 *OpenFlow Tables*

OpenFlow switches may be classified into two types: OpenFlow-only, and OpenFlow-hybrid. OpenFlow-only switches support only OpenFlow operations, and all packets are processed by the OpenFlow pipeline. OpenFlow-hybrid switches support both OpenFlow operations and normal Ethernet (or any Layer 2 techniques) operations. These switches must classify traffic and route it to either the OpenFlow pipeline or the normal pipeline. In this kind of switch packets may go from the OpenFlow pipeline to the normal pipeline through reserved ports. The main components of the OpenFlow pipeline are shown in Figure 1.4.

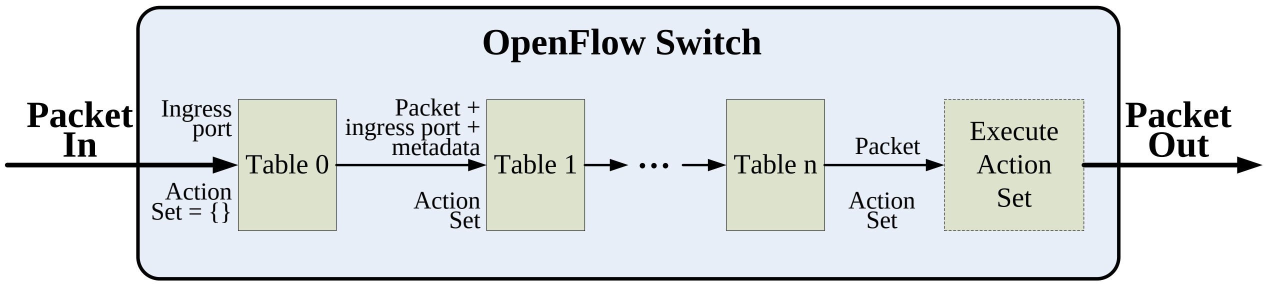


Figure 1.4: Packet flow through the processing pipeline [30]

The OpenFlow pipeline processing defines interactions between packets and flow tables containing multiple flow entries. One or more flow tables may be implemented in the OpenFlow switch. The pipeline processing is greatly simplified when only a single flow table is used. In the case of multiple flow tables they are sequentially numbered, starting at 0. During pipeline processing the OpenFlow packet is first matched against flow entries of flow table 0. Depending on the outcome of the match in the first table, other flow tables may be used. If the flow entry is found, the instruction set included in that flow entry is executed. The packet may be directed to another flow table, where the same process is repeated again. If the packet is not directed to another flow entry, pipeline processing stops at this table. Next, the packet is processed according to the associated action set and forwarded. Unmatched packets are: drooped, passed to another table or to the controller over the control channel.

Each flow table contains flow entries. Each flow table entry consists of the following fields: Match Fields, Priority, Counters, Instructions, Timeouts, and Cookie:

- Match Fields – to match against packets. These consist of the ingress port and packet headers, and optionally metadata specified by a previous table.

- Priority – matching precedence of the flow entry.
- Counters – are updated when packets are matched.
- Instructions – to modify the action set or pipeline processing.
- Timeouts – maximum amount of time or idle time before flow is expired by the switch.
- Cookie – opaque data value chosen by the controller. May be used by the controller to filter flow statistics, flow modification and flow deletion.

A flow table entry is identified by its match fields and priority: the match fields and priority taken together identify a unique flow entry in the flow table. The flow entry that wildcards all fields (all fields omitted) and has priority equal to 0 is called the table-miss flow entry.

Flow entries may be removed from flow tables in two ways: at the request of the controller or by the switch flow expiry mechanism. This mechanism is based on the two parameters: idle_timeout and hard_timeout. These parameters are associated with each flow entry. If the hard_timeout_field is non-zero, the entry's arrival time must be noted by the switch and the flow entry has to be removed after the given number of seconds. If the idle_time_field is non-zero, the arrival time of the last packet of the flow must be noted by the switch, and the flow entry has to be removed when it has matched no packets in the given number of seconds. When one of the timeouts is exceeded the associated flow entries must be removed from the flow table. Flow entries may be also removed by the controller using delete flow table modification messages. After deletion of a flow entry the switch must check the flow entry's flag. If the flag is set, the switch must send a flow removed message to the controller. Each flow removed message consists of the following: a complete description of the flow entry, the reason for removal (expiry or deletion), the flow duration at the time of removal, and the flow statistics at the time of removal.

1.3.2.3 *OpenFlow Channel*

The OpenFlow channel is the interface that connects each OpenFlow switch to a controller. It is usually encrypted and may be run over the TCP/IP protocol stack. Using the OpenFlow channel the controller configures and manages the switch, receives messages about events from the switch, and sends packets out the switch. A typical OpenFlow controller manages multiple OpenFlow channels, each one to a different OpenFlow switch. An OpenFlow switch may have one OpenFlow channel to a single controller, or multiple channels for reliability, each to a different controller. The OpenFlow switch always initiates connections to an OpenFlow controller.

All OpenFlow Channel messages must be formatted according to the OpenFlow protocol, which supports three message types: controller-to-switch, asynchronous, and symmetric, each with multiple sub-types. Controller-to-switch messages are initiated by the controller and used to directly manage or inspect the state of the switch. Asynchronous messages are initiated by the switch and used to update the controller about network events and changes to the switch state. Symmetric messages are initiated by either the switch or the controller and sent without solicitation.

The OpenFlow protocol provides reliable message delivery and processing. It does not automatically provide acknowledgements or ensure ordered message processing. If the OpenFlow channel fails, the switch may have gone into "fail standalone mode".

Switches must process every message received from a controller, and possibly generate a reply. If a switch cannot completely process a message received from a controller, it must send back an error message. Packets may be silently dropped after OpenFlow processing due to congestion at the switch, QoS policy, or if sent to a blocked or invalid port. Switches must send to the controller all asynchronous messages generated by the OpenFlow state changes, such as flow-removed, port-status or packet-in messages to ensure that the controller has the actual knowledge about the switch state. Packets may be dropped also when they fail to match any entries in a flow table, and that table's default action is to send to the controller. Controllers are free to ignore messages they receive, but must respond to echo messages to prevent the switch from terminating the connection.

The order of messages can be ensured through the use of barrier messages. Messages must not be reordered across a barrier message and the barrier message must be processed only when all prior messages have been processed. More precisely:

- messages before a barrier must be fully processed before the barrier, including sending any resulting replies or errors;
- the barrier must then be processed and a barrier reply sent;
- messages after the barrier may then begin processing.

If two messages from the controller depend on each other they must be separated by a barrier message.

1.3.3 OpenFlow Control Framework

The ALIEN project will extend the OpenFlow control framework developed within the OFELIA project to support abstraction of network information of equipment that are alien to the OpenFlow technology such as optical network elements, legacy Layer2 switches, network processors and programmable hardware (FPGA) [31].

The OFELIA Control Framework can be defined as the control plane application for the OFELIA FP7 facility. The main purpose of the framework is to automate, simplify and authorize users to create network slices and deploy resources available within OFELIA islands for various types of experimental projects. The following principles have guided the development work for the OFELIA control framework:

- **Resource allocation**: the user should be able to allocate or book resources in an easy way. Within the OFELIA testbed the different basic types of resources will be: OpenFlow resources (such as OpenFlow enabled switches, switch ports, traffic flows), hosts: (either virtual or physical), in-cluster VMs or any other resource that partners want to include in the OFELIA facility.
- **Experiment and project based resource allocation**: the resource allocation must be made per project and slice. A slice is defined as the smallest indivisible entity that is composed by the resources necessary to carry out an experiment. A project may be composed of one or more slices.

OFELIA adopts the Enterprise GENI (E-GENI) Control Framework (CF) for its facility as a base over which a lot of new features and functionalities are added according to OFELIA's requirements. The E-GENI control framework is based on SFA (a federation framework which defines a set of rules by which two or more experimental entities can be federated).

The overall OFELIA high-level system architecture is composed of the following layers:

- Top Layer: contains entities responsible for defining policies for the usage of the facilities;

- Middle Layer: contains the CF components which manage and monitor the applications and devices in the physical substrate layer;

- Bottom Layer: set of physical resources which include the IT resources (servers, processors) and Network resources (routers, switches, links, wireless devices, and optical components) which we identify as the physical substrate.

In this architecture, the User Interface (UI) is a centralized entity which will talk to the all clearing houses in the lower layers. The Clearing house/Slice manager is responsible for communicating with all the Aggregate Managers through its south bound interface to collect all information regarding the available resources and present it to the UI layer.

The Middle Layer is composed of:

- Aggregate Managers, responsible for a set of components and interacting with a set of Resource Managers. Depending on how slices are built in the OFELIA infrastructure, AMs may form a hierarchical structure.

- Resource Managers, providing interfaces for specific devices or resources, e.g., a switch, a server, and virtual machine, or a subset of the TCAM entries on switches. RMs perform resource management and virtualization.

1.4 ForCES

The basic idea of ForCES (Forwarding and Control Element Separation) is the separation of the forwarding plane and control plane in network elements. The ForCES specification contains a modelling language which allows flexible definition of the flow tables and flow logic. ForCES flow logic include Logical Functional Blocks (LFBs) connected in flow logic that is described in the logic of direct graphs augmented by passage of Metadata and grouping concepts.

ForCES work in IETF has defined a new environment to build network devices that split the network devices into control plane and forwarding plane units. For example, a router could be considered a network element (NE) with a control plane running the router protocol and a data plane forwarding IP traffic.

The drive to have a ForCES NE device split arose from the desire to build hardware forwarding tables out of flexible hardware components. These hardware devices included Network Processors and network specific ASICs.

The ForCES environment defines requirements [7], goals [32], architecture and protocol requirements [7], a controller-forwarder communication protocol [9]. ForCES also describes a policy on how to building the forwarding engine out of a set of logical functional blocks (LFBs)which are connected as a directed graph [11]. ForCES allows many different Forwarding Engines (FE) to linked to Controller Engines (CE) via the protocols. ForCES provides a modelling language [11] to describe these FE devices so that controllers can load control the devices, load forwarding tables, and keep track of statistics. ForCES RFCs also define how the ForCES protocol runs over SCTP [10].

1.4.1 Forces Definitions

1.4.1.1 *Force Forwarding Element (F-FE)*

A logical entity that implements the ForCES Protocol. FEs use the underlying hardware to provide per-packet processing and handling as directed by a CE via the ForCES Protocol [7]. ForCES forwarding FE supports forwarding rules insertion.

1.4.1.2 *ForCES Control Element (F-CE)*

A logical entity that implements the ForCES Protocol and uses it to instruct one or more FEs on how to process packets. CEs handle functionality such as the execution of control and signalling protocols [7]. The ForCES CE controller may be located within the same hardware box on a different blade or across an Ethernet connection, or across a L3 Link (if security used).

1.4.1.3 *ForCES Network Element (F-NE)*

An entity is composed of one or more CEs and one or more FEs. To entities outside a NE, the NE represents a single point of management. An NE usually hides its internal organization from external entities and represents a single point of management to entities outside the NE [7]. The NE's single point of management can be at the IP layer, the Ethernet layer, and at a virtual layer. In this document, the network element is examined as being the set of network functions in the hardware that collaborates to act like a switch. This less strict definition allows ForCES to be compared with the Open Flow work.

1.4.1.4 *ForCES Pre-Association Phase (F-Pre-A)*

ForCES defines the Pre- Association Phase (F-Pre-A) as "the period of time during which a FE Manager (see below) and a CE Manager (see below) are determining which FE is a part of the network element" [7].

1.4.1.5 *FE Manager(F-FE-Mgr) – ForCES (F-FE-Mgr)*

This is a logical entity that operates in the Pre-Association Phase and is responsible for determining to which CE(s) a FE should communicate. This process is called CE Discovery and may involve the FE manager learning capabilities of available CEs [7].

1.4.1.6 *CE Manager (CE-Mgr) – ForCEs CE-MGR (F-CE-Mgr)*

This is a logical entity that operates in the pre-association phase and is responsible for determining to which FE(s) a CE should communicate. This process is called FE discovery and may involve the CE manager learning the capabilities of available FEs. The CE manager may use anything from statics configuration to a pre-association phase protocol [7].

1.4.1.7 *ForCES Protocol (ForCES-Proto)*

While there may be multiple protocols used within the overall ForCES architecture, the term "ForCES protocol" refers to only the post-association phase protocol [7].

The ForCES protocol operates between the "FP reference points" of the ForCES architecture (as shown in Figure 1.5) [9].

Basically, the ForCES protocol works in a master-slave mode in which the FEs are slaves and the CEs are masters [9].

The location and exact instantiation of the CE logical entities associated with the FE logical entity is flexible. The CE entities could reside on a process on a local switch communicating to other process off the local switch.

1.4.1.8 *ForCES Protocol Layer (ForCES PL)*

This is a layer in the ForCES protocol architecture that defines the ForCES protocol messages, the protocol state transfer scheme, and the ForCES protocol architecture itself (including requirements of ForCES TML)" [9]. This layer is defined in [9].

1.4.1.9 *ForCES Protocol Transport Mapping Layer (ForCES TML)*

This is a layer in the ForCES protocol architecture that uses the capabilities of existing transport protocols to specifically address protocol message transportation issues, such as how the protocol messages are mapped to different transport media (like TCP, IP, ATM, Ethernet, etc.), and how to achieve and implement reliability, multicast, ordering, etc. The ForCES TML specifications are detailed in separate ForCES documents, one for each TML [9].

The ForCES TMLs focused on are STCP and SSL. TM handles transport of messages (reliable or non-reliable), "congestion control", "multicast", ordering, and other things [9].

1.4.1.10 *LFB (Logical Function Block)*

This is the basic building block that is operated on by the ForCES protocol. The LFB is a well-defined, logically separable functional block that resides in an FE and is controlled by the CE via the ForCES protocol. The LFB may reside at the FE's data path and process packets or may be purely an FE control or configuration entity that is operated on by the CE. Note that the LFB is a functionally accurate abstraction of the FE's processing capabilities, but not a hardware-accurate representation of the FE implementation.

1.4.1.11 *LFB Class and LFB Instance*

LFBs are categorized by LFB classes. An LFB instance represents an LFB class (or type) existence. There may be multiple instances of the same LFB class (or type) in an FE. An LFB class is represented by an LFB class ID, and an LFB instance is represented by an LFB instance ID. As a result, an LFB class ID associated with an LFB instance ID uniquely specifies an LFB existence.

1.4.1.12 *Physical Forwarding Element*

This is the physical element that forwards the packets.

1.4.2 Forces Building Blocks

The building blocks within the ForCES architecture are the CEs (controller elements), FEs (forwarding elements), and an interconnect protocol between CE(s) and FE(s). ForCES also recognizes the logical functions of a FE-Manager and a CE Manager. Figure 1.5 shows a diagram that details interaction between all these components [9].

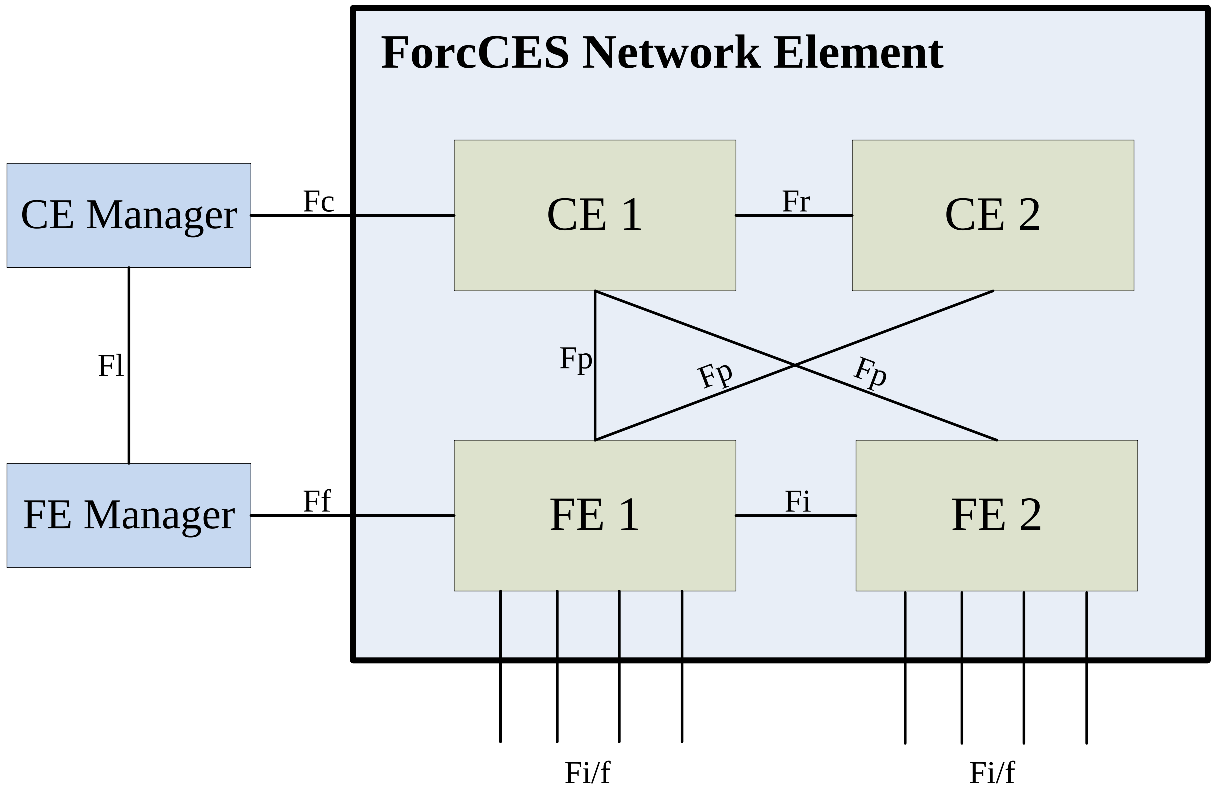


Figure 1.5: Diagram of interactions between CEs, FEs, CE-Manager and FE-Manager [9]
Fp: CE-FE interface,
Fi: FE-FE interface,
Fr: CE-CE interface,
Fc: Interface between the CE manager and a CE
Ff: Interface between the FE manager and an FE,
Fl: Interface between the CE manager and the FE manager,
Fi/f: FE external interface

The ForCES CE controls switching, signalling, routing, and management protocols. Each CE is a logical unit which may be located within the same box, different boxes, or across the network. ForCES architecture [33] allows CEs to control forwarding in multiple FEs.

ForCES defines logical Forwarding Elements (FEs) that reside on a variety of physical forwarding elements (PFE) such as a "single blade (PFE)", partition within blade, or multiple PFEs in a single box, or among multiple boxes [33]. The ForCES logical FEs could also be run within Virtual Machines (VMs) within a single box or a set of boxes or a cloud.

A single FE may be connected to multiple CEs providing strong redundancy. FE internal processing is described in terms of Logical Forwarding Blocks (LFBs) connected together in a directed graph that "receive process, modify and transmit packets along with metadata" [9]. The FE model determines the LFBs, the topological description, the operational characteristics, and the configuration parameters of each FE.

The Forces Logical Forwarding Block (LFBs) Library FORCES-LFB provides the class descriptions for Ethernet, IP Packet Validation, IP Forwarding LFBs, and Redirection, MetaData, and Scheduling. Forces-LFB document demonstrates how these logical blocks can be placed within a machine to support IP Forwarding (IPv4/IPv6) for unicast & multicast and ARP processing.

ForCES architecture [33] allows CEs to control forwarding in multiple FEs. ForCES also recognized the logical functions of a FE-Manager and a CE-Manager. The FE manager determines the CE(s) each FE should communicate with. The CE manager determines which FEs each CE should communicate with. The ForCES defines the FE Manager and CE-Manager to operate in a "pre-association" phase of the communication to set-up the FORCES communication path. Similarities between the functions of the CE-Mangers and FE managers of ForCES and modern hypervisors may come from the creative interplay of early open source communities. Applications directly interacting with ForCES components (CEs or CE-Managers or FE-Manager) could be described as interactions with the CEs or CE-Managers.

1.5 Conclusions

This section shows, that the idea of Software Defined Networking draws an attention of significant number of research-based institutions (e.g. universities, vendors, standardization organizations) – the main players on the networking market. The concept of SDN is gaining attention as a viable solution which can address the simultaneous needs for virtualization, manageability, security, and agility in networks. Benefits of the SDN architecture, highlighted in this section, confirm big importance of this concept for network vendors and operators. The OpenFlow protocol, generally saying the OpenFlow ecosystem, becomes a key solution for SDN networks. Another solution for SDN – as an alternative to OpenFlow and Cisco ONE – is ForCES introduced by IETF. The ALIEN project will focus on the OpenFlow environment and will continue developments already started in the OFELIA project.

Todays' networks have been built using different kind of devices like switches, routers or optical equipment. Many of these devices are non-OF equipment, not able to support the concept of SDN. The development of the abstraction layer, as the ALINE project proposes, will open opportunities for owners of such equipment to participate in experiments in the SDN domain.

2 Hardware Abstraction Layer (HAL)

2.1 Overall Information

Let us start with some brief consideration of the architectural constraints and restrictions imposed by split architectures. We will focus on OpenFlow as it is the dominant framework in the SDN arena currently. We expect the reader to be familiar with one of the recent OpenFlow specifications, but the observations mentioned here apply to other SDN approaches (like IETF's ForCES framework [8], [34]) as well. In split architecture, a network element is logically fragmented into three modules for control, forwarding, and packet processing. Though logically separated, both forwarding and processing modules are tightly interwoven and will most likely reside on the same network element (called a data path element in OpenFlow). However, with SDN we gain freedom to develop and deploy control modules outside of a network element and control the latter via a well-defined control interface. Besides the constraints imposed by relocating the control module in terms of signalling delay, available control connection capacity, and so on, both control plane and data plane need a common shared data model for synchronizing commands emitted by the control module and interpreted by the forwarding/processing logic in the network element.

OpenFlow's data model is no exception to this rule and provides an abstract view of the data path element towards the control plane, its device state, and available processing capabilities. The data model and its accuracy for depicting the data path element's internal structures and capabilities, defines the network element's programmability: exposing more details of the underlying forwarding and processing engines and pipelines towards the control module may allow control plane developers to write control plane logic modules better suited to hardware capabilities, but limits the architectural freedom for designing different architectures for data path elements at the same time. Defining a restricted data model that hides many of the network element's details imposes a trade-off, as it simplifies writing suitable control logic, but may also result in worse performance numbers in terms of forwarded or processed packets.

The authors of the OpenFlow specification followed a pragmatic approach as they adopted a stable and well-proven interface for controlling network elements: the interface used for controlling the switching chip-set itself. Following such APIs has two implications: (a) the level of detail defined for OpenFlow's data model cannot be more accurate than the level of detail exposed by the switching chip-set API, and (b) the set of available processing capabilities is restricted to the set of processing instructions defined by the switching chip-set. As the switching chip-sets expose processing commands that are low-level in nature (setting a VLAN ID, changing a MAC address, pushing a MPLS header on the frame, and so on), the control plane developer in OpenFlow ends up with the need to program data path elements with some kind of network element assembler. Initially, the set of supported processing functions in OpenFlow was limited to those protocols typically found in campus networking environments like Ethernet and IPv4; subsequent versions introduced support for new protocols like MPLS, IPv6, or GRE encapsulation. However, previous projects like SPARC (Split Architectures for Carrier-Grade Networks [35]) and CHANGE [36] have documented the need for more advanced protocols in SDN.

The first non-software based OpenFlow implementations were deployed on ASICs (application-specific integrated circuits), so the lack of flexibility in the programming API did not impose a significant restriction to the OpenFlow protocol as ASICs themselves are non-programmable. In contrast to these devices, network processor units, FPGA based chip-sets or even general purpose processing units like Intel's x86 CPUs and the associated DPDK framework [37] claim reconfigurable processing capabilities while running their NICs at line speed (one of the compromises is potentially a limited number of NIC ports attached to these units, though). Here, two major drawbacks of the OpenFlow framework become apparent: the OpenFlow protocol lacks a means to program such devices at run-time and OpenFlow's data model cannot express the details of these platforms. To sum up, the requirements for an advanced data model (hardware abstraction) are:

- Enable the OpenFlow data model to express hardware specific details to the control plane. However, a simplified view on the network element should remain possible unless the control plane requests a more detailed view in order to avoid burdening the control plane developer with (probably unwanted) high complexity.

- Provide a means to define new processing actions for programmable network elements when supported at run-time. This implies a definition of packet frame formats, semantics of specific header fields for packet matching, and processing functions.

- Based on the low-level processing commands defined in OpenFlow, allow the definition of more complex compound commands in order to reduce complexity as seen by the control plane. Such compound commands act like function definitions, as they cluster a sequence of low-level processing commands in a well-defined functional block. This may also reduce the amount of state maintained by a data path element. The control plane should be enabled to define new and erase existing function calls at run-time.

2.2 HAL in Different Operating Systems

We have seen in the previous section that a mismatch exists between the level of abstraction OpenFlow provides and the capabilities of the underlying hardware (or software) implementation of the network element. In the 1980s researchers tackled similar problems in the area of operating systems' research. Let us briefly revisit their findings: Traditional, monolithic kernels offer an abstraction of the underlying hardware, providing a well-known (standardized) application programming interface for application developers and hiding hardware specific details. With the advent of new networking protocols, file systems, and hardware devices, these kernels became more and more bloated stimulating research on and a series of micro-kernel implementations that provided only a limited set of functions, relocating higher level services towards the application space (e.g. device drivers, networking stacks, etc.). A micro-kernel is a kernel that is minimal (e.g. see the L4 micro kernel [38]), i.e. it provides the rudimentary set of functions for process/thread/memory management and inter-process communication, but it still defines some model for a hardware abstraction. Contrary to this, the exokernel developed at MIT [39] can be seen as an even more radical approach, as it defines only a shim layer organizing access to hardware resources among various so-called library operation systems and reveals the true hardware layout without any abstraction [40]. The authors of [38] state that its design was inspired by the end-to-end principle: an operating system should avoid definitions of a virtualized hardware abstraction as the application designer knows best his application's requirements and any type of abstraction will lead to sub-optimal performance.

What do we learn from the preceding paragraph for creation of a Hardware Abstraction Layer for an SDN/OpenFlow environment? First of all, contrary to general purpose processors and computing architectures, hardware architectures for network elements differ significantly in their exposed features and APIs. Dedicated switching chip-sets (ASICs or FPGAs) compete with network processors (probably enriched by dedicated hardware units for conducting specific packet oriented operations and optimized software libraries) and general purpose processing units like Intel's x86 family of CPUs for running network code on Linux or BSD. Some of the characteristics typically seen with ASIC designs (compared to a reprogrammable FPGA) are: (a) the hard wired and immutable set of parsing, processing, and forwarding actions offered, and (b) a proprietary API accessible via a system library usually located on an associated embedded system. When comparing the OpenFlow architecture with its processing and forwarding actions and matching structure, some similarities to existing ASIC APIs offered by major chip-set vendors seem to be undeniable. On the other end of the hardware spectrum, the general purpose CPUs (like the x86 family) offer the highest flexibility in defining APIs ranging from the well-known Linux/BSD network interfaces up to specially optimized APIs as offered by Intel's Data Plane Development Kit (DPDK) [37]. NPUs define some kind of compromise: Cavium's OCTEON family of network processors is based on the MIPS architecture including dedicated packet processing optimized hardware and providing a software development kit for the (again) proprietary API. To sum up this paragraph, one of the challenges for a Hardware Abstraction Layer is the need to cope with the heterogeneity of the existing hardware architectures and associated APIs while not limiting control plane designers to write high performance control plane applications due to a high level and inflexible data path model.

In 2009, the OpenFlow core development team proposed a general model for mapping the virtual data path model as defined in the OpenFlow specification to hardware environments [41]. The document defines the APIs and functional building blocks for more detailed data path architecture: in principle, a data path is split into three functional layers: the highest layer is defined by a hardware-agnostic software based data path and an OpenFlow management endpoint for attaching to the control plane. At the bottom, a (vendor defined) hardware driver exposes a (potentially proprietary API) towards the agnostic parts of the network element. As a consequence, an abstraction layer binds both the hardware-agnostic OF logic and its downward interface and the hardware-specific driver to each other, thus hiding effectively any vendor-specific details. The architecture defines three reference points (APIs): (1) The *OpenFlow protocol* defines the highest abstraction for the data path and is used to expose its details towards the control plane. (2) The *OpenFlow Hardware Abstraction API* defines an agnostic interface that binds OF logic and adaptation module, and (3) the *Vendor Hardware API* actually attaches a vendor specific driver to the abstraction layer. The draft HAL API was motivated by the need to hide and protect proprietary code and APIs for hardware chip-sets and the desire to introduce a modular data path software architecture capable of replacing parts of a data path without having to rewrite significant portions of the overall system. It is worth mentioning that this document was not updated since release 0.4 in 2009 and has not been officially readdressed by one of the succeeding OpenFlow specifications published so far.

One example for specific hardware architectures is the recently published Intel communications architecture (Crystal Forest): The need for generating new fields of revenue for service providers is under discussion currently and more powerful processing capabilities are expected to help building enhanced networks with advanced services [42]. Intel addresses these needs by positioning its x86 architecture for software based network processing including a set of advanced network interface cards and IO chip-sets (e.g. Cave Creek / Intel DH89xx PCH) that have been released in Q4 2012. However, the complexity of multiple cores, parallel DMA access, NUMA architectures, efficient utilization of PCI buses, etc. builds up a significant threshold for creating high performance network applications. A Data Plane Development Kit (DPDK) [37] was released end of 2012 for network application developers to ease handling of complexity induced by the platform. The DPDK defines an environment abstraction layer (EAL) that hides all platform specific details in terms of memory management (allocation, copying) and transfer of packets between the network interface cards and the processing units. The DPDK is a typical example of a vendor specific API that may be used within a vendor-specific driver to implement the lower-level parts of a data path element.

In the preceding paragraphs we have seen the need for modular software architecture for data path elements in order to decouple hardware specific constraints from the device agnostic parts, thereby easing maintenance and evolution of a data path implementation. The data path model defined by OpenFlow binding the north- and southbound ends of the control channel remains unaltered. We have seen that forwarding and processing control of OpenFlow is largely designed along the switching chip-set APIs, forcing effectively control plane developers to program the data path in some form of low-level switching chip-set assembler (or network/SDN assembler), probably an error prone approach. The Frenetic framework introduced in [43] addresses the needs for a more descriptive, easy to use high level programming language that provides frequent programming constructs (like a difference among sets of values or negated values like all IP addresses not within a specific range, etc.). Frenetic addresses the core problems of the two-tier programming model of OpenFlow: in fact, a control plane application is a distributed application as parts of it run within the control plane and parts on data path element(s), burdening the programmer with synchronization needs among the distributed components. Frenetic's network programming language aims towards a declarative and modular design, a single-tier programming model, race-free semantics, and cost control. The latter refers to the fact, that some constructs (like a large number of wildcard related actions) causes a significant burden in terms of processing on a data path element thus affecting the element's overall performance. Cost control in Frenetic is focusing on advanced statistics queries. However, it seems that the proposed solution addresses a potential problem in the OpenFlow protocol itself (polling statistics vs. setting pre-defined rules for asynchronous notifications) that was addressed recently in OpenFlow version 1.3.1. A framework can be also defined for sending asynchronous notifications from the data path element towards the control plane based on configurable conditions like exceeding a threshold value (e.g. packet rate). Another frequently seen problem with early deployments of OpenFlow and addressed by Frenetic relates to the deficiencies of the programming model of the NOX OpenFlow controller. A NOX controller may host multiple network applications for controlling a domain of OpenFlow enabled data path elements, but does not provide an adequate model for isolating these applications from each other, thus two applications may specify contradicting commands leading to unintended state in the network. While this category of problems can be seen as a deficiency of NOX itself, coordination of multiple control instances is a common problem to SDN control planes. The SPARC [35] project addressed that problem by introducing flowspace registrations where a data path element is controlled by several controllers in parallel and multiplexing of control messages is done by each controller specifying the part of the overall flowspace it is willing to control (see SPARC's final architectural considerations in deliverable D3.3 [44] for details).

2.3 Conclusions

From the existing literature we can draw some initial conclusions:

- The abstract data model for a virtualized networking element adopted by OpenFlow limits control plane designers to write high performance network applications as it hides necessary details of the data path element. A more flexible data path model may expose a more detailed view on the hardware platform upon request by control plane designers.

- The forwarding and processing framework of OpenFlow is based on a typical switching chip-set API and as such, resembles a low-level network assembler. This low-level interface makes programming a tedious, potentially error-prone task. A high level programming language for programming entire SDN domains is desirable.

- Heterogeneity of hardware platforms has not been addressed adequately by OpenFlow yet. While defining immutable sets of processing commands is appropriate for ASIC based designs, general purpose programming environments (NetFPGA, Intel DPDK, etc.) offer far more flexibility to control plane designers, e.g. to define new actions and download these to the network element at run-time

- A modular software design for data path elements allows a separate evolution of OpenFlow management endpoint and hardware specific drivers.

In this project we plan to put specific attention to all abovementioned limitations, while designing the HAL for non-OpenFlow equipment.

3 Network and Protocol Description Languages

3.1 VXDL

Virtual Resources and Interconnection Networks Description Language (VXDL) [45] is a language used for specification and modelling of virtual resources and interconnection networks. Besides, it allows the description of end resources, including also virtual routers and timeline. The main motivation for developing this language was the lack of a similar proposal in the context of data grid applications, where both the description of resources and modelling of the interconnection network should be addressed.

Data grid applications require access to infrastructures with enough capacity for high performance data movement coordinated with computational resources. The resources available in a grid can be shared by multiple users for different computational purposes. In this context dynamic infrastructure provisioning is needed, which combines end resources and network virtualization. On one hand, several research projects have been proposed to cope with efficient and flexible resource sharing in grids. Several languages have been proposed to describe computer resources, such as ClassAd, vgDL or SWORD, all of them with different grammar, parameters and implementation particularities. On the other hand, another research projects have explored dynamic lambda path or bandwidth provisioning by end users. In networking description, for example, NDL, which is based on XML/RDF, is one of the few languages proposed, but does not include any end resource description. Standardization efforts have been done by groups like OGF NML-WG. However, both issues (end resources and network description) have not been covered simultaneously by any of them. Therefore, the modelling and specifying of the interconnection of both type of resources is still an open issue. In this context, virtualization enable to split the physical resources (computational and network resources) and to share them between multiple virtual entities.

VXDL has been proposed as a language for virtual resource interconnection network specification, which allows extending the detailed definition to all components in the virtual infrastructure. The main objective of VXDL is to integrate the network interconnection, the virtual constraints and the timeline with the description of resources.

Virtual infrastructure should be properly defined, since it is fundamental for VXDL. A virtual infrastructure is an aggregation of virtual resources interconnected by a network. The efficient decoupling of application specifications from physical resources is enabled by the virtualization layer. With regard to hardware, the abstraction layer allows the creation of multiple and isolated virtual clusters on the same physical resources at the same time. A virtual cluster is a group of machines configured for a joint purpose. When dealing with Grid computing, the resources can be spread in multiple sites interconnected over wide area links. A simple abstraction of a virtual cluster does not cope with network QoS and secure communication channels. The inability to enforce network QoS may prevent this approach being useful. In this context, the automated provisioning of lightpaths and virtual private network requests is emerging. However, VXDL tries to combine machine virtualization with network virtualization and bandwidth reservation through service overlays.

VXDL aims to specify the interconnection of virtual resources into a virtual infrastructure (Figure 3.1). The VXDL language is able to describe the following elements: (1) individual resources and groups; (2) elementary functions assigned to the resource; (3) network topology, including virtual representation of routers and links in terms of QoS metrics; (4) applications and tools needed for each components; and (5) execution timeline of an application. The grammar of VXDL is divided in: the general description of the Virtual Infrastructure, the description of each Virtual Resource, the description of the Virtual Topology and the description of the Virtual Timeline.

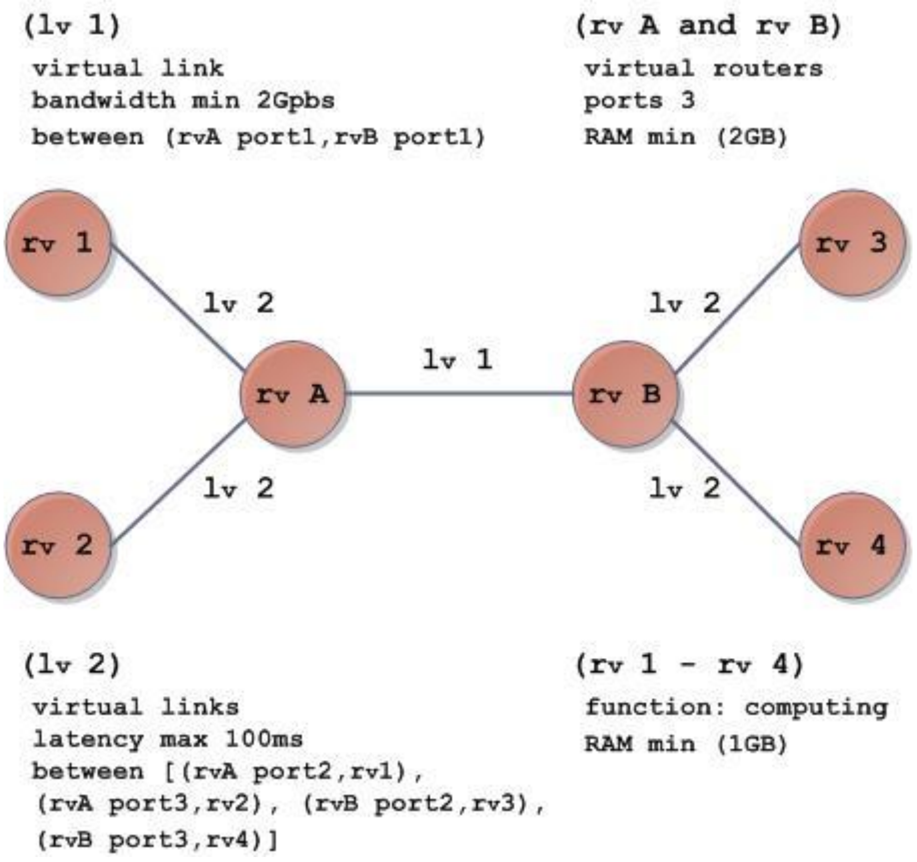


Figure 3.1: Virtual infrastructure represented by VXDL [45]

The Virtual Infrastructure general description [46] defines the general attributes of a virtual infrastructure, such as the type of infrastructure, the reservation period, the security level, the geographical location, the owner and the list of invited users that have access to the virtual infrastructure.

Regarding the Virtual Resources description, this part of the grammar allows the description of all necessary components (nodes and clusters) and the creation of groups between them. These end hosts and host groups are the vertices of a resources graph and VXDL allows their basic parameterization, such as minimum or maximum values for CPU speed and memory size. Elementary functions of a component are used to identify them, such as computing, storage, network_sensor, or router. In this context, a virtual infrastructure can be instantiated by the allocation of distributed resources, for example, specifying the physical location of the component. This enables the local dependency of specific executions.

The third part of the grammar, the Virtual Network Topology, allows the user to specify the network topology requested to connect the resources. Links define the interconnection between any virtual resource (e.g. end hosts, host groups or virtual routers). VXDL grammar defines the source and destination pair for each link. To simplify complex infrastructures, the same link can be applied to different pairs of resources. Moreover, latency and bandwidth parameters can be detailed to all links needed. The goal of detailing the network topology is to enable communication-sensitive applications, such as transmission of high data volume or low latency applications. Consequently, VXDL allows specifying both network components and topology, but there are specific details which are not covered, such as the pricing scheme or physical information about links. However, all those parameters can be detailed as external. In VXDL the topology can be represented by graphs, where the vertices represent the definitions of the components (i.e. nodes, clusters or routers) and the edges identify the links between them (i.e. the network topology).

The fourth part of the grammar, the Virtual Timeline, is used to specify the moment when the resources are needed, since any virtual infrastructure can be permanent, semi-permanent or temporary. Typically, the resources are requested for a certain period of time or time slots, and then released in order to be used by other researchers. Time slots duration depends on the specific management framework and is configured by the manager. The defined grammar allows detailing parameters such as *start*, *after* or *until* when asking for resources. Periods are delimited by temporal mark, so a period can be activated after the end of another period. This introduces a rich scenario for timeline execution.

In a nutshell, VXDL is a language for defining distributed virtual infrastructures, which makes it possible to specify a complete environment based on the proposed grammar. It covers the definition of components, the network topology and the timeline description, which are the three parts of the grammar. Although it is mainly oriented to the Grid, VXDL can be applied in other scenarios. The main contributions are the possibility to describe the timeline (basic for Grid) and the complete definition of the network topology, with rules for each link. As a consequence, the scheduling and resource utilization are improved by means of VXDL.

3.2 Network Description Language (NDL)

The Network Description Language (NDL) [47] was developed by the System and Network Engineering (SNE) research group from the University of Amsterdam (UvA) as a necessary element to cope with hybrid networks. Optical hybrid networks consist of an IP routed part and a circuit switched optical part, which is commonly known as a *lightpath*. In this type of scenarios, the end-users are able to provision lightpaths on demand through the network in order to move large amounts of data or assure a fixed quality of service in terms of bandwidth, jitter or delay. Thus, Optical Private Networks (OPNs) are dynamically created with a different topology depending on the application. Currently, most lightpaths are configured manually, which can take days or weeks. Moreover, when the lightpath involves multiple domains it could take even longer.

The NDL is an ontology based on Resource Description Framework (RDF). The main goal of RDF is the description of topologies in optical networks in a machine readable format. NDL provides a common semantic for the unambiguous communication between the application, the network and the service provider. Moreover, inter-domain network graphs can be created with different abstraction levels to assist service discovery applications and lightpath provisioning. One of the advantages of relying on RDF is that already established semantic web tools can be reused.

NDL has been developed based on the requirements imposed by two significant hybrid networks: SURFnet6 and GLIF. The former is the Dutch NREN with 6000 km of dark fibre, which uses Nortel OME6500 DWDM and TDM equipment and Avici routers. The latter is the Global Lambda Integrated Facility, which is a virtual organization of research networks and institutions whose aim is to build a worldwide networking facility for scientific research. GLIF is formed of optical exchange points, GOLEs (GLIF Open Lightpath Exchanges) and links between them. The applications can take advantage of the reconfiguration capability of the infrastructure by dynamically assigning resources for a period of time.

The integration of dynamic lightpath provisioning with the middleware used in e-Science requires the automatic topology discovery and pathfinding across multiple administrative domains. A common understanding of networking resources and their physical topology is a key requirement that NDL tries to cover. Thus, NDL provides a simple schema which presents an overview of the network and the relation between the devices. This schema can be used by applications to automatically provision lightpaths.

Prior to introducing NDL, the RDF should be described. RDF was conceived to represent information about Web resources, providing a framework for expressing metadata between applications without meaning loss. This information is expressed in triplets: (1) the subject, which is the resource to describe; (2) the property to describe; and (3) the object, which is the value of the property. A set of triplets compose a graph, and complex graphs can be created based on the fact that an object can also be the subject of another triplet. One common way of expressing RDF graphs is RDF/XML, where the graph is encoded in XML format. In RDF, the terminology issue is solved by using URIs.

Therefore, NDL can be described as a simple ontology in RDF to describe physical networks. The final schema consists of four classes and eight properties (Figure 3.2). First of all, NDL defines four classes to express the resource type: (1) *Location*, the physical place where it is located; (2) *Device*, the physical resource which is connected to the network; (3) *Interface*, used to connect devices to the network; and (4) *Link*, which is an abstract connection between two interfaces. Moreover, NDL has eight properties to define the relations between instances of the previously introduced classes: (1) *name*, the relation between a resource and its name; (2) *description*, a human-readable description of a resource; (3) *locatedAt*, the relation between resource and location; (4) *hasInterface*, the relation between devices and interfaces; (5) *connectedTo*, a physical connection between two interfaces or between an interface and a link; (6) *capacity*, the bandwidth of an interface or a link; (7) *encodingType*, which defines the encoding used on the interface or link; and (8) *encodingLabel*, which provides additional details about the encoding. All these classes and properties allow the description of networks, cables, capacities and transport types. For instance, GMPLS terminology can be used as a separate namespace to encode its values by means of *capacity*, *encodingType* and *encodingLabel* properties.

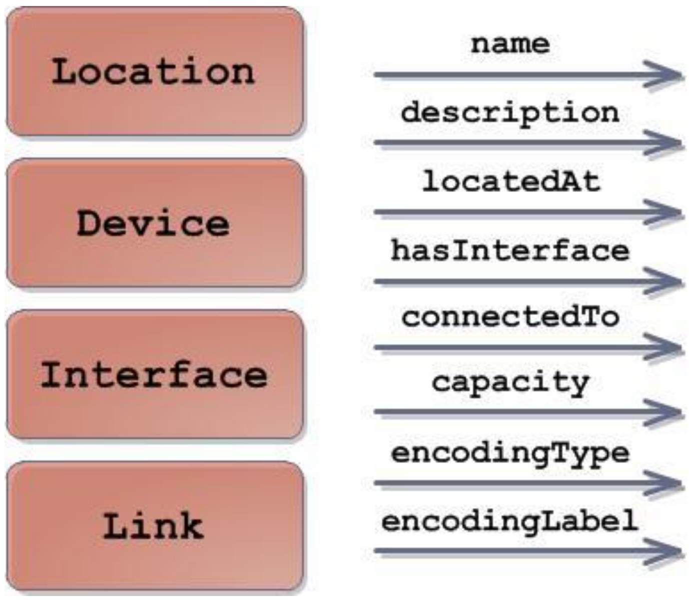


Figure 3.2: Overview of the Network Description Language [47]

Sometimes, there is no need to show the complete description of a network. In inter-domain scenarios, it is almost mandatory to share as little information as possible to avoid exposing too many details about the network. In those cases, and starting from the network description in NDL format, it is possible to extract just the required information. The W3C created a query language for RDF, SPARQL, which can be used with NDL. SPARQL is an SQL-like query language which makes use of a simple syntax to specify variables and triplet templates to retrieve information. For instance, a simple SPARQL query can be sent to automatically retrieve the connections between the devices and their names.

The Network Markup Language Working Group (NML-WG) within the Open Grid Forum (OGF) has worked to get this work standardized. Thus, the modelling effort done in NDL has largely been incorporated and expanded in the NML schemas.

Apart from solving many operational issues of hybrid networks, NDL has been tested in several other scenarios. For instance, it allows the automatic generation of network maps by means of already available web tools for RDF. In the context of GLIF, it has also enhanced the interoperability and information exchange between different domains. Finally, NDL has facilitated the creation of algorithms for lightpath finding and their later setup in SURFnet6.

3.3 INDL

The Infrastructure and Network Description Language (INDL) [48] is an improved version of NDL developed by the same research group, SNE at UVA. One of the main limitations of NDL is that it is completely oriented to describe the network infrastructure, but it does not cover the physical resources. Therefore, INDL provides a technology independent description of complete computing infrastructures, which involves both network infrastructure and resources, by means of a semantic approach. It also provides the mechanisms to describe virtualization of resources or services. Finally, INDL can be extended to describe federation of computing infrastructures.

An infrastructure modelling framework should provide virtualization and management of computing infrastructure, which includes description, discovery, modelling, layout and monitoring of resources. The INDL is designed to describe computing and cloud infrastructures in a technology independent, extensible and linkable manner. Because of this, it is based on semantic web technologies.

INDL was designed taking into account other available models to describe network and computing infrastructures, such as: Open Cloud Computing Interface (OCCI) based on UML (OGF working group); Common Information Model (CIM) which provides a detailed XML schema (developed at DMTF); Virtual resources and interconnection networks description language (VXDL) which uses XML schema; Resource Specification (RSpec) which also uses XML schemas and is used in the GENI project; or NDL-OWL also used in the GENI project and based on Semantic web.

As in NDL, INDL is based on Semantic Web paradigm, Resource Description Language (RDF) and Web Ontology Language (OWL), instead of XML schema or UML. The data is represented in triplets (*subject*, *predicate*, *object*), which provide specific information about a *subject*. OWL is used to describe the ontologies, which means that it specifies the vocabularies of triplets and the possible types of predicates. The main advantage of using OWL is that semantic graphs can be easily mapped from the computing infrastructure. Moreover, it provides a proper separation between semantics and syntax. The OWL Schema defines the ontology and semantics, whereas XML/RDF is used to define the syntax. XML/RDF uses XML to describe the RDF triplets.

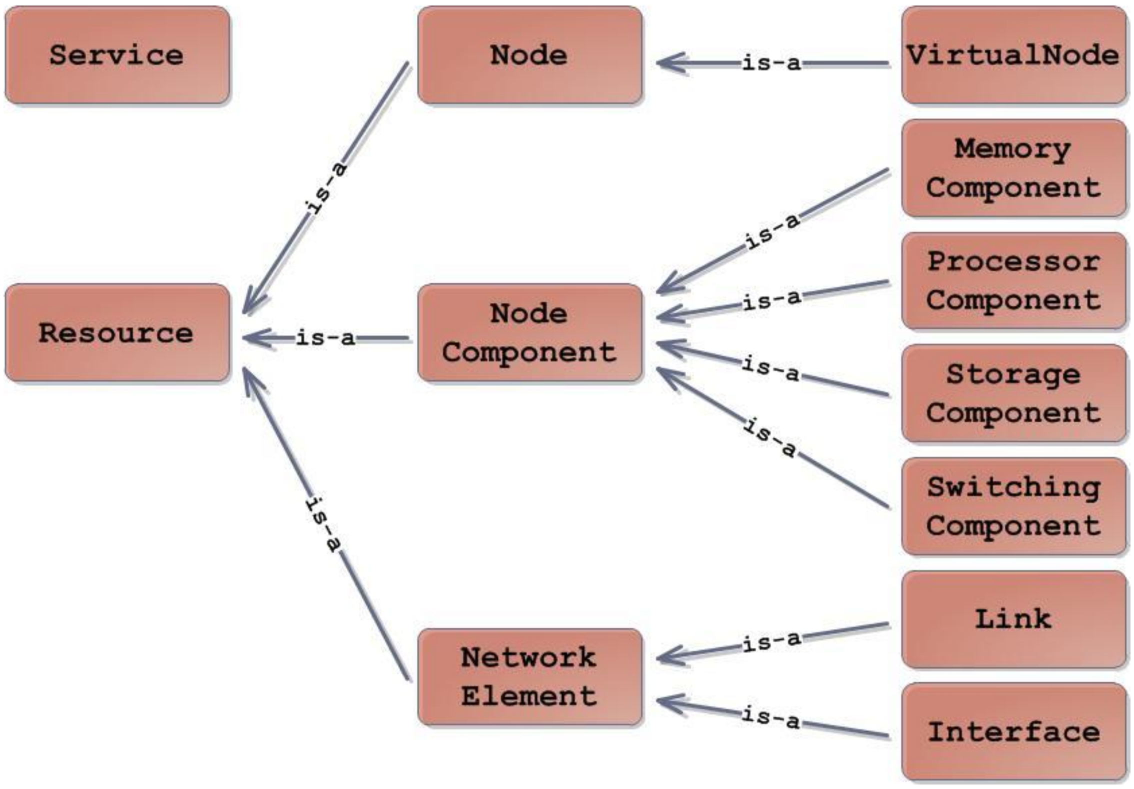


Figure 3.3: INDL Ontology [48]

Based on the requirements imposed by GEYSERS and NOVI (both are FP7 projects), INDL has improved NDL by basically adding support for IT resources and virtualization. Currently, the INDL ontology has two main classes: *Resource* and *Service*. The *Resources* are identified by a unique URI, and name and three subclasses are defined for it: *Node*, *Network Element* and *Node Component* (Figure 3.3). Regarding the *Service* class, different subclasses can be defined depending on the specific domain in which INDL is applied.

The virtualization support is modelled by introducing the *VirtualNode* element, which is a subclass of *Node*. In order to implement a virtual node, the *Node* class is used. In the end, the *Node* class can be a physical or a virtual resource, even a layer of virtualization can be defined.

A *Node Component* is part of a *Node*, and a *Node* consists of a number of *Node Components*. The *Node Component* is an abstract class which specifies the basic components of machines, such as: (1) *Memory Component* which describes the memory size; (2) *Processor Component*, which specifies the number of cores and speed of CPUs; (3) *Storage*, which defines the available space for local storage; and (4) *Switching Component*, which handles the network traffic from Interface to Interface.

The *Network* element is an abstract class to model network connectivity and has two subclasses: *Interface* and *Link*. On the one hand, the Interface class defines the point of connection of a *Node* to the network. Each node can have multiple inbound and outbound interfaces. The internal relations between inbound and outbound interfaces define the switching inside the node, and are modelled by the *switchTo* property of Interface. On the other hand, the *Link* class define an unidirectional connection between two *Nodes*. In general, the network connectivity is defined as unidirectional. Each link is connected to two Interface classes, one acting as source and the other as sink. For a bidirectional connection, two links need to be defined.

The main benefit from using INDL is that virtualization, functionality and connectivity are decoupled. This characteristic makes it easier to extend part of the ontology (e.g. defining new functionalities) without impacting the rest of the model (e.g. virtualization and connectivity). Furthermore, the ontology allows using the same connectivity and functionality models for both physical and virtual nodes.

3.4 NetPDL

Currently, most applications use their own packet description hardwired in their code. Network Protocol Description Language (NetPDL) [49] was proposed as an application-independent protocol description database, which potentially can be shared among several applications. In a nutshell, NetPDL is an extensible XML-based language for packet header description. The shared database for protocol description is just one of the potential use cases, but there are other possibilities.

Dealing with packet headers is a common task in several network applications, such as packet sniffing, firewalling, traffic analysis or packet routing. However, there is no solution to delegate the processing of packet header to a single optimized component. Each application is responsible for not just processing these headers, but also defines them in an implementation dependant fashion. A kind of universal protocol header database can benefit all these applications, which currently are using proprietary syntax to describe packet headers in the code. As a consequence, supporting a new protocol requires the generation of a new executable. NetPDL enables the creation of a shared database with all the updated description of network protocol headers.

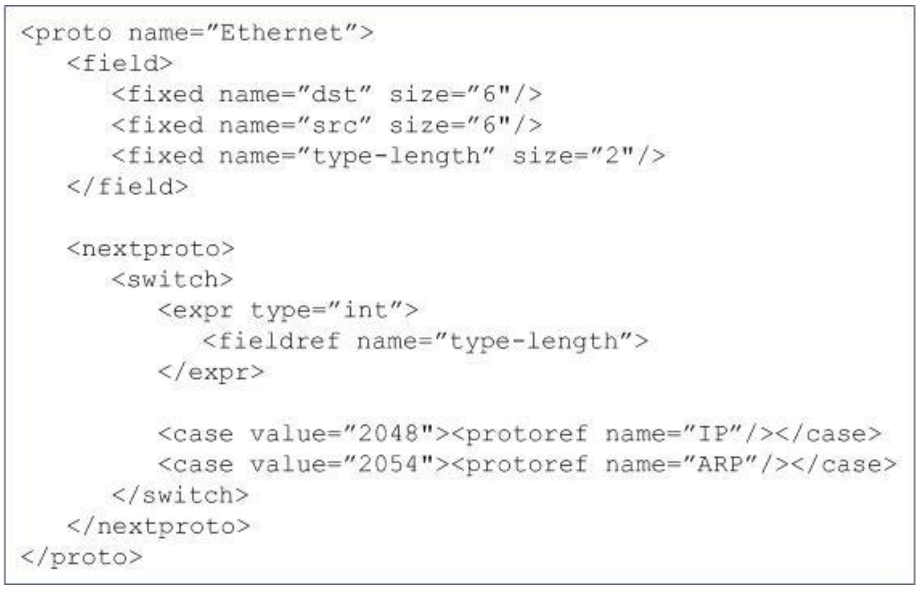
The main design objective of NetPDL is simplicity in its description of packet header formats and protocol encapsulations. Because of this, NetPDL is based on eXtensible Markup Language (XML), which is usually parsed at run-time by applications.

This allows dynamic updating of the protocol header database, which enables the transparent support of newer versions of protocol definitions.

NetPDL mainly focuses on packet header description, protocol encapsulation description and extensibility. There are other proposals used by some applications, such as Libpcap, Analyzer, FALCON, GASP, SPY, NPL, Solidum PAX Pattern Description Language, ASN.1, ACT ONE or ABNF, but none of them covers all the requirements. Most of them do not have a proper support for extensibility and they are also too complex.

Extensibility is one of the key design features of NetPDL and one of the main reasons for choosing XML as the base for building the protocol description language. New XML elements and attributes can easily be added while maintaining backward compatibility. Moreover, it can also be edited simply by basic text editors. Furthermore, XML introduces portability across platforms and the possibility enforcing strong syntactical validation with application-independent tools according to DTD and XML Schema standards.

As previously stated, the target of NetPDL is to describe packets as defined by network protocol specifications, which includes: (1) the packet format, a list of fields and their format; and (2) the protocol encapsulation, which defines the rules that determines the way the sequence of bytes constituting the payload should be interpreted. Additionally, the design objectives are: (1) Simplicity, with an intuitive syntax that can be easily understood and written using a text editor; (2) Completeness, including enough primitives to describe any packet header or the possibility to define external plug-ins; (3) Extensibility, to support the addition of new primitives and backward compatibility; and (4) Efficiency, with a performance comparable to custom code based on hardwire packet descriptions.



```
<proto name=”Ethernet”>
   <field>
      <fixed name=”dst” size=”6"/>
      <fixed name=”src” size=”6"/>
      <fixed name=”type-length” size=”2"/>
   </field>

   <nextproto>
      <switch>
         <expr type=”int”>
            <fieldref name=”type-length”>
         </expr>

         <case value=”2048"><protoref name=”IP”/></case>
         <case value=”2054"><protoref name=”ARP”/></case>
      </switch>
   </nextproto>
</proto>
```

Figure 3.4: Part of the NetPDL description of the Ethernet header [49]

The general structure of a NetPDL document starts with a *<netpdl>* tag. A set of protocol descriptions can be referenced by using a *<proto>* element. Each of these descriptions includes a group of *<fields>* tags to specify header formats and *<next-proto>* elements to define the encapsulation. The processing of a packet is done by matching the byte sequence with the elements of the description in the same order defined in the NetPDL file. When no suitable protocol description is available, there are two predefined protocols for processing the remaining data: (1) _*startproto*, for the first encapsulation; and (2) _*defaultproto*, for the last protocol.

Although the majority of header fields are defined with a fixed length, there are other fields with variable length that can be obtained only when processing the packet. NetPDL defines six basic types to categorize a field: (1) *<fixed>*, for fields with a fix length; (2) *<masked>*, which identifies the part of the header that contains the bit fields; (3) *<bit>*, for a bit field; (4) *<variable>*, for fields with a variable length that can be described by specific attributes; (5) *<line>*, for line fields ended by a carriage return or line feed; and (6) *<padding>*, which is used to align a protocol header to a 16 or 32 bit limit. In general, a field can be completely characterized by its length, its position and the number of occurrences. However, the latter two attributes are not always needed.

In order to obtain an advanced description of packet headers, some additional elements are defined: (1) field blocks; (2) conditional elements; (3) expressions; (4) repeating a field; (5) lookahead operands; and (6) custom plugins. First of all, the *<block>* element is used to improve readability by tagging a portion of code, similar to a macro expression. The *<includeblk>* tag is used to reference certain *<block>* content. Therefore, when an *<includeblk>* tag is found, it is replaced by the contents included in the *<block>* element. This capacity is used to isolate a portion of NetPDL code and adds compactness, since the same block can be reused several times. The second type of elements, the *<switch>*-*<case>* and *<if>* elements allow the conditional inclusion of a field or a value depending on the value of another field. The former introduces the capacity to define multiple alternative descriptions, whereas the latter element enables the capacity to include some fields depending on the evaluation of an arbitrarily complex condition. The *<expr>* element can be used to define the condition. Regarding the third type of elements, mathematical, logical and string expressions are possible in NetPDL descriptions to improve the conditional elements. Native XML structures are used to define these expressions. Concerning the fourth type of elements, the *<loop>* element allows the repetition of a field or group of fields, with different possibilities: size-bounded loop (depending on a given value), occurrence-bounded loop (depending on the number of times), while-bounded loop (depending on a condition), and do-bounded loop (same as while-bounded loop but at least once is present). The *<loopctrl>* element permits to restart or interrupt a corresponding repetition (similar to break and continue instructions). The fifth type of elements, the *lookahead* operands allow to look at some bytes before evaluating certain expressions. Finally, the *<plugin>* element enables to delegate the handling of some protocol header feature to external code. This is very useful when dealing with complex structures.

As previously mentioned, one of the main objectives of NetPDL is to describe the protocol encapsulation, which defines how to interpret the following bytes. Protocol encapsulation is handled by the *<nextproto>* element. Usually, the encapsulation is based on the value of one or more header fields, so the relationship between this value and the corresponding encapsulated protocol should be specify. The *<protoref>* element is used exactly for defining this relationship. Sometimes, *<switch>*-*<case>* or *<if>* elements are used to evaluate a condition before assigning the correct encapsulated protocol. In other cases, further processing is needed the proper identification is made by evaluating a value of a field within their own header. For this reason, the *<presentif>* element supports to define a condition while processing the packet header. The use of a version field is very typical in these cases.

Finally, NetPDL allows the declaration of variables, which can be manipulated at run time. Depending on the validity and scope of a variable, there are different categories of variables. On the one hand, a variable can be *volatile* is it is only valid while processing one packet, or *static* if its value is preserved through different packets. On the other hand, a variable can be *local* if it is only valid when processing fields of a single protocol header, or *global* if the variable is valid while processing the entire packet. Therefore, four categories are possible: *local-volatile*, *local-static*, *global-volatile*, and *global-static*. A number of predefined global variables are present in each NetPDL engine, such as the link-layer type, the total number of bytes for processing, the number of bytes already processed or a timestamp. The user is able to create their own variables.

One of the main improvements of NetPDL is the extensibility of the language. New attributes and elements, including their validity range, can be added very easily while maintaining backwards compatibility. One example of this extensibility is the NetPDL Visualization Extension, which has been designed to support protocol analysers. It provides additional information needed for displaying the packet fields. Two types of views are defined: a *summary view* and a *detailed view*. Although XSL Transformations can be used, this implies to learn a new programming language to handle visualization. By contrast, NetPDL Visualization Extension defines only a few visualization primitives. All the relevant visualization elements are defined through the <*showtemplate*> *element*, and the most important attributes are: (1) *showtype*, which determines the format of each byte (hexadecimal, ascii, binary or decimal); (2) *showgrp*, which specifies how the bytes must be grouped; and (3) *showsep*, which defines the separator string between the groups. The <*showdtl*> element can be used to describe more sophisticated displaying rules. Next, the <*showmap*> element compares the value of a field against a set of choices to define the displaying format. Additionally, a set of primitives are defined for creating a summary view of each packet.

In summary, NetPDL is an extensible XML-based language for describing the format of protocol headers. New protocols can be easily specified with the basic NetPDL primitives already defined. Moreover, these primitives can be also easily extended to cope with specific needs. The syntax is easy to understood and an easy to implement parsers thanks to a large number of XML tools and libraries.

3.5 Conclusions

This section presents some of the most relevant network and protocol description languages related to the work defined in the ALIEN project. Next, all these languages are summarized and compared.

The VXDL is a language that includes both network and end resources description. It is also useful when dealing with virtual infrastructures, since it was originally designed for Grid computing. Due to its origins in the Grid, the time-based sharing of resources is supported by the language, which is able to describe the execution timeline.

The NDL is a very simple language in XML/RDF useful to describe the network infrastructure and its topology, but it does not support the description of end resources. It was designed to cope with optical hybrid networks; therefore it supports the description of optical circuit switched resources. One of the main strengths of NDL is the set of parsers (since it is based on XML) and tools available to process the data. For instance, these tools can generate different views of the same resources very easily and just expose part of the internals of the network.

One of the main limitations of NDL is the lack of support for describing end resources. The INDL was designed to overcome this limitation. It is based on Semantic Web paradigm and Web Ontology Language (OWL). In a nutshell, INDL adds the computing infrastructure and virtualization to the NDL.

The NetPDL is quite different from the previous languages, since it is a language to describe new protocols (e.g. headers) instead of network infrastructure or computational resources. NetPDL is useful to have a common language to describe protocols and also as a known mechanism to extend those protocols. This ability could be useful to describe new extensions to the OpenFlow protocol (e.g. matching).

There are several areas in which these languages can be useful for the ALIEN project, such as the description of the network infrastructure or the time-based sharing of resources. Regarding the description of network resources, there are two separate targets in which this capacity would be needed. On the one hand, the design of the Hardware Abstraction Layer

(HAL) could benefit from it when dealing with the emulation of a set of interconnected network elements as a single resource. The description of the physical connectivity would be a valuable input to automate this process, mainly when the auto-discovery is not an option. On the other hand, the description of the topology can be also relevant for the CCN application in order to take decisions based on this description, use different paths and avoid loops. Initially, NDL will be explored to support the description of the network resources. Eventually, INDL could be used to add computational resources to this description.

Regarding the time-based sharing of resources, the VXDL ability to describe this type of sharing in the context of Grid computing could be explored when designing the integration of ALIEN resources under the OFELIA Control Framework (OCF) in a time-based manner. The direct adoption of this language for this task could not be the best option due to compatibility issues that could arise with the current OCF development. However, VXDL could be a reference when designing this integration based on time slots.

To conclude, the integration of these languages in the ALIEN project will depend on the specific necessities and restrictions of each module.

4 Intermediate Representation Languages

The “intermediate representation” is the term from computer science. It is a language or data structure used during internal steps of computer program compilation. The intermediate representation allows for computer program analysis, optimization and transformations to another shape (e.g.: machine code). It could be some graph data structure like AST (Abstract Syntax Tree) or DAG (Direct Acyclic Graphs), which will represent logical dependencies between various parts of program. There could be also used, a dedicated low-level intermediate language for representing an input program. Such intermediate language is composed of instructions representing basic operations, doesn’t contain control flow information and operates on unlimited number of variables/registers. Sometimes very popular programming language with good performance become an intermediate language for implementing a new languages (many modern high-level languages like Python, Ruby, Perl, Haskell, etc are implemented in C).

4.1 Hardware Description Language (HDL)

The perfect definition of Hardware Description Languages can be found in Wikipedia [50]. It states: "*in electronics, a hardware description language or HDL is any language from a class of computer languages, specification languages, or modeling languages for formal description and design of electronic circuits, and most commonly, digital logic. It describes the circuit's functionality and operation, its design and organization, and tests to verify its operation by means of simulation*".

There are two basic ways to describe digital circuits [51], [52], [53], [54]:

- behavioral - the designer defines the relationship between outputs and inputs of the system, and the system implementation is realized by a specialized compiler;
- functional - the designer defines the functional blocks and the relationships between them. The functional blocks can be very simple, such as elementary logic functions, as well as complex, such as memory, registers, adders or even more complicated.

HDL allows various optimizations, such as minimization of functions, simplifying the circuit by the use of prefabricated elements, designed to test systems and their functional simulation.

The history of hardware description languages dates back to the 1950s, it became necessary to automate certain parts of design processes. HDL language first developed in the United States, leading to ANSI standards, and later international standards such as those from IEEE.

The hardware description languages are very convenient, because they allow description of hardware without reliance on a particular technology. Actually there exist more than 20 different hardware description languages, but the most popular are VHDL and Verilog.

4.1.1 VHDL

An early version of Very High Speed Integrated Circuits Hardware Description Language (VHDL) was developed at the U.S Department of Defence in the 1980s. The first IEEE standard for this language comes from 1987 [55]. After many changes and updates, the current version is defined in IEEE 1076-2008 [56]. Currently the language is used for the definition of ASIC chips and mainly programming FPGA circuits. In preparing FPGA designs [57], various design environments can be used (for example Xilinx ISE, Altera Quantas or Mentor Graphics HDL Designer). These IDEs translate VHDL code into an RTL (Resistor-Transistor Logic) [58] schematic of the desired circuit. Such a prepared schematic can be verified with simulation tools. These tools can generate input signals for functionality to be tested and present the waveform at different stages for the designed circuit. The designer can observe the behaviour of particular blocks and accept or redesign the project. The final stage of preparation usable version of VHDL project is the translation of VHDL model into physical structure of particular FPGA chip, i.e. into its internal gates and wires. The execution of a project consists of downloading prepared files onto chip using a dedicated interface. Figure 4.1 shows an example of VHDL code for a simple bidirectional counter [59]. Typical functionality of hardware languages allows to define many concurrent activities. The code in Figure 4.1 describes only one, it can be run simultaneously with many others.

```
entity counter is
  Port ( CLOCK : in  STD_LOGIC;
    DIRECTION : in  STD_LOGIC;
    COUNT_OUT : out  STD_LOGIC_VECTOR (3 downto 0));
end counter;

architecture Behavioral of counter is
signal count_int : std_logic_vector(3 downto 0) := "0000";
begin
process (CLOCK)
begin
  if CLOCK='1' and CLOCK'event then
    if DIRECTION='1' then
      count_int <= count_int + 1;
    else
      count_int <= count_int - 1;
    end if;
  end if;
end process;
COUNT_OUT <= count_int;
end Behavioral;
```

Figure 4.1: Example of VHDL code (code for simple counter)

The main advantage of VHDL is that it allows the behaviour of the required system to be described and verified (simulated with dedicated tools) before synthesis tools translate the design into real hardware (gates and wires). It allows debugging of the system in software before a hardware implementation takes place.

4.1.2 Verilog

Verilog is standardized as IEEE 1364. There are several implementations of Verilog (SystemVerilog, Verilog 2001, Verilog 2005 [53], [54]). Theirs functionality is similar to VHDL. The newest Verilog standard comes from 2009. In its usage, this language is the same as in case of VHDL. The majority of tools can work with both languages and it depends on the particular designer which language will be used for a particular module. In some cases, the blocks of code written in VHDL and Verlilog can be mixed on a single chip. The set of keywords in Verilog is similar (but not identical) to the analogous set for VHDL. The same functionality can be obtained with both languages but in a different way. The differences are visible especially in features for simulations. The interesting fact about both languages is that not all functionality can be implemented in hardware - some sections are used only in the development process. These sections allow debugging in a more effective and convenient way. When a file is prepared for downloading to a physical device, many restrictions are checked. For example, the longest signal paths and the clocking (frequency conditions) are analyzed.

The Figure 4.2 shows example code for a bidirectional counter written in Verilog code [59]. The same functionality is realized by code from the Figure 4.1.

```
module counter(CLOCK, DIRECTION, COUNT_OUT);
input CLOCK;
input DIRECTION;
output [3:0] COUNT_OUT;
);

reg [3:0] count_int = 0;
  always @(posedge CLOCK)
    if (DIRECTION)
      count_int <= count_int + 1;
    else
      count_int <= count_int - 1;

assign COUNT_OUT = count_int;
endmodule
```

Figure 4.2: Example of Verilog code (code for simple counter)

4.2 LLVM

LLVM (formerly the Low Level Virtual Machine) is a framework and infrastructure [LLVM] for the code compilation, containing a set of modular tool-chain components and libraries (e.g., assemblers, compilers, debuggers, optimizers, code generators debuggers, linkers etc.). The LLVM compiler infrastructure, thanks to well-defined interfaces, provides reusable and replaceable modules for building custom compilers and, in consequence, reducing the time and cost of providing of a compilation environment for any programming language. Now, LLVM is used as a common infrastructure to implement a broad variety of statically and runtime compiled languages (e.g. C, C++, Java, .NET, Python, Ruby, Scheme, Haskell, D, as well as countless lesser known languages).

LLVM was designed as an alternative to monolithic compilers such as GCC, where there is no way to reuse pieces, and there is very little sharing across language implementation projects. LLVM follows three-phase design pattern for compilers [60], where major components of the compiler are the following:

- Frontend – parses source code, applies preprocessing, makes lexical and syntactic analysis, checks variable types, reports code errors and builds an intermediate representation (IR) to represent the input code.

- Optimizer (middle-end) – transforms an intermediate representation many times for purpose of the code optimization (removal of unreachable code, propagation of constant values, eliminating redundant computations) in order to improve the code's running time.

- Backend – translates an intermediate representation into the machine code (e.g.: replace generic IR instructions with target instructions, assigns registers for the program variables) with usage of specific features of the target CPU architecture like special instructions or parallel execution.

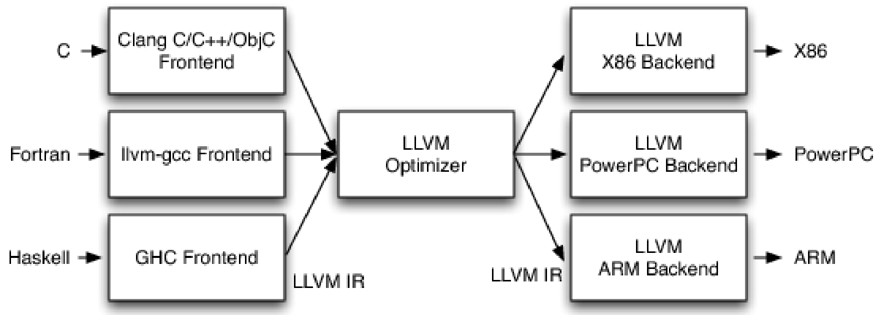


Figure 4.3: LLVM's implementation of the Three-Phase Design [61]

The compiler decomposition into the frontend (see Figure 4.3), the middle-end and the backend means that a developer of a new language can focus only on providing a good implementation of a new language fronted and reuse optimizers and backends that already exist for many target platforms (e.g.: x86, ARM, PowerPC, SPARC, etc). Additionally, if a developer implements a backend that provides a faster code execution for a given platform, then this work can be easily adapted for all already existing languages within the LLVM framework.

A modular design of LLVM components with usage of a single IR representation, exchanged between LLVM components, gives additional opportunities for users. The user has the freedom to decide how LLVM components will be used, i.e. what is order of LLVM components within the compiler tool-chain. As an example, in Figure 4.4 install-time optimization use case [61] is presented. Install-time optimization is the idea of delaying code generation even later than link time in order to find out the specifics of the targeted device and applying the best possible optimizations to generated machine codes. Within this use-case, we can see that LLVM optimizers can be used in many places of a compilation process and the compilation process itself can be easily adapted to the complex scenario (i.e. usage of different languages in a project with usage of compilation-time, link-time and install-time optimizations).

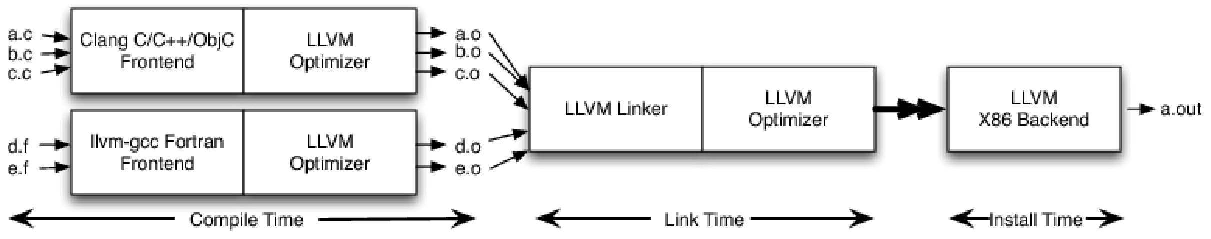


Figure 4.4: Install-time optimization with usage of the LLVM infrastructure [61]

The LLVM framework is built around a well-documented intermediate representation (IR) of code [62]. LLVM defines a common, low-level IR code representation in Static Single Assignment (SSA) form, with several novel features. LLVM contains a simple, language-independent type-system that exposes the primitives commonly used to implement high-level languages features like a mechanism for implementing the exception handling. The LLVM IR representation describes a program using an abstract RISC-like instruction set but with key higher level information for effective analysis. The LLVM code representation is designed to be used in three different forms: as an in-memory compiler IR, as an on-disk bit-code representation (suitable for fast loading by a Just-In-Time compiler), and as a human readable assembly language representation. All three forms of LLVM IR are equivalent. By providing type information, LLVM IR can be used as the target of optimizations.

```
; file: helloworld.s

; External declaration of the puts function
declare i32 @puts (i8*)

; Declare the string constant as a global constant
@global_str = constant [13 x i8] c"Hello World!\00"

; Definition of main function
define i32 @main() {

    ; Convert array [13 x i8] to i8's pointer
    %temp = getelementptr [13 x i8]* @global_str, i64 0, i64 0

    ; Call puts function to write out the string to stdout
    call i32 @puts(i8* %temp)

    ret i32 0
}
```

Figure 4.5: Hello World example of IR human readable notation [63]

In Figure 4.5, we see an IR code example which prints “Hello World!” message (language tutorials always start with such example). The IR code [62] is similar to assembly languages because it contains low-level instructions and memory model that is only slightly richer than in a standard assembler. In the LLVM IR code, we can put comments (begins with `;` character), defines global identifiers (begins with `@` character) and local identifiers (begins with `%` character). Any variables and constants have strict typing. The integer type is declared as `iN` (e.g.: i8 – 8-bit integer, i32 – 32-bit integer, etc). There are also derived variables types like:

- arrays (e.g. `[40 x i32]` – array of forty 32-bit integer values),
- vectors (e.g. `<8 x float>` – vector of eight 32-bit floating-point values),

- pointers (e.g. `[4 x i32]*` – a pointer to array of four i32 values),
- structs (e.g. `{ i32, i8, float }` – a structure of 32-bit integer, 8-bit integer and float value),
- and functions (e.g. `{i32, i32} (i32)` – a function taking an i32, returning a structure containing two i32 values).

The LLVM instruction set captures the key operations of ordinary processors but avoids machine-specific constraints such as physical registers, pipelines, and low-level calling conventions. The LLVM IR declares the following rich set of instructions [62]:

- termination instructions (e.g. `ret`, `br`, `switch`, `invoke`, `resume`, etc),
- binary operations (e.g. `add`, `fadd`, `sub`, `fsub`, `mul`, `fmul`, `udiv`, `sdiv`, `fdiv`, etc),
- bitwise binary operations (e.g. `shl`, `lshr`, `ashl`, `and`, `or`, `xor`),
- vector operations (e.g. `extractelement`, `insertelement`, `shufflevector`),
- memory operations (e.g. `alloca`, `load`, `store`, `fence`, `getelementptr`, etc),
- conversion operations (e.g. `trunk … to`, `zext .. to`, `pfext .. to`, `ptrtoint .. to`, `bitcast … to`, etc),
- and others.

The LLVM IR code can be compiled and executed using LLI tool from the LLVM infrastructure (see Figure 4.6).

```
Host# lli helloworld.s
Hello World!
```

Figure 4.6: Compilation and execution of LLVM IR "Hello World" example with usage of LLI tool [63]

The LLVM IR has no direct notion of high-level constructs such as classes, inheritance, or exception handling semantics and only provides primitives to implement those features of supported languages. LLVM also does not specify a runtime system or a particular object model. The LLVM is used instead to the implementation of runtime systems for higher level languages. LLVM does not guarantee type safety, memory safety, or language interoperability and this way LLVM cannot be a direct intermediate IR form of high-level languages.

4.3 CUDA

CUDA (Compute Unified Device Architecture) is a parallel computing platform and programming model invented by NVIDIA. [64] This architecture is used on all NVIDIA hardware to harness the power of the graphics processing unit (GPU) and hence increase computing performance. To simplify programming the CUDA platform is provided with CUDA-accelerated libraries, and extensions to standard programming languages, including C, C++, Fortran, JAVA and Python. The programmers write source codes in one of these programming languages and then compile them with the provided CUDA compiler (for C language it is ‘nvcc’). In Figure 4.7 you can see the NVIDIA’s CUDA architecture.

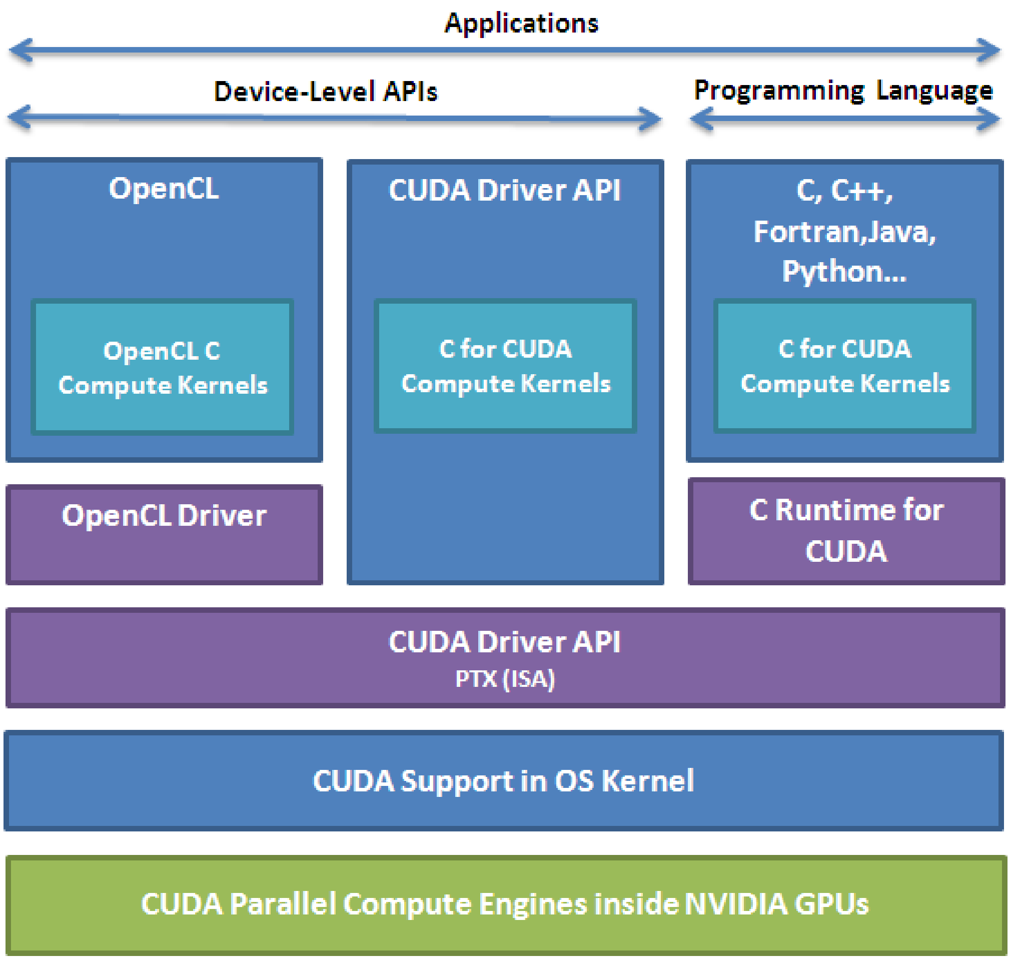

Applications
Device-Level APIs
Programming Language
OpenCL
OpenCL C Compute Kernels
CUDA Driver API
C for CUDA Compute Kernels
C, C++, Fortran,Java, Python...
C for CUDA Compute Kernels
OpenCL Driver
C Runtime for CUDA
CUDA Driver API
PTX (ISA)
CUDA Support in OS Kernel
CUDA Parallel Compute Engines inside NVIDIA GPUs

Figure 4.7: NVIDIA’s CUDA architecture [65]

The most interesting feature of the CUDA architecture in the context of the Intermediate representation languages is the PTX (Parallel Threading Execution). PTX is a low-level, parallel thread execution virtual machine and instruction set architecture (ISA). PTX exposes the GPU as a data-parallel computing device.

PTX programs are a collection of text source modules (files). PTX source modules have an assembly-language style syntax with instruction operation codes and operands. Pseudo-operations specify symbol and addressing management. The ptxas (optimizing backend compiler) optimizes and assembles PTX source modules to produce corresponding binary object files.

Source PTX modules are ASCII text. Lines are separated by the newline character (*n*). All whitespace characters are equivalent; whitespace is ignored except for its use in separating tokens in the language. The C preprocessor cpp may be used to process PTX source modules. Lines beginning with # are preprocessor directives. The following are common preprocessor directives:

```
#include, #define, #if, #ifdef, #else, #endif, #line, #file
```

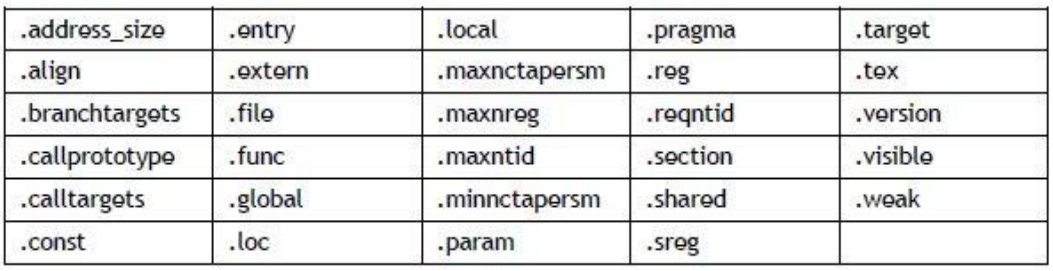
A PTX statement is either a directive or an instruction. Statements begin with an optional label and end with a semicolon. An simple example of code using PTX is presented in Figure 4.8 [66]:

```
        .reg     .b32 r1, r2;
        .global  .f32  array[N];

start:  mov.b32   r1, %tid.x;
        shl.b32   r1, r1, 2;          // shift thread id by 2 bits
        ld.global.b32 r2, array[r1];  // thread[tid] gets array[tid]
        add.f32   r2, r2, 0.5;        // add 1/2
```

Figure 4.8: Listing of example PTX code [66]

Directive keywords begin with a dot, so no conflict is possible with user-defined identifiers. The directives in PTX are listed in Figure 4.9 [65].

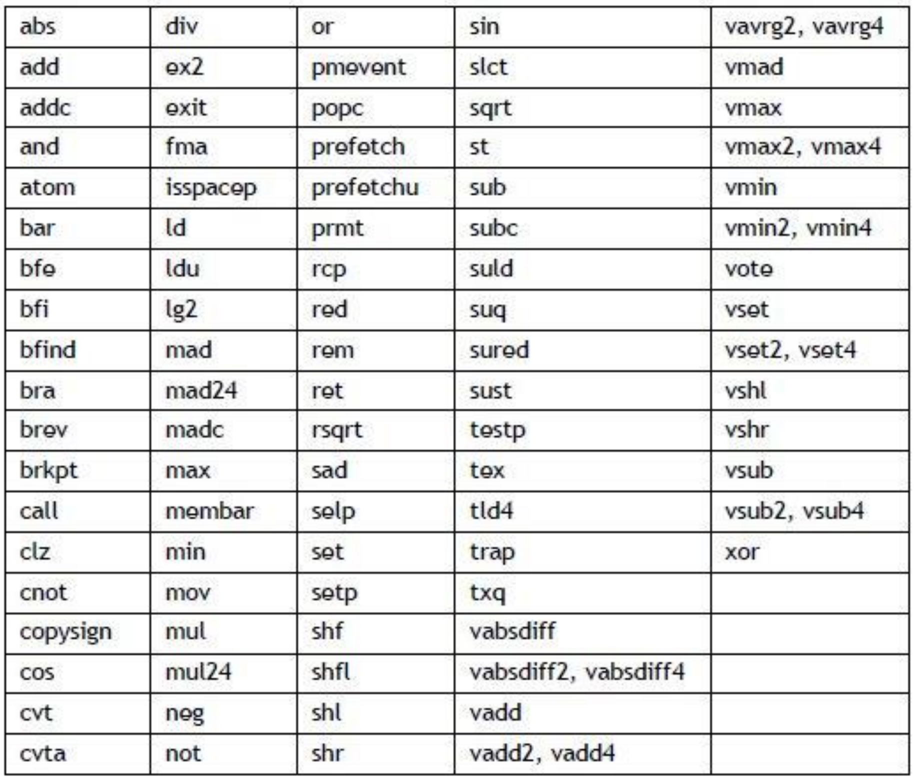


.address_size	.entry	.local	.pragma	.target
.align	.extern	.maxnctapersm	.reg	.tex
.branchtargets	.file	.maxnreg	.reqntid	.version
.callprototype	.func	.maxntid	.section	.visible
.calltargets	.global	.minnctapersm	.shared	.weak
.const	.loc	.param	.sreg	

Figure 4.9: PTX directives [65]

Instructions are formed from an instruction opcode followed by a comma-separated list of zero or more operands, and terminated with a semicolon. Operands may be register variables, constant expressions, address expressions, or label names. Instructions have an optional guard predicate which controls conditional execution. The guard predicate follows the optional label and precedes the opcode, and is written as *@p*, where p is a predicate register. The guard predicate may be optionally negated, written as *@!p*. The destination operand is first, followed by source operands.

Instruction keywords are listed in Figure 4.10.



abs	div	or	sin	vavrg2, vavrg4
add	ex2	pmevent	slct	vmad
addc	exit	popc	sqrt	vmax
and	fma	prefetch	st	vmax2, vmax4
atom	isspacep	prefetchu	sub	vmin
bar	ld	prmt	subc	vmin2, vmin4
bfe	ldu	rcp	suld	vote
bfi	lg2	red	suq	vset
bfind	mad	rem	sured	vset2, vset4
bra	mad24	ret	sust	vshl
brev	madc	rsqrt	testp	vshr
brkpt	max	sad	tex	vsub
call	membar	selp	tld4	vsub2, vsub4
clz	min	set	trap	xor
cnot	mov	setp	txq	
copysign	mul	shf	vabsdiff	
cos	mul24	shfl	vabsdiff2, vabsdiff4	
cvt	neg	shl	vadd	
cvta	not	shr	vadd2, vadd4	

Figure 4.10: PTX instruction keywords [65]

With the existence of multiple NVIDIA GPU architectures, it is not always known at compile time on what type of GPU the application will run. The importance of this issue is directly proportional to the number of different GPUs. NVCC, together with the CUDA runtime system, provides three mechanisms for dealing with this:

1. Storing more than one generated code instance embedded in the executable.

2. Allowing PTX intermediate representations as generated code.

3. Maintaining device code repositories external to the executable, in directory trees, or in zip files.

More than one compiled code instance for the same device code occurring in the CUDA source allows the CUDA runtime system to select an instance that is compatible with the current GPU, which is the GPU on which the runtime system is about to launch the code. If more than one compatible code instances are found, then the runtime system can select the ‘most appropriate’, and in case the most appropriate code instance is still PTX intermediate code, the runtime system may decide to compile it for the current GPU. PTX intermediate code is especially useful for distributed libraries.

External code repositories allow fine tuning as more of the compilation environment becomes known: because such repositories are directory trees in an open format (normal directory or zip format), any PTX code that it contains can be ‘hand- compiled’ after distribution. One particular way of such fine tuning is to use runtime compilation while enabling a device code translation cache: this will result in a new code repository, or it will extend an existing one [66].

CUDA also allows 'Just-in-Time Compilation' of the PTX code. Any PTX code loaded by an application at runtime is compiled further to binary code by the device driver. This is called just-in-time compilation. Just-In-Time compilation increases application load time, but allows the application to benefit from any new compiler improvements coming with each new device driver. It is also the only way for applications to run on devices that did not exist at the time the application was compiled, as detailed in Application Compatibility [65].

When the device driver just-in-time compiles some PTX code for some application, it automatically caches a copy of the generated binary code in order to avoid repeating the compilation in subsequent invocations of the application. The cache – referred to as compute cache – is automatically invalidated when the device driver is upgraded, so that applications can benefit from the improvements in the new Just-In-Time compiler built into the device driver.

4.4 OpenCL

Open Computing Language (OpenCL) is a computing framework for heterogeneous platforms proposed by Apple as a solution for general-purpose computing on graphics processing units (GPGPU). OpenCL provides developers with a unified interface for computation on central processing units (CPUs), graphics processing units (GPUs), digital signal processors (DSPs) and other processors. OpenCL programming language is based on C99 [67] and provides low level access to various computing resources (more low level than CUDA).

The Platform model for OpenCL (see Figure 4.11) consist of a host connected to one or more OpenCL devices (e.g. CPU or GPU). An OpenCL device is divided into one or more compute units (CU) (e.g. single core in CPU or GPU) which are further divided into one or more processing elements (PE) (i.e. virtual scalar processors).

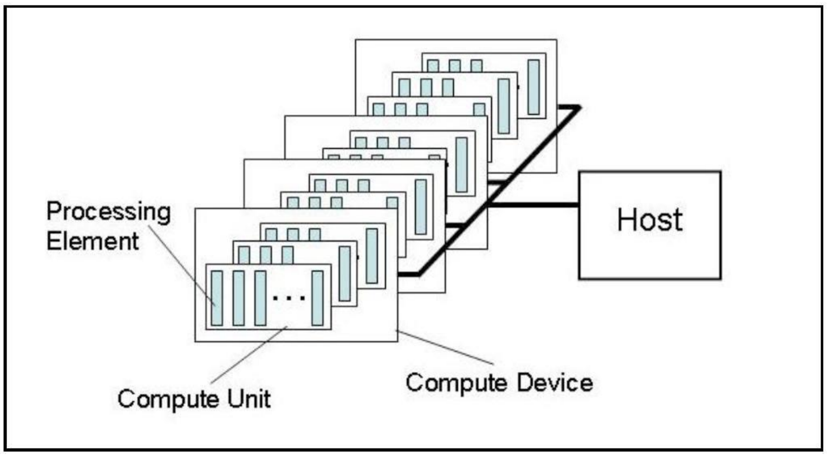


Figure 4.11: OpenCL platform model

Execution of an OpenCL program occurs in two parts: kernels that execute on one or more OpenCL devices and a host program that executes on the host. The host program defines the context for the kernels and manages their execution [68]. A kernel is a function declared in a program and executed on an OpenCL device. Example of simple kernel code is presented on Figure 4.12. Kernels are dynamically compiled from intermediate representation into native code with JIT compiler.

```
__kernel void dot_product (__global const float4 *a,
                           __global const float4 *b,
                           __global float4 *c)
{
    Int gid = get_global_id(0);
    C[gid] = a[gid] + b[gid];
}
```

Figure 4.12: Listing of example kernel code [69]

The host program is responsible for configuring environment, preparing kernels, configuring CUs and PEs, setting kernel arguments and starting its execution. Example of a host program is presented on Figure 4.13.

```
// create the OpenCL context on a GPU device
context = clCreateContextFromType(0, CL_DEVICE_TYPE_GPU, NULL, NULL, NULL);

// get the list of GPU devices associated with context
clGetContextInfo(context, CL_CONTEXT_DEVICES, 0, NULL, &cb);
devices = malloc(cb);
clGetContextInfo(context, CL_CONTEXT_DEVICES, cb, devices, NULL);

// create a command-queue
cmd_queue= clCreateCommandQueue(context, devices[0], 0, NULL);
free(devices);

// allocate the buffer memory objects
memobjs[0] = clCreateBuffer(context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
                            sizeof(cl_float4) * n, srcA, NULL);
memobjs[1] = clCreateBuffer(context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
                            sizeof(cl_float4) * n, srcB, NULL);
memobjs[2] = clCreateBuffer(context, CL_MEM_READ_WRITE, sizeof(cl_float) * n, NULL, NULL);

// create the program
program = clCreateProgramWithSource(context, 1, (const char**)&program_source, NULL, NULL);

// build the program
err = clBuildProgram(program, 0, NULL, NULL, NULL, NULL);

// create the kernel
kernel = clCreateKernel(program, "dot_product", NULL);

// set the args values
err = clSetKernelArg(kernel, 0, sizeof(cl_mem), (void *) &memobjs[0]);
err |= clSetKernelArg(kernel, 1, sizeof(cl_mem), (void *) &memobjs[1]);
err |= clSetKernelArg(kernel, 2, sizeof(cl_mem), (void *) &memobjs[2]);

// set work-item dimensions
global_work_size[0] = n;
local_work_size[0]= 1;

// execute kernel
err = clEnqueueNDRangeKernel(cmd_queue, kernel, 1, NULL, global_work_size, local_work_size, 0,
NULL, NULL);
```

Figure 4.13: Listing of example host program [69]

OpenCL currently offers portability only at the source code level because specification does not cover implementation aspects. Each hardware vendor provides its own implementation of the OpenCL framework and thus there is no standard intermediate representation for OpenCL. From the three biggest vendors, two (AMD and Intel) are using a LLVM IR in its implementations and one (nVidia) is using its own PTX language. The Khronos Group which is responsible for maintaining the development of OpenCL specifications intends to change this situation by creating the OpenCL Standard Portable Intermediate Representation (SPIR). The SPIR will be based on LLVM IR because many of the OpenCL vendors already base their technology on LLVM and this makes LLVM IR the de facto OpenCL IR and a natural candidate for SPIR.

4.5 JVM

The Java Platform [70] acts as a buffer between a running Java program and the underlying hardware and operating system. Java programs are compiled to run on a Java Virtual Machine (JVM). The virtual machine runs the program and the JVM mechanisms gives the program access to the underlying computer's resources. No matter where a Java program goes, it need only interact with the Java Platform. It needn't worry about the underlying hardware and operating system [71]. Java platform execute a program in three phases (see Figure 4.14):

- first Javac compiling the source code into bytecode,
- secondly passing the bytecode to the virtual machine,
- finally the JVM executes the bytecode as machine code

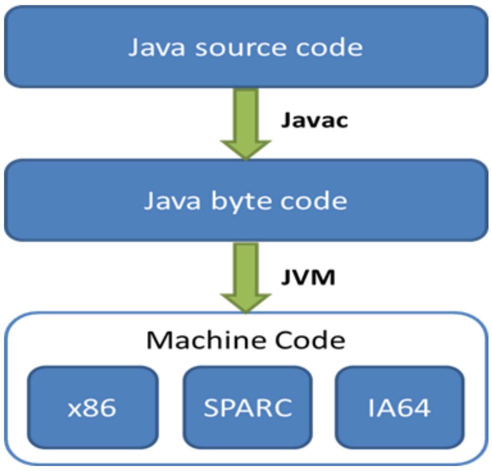


Figure 4.14: Java runtime environment

Because the Java programming language is platform independent so are the ranges and behaviour of its primitive types inserted in the language [70]. In C or C++ languages, the range of the primitive type 'int' is determined by its size, and its size is determined by the target platform. The size of an 'int' in C or C++ is generally chosen by the compiler to match the word size of the platform for which the program is compiled. This means that a C++ program might have different behaviour when compiled for different platforms because the ranges of the primitive types are not consistent across the platforms. For example an 'int' in Java behaves as a signed 32-bit two's complement number, a 'float' is the 32-bit IEEE 754 floating point standard and no depend on underlying platform. This consistency is also reflected in the internals of the Java virtual machine, which has primitive data types that match those of the language. It's guarantee that primitive types behave the same on all platforms, the Java language itself promotes the platform independence of Java programs [71].

The Java virtual machine [72] is a component of the Java technology responsible for hardware and operating system independence. The Java virtual machine is an abstract computing machine which has an instruction set and use various memory resources during execution. JVM work on binary files. File with extension .class, which is a compiled Java program to bytecode, is an input for JVM. The class file itself is a binary file that cannot be understood by a human. Binary machine code is an output for the platform that the proceeding JVM was prepared. Please see an example of Java code and bytecode in Table 4.1. Java bytecode is platform-independent code. It is executable on the hardware where the JVM has been installed. The size of the compiled code (output from 'Javac') is of a similar size to the source code. It's make easy to transfer and execute the compiled code via the network [71].

Java program	Java bytecode and mnemonics

`class Test {`
 	
`    public static void doSth() {`
`        int i = 0;`
`        for (;;) {`
`            i += 1;`
`            i *= 2;`
`        }`
`    }`
`}`
	0 iconst_0 // 03
	1 istore_0 // 3b
	2 iinc 0, 1 // 84 00 01
	5 iload_0 // 1a
	6 iconst_2 // 05
	7 imul // 68
	8 istore_0 // 3b
	9 goto 2 // a7 ff f9

Table 4.1: Java code and bytecode example [71]

The role of the JVM is illustrated on Figure 4.15, The class loader subsystem is a mechanism for loading types (classes and interfaces) given fully qualified names. The class loader subsystem is responsible for more than just locating and importing the binary data for classes. It must also verify the correctness of imported classes, allocate and initialize memory for class variables, and assist in the resolution of symbolic references. Each Java virtual machine also has an execution engine, a mechanism responsible for executing the instructions contained in the methods of loaded classes. The behavior of the execution engine is defined in terms of an instruction set. For each instruction, the specification describes in detail what an implementation should do when it encounters the instruction. There is no specification for the implementation of JVM execution engines. The implementations can interpret, just-in-time compile, execute natively in silicon, use a combination of these, or use some other new technique. Concrete implementations may use a variety of techniques, are either software, hardware, or a combination of both. A runtime instance of an execution engine is a thread. When a Java virtual machine runs a program, it needs memory to store the bytecodes and other information it extracted from loaded class files, objects, parameters to methods, return values and local variables. The Java virtual machine organizes the memory to execute a program into several runtime data areas [71].

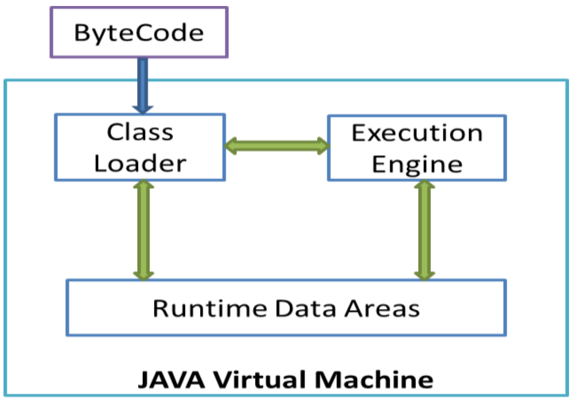


Figure 4.15: View of Java Virtual Machine

The JIT compiler (see Figure 4.16) converts the bytecode to native code that is executed directly by Operating System and optimizes it. An adaptive optimizing virtual machine begins by interpreting all code and monitors the execution of that code. Most programs spend up to 80 percent of their executing time in 20 percent of the code [70], [71]. This kind of code is called program's “hot spot”. By monitoring the program execution, the virtual machine can figure out which methods represent the program's "hot spot" [73]. JIT compiler in JVM compiles most frequently used parts of bytecode to native code upon first execution, then during re-calls of “hot spot” bytecode JVM executes it from the native code [71].

A JIT compiler takes more time to compile the code than the interpreter to interpret the code as it runs. Therefore, if the code is to be executed just once, it is better to interpret it instead of compiling. The JVMs that use the JIT compiler internally check how frequently the method is executed and compile the method only when the frequency is higher than a certain level [74]. JIT use intermediate representation of bytecode and usually optimize it before native code will be generated, see Figure 4.16.

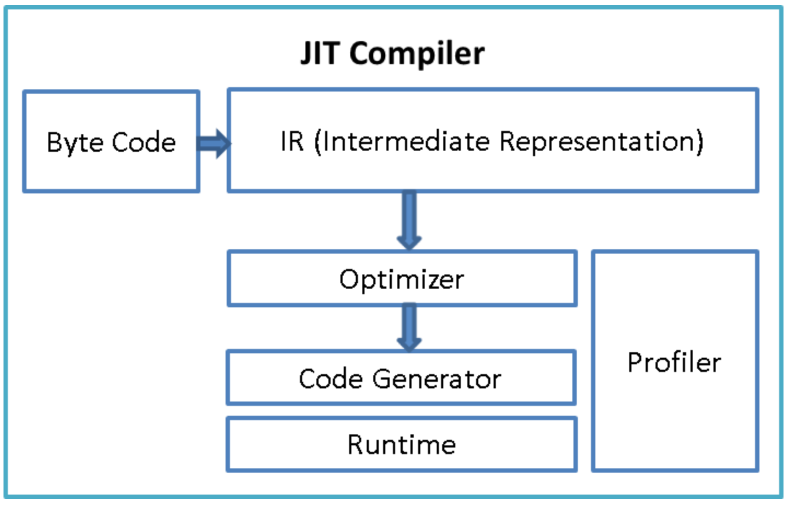


Figure 4.16: Compilation Just In Time blocks

JIT does optimization through the following mechanism [74]:

- Inline methods - instead of calling method on an instance of the object it copies the method to caller code.
- Replace interface with direct method calls for method implemented only once to eliminate calling of virtual functions overhead
- Join adjacent synchronized blocks on the same object
- Eliminate dead code
- Drop memory write for non-volatile variables
- Remove pre-checking *NullPointerException* and *IndexOutOfBoundsException*

4.6 PacketC

The packetC language [75] was developed by CloudShield Technologies, Inc. [76], in partnership with multiple partners worldwide including the US government, federal system integrators, telecommunication system providers and independent software vendors. The language is open for the implementation on numerous platforms in order to develop a common standard for developing network application. PacketC ensures high-performance parallel processing, secure code, network-centric processing and is designed to be used with a runtime environment that provides parallel processing [75], [77].

PacketC has been designed to maximize application reliability and security through object-oriented features, improved error handling, and strict typing. This language uses the C99 variant of the C language [67] for operators (arithmetic, logical and bitwise), conditional statements and overall syntax. The grammar of the packet language is similar to C and deviates only when is relevant to the specific problem domain. A simple example of packetC code to deny pings on a network is presented in Figure 4.17.

```
packet module DENY_PING;
#include <cloudshield.ph>
void main($PIB pib)
{
      if ( pib.l3Type == L3TYPE_IPV4 && pib.l4Type == L4TYPE_ICMP )
          pib.action = DROP_PACKET;
}
```

Figure 4.17: Deny pings in packetC language [75]

PacketC frees network application developers from the mechanics of managing parallel processes. Also abstracting the details of specialized devices increases the resulting applications' portability [78]. The relationship between the parallel Packet Processing model, the packetC language and hardware platforms is depicted on Figure 4.18.

The packetC language is designed to be **compiled into optimized bytecodes** that are executed by a packetC native processor (e.g. Cloudshield's DPPM processors [76]) or by an appropriate Virtual Machine (VM) [77]. A bytecode output for the packetC virtual machine allows for heterogeneous hardware platforms to execute the code in a predictable manner. The approach employed by the packetC system follows the approach familiar to Java p-code. The virtual machine is like a lightweight bytecode virtualization layer found in emulated embedded systems or Java programs. The bytecode representation can employ the specialized instruction required and leave the implementation to packetC native processor or a virtual machine providing an equivalent implementation. The packetC code is assumed to be executed in a runtime environment that either provides or emulates:

- Arithmetic and logical operations for unsigned integer with sizes of 8, 16, 32 and 64 bits,
- Structures in which the fields that are declared first are stored at lower addresses,
- Multiple-byte integers stored in big endian order with the most significant byte stored in the lowest number addresses,
- Little-endian bit fields with bytes stored in big-endian order,
- Management of packet receipt, buffering and transmission,
- Basic packet structure interpretation and underlying functions for IP packet cleanup,

Fundamental primitives for structured and unstructured content analysis to support database and search set expectations.

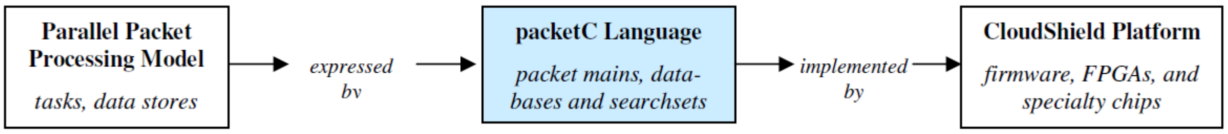

Parallel Packet Processing Model
tasks, data stores
expressed by
packetC Language
packet mains, data-bases and searchsets
implemented by
CloudShield Platform
firmware, FPGAs, and specialty chips

Figure 4.18: Relationships: among the parallel processing model, packetC language, hardware platform

The elements mentioned above may be provided by a hardware platform, operating system, packetC virtual machine environment or the compiler itself. The packetC system provides these capabilities such that code does not change from one platform to another.

The language proposed by Duncan and Jungck lets users develop parallel networking applications by writing a single short program rather than a collection of low-level tasks. PacketC adds the following data type extensions [79]:

- **Descriptors** – *struct* that describes a network protocol and is mapped to whatever location within a packet that contains that protocol structure,
- **Databases** – structure aggregates divided into *data* and *mask* for matching against packet content,
- **Searchsets** – aggregates of strings or regular expressions to match against packet content,
- **References** – provide capability for chaining together successive databases and searchsets operations that are contingent on a previous operation's result.

These data types and operations on them can be implemented by general-purpose or specialized processors. Figure 4.19 shows how to use virtual machine to isolate processor specifics. A high-level packetC source code is translated by a compiler tool-chain into **a virtual machine bytecode that is an intermediate representation**. After that operation, interpreters transform bytecode into chip-specific commands for variety of processors. This approach allows developers to focus on network matters rather than machine details so knowledge of what processors types are in the system is required only of the interpreter implementers [79].

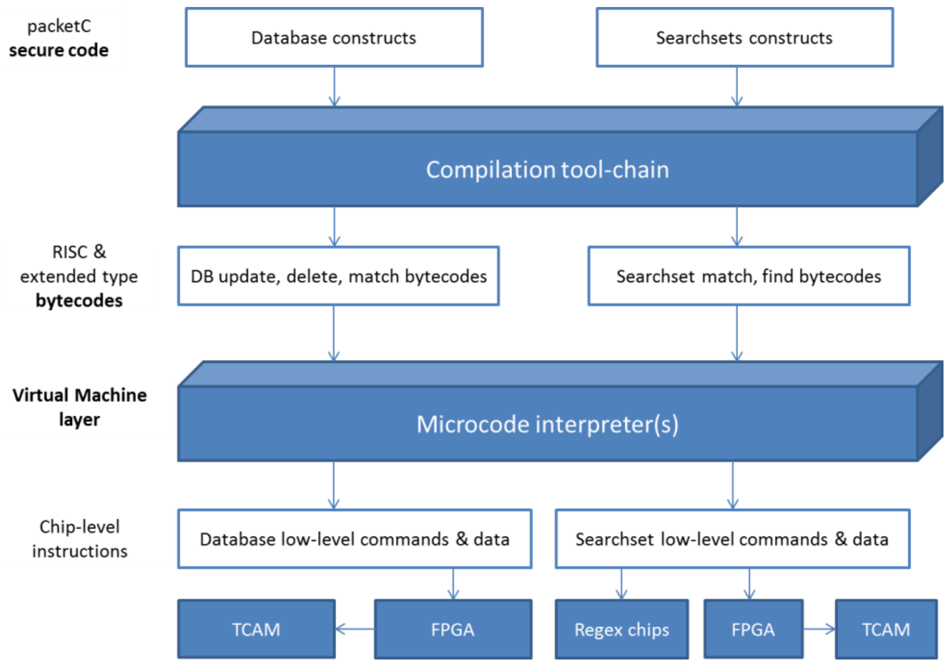


Figure 4.19: Using interpreted virtual machine instructions to hide and isolate machine control for heterogeneous processors [79]

4.7 Conclusions

As presented above Intermediate Representation Language (IRL) is a form of language used by an abstraction layer to generate a machine code for the target hardware. The IRL consists of platform-independent instruction sets that lies between high-level and low-level program languages. In a computer science domain intermediate representation languages provide a portability of code between different hardware architectures.

An abstracted view of datapath entities towards the control plane is used by the OpenFlow protocol to control the behavior of network elements. A common data model of the device hides the complexity of internal structures of data processing entities. In OpenFlow such device abstraction is provided by flow tables and actions (pipeline in OF above v.1.2). The IRL, as described above, hides the complexity of underlying hardware in a similar way. The IRL with its abstracted machine ensures that a common language (e.g. bytecode in Java) is transferred into the hardware specific machine code. The IRL can be supported then on any device that implements the abstraction layer.

The analogy of the IRL's abstraction data model and the OpenFlow protocol abstraction could be used during the Hardware Abstraction Layer (HAL) design. The OpenFlow device abstraction can be used as a common abstraction for alien hardware and a base for the HAL design.

5 Network Virtualization

5.1 Definition of Network Virtualization and Existing Approaches

Although the concept of virtualizing networks has been in IT literature for many years and it has been used in the industry, recently the shortcomings of the current Internet architecture to support new protocols and services has focused significant attention on network virtualization (NV) as a solution to this problem. While network virtualization can be seen as a tool to evaluate new network architectures, it can also be seen as one of the fundamental attributes of future Internet. In any case, network virtualization has been the first choice to test any new network technology and architecture before making it available for public. This has led network virtualization to be seen as a breakthrough in networking [80].

A definition of network virtualization which could encompass conventional and unconventional virtual networks is [81]: *"Network Virtualization is any form of partitioning or combining a set of network resources, and presenting (abstracting) it to users such that each user, through its set of the partitioned or combined resources has a unique, separate view of the network. Resources can be fundamental (nodes, links) or derived (topologies), and can be virtualized recursively. Node and link virtualization involve resource partition/combination/abstraction; and topology virtualization involves new address (another fundamental resource we have identified) spaces."*

While network virtualization is still evolving, and there are many projects in research communities around the world investigating new NV mechanisms, Software Defined Networking has recently gained a lot of attention. Several commercial SDN products are available to introduce network virtualization. To draw a better picture, the next sections will review conventional approaches in academia and SDN approaches in industry toward network virtualization.

5.1.1 Academic Oriented

As the Internet received huge interest and gained rapid growth in thirty years, new services have been added to it over the time which has led it to be a complex architecture to manage and maintain. All these new services have been designed on top the TCP/IP protocol suite which has not been designed to serve these services when it was created. In the networking community many have debated whether the current state of the Internet is about to be ossified and it is facing serious challenges in terms of scalability and security and network virtualization could therefore be a natural solution to deploy testbeds as virtualized infrastructure to investigate and overcome Internet obstacles [80], [82].

Among recent existing major activities and research projects, European projects (OFELIA, GEYSERS, NOVI, MANTICORE, FEDERICA) and the US project VINI consider addressing network infrastructure virtualization at a national and/or international level with National Research and Education Network (NREN) connectivity in some of them.

In the following sections, a brief review is provided to give a general idea of conventional approaches towards network virtualization in research communities. While it might seem that all projects have similar aims and objectives at first glance, the following points could give one a basis to evaluate and differentiate these projects in virtualization context:

- architecture overview,
- industry oriented.

5.1.1.1 *Architecture Overview*

FEDERICA [83]: dedicated “slices” of network infrastructure are provided which can be used simultaneously by different group with various control granularities and no disruption. These slices are created by a unique combination of virtualization techniques and network control mechanisms. Each slice gives the ability to run virtual overlay networks (L2 and L3) down to the lowest possible network layer. Slices, as part of the FEDERICA infrastructure, are built on top of existing NREN networks. By building each slice, a set of virtual nodes, e.g., switches, routers and machines are implemented in the FEDERICA infrastructure and interconnected by virtual Ethernet links. The slice could be accessed either directly or via a gateway through the Internet [84].

MANTICORE [85]: Following the infrastructure as a service (IaaS) paradigm, MANTYCORE enables NRENs and other e-infrastructure providers to represent their infrastructure resources (routers, switches, optical devices and IP networks) as a service to virtual research groups. It offers remote access and control of infrastructure resources to clients via web services. A marketplace represents all the resources and services between providers and customers in a unified virtual resource pool platform which facilitates network resources and features advertisement [86].

GEYSERS [87]: The aim of this project is to define and implement a new photonic network architecture with the capability of optical network and IT resource provisioning for network operators which will deliver end-to-end services. It adopts the IaaS and service-oriented concepts to enable infrastructure provisioning flexibility while at the same time separates the functional aspects of every entity involved in the converged service provisioning by enabling a layer-based structure in its architecture. Following the layered architecture, the Network Control Plane (NCP) layer is responsible for dynamic network connectivity provisioning and also control/management of the logical network infrastructure. Logical Infrastructure Composition Layer (LICL) partitions the physical infrastructure and Service Middleware Layer in an intermediate layer between client’s applications and NCP offering converged services [88].

NOVI [89]: Resources belonging to various levels, i.e. networking, storage and processing are in principle managed by separate yet interworking providers. NOVI will concentrate on methods, algorithms and information systems that will enable users to work within enriched isolated slices, baskets of virtualised resources and services provided by federated infrastructures. It will investigate federation data, control, monitoring and provisioning planes of constituent FI infrastructures. NOVI proposes and tests resource description data models and abstraction algorithms, incorporating Semantic Web concepts in order to give the user the ability to efficiently identify and correlate virtual resources. NOVI has Control and Management plans which offer an API to present control and management services. By providing an API for control and management, NOVI will try to enhance federation approaches to facilitate slice control and management within federation of heterogeneous virtualized infrastructure [90].

GENI [91]: It provides an environment for investigating and evaluating new network architectures and protocols. In overview, it consists of three levels: a physical substrate which represents GENI's physical resources (routers, links and switches), user services which represents the services available for users for their experiments and the GENI Management Core (GMC) which is a framework to bind user services with underlying physical infrastructure.

VINI [92]: project VINI creates a virtual network infrastructure allowing researchers to deploy and evaluate their protocols and services in its test-bed. VINI is associated with PlanetLab network and adds layer 2 virtual networks to PlanetLab slices which means each virtual machine in PlanetLab is connected with Ethernet point-to-point virtual links. Leveraging from real routing software, traffic loads and network events, researchers have the ability to design and implement their experiments on a shared physical infrastructure provided by VINI project [93].

A taxonomy of network virtualization mechanisms is presented in Table 5.1.

	Layer 3 virtualization	Layer 2 virtualization	Layer 1 virtualization
FEDERICA	By leveraging Juno's virtualization in L2/L3 switches and software router [84]	By leveraging Juno's virtualization in L2/L3 switches and VMware ESXi virtual switches [84]	Not supported
MANTICORE	By means of Web Services it provides an IP interface for every user [86]	By means of Web Services it provides an interface for L2 [86]	Not supported
GEYSERS	Not supported	Not supported	By creating an abstraction of heterogeneous network resources, a virtual optical node is created which could be either a partition of a single optical node or aggregation of multiple nodes [88]
GENI	By implementing Packet System Processing (PPS) in Programmable Core Node (PCN) which includes a high-speed programmable device supporting multiple virtual routers [91]	By implementing Circuit Processing System (CPS) in PCN containing collection of circuit-oriented components [91]	By implementing Circuit Processing System (CPS) in PCN containing collection of circuit-oriented components [91]
VINI	By using vNet software module which creates a form of raw IP with isolated traffic for each slice [93]	By emulation of L2 functionality over IP [93]	Not supported
NOVI	By implementing NSwitch in each domain. The NSwitch is developed by NOVI [94]	By implementing NSwitch in each domain. The NSwitch is developed by NOVI [94]	Not supported

Table 5.1: Taxonomy of network virtualization mechanisms

5.1.1.2 *Industry Oriented*

Although there are many commercial approaches and technologies to create virtual networks, we review here those approaches which are related to Software Defined Networking.

Cloud computing is an emerging IT trend that brings several benefits such as reduced run time and response time, minimized risk of deploying physical infrastructure and increased pace of innovation. Although network virtualization has been used in cloud computing, it could not keep pace of innovation as fast as compute and storage virtualization and still remains as a barrier for more innovation in cloud computing services [95].

In a cloud data centre where thousands of Virtual Machines (VM) communicate at layer 2, VLAN tagging is used to create logical segmentations and virtual networks. However, this approach is not scalable because of limitation in maximum number of VLAN tags (only 4094). It also leads to cloud infrastructure segmentation when it is expanded between several remote locations. To overcome this impasse a solution is to have an overlay layer 2 (L2) network on layer 3 (L3) network to prevent logical L2 segmentation [96].

In order to address these issues and make virtual networks more flexible and easier for administration and maintenance, a few network protocols such as STT [97], VXLAN [98] and VNGRE [95] have been introduced to facilitate overlay L2 on L3 networks and have been used in commercial products.

Nicira's [99] approach toward network virtualization in data centre is to decouple network services from underlying physical network hardware, as happens in server virtualization, which provides an environment for creation of agile, virtual networks [100]. Stateless Transport Tunnelling Protocol for network virtualization (STT) was introduced by Nicira Inc. STT in its core was designed to provide all the high performance feature that is available in the NIC in tunnelling mode, while keeping the flexibility of software for network virtualization function [97].

Another Software Defined Networking approach was introduced by Big Switch [25]. Inspired by OpenFlow, Big Switch has produced its own product, Big Network Controller. This controller resides on top of both OpenFlow hypervisor switches and physical switches and provides a unified abstraction of underlying network represented by an API. This API gives the ability to dynamically and automatically control the network and supports any data centre network protocol [26].

Vyatta [101] pursues another approach by creating SDN compatible virtual routers, firewall and VPNs to connect remote L2 SDN networks to each other. These virtual entities can be operated as virtual machines and could be replicated and located where needed. This will give flexibility to the users to provision, control and maintain their virtual network [102].

Midokura [103] is another company which has different vision of virtual networks and SDN in the cloud. Using the Open vSwitch kernel module, Midokura creates a distributed packet processing engine with L2-L4 forwarding capability. By running an agent as a control plane on each virtual network configuration, the flow state is built on the local control plane and since it can process packets, load balancing and firewall policies can be deployed without forwarding the packet to another machine.

5.2 Network Virtualization in OpenFlow Networks

5.2.1 FlowVisor-Based

5.2.1.1 *FlowVisor*

At present, FlowVisor [104] is the most popular SDN based implementation to instantiate virtual networks. FlowVisor is a special purpose OpenFlow controller which acts as a virtualization layer between the OpenFlow-enabled switches and multiple controllers. In the network architecture, FlowVisor is placed between the OpenFlow network and the controllers acting as a transparent proxy (see Figure 5.1). With this approach, FlowVisor "slices" the OpenFlow network by intercepting all the OpenFlow-protocol messages sent by the switches to the controllers and forwarding them to the correct controller accordingly to pre-defined policies (this process is called network slicing). FlowVisor allows the configuration of several logical topologies (also called slices) which links and nodes are a subset of the physical topology. FlowVisor generally hosts multiple guest controllers, one controller per slice (see Figure 5.1) and it ensures that a controller can observe and control its own slice only, while isolating one slice from another.

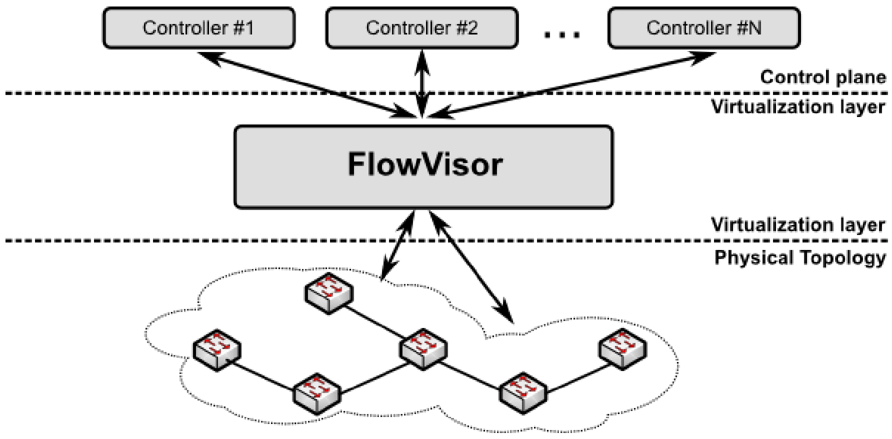


Figure 5.1: FlowVisor is logically placed between the physical network and the slice controllers

As stated above, FlowVisor logically sits between the physical network and the controllers and intercepts all the messages exchanged between switches and controllers through the control channel. These messages contain a structure with fields which include (among others): in port (port of the switch on which the packet was received) and 128 bytes of the flow header.

FlowVisor intercepts these messages and sends them to the correct controller on the basis of predefined policies. These policies are defined by using the headers of flows, the switch identifier (also called datapath_id) and the input port.

For instance, a slice can be defined as the set of OpenFlow messages coming from specified subsets of switches and containing flow headers with destination and source ip addresses belonging to the subnet 192.168.0.0/24.

Although a switch can belong to multiple slices, each slice only has control over its own flows. When a controller tries to control flows outside its flowspace, FlowVisor can either modify the control message or simply reject that message.

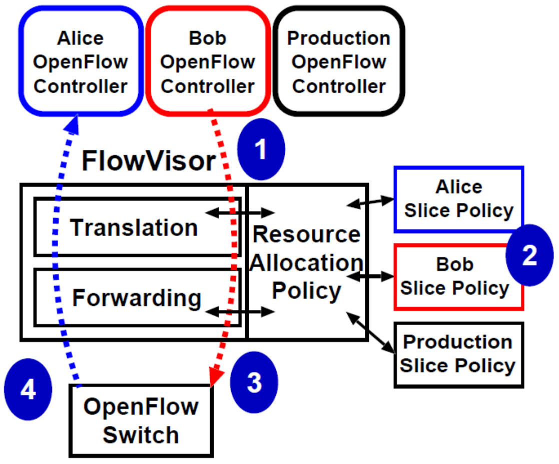


Figure 5.2: FlowVisor workflow [80]

Figure 5.2 depicts the workflow for messages going from a controller to an OpenFlow switch and from a switch to a controller. In step (1) of the figure, the message is intercepted by FlowVisor which, accordingly to the user's policies (2), rewrites or rejects the message and finally sends it to the switch (3). Messages from switches (4) are only forwarded to the controller with the matching slice policies.

Along with the flowspace isolation mentioned above (achieved by rewriting the rules in the messages), FlowVisor also provides bandwidth isolation among slices by marking the VLAN priority bits of the flow headers.

Furthermore, FlowVisor only proxies connections to a controller for switches that are included in the virtual topology of the controller and rewrites messages of these connections to only report to the controller the switch ports included in the virtual topology.

FlowVisor also limits the number of flow entries that a slice can install on the network and ensures that the limit is never exceeded.

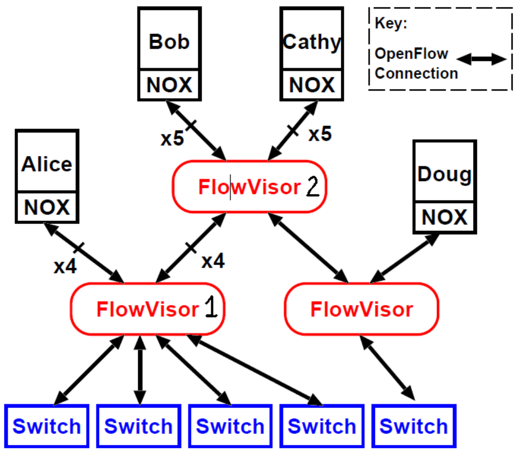


Figure 5.3: FlowVisor can recursively alice a virtual slice [80]

Another interesting feature of FlowVisor is its ability to recursively slice a virtual slice (see Figure 5.3). For instance, all the messages containing flow headers with source and destination IP address of subnet 192.168.0.0/24 are sent from FlowVisor 1 to FlowVisor 2 (see Figure 5.3). Then FlowVisor 2 could send the subset of messages with TCP destination port 80 to the Bob's NOX controller while all the other messages to the Cathy's controller.

FlowVisor has been successfully applied as a Network Virtualization layer in several wired and wireless scenarios. FlowVisor has been applied using different network technologies in several different deployment experiments including: GENI [91] and OFELIA [105].

5.2.1.2 *Optical FlowVisor*

Combining optical network virtualization technologies and SDN enables service providers to efficiently and dynamically reconfigure their network and support new services on demand with less manual intervention, hardware deployment and configuration. Inspired by the FlowVisor architecture, an OpenFlow-based packet switching over optical network convergence providing network virtualization (Optical FlowVisor) has been proposed.

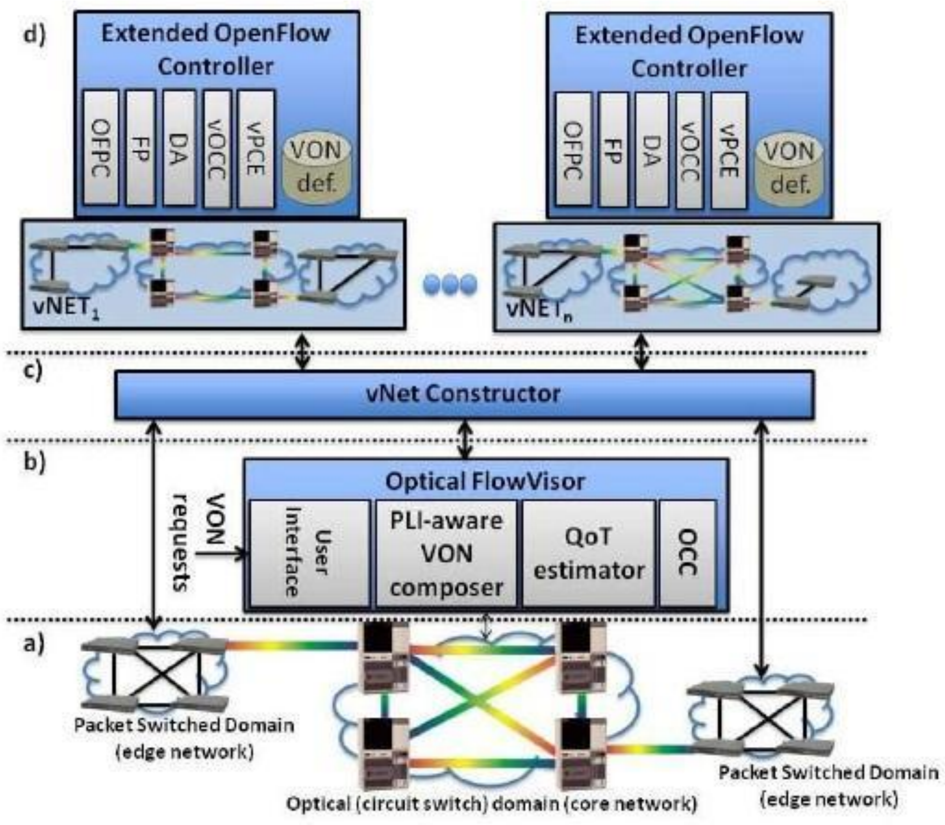

d)
Extended OpenFlow Controller
OFPC
FP
DA
vOCC
vPCE
VON def.
Extended OpenFlow Controller
OFPC
FP
DA
vOCC
vPCE
VON def.
vNET1
vNETn
c)
vNet Constructor
b)
VON requests
Optical FlowVisor
User Interface
PLI-aware VON composer
QoT estimator
OCC
a)
Packet Switched Domain (edge network)
Optical (circuit switch) domain (core network)
Packet Switched Domain (edge network)

Figure 5.4: The architecture of Optical FlowVisor [106]

A virtual optical network (VON) consists of set of virtual optical switches interconnected by virtual optical links. The proposed architecture has the following elements: a) an optical physical infrastructure, b) Optical FlowVisor (OFV) which is an access proxy between virtual networks and the physical infrastructure, c) vNet constructor responsible for both configuring virtual network and corresponding extended OpenFlow controller, d) extended OpenFlow control layer responsible for controlling virtual network.

The OFV user interface handles requests received by network operators which specify virtual network topologies and generates VONs utilizing available physical resources. Inside OFV, Optical Connection Controller provides access to VON to the physical infrastructure and controls cross connections in the optical switched network.

OVF's output is a VON description which is handed to the vNet constructor to create and configure the virtual network and the corresponding extended OpenFlow controller [106].

5.2.1.3 *VeRTIGO*

As highlighted by the authors in [107], one of the major limitations of FlowVisor is the inability to establish virtual topologies completely independent from the underpinning physical topology. As a consequence, FlowVisor is unable to provide researchers flexibility in designing their experiments with arbitrary network topologies on a defined physical infrastructure.

VeRTIGO [108], a software architecture built on top of FlowVisor, provides the functionalities to overcome the above-mentioned FlowVisor's limitation by allowing the instantiation of generalized virtual topologies in an OpenFlow network through the implementation of virtual links as aggregation of multiple physical links and OpenFlow-enabled switches.

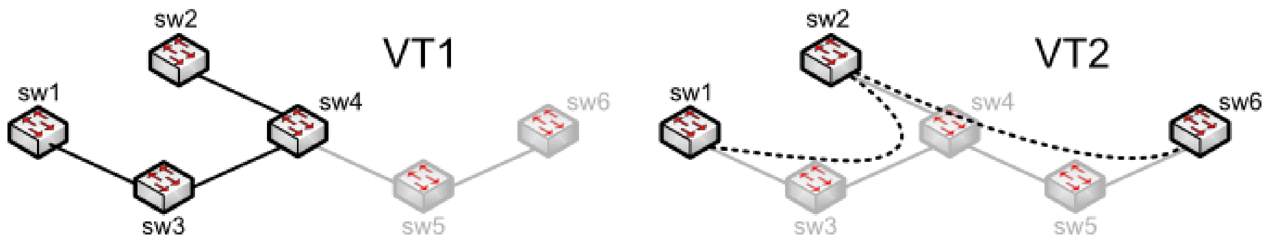


Figure 5.5: VT1 is an example of FlowVisor's slice while VT2 is an example of VeRTIGO's virtual topology

Like FlowVisor, VeRTIGO sits between the physical hardware and the guest OpenFlow controllers and enables the implementation of virtual topologies (see the VT2 example in Figure 5.5). Unlike FlowVisor, VeRTIGO directly controls part of the OpenFlow network with the purpose of enabling the definition of logical topologies completely decoupled from the underlying physical network.

VeRTIGO's capability to provide virtual instances of the physical network is grounded on two basic virtual elements: Virtual Links and Virtual Ports. These two elements are used to instantiate arbitrary network topologies including virtual links between not adjacent switches which are exposed to the OpenFlow controller as part of the network.

Virtual topologies instantiated through VeRTIGO are used to either simplify or deeply customize the topology of the network overcoming the physical constraints.

This customization can be useful for researchers willing to test their new protocols on a particular configuration of the network. VeRTIGO could also reduce the complexity of the network allowing the network administrator of a company to configure only the nodes at the edges of his/her network.

VeRTIGO is under development stage within the OFELIA European project and aims at offering an advanced network virtualization framework for the OpenFlow islands of the OFELIA facility.

5.2.2 FlowVisor-Less

Even though FlowVisor is considered the most usual tool to introduce a Network Virtualization layer into an OpenFlow network, several authors have recently proposed alternative approaches (and therefore to extensions as discussed in the previous section). The virtual architectures implied by FlowVisor have some known limitations, among them:

- When scaling to hundred or more controllers, FlowVisor could potentially experience some scalability issues (but in their original paper, FlowVisor's authors have implied it could scale to thousand of controllers by postulating FlowVisor's CPU and bandwidth consumption scale linearly with the number of controllers); furthermore FlowVisor is, itself a single point of failure.

- FlowVisor' slices are not allowed to modify the configuration of switches. This could be an interesting feature to enable virtualization in the wireless domain.

- Giving that FlowVisor is inherently bound to a single version of the OF protocol, it is not able to handle network scenarios where switches and controllers based on different versions of the OF protocol are running;

- Adding functional extensions to network switches (either OAM functionalities, novel forwarding mechanisms, pseudowires or a new matching structure) is difficult; FlowVisor is in fact bounded to the OpenFlow protocol version, therefore for every OF protocol update, FlowVisor has to be also updated.

- The global flowspace must be divided logically among all the network slices. Therefore there's no way for two FlowVisor' slices to share flowspace and simultaneously prevent them from interfering with each other's traffic.

- FlowVisor works as a hypervisor into the management plane; therefore it requires placing trust in a large and potentially buggy software code.

- FlowVisor do not provide a way to do not provide methods to verify in a formal way that a certain network has the required isolation properties.

While some of these limitations can be addressed by proper extensions of FlowVisor itself, others cannot be overcome since they are strictly related to the architectural properties of FlowVisor. Recently, some authors have been proposing alternative approaches to virtualize OpenFlow networks.

In [109] authors propose an "integrated OpenFlow virtualization framework" able not only to handle multiple instances of OpenFlow switches (with different forwarding capabilities and OpenFlow versions), but also to run and configure controllers designed for managing a virtual network and properly handle QoS in the network. A key point (and a constraint) of the proposed architecture is the availability of data plane elements with virtualization capabilities, i.e. elements which are able to run multiple instances (and versions) of OpenFlow switches together with a logic mapping the physical interfaces of the switches to the virtual ones. In particular, the proposed framework implements management and control functions for managing these virtual switches, by instantiating them, by starting/suspending or stopping them, by assigning physical ports to the virtual ones, etc. Furthermore, the framework maintains the topology of the managed virtual network, e.g. by reconfiguring the physical-to-virtual port assignment in case of a physical failure. The authors claims that FlowVisor can be easily managed through the proposed framework.

In [110] the authors takes a slightly different approach, by emphasizing what is really important when performing network slicing: traffic isolation among network users. While several approaches to perform such isolation have been proposed all over the years, from VLAN to firewalls up to FlowVisor (when dealing with OF networks), none of the proposed mechanisms is completely satisfactory. The authors claims that instead of relying on low-level mechanisms (like VLANs), middle-boxes (like firewalls) or complicated hypervisors (like FlowVisor), when it comes to Software Defined Networking, languages for programming networks should be “equipped with intuitive and composable constructs” to be leveraged when some isolation properties (including traffic, physical and control isolation) must be guaranteed. This method has in fact several advantages over all the existing mechanisms and, moreover, it provides a way to formally verify that a network has the needed isolation properties. In short, instead of proposing a new mechanism, the authors propose to move researchers attention from the mechanism itself to the implementation of a compiler for programs running on the controllers which should simplify programmers life through a set of simplified commands while leaving the effective implementation of the isolation to some of the existing mechanisms, somehow keeping their implementation complexities to a compiler that can properly interpret those commands and translate them into switch configurations.

In [111] the authors propose a novel OpenFlow virtualization mechanism with highly scalable multitenancy. Compared to existing mechanisms (FlowVisor and VeRTIGO), the flow space is not globally distributed among tenants but each tenants can handle any flow space values freely thanks to a flow and packet-header translation mechanism. One of the key points of the proposed architecture is the need for an Open vSwitch instance to be used as packet header translator on the host/VM side. While this is widely available on Linux machines, it is not on other OS-based ones. So, while the proposed framework could work well in a purely experimental scenario where all the hosts are controlled and can be standardized, in other application scenario this can be a severe constraint since a NV framework should be independent from the hosts architecture. It is also not clear how much this scenario can scale especially when dealing with hundreds or thousands of nodes and hosts.

5.3 Conclusions

Despite the significant breakthrough in virtualization technologies in IT, virtualization in data network is still a bottleneck in terms of performance and management. Part of the problem is that network resources are distributed in different locations in a logical network topology which makes them difficult to control and manage in an efficient and centrally controlled way. Software Defined Networking by dissecting the control plane from data plane on each network device and placing it on a central box, promises to tackle current network obstacles and among them network virtualization. Although SDN in its concepts tries to provide programmability and flexibility by abstracting the data plane, it does not address the network virtualization problems directly. Previous researches and projects about network virtualization have tried to investigate all aspects of the problem with different approaches to realize pros and cons of each solution. Having said that, in ALIEN Project a network virtualization solution that meets the SDN requirements with regards to previous and current network virtualization approaches will be provided.

6 OFELIA

6.1 Overall Information

OFELIA [105] is a collaborative project funded by the 7th EU Framework Project (ICT Work Programme), “OpenFlow in Europe: Linking Infrastructure and Applications”. It is a 3 year project, from October 2010 to September 2013, with a total cost of 6.3M€ (4.45 million Euros funding from European Commission).

The increasing complexity of the current Internet has shown the inherent limitations of its architectural design. Different research initiatives, such as Future Internet in Europe and Clean Slate Design in USA, have done significant efforts to overcome these limitations. In order to test novel research proposals from these initiatives at scale, an appropriate experimental facility is required. According to this, OFELIA cooperates with international research initiatives such as GENI in USA and AKARI project in Japan.

The main objective of the project is to create the first OpenFlow based experimental facility at Europe. The facility allows researchers not only to experiment by using the network but to control the network itself. The researchers have the ability to extend/modify the network in a dynamic way by adding or removing resources. OpenFlow supports the virtualization and delegation of the network’s control plane through a secure and standard interface (i.e. the OpenFlow protocol).

Initially, there were ten partners including operators, vendors and research institutions, and five interconnected islands based on OpenFlow technology. This OpenFlow infrastructure provides a unique facility to experiment on multi-layer and multi-technology networks. By means of two Open Calls, the number of partners and interconnected islands has increased. Two additional partners were added after the first Open Call (deadline March 30th, 2011) and five extra partners after the second one (deadline April 18th, 2011). Nowadays, OFELIA consists of 10 OpenFlow-enabled islands at academic institutions. However, all of them are not yet fully deployed or operational for testing.

Currently, the project consortium is composed of the following organizations: European Center for Information and Communication Technologies (EICT), Germany (coordinating partner); Deutsche Telekom Laboratories, Germany; University of Essex, UK (this is now transferring to the University of Bristol); i2CAT Foundation, Research and Innovation in the Internet Area, Spain; Technische Universität Berlin (TUB), Germany; NEC Europe Ltd, UK; Interdisciplinary Institute for Broadband Technology (IBBT), Belgium; Eidgenössische Technische Hochschule Zürich (ETH), Switzerland; The Board of Trustees of the Leland Stanford Junior University, USA; ADVA AG Optical Networking, Germany; Consorzio Nazionale Interuniversitario per le Telecomunicazioni (CNIT), Italy; Center for REsearch And Telecommunication Experimentation for NETworked communities (Create-Net), Italy; Centre Tecnològic de Telecomunicacions de Catalunya (CTTC), Spain;

Lancaster University, UK; Instituto de Telecomunicações Aveiro (ITAV), Portugal; University of São Paulo (USP), Brazil; Federal University of Uberlândia (UFU), Brazil.

As previously mentioned, OFELIA has built the first OpenFlow switching testbed in Europe (there are large OpenFlow testbeds in USA and Japan), which is fundamental for Future Internet experimentation. However, several scientific challenges needed to be addressed during the project lifetime in order to provide the proper virtualization support for a large-scale and flexible scenario. The facility enables the testing of novel control and routing algorithms at scale, since there are poor support for experiments in legacy routers and switches. Therefore, OFELIA creates a real-world experimental networking substrate with several interesting characteristics. First of all, it allows the disclosure of a flexible control plane down to individual flows. It also provides a protocol agnostic facility and exposes the programmability of network resources in a scalable manner. Regarding the OpenFlow technology itself, OFELIA allows the deployment and testing of new controllers and control applications.

The main objective of OFELIA is the creation of a research facility with the aforementioned characteristics. Consequently, there are three aspects to consider: virtualization, multi-domain and extensions to other technologies. The virtualization approach is based on FlowVisor, which enforces the isolation between slices and enables the delegation of the control plane to the researchers. It was improved by one of the projects accepted in the 1st Open Call, VeRTIGO (from CREATE-NET), which adds support for logical topologies to FlowVisor. From the standpoint of the facility, the automatic creation/management of slices is mandatory for both FlowVisor and VeRTIGO. Regarding the multi-domain issue, some extensions to controllers are expected to achieve the federation of islands. Although there have been some efforts in the community, there are still several issues to solve. Since OFELIA was conceived as a facility for Future Internet proposals, the experimentation with optical and wireless technologies is expected. Finally, some OpenFlow extensions are needed to cope with multi-layer and multi-domain experiments. In this multi- scenario, any layer or domain borders require flow processing (extending the filter format description to generic labels) and additional developments are needed to interface these elements towards the controller. Moreover, non-IP experiments are also possible such as content-based addressing approaches.

6.2 Facility and islands

OFELIA was conceived to grow as a facility in three phases. One reason for this was the obligation to provide early access to the facility due to the FP7 call restrictions, no later than month 6. Another reason was the necessity to evolve the facility during the project lifetime. This evolution is a consequence of the requirement to incorporate the feedback of users and accommodate their needs. Additionally, the gradually extension of the facility to other locations and test facilities also caused this evolution.

The first phase (March 2011) focused on the setup of islands. This first phase aimed to implement the first operative version in a short period of time and made the facility available for users in month 6. Initially, there were five islands located in different universities and research institutions in Europe: University of Essex (UK), TU Berlin, IBBT Gent, i2CAT Barcelona and ETH Zürich. In this phase, researchers accessing the OFELIA facility were provided with a network slice, which is basically formed of three different resources: (1) a set of network nodes with OpenFlow 1.0 support, which in most cases are slices of the flowspace of the physical NEC switches; (2) a virtual machine to run the OpenFlow controller, which controls the network resources; and (3) a set of virtual machines to act as end-hosts. The researchers were able to access the delegated virtual machines from the outside of the facility by means of SSH.

In this phase, the five islands were reachable through bridged OpenVPN connections between them. These interconnections were expected to be improved by installing dedicated GbE circuits in a later phase. From the outside, researchers needed to set up a routed OpenVPN connection to IBBT in order to interact with the facility and get access to their network slices. Then, IBBT became the central access point to the OFELIA facility.

The second phase (March 2012) was oriented to improve the interconnection of the islands. As previously stated, one of the main concerns about the early setup of the facility is the lack of dedicated resources and limited performance of these interconnections. In this phase, the connectivity was improved to a minimum of 1Gbit/s Ethernet tunnels in star configuration. The central node of this topology was located at IBBT in Gent, which became the hub of OFELIA. Since the external access to the facility was centralized at IBBT, this election makes sense.

The third and last phase focuses on the customization of the facility in order to support the OFELIA researches to deploy their innovative usage scenarios. The Open Call process invites new partners to the consortium and assigns dedicated funding to implement and deploy their new use cases over the facility. Concrete use cases arises new requirements that need to be addressed and new partners contribute to add this support to the OFELIA portfolio.

In this final phase, the automated provisioning of OpenFlow slices across multiple islands needs to be available for researchers. Furthermore, the federation of OFELIA with other FIRE facilities is also expected.

Currently, OFELIA is formed of ten OpenFlow-enabled islands at different academic institutions both in Europe and Brazil, which are: IBBT (Gent, Belgium), University of Bristol (Bristol, UK), ETH (Zurich, Switzerland), i2CAT (Barcelona, Spain), TUB (Berlin, Germany), Create-Net (Trento, Italy), CNIT (1 in Roma and 2 locations in Pisa, Italy), and UFU (Brazil). All islands can be seen in Figure 6.1. However, only six islands are currently available which are detailed in the following subsections.

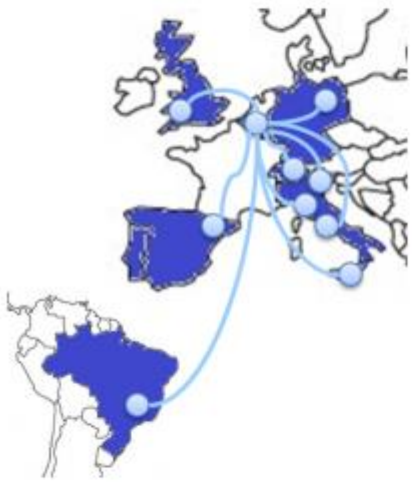


Figure 6.1: OFELIA facility and islands

6.2.1 IBBT Island

The IBBT testbed consists of two laboratories: Wilab and Virtual Wall. Both laboratories are available at OFELIA and focus on different technologies.

The Wilab (w-iLab.t) is an experimental wireless testbed deployed in the IBBT building, in which researchers are able to upload executables, associate them with wireless motes to create a job and schedule the job to be run on wilab. All the messages from each experiment are available for later processing by the corresponding researcher. Wilab enables research in sensor network programming, protocols and applications. There are 170 TMote Sky sensor motes deployed at IBBT. Each mote consists of: TI MSP430 processor 8Mhz, 10KB of RAM, 1Mbit of Flash, Chipcon CC2420 radio at 2.4GHz (indoor range 100 m) and sensors for light, temperature and humidity. Nodes run the TinyOS operating system and NesC is used as programming language.

The Virtual Wall (iLab.t) consists of 100 servers interconnected by a non-blocking Force10 switch. The infrastructure allows researchers to configure any network topologies adding enough flexibility to change completely the network behaviour and how the traffic is handled by each node. Each server has the following specification: dual CPU and dual core, 4GB RAM, 4x 80GB HDD. There are 60x6 and 40x4 GbE experimental network interfaces interconnected by a Force10 E1200 switch with 576x Gb/s ports, 8x10Gb/s ports and 1.6 Tb/s backplane. If needed, links between network nodes can be shaped by restricting the bandwidth or adding some delay. Furthermore, researchers get root access to the nodes and are able to customize the OS images. However, a specific image has been created in advance for OFELIA with support for OpenFlow v1.0, which includes: OpenVSwitch v1.1.0, NOX controller v0.9.0 and credentials to get access to OFELIA.

6.2.2 University of Essex Island

The OFELIA research team from the University of Essex have changed their affiliation to University of Bristol, so the island is expected to be moved also to their new location in Bristol. The island consists of four OpenFlow enabled switches (NEC IP8800/S3640-24T2XW) and two dedicated physical servers (Dell Power Edge 1950, Intel Xeon® Quad Core X5355 2.66GHz, 8GB RAM and 400GB HDD). The virtual machines provided to researchers are based on XEN virtualization (XEN hypervisor on top of Debian 6) and automatically created by the OFELIA control framework. Moreover, there are four additional servers, which are part of the island infrastructure but not accessible by researchers: one for the OFELIA control framework; another one for the FlowVisor used to slice the resources; a third server to run a SNAC (controller for the administrator of the island); and a last server to run the OpenVPN that connects the island to IBBT hub. The island has a very good connectivity to GEANT and JANET (NREN from UK), and also to Brazil and USA via Internet2. Additionally, optical equipment (ADVA FSP 3000 Optical Add/Drop Multiplexers) is expected to be connected to the island.

6.2.3 ETH Island

The ETH Island is located in Zurich and hosted inside the Communication Systems Group. Basically, the island is formed of three OpenFlow enabled switches (NEC IP8800/S3640-24T2XW) and three 64-bit dedicated servers with Debian Squeeze and 36 GB of RAM. The switching topology is completely meshed. Regarding the servers, one of them is dedicated to OFELIA related issues, such as the control framework, FlowVisor and OpenVPN tunnel. The remaining two servers are devoted to host the virtual machines delegated to researchers based on XEN virtualization. Each VM has three network interfaces: the control interface (eth0), and two experimental interfaces (eth1 and eth2).

6.2.4 TUB Island

This island is located in the Campus-Network at the Technical University of Berlin. Concerning the OpenFlow enabled switches, there are five switches (four NEC IP8800/S3640-48TW and one HP5400); however, only three of them are currently available at OFELIA. These three switches are connected in a full mesh topology. There are also two servers (E3-1240 Quad Core with 16 GB RAM) running Debian Squeeze for the virtual machines delegated to researchers. Both servers are connected to the three switches (eth1, eth2 and eth3) and use an additional interface (eth4) for the control traffic. Additionally, an Ixia T400 traffic generator and performance analyser is connected to the infrastructure with eight port GBit interface cards. Finally, the OFELIA control framework and the OpenVPN gateway are hosted in another server, an IBM server with two Intel Xeon CPUs (Quad core) and 2.4 GHz. Therefore, the experimental traffic is bridged and tunnelled to other islands by means of a VPN.

6.2.5 I2CAT Island

The i2CAT island comprises eight OpenFlow enabled switches (five NEC IP8800/S3640-24T2XW and three HP E3500-48G-PoE) in a full mesh topology; however, only the NEC devices are currently deployed and available through OFELIA. There are also five dedicated 64-bit servers (SuperMicro SYS-6010T-T) running Debian Squeeze with 12 GB of RAM. One of these servers hosts the OFELIA control framework, the FlowVisor to slice the network and the OpenVPN to connect the islands. The remaining four servers are used to host the virtual machines automatically delegated to end users by means of the control framework. The virtual machines are based on XEN server running over Debian.

6.2.6 Create-Net Island

The Create-Net Island deployed in Trento (Italy) is the first extension to the original OFELIA infrastructure which is currently up and running. Regarding the OpenFlow enabled switches, the island is formed of three NEC switches (NEC IP8800/S3640-24T2XW), two HP switches (HP ProCurve 3500) and four NetFPGA cards. Concerning the server machines, there are two Dell PowerEdge (1850 32-bit with 8GB and 1750 32-bit with 5 GB) and three server-class PCs (Debian Squeeze, 64-bit, 16 GB of RAM). The latter three servers are devoted to host the virtual machines of researches and are based on XEN Server, whereas one Dell server hosts the OFELIA control framework and the other Dell server hosts both the FlowVisor and VeRTIGO. Create-Net entered in the consortium in the 1st Open Call to deploy the support for logical topologies in OFELIA (i.e. VeRTIGO) and is the first island with this support.

6.3 Control Framework

The OFELIA Control Framework (OCF) [112] can be defined as the orchestration software for the OFELIA FP7 [105] facility. The main purpose of the framework is to arbitrate, automate and simplify experiment life-cycle within the facility. Original directives were taken from the ”OFELIA Basic Use Case” [113] and the experience of previous OpenFlow testbeds already running, like GENI [91] project.

OFELIA embraced the Enterprise GENI (E-GENI) [114] control framework for its facility as a base over which a lot of new features and functionalities are added as per OFELIA’s requirements. The E-GENI control framework was based on SFA

(Slice-based Facility Architecture) [115]. SFA is a federation framework that defines a set of rules by which two or more experimental entities can be federated.

The most used federation scenarios are:

- **Centralized:** a single Clearinghouse manages all the underlying components of control framework across the experimental facilities. This type of federation is used e.g. in the Panlab [116] architecture.
- **Distributed:** offers more flexibility and redundancy, because every Clearinghouse in every facility will have full access to the components in the lower layers of the control framework. The OCF is based on this type of federation.

The OFELIA Control Framework fulfils a set of requirements defined in the "OFELIA Basic Use Cases":

- **Resource allocation and instantiation:** the OCF software supports resource allocation, instantiation and de-allocation for generically any types of resource (e.g. an OpenFlow network slice or a virtual-machine).
- **Experiment/project based resource allocation:** the resource allocation/de-allocation is made per project and slice (e.g. the smallest indivisible entity composed by the resources necessary to carry out an experiment). Slices are totally isolated from each other, even though they might share the same infrastructure substrate. The isolation among slices is fulfilled using a unique header field assigned for slicing purposes (currently VLAN ids).
- **Federation and island autonomy:** the software architecture inherently supports internal (between OFELIA islands) and external federation with other testbeds. For this reason the OCF is based on a distributed scenario for managing the federation.
- **AA and policy framework:** OCF supports mechanisms for authentication and authorization (in several scopes) along with a strong policy framework (also in several scopes or layers).
- **Usability:** experimenters have access to comprehensive and easy to use user interface(s). In this sense, the main focus of the development has been towards a web-based user interface.

6.3.1 OCF Current Architecture

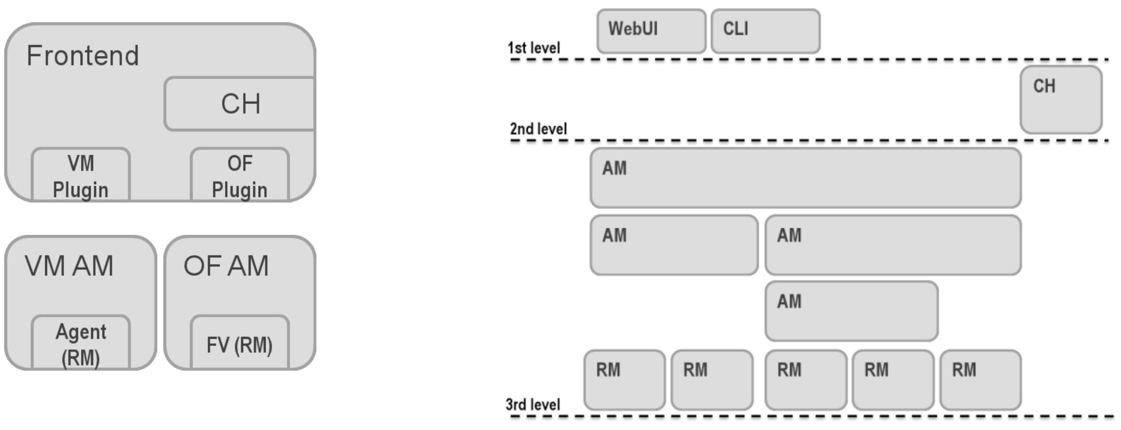
The current architecture design of the OCF software, corresponding to a single testbed, is the one depicted in Figure 6.2(a). This architecture is influenced by the open-source software used to manage OpenFlow resources (Expedient and Opt-In). One of the main requirements of OFELIA was that users could access the physical resources (switches and virtual machines) since the very beginning of the project. As a first step, the internal board in charge of the implementation decided to use this standard software and to modify/extend it to match the requirements. Nevertheless, as part of the project work-plan and following an iterative approach, architecture has and is evolving along with the implementation, as it will be shown in section 6.3.2. However most of the fundamental concepts behind the evolved architecture are already present in the design. Figure 6.2(a) shows the basic structure of a single testbed software stack, composed fundamentally by two types of components and formally corresponding to two separate layers:

- **Frontend layer:** This layer encapsulates two functions: the user interface (web frontend) and the so-called Clearinghouse (GENI terminology). The Clearinghouse stores information about projects, slices and user state, and is in charge of user authentication and authorization. This component plays two main roles. On one hand, it deals with the management of users, projects and slices within each island. The Clearinghouse accesses the OFELIA LDAP for synchronizing user accounts and privileges among islands and provide a unified authentication framework (Single Sign In). On the other hand, it also acts as the AJAX-enabled web-based user interface. The communication with the aggregate managers below is performed by means of specific plugins developed in Python. Each plugin provides the means to communicate and translate the resource specific description provided by an Aggregate Manager (see next bullet). The Clearinghouse is based on Expedient, a module originally developed by Stanford University, but highly adapted and extended in areas like the experiment workflow and the Web UI.

- **Resource/Aggregate Managers (RM/AM) layer:** This layer takes care of resource management, hence maintaining reservation state, performing resource allocation, setup, monitoring and de-allocation. Currently two AMs with the corresponding RMs are supported:

    - **Virtual Machine Aggregate Manager:** deals with the management of the virtualized servers provided by the OFELIA facility to host OFELIA user's virtual machines. The current implementation supports XEN, although AM design and implementation is inherently hypervisor-agnostic. The VM AM handles the requests and takes care of the provisioning, instantiation and deinstantiation of VMs.

    - **OpenFlow Aggregate Manager:** is currently based on the Stanford's tool Opt-in Manager. The main objective of this package is to control and administer the FlowVisor configuration, the tool in charge of slicing the OpenFlow network and multiplex concurrent usage (see Section 5).



(a) (b)

Figure 6.2: OFELIA Control Framework Architectures for a single testbed: (a) current version v0.22, (b) future version v1.x

Federation, not shown in the figure for simplicity, is accomplished by allowing higher layer component instances (Frontends) to connect to other testbeds RM/AMs, effectively allowing users to use resources across several testbeds. Obviously, a common slicing mechanism must be adopted between federated testbeds in order to assure the separation among users' traffic.

Nevertheless, this architecture presents some limitations. First, the higher layer component (frontend) should be split into two logically separated modules, the User Interfaces (UIs, e.g. Web Graphical User Interface and Command Line Interface) and the Clearinghouse (CH), along with a formal definition of the interfaces among these components and the interaction with the AM/RM layer.

The federation is achieved using at least two logical networks with different purposes:

- **Experimental network:** interconnects the OpenFlow switches and Virtual Machines (VMs) accessible for experimenting.
- **Control network:** the facility contains some hardware to control and manage the facility. Besides the OFELIA specific services (Expedient, OpenFlow AM and VT AM, FlowVisor), also more generic services are provided, like an LDAP server for storing the credentials, DNS server, NFS server, etc. The purpose of this network is to interconnect these servers and the infrastructure under control, plus it gives the users access to their experiments (slices).

6.3.2 OCF Software Architecture for Future Versions

Figure 6.2(b) shows the planned evolution of the software architecture design, intended to be implemented starting from version 1.0 of the OCF on in order to overcome the limitations previously described. The architecture is highly influenced by other testbed projects such as GENI. The first release of OCF v1.0 is expected for September 2013, according to the implementation roadmap.

In this new architecture there is a formal definition of three separate layers;

- the User Interface layer
- the ClearingHouse layer
- the AM/RM layer

Moreover, interactions and interfaces among these layers are formally defined. Federation is still taking place allowing user interface to interact with other testbeds' AM/RM layers. The new architecture is based on the assumption that AMs and RMs may form a hierarchical chain, also due to the fact that they have an unique authorization and authentication frameworks and common APIs. This will be achieved since all the different AM/RMs will be based on the AMsoil [117], a software package framework that will act as the base toolset to build OFELIA AMs. It encapsulates common tasks which are performed by every AM/RM, such as Authentication & Authorization, interfaces such as the native OFELIA API, GENI API v3 or SFA wrapper and common AM/RM abstractions and mechanisms, like booking and monitoring logic.

Both the first and the third layer will be substantially modified starting from OCF v1.0. In particular, the current OpenFlow AM (OptIn) will be replaced with an extended version of FOAM [118], that is actually the OF AM used in GENI. The OFELIA FOAM will be enhanced with the support of virtualization (VeRTIGO) [119], optical and wireless resources. Moreover, the UI level will be updated with a new version of the Graphical User Interface, that will support more generic resources, and with the inclusion of OMNI, the UI adopted by GENI.

The aggregation of AMs is something allowed by the architecture itself and may be implemented depending on the specific management requirements (resource aggregation). The architecture also defines that AM/RM component may implement policies locally according to the operational needs.

6.4 Conclusions

The OFELIA experimental facility is formed of ten OpenFlow-enabled islands at different academic institutions of both Europe and Brazil. The facility allows researchers not only to experiment by using the network but also to control the network itself. The researchers have the ability to extend/modify the network in a dynamic way by adding or removing resources.

In OFELIA, the whole life-cycle of experiments is managed through the OFELIA Control Framework (OCF). The OCF is a framework which provides the experimenters a comprehensive and easy to use user interface for requesting network and computational resources. The OCF also relies on FlowVisor and VeRTIGO as OpenFlow resource managers and XEN as manager for the computational resources.

Given the modular architecture of the OCF, the ALIEN hardware can be easily integrated into the OFELIA facility by implementing specific aggregate and resource managers and a plugin for the GUI of the OCF. The ALIEN project can leverage on the OCF to enable the OFELIA users/experimenters to use the ALIEN hardware through a single interface and hence, to conduct their experiments on a heterogeneous testbed composed of OpenFlow and non-OpenFlow capable hardware simultaneously.

7 Security Aspects

This section discusses the state of the art in security as it applies to the ALIEN project. In specific it covers security aspects related to Software Defined Networking and intermediate representation languages. While there are some reasons to believe that software defined networks will help with security issues, however, there are some specific challenges which must be met. No comprehensive review paper yet seems to exist in the academic literature but high level overviews are given in articles such as [120] and from an enterprise network perspective [121]. While SDN in general does not specify a specific technology for the control plane, Open Flow is becoming increasingly widely spread and much of the discussion in this section is in the context of Open Flow as a control plane technology although many of the conclusions would be general to any controller which was equally or more flexible.

7.1 Specific Security Aspects in SDN

7.1.1 Isolation

In a typical SDN situation, multiple tenants share virtualised slices of the network infrastructure. Hence, there is opportunity for lack of resource isolation leading to attacks between SDN tenants.

This lack of isolation can manifest itself as a confidentiality risk (by e.g. enabling side channel attacks) or an availability risk (by e.g. resource starvation). In current virtual network implementations, isolation is applied at various levels and using different techniques. In the case of FlowVisor [107], OpenFlow is used to provide isolation at the control and forwarding level; other techniques are required for CPU isolation [122] and bandwidth isolation [123]. Whereas FlowVisor relies on running parallel OpenFlow instances to achieve its isolation properties, it is also possible to achieve this by compiling independent network slices into a single set of flow rules that implement them [110]. Although this provides performance and verifiability benefits, it can increase system complexity.

Independently of how control and forwarding is achieved, current proposals rely on additional techniques for CPU and traffic load isolation. Hence, in order to protect tenants in SDN deployments from traffic injected into the shared network infrastructure, adequate support for traffic isolation must be deployed in the SDN-enabled network elements themselves. Failure of isolation can lead to potential attack vectors. For example, shared physical paths with insufficient isolation could allow users to deduce the nature of the traffic pattern of other users who share those physical paths even if they are isolated virtually.

7.1.2 Protecting the Controller

One of the reasons why SDN is attractive is that it provides increased flexibility by centralising policy deployment. This has the effect, however, of turning the SDN controller into a single point of failure. In effect, any mechanism that achieves DoS on the controller can then be leveraged to DoS the entire network. In general terms, two kinds of attacks can be distinguished: attacks that target the controller directly, and attacks that target the control channel between the controller and SDN-enabled network elements [120].

7.1.2.1 *Direct Attacks on the SDN Controller*

Since in many cases the controller has wide-ranging control capabilities over the network, its compromise or disabling could present an endless series of security threats. This means that SDN controllers must be both logically and physically secure, with robust authentication and authorization mechanisms. In addition, the software components of the controller must be robust, and hardened against privilege escalation and arbitrary execution threats by e.g. buffer overflow errors. Operationally, the controller must also be supported by secure procedures that correctly specify which users, processes and hardware platforms have what privileges over its hardware and software. Unnecessary services should be removed from the controller, and incoming connections to it should only be accepted if adequately authenticated. Finally, operating system security mechanisms such as digitally signed application installation and application executable fingerprinting should be in effect in the controller.

7.1.2.2 *Attacks on the SDN Control Channel*

The communications channel between the controller and the SDN-enabled network elements (NEs) can be used to perform attacks on both. Hence, this channel must be adequately secured. In OpenFlow 1.3.1 then the channel “is usually encrypted by TLS but may be run directly over TCP” and is from a “user-configurable (but otherwise fixed) IP address”. By requiring strong mutual authentication in these interactions, the risk for impersonation in both directions is mitigated. Furthermore, since communication failures between controllers and NEs could potentially disrupt the operation of the entire network, adequate capacity and survivability is required in order to ensure that the network remains operational even on high-load scenarios such as a DDoS attack or a flashcrowd. This has led to some network architectures relying on dedicated capacity for controller-NE traffic. However, if such a design is to be effective, the controller itself should be made resilient against excessive application load. This makes the deployment of the controller as part of a cluster or elastic cloud infrastructure particularly attractive, as long as adequate fail-over mechanisms are present between the controller and its standby counterparts.

7.1.3 Dynamic Traffic Policies

In an SDN-enabled network, traffic policies can be updated in near-real-time simply by effecting the corresponding changes in the controller. This means that SDN tenants may not have any assurance, a priori, of the treatment that a particular traffic flow may receive. This may lead to a mismatch between the characteristics of flow paths as presupposed by the end users and those that the actual paths have. This is of crucial importance for scenarios in which the SDN controller is compromised; this can lead to man in the middle attacks that are difficult to defend against.

From a practical perspective, the risk of dynamic traffic policies having unexpected effects is normally dealt with by using policy conflict detection. For instance, both VeriFlow [124] and FortNOX [125] both rely on the real-time verification of flow rules as they are inserted in the controller in order to detect whether they break network-wide invariants [124] or contradict previously inserted security rules [125].

Another source of complexity when dealing with dynamic traffic policies is their composition in such a way that desirable delegation of authority and conflict resolution properties are preserved. One proposal to achieve this, HFT [126], organizes policies as trees in which each subtree can independently be used to determine what action to take over a given packet. When conflicts arise between different tree nodes, they can be resolved using user-defined resolution operators which exist at every node.

7.1.4 Risks and benefits of virtualisation

By decoupling the control and forwarding paths, SDN opens the door for the deployment of virtualised network elements. This means that components traditionally used to demarcate the network perimeter, such as Firewalls or Intrusion Prevention Systems (IPS) can now be instantiated in various points within the physical network. However, with this benefit comes a risk. Firewalls are often used to guard a network perimeter and separate a network into an untrusted outside network, a semi-trusted de-militarised zone and a trusted inner network. With a virtualised network infrastructure this very concept of a network perimeter becomes slippery.

The freedom to deploy virtual network elements allows an unprecedented capability to deploy security policies in a flexible, modular fashion. For instance, FRESCO [127] allows the compilation of flow rules on the basis of 16 commonly reusable modules that provide capabilities such as packet filtering and other firewall functions, scan detection, attack deflection or intrusion detection logic. By allowing flexible and quickly deployable security policies, then security can be associated with virtual machine instances rather than with physical machines. This means that the security policies can follow and be appropriate for the virtual machines running on that infrastructure and, in a scalable deployment, where virtual machines are created and destroyed on demand, they can be created within a virtual infrastructure which has appropriate policies in place for those tasks.

Imaginative use of network virtualisation could provide new methods to meet old threats. For example [128], provides functionality to make attacks more difficult by constantly changing the public-facing IP address of hosts while maintaining a continuous assignment within the network. Hence, named hosts are reachable via virtual IP addresses advertised using DNS, but real IP addresses are only reachable by authorized entities. This helps mitigate the possibility of DDoS style attacks on hosts.

Although the increased flexibility provided by SDN can help reduce capital and operational expenses, it can also increase the complexity of SDN deployments. This has led to a number of commercial platforms [129], [130] that help system administrators manage the challenges specific to virtualised networks and SDN.

7.2 Conclusions

7.2.1 Implication of Security Aspects

As can be seen from the previous section, SDN has both benefits and costs in terms of network security. In some cases, research is already showing the benefits that network virtualisation could have. In other cases, however, it is clear that the eventual outlook for SDN security will depend on to what extent the controllers and their interactions with network elements are developed with security aspects in mind.

Certain new attack vectors identified in the previous section arise from the presence of a centralised controller (for example an OpenFlow controller). Attacks which compromise this controller or the interaction between the controller and the network elements could have severe security risks and, in particular, enable more subtle and problematic attacks than traditional man-in-the-middle mechanisms. A compromised SDN controller would have a great deal potential for attack and privilege escalation.

Provided that virtualised network elements do give the promised isolation then this would allow greater freedom diverse users to share the same network infrastructure without having to mutually trust each other. However, failures of such isolation could lead to the possibility of side-channel attacks and leakage of information (for example about traffic patterns) between networks.

DDoS is a well-known attack vector that works in a number of circumstances. There is good reason to think that SDN could mitigate such attacks, for example, simply by providing scalability through a virtualised infrastructure (which might allow more physical resources to be brought into play if a given link were overwhelmed). In addition more subtle defences such as [128] could use the SDN flexibility to specifically guard against DDoS attacks. Conversely, however, the presence of the controller might, itself, present opportunities as a point of failure and provision must be made for failover in case of attacks on the controller itself.

In summary then, there are many reasons to believe that SDN could deliver increased security for network infrastructures, however, this will only be the case if steps are taken to mitigate the known risks.

7.2.2 Intermediate Representation Languages

Research in the area of security in programming languages as well as their intermediate representations is still continued. Analysis of vulnerabilities and new attack techniques is an endless process. Inducting new more complex technologies causes threat of appearing new software vulnerabilities. The consequences of leaving unpredictable backdoors may become a significant danger for appearing new exploit and in fact serious danger of system takeover. The security aspects of language syntax and architecture for intermediate representation layer should be always taken into account during designing process.

Intermediate languages as an additional abstraction layer allows to create additional level of protection. These languages are often comparatively simple and thus are a good starting point for a various methods of a static code analysis which can be performed while compilation or before execution. IL allows also to create an additional level of permissions. In Common

Intermediate Language (CIL) infrastructure a starting code is assigned to one of defined code groups and verified against proper security policy.

Important security issue which is addressed only by GPGPU Intermediate Languages is enforcement of internal separation of resources which are visible from outside as a one resource i.e. graphic cards. This functionality is implemented e.g. in OpenCL from version 1.2 as a Device Partitioning.

Different security aspects are presented based on Java and PacketC. Important security issues of those two intermediate languages are briefly characterized below. It could become further inspiration during work on HRM and HAL.

Java Platform provides a safe and secure solutions for running applications. Compile-time data type checking and automatic memory management leads to more robust code and reduces memory corruption and vulnerabilities. Bytecode verification ensures code conforms to the JVM specification and prevents hostile code from corrupting the runtime environment. Class loaders ensure that untrusted code cannot interfere with the running of other Java programs [131].

Built-in language security features enforced by the Java compiler and virtual machine are:

- strong data typing,
- automatic memory management,
- bytecode verification,
- secure class loading.

PacketC language (described in Section 4.6) dedicated especially for network purposes is more secure language than C99 [67] and has been designed to maximize application reliability [75]. Some object-oriented features included in packetC make this approach more secure and error-free by:

- strong typing,
- exception handling implemented using try-catch-throw concept,
- simplifying type declaration system to prevent typing conflicts,
- eliminating pointers while providing flexibility for secure dynamic references.

Both security vulnerabilities description and proposed solutions are planned to be taken into account during HRM and HAL design phase.

8 Conclusions and next steps

The very fast innovations reflected in many competitive solutions within the electronic hardware is the source of heterogeneity problems which manifests in the existence of problems during managing, controlling and using of larger sets of the hardware. This document presents different approaches which have been designed to solve aspects of the heterogeneity existing within IT and networking areas: 1) network node forwarding representations (e.g. OpenFlow) shared between control and forwarding layer in SDN networks, 2) a common data modelling languages capable of representing some parts of network environment (e.g.: VXDL, INDL, NetPDL), 3) silicon circuits description languages, 4) intermediate languages translating high-level code into low-level instructions of the broad spectrum of existing processors.

One of solutions for solving a heterogeneity problem is to make a good abstraction which will be capable of describing and operating over different kinds of the hardware and its background technology. Currently, the most promising abstraction of network hardware elements is OpenFlow, however this document presents the limitations related to OpenFlow like Ethernet-centric data modeling, to close relation on ASIC chip-set API and hiding very specific functionalities of the hardware which can be useful. In order to deal with these limitations, extensions of OpenFlow protocol should be proposed which will contain more flexible data path model and will operate over more high level programming representation. The proposed solution should be open for new kinds of hardware platforms and programming environments (NetFPGA,, Intel DPDK, etc.).

From the point of view of intermediate representation languages, a modular design of software components is a key factor to successfully deal with the hardware heterogeneity and straightforward adaptation to new requirements. In the document several solutions have been elaborated allowing to transform high-level operations into a low-level hardware-specific machine code. These factors should be applied in the HAL solution developed within the project. On the other hand, network description languages (VXDL or INDL) could be used by OFELIA resource management software whereas NetPDL could allow network element learning of new protocols or protocol extensions (i.e.: flow matching of a new field and its modification) which will enable of faster and easier network protocol stack evolution/revolution towards Future Internet solutions.

This deliverable will drive the work in T2.2 of the ALIEN project, where the design of Hardware Abstraction Layer (HAL) and Hardware Description Language (HDL) for non OpenFlow capable devices (alien hardware) will take place. The HAL should interface with different type of alien hardware and hide their complexity as well as technology and vendor specific features from OpenFlow control framework. The HDL for alien hardware should provide a uniform representation of any type of alien hardware.

9 References

[1] D. Dineley. Software-defined networking.

[2] K. Greene. (2009) MIT Technology Review. [Online]. http://www.technologyreview.com/biotech/22120/

[3] T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, J. Turner N. McKeown, "OpenFlow: Enabling Innovation in Campus Networks," in *ACM SIGCOM Computer Communication Review*, vol. 38, April 2008, pp. 69-74.

[4] OpenFlow Switch Consortium. (2011, February) OpenFlow version 1.1.0 Switch Specification. [Online]. http://www.openflow.org/documents/openflow-spec-v1.1.0.pdf

[5] Cisco Open Network Environment (ONE). [Online]. http://newsroom.cisco.com/cisco_one

[6] Cisco Open Network Environment. [Online]. http://www.cisco.com/web/solutions/trends/open_network_environment/index.html

[7] T. Anderson H. Khosravi, "Requirements for Separation of IP Control and Forwarding," RFC3654, November November 2003.

[8] R. Dantu, T. Anderson, R. Gopal L. Yang, "Forwarding and Control Element Separation (ForCES) Framework," RFC3746, April 2004.

[9] J. Hadi Salim, R. Haas, H. Khosravi, W. Wang, L. Dong, R. Gopal, J. Halpern A. Doria, "Forwarding and Control Element Separation (ForCES) Protocol Specification," RFC5810, March 2010.

[10] K. Ogawa J. Hadi Salim, "SCTP-Based Transport Mapping Layer (TML) for the Forwarding and Control Element Separation (ForCES) Protocol," RFC5811, March 2010.

[11] J. Hadi Salim J. Halpern, "Forwarding and Control Element Separation (ForCES) Forwarding Element Model," RFC5812, March 2010.

[12] R. Haas, "Forwarding and Control Element Separation (ForCES) MIB," RFC5813, March 2010.

[13] H. Khosravi, A. Doria, X. Wang, K. Ogawa A. Crouch, "Forwarding and Control Element Separation (ForCES) Applicability Statement," RFC6041, October 2010.

[14] K. Ogawa, W. Wang, J. Hadi Salim E. Haleplidis, "Implementation Report for Forwarding and Control Element Separation (ForCES)," RFC6053, November 2010.

[15] O. Koufopavlou, S. Denazis E. Haleplidis, "Forwarding and Control Element Separation (ForCES) Implementation Experience," RFC6369, September 2011.

[16] B. Heller, N. McKeown B. Lantz, "A Network in a Laptop: Rapid Prototyping for Software-Defined Networks," , New York, NY, USA, 2010, pp. 19:1-19:6.

[17] T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, N. McKeown N. Gude, "Nox: Towards an Operating System for Networks," no. 38, pp. 105-110, July 2008.

[18] About NOX | NOX Repo. [Online]. http://www.noxrepo.org/nox/about-nox

[19] About POX | NOX Repo. [Online]. http://www.noxrepo.org/pox/about-pox

[20] Beacon: a Java-based OpenFlow Control Platform. [Online]. http://www.beaconcontroller.net/

[21] Home - Beacon - Confluence. [Online]. https://openflow.stanford.edu/display/Beacon/Home

[22] Floodlight OpenFlow Controller. [Online]. http://floodlight.openflowhub.org

[23] Trema. [Online]. http://trema.github.com/trema

[24] Ryu. [Online]. http://osrg.github.com/ryu

[25] Big Switch. [Online]. http://www.bigswitch.com

[26] Big Network Controller. [Online]. http://www.bigswitch.com/sites/default/files/sdn_resources/bnc_datasheet.pdf

[27] M. Scharf, T. V. Lakshman, V. Hilt V. K. Gurbani, "Abstracting Network State in Software Defined Networks (SDN) for Rendezvous Services," in *IEEE International Conference on Communications ICC*, 2012, pp. 6627-6632.

[28] Open Networking Foundation, "Software-Defined Networking: The New Norm for Networks," April 13, 2012.

[29] Open Networking Foundation: Into to OpenFlow. [Online]. http://www.opennetworking.org/standards/into-to-openflow

[30] Open Networking Foundation, "OpenFlow Switch Specification," September 6, 2012.

[31] ALIEN: Description od Work (internal).

[32] J. Schaad, "Use of Advanced Encription Standard (AES) Encription Algorithm in Cyptographic Message Syntax (CMS)," RFC3565, July, 2003.

[33] R. Dantu, T. Anderson, R. Gopal L. Yang, "Forwarding and Control Element Separation (ForCES) Framework," RFC3746, April, 2004.

[34] Forwarding and Control Element Separation (ForCES). [Online]. http://datatracker.ietf.org/wg/forces/charter/

[35] SPARC - Split Architecture for Carrier Grade Networks. [Online]. http://www.fp7-sparc.eu

[36] CHANGE. [Online]. http://www.change-project.eu/

[37] Packet Processing on Intel Architecture. [Online]. http://www.intel.com/p/en_US/embedded/hwsw/technology/packet-processing

[38] L4 micro kernel. [Online]. http://www.l4hq.org

[39] MIT Exokernel Operating System. [Online]. http://pdos.csail.mit.edu/exo.html

[40] M. F. Kaashoek, J. O'Toole Jr. R. E. Dawson, "Exokernel: An Operating System Architecture for Application-Level Resource Management," in *in Proceedings of the 15th ACM Symposium on Operating Systems Principles, SOSP '95*, Cooper Mountain Resort, Colorado, December, 1995, pp. 251-266.

[41] M. Hara, B. Heller, B. Lantz, J. Pettit, B. Pfaff, D. Talayco D. Erickson, "OpenFloe Hardware Abstraction API Specification - DRAFT," Version 04 based on OpenFlow 0.9.0 September 2009. [Online]. http://www.openflow.org/wk/images/c/c2/Of-hw-api-draft-0.4.pdf

[42] Software Defined Networking and Software-Based Services with Intel Processors. [Online]. http://www.intel.com/content/www/us/en/communications/communications-sw-defined-networking-paper.html

[43] R. Harrison, M. J. Freedman, Ch. Monsato, J. Rexford, A. Story, D. Walker N. Foster, "Frenetic: A Network Programming Language," in *ICFP'11*, Tokyo, Japan, September 2011.

[44] D3.3 - Split Architecture for Large Scale Wide Area Networks. [Online]. http://www.fp7-sparc.eu/assets/deliverables/SPARC_D3.3_Split_Architecture_for_Large_Scale_Wide_Area_Networks.pdf

[45] P. V. B. Primet, A. S. Charao G. P. Koslovski, "VXDL: Virtual Resources and Interconnection Networks Description Language," in *2nd International Conference on Networks for Grid Applications*, 2008.

[46] VXDL: Virtual Private eXecution Infrastructure Description Language. [Online]. http://www.ens-lyon.fr/LIP/RESO/Software/vxdl/home.html

[47] P. Grosso, R. van der Pol, A. Toonk, C. de Laat J. van der Ham, "Using the Network Description Language in Optical Networks," in *10th IFIP/IEEE International Symposium on Integrated Network Management*, 2007.

[48] J. van Ham, P. Grosso, C. de Laat M. Ghijsen, "Towards an Infrastrucure Description Language for Modeling Computing Infrastructures," in *10th IEEE International Symposium on Parallel and Distributed Processing with Applications, ISPA 2012*, 2012.

[49] M. Baldi F. Risso, "NetPDL: An Extensible XML Based Language for Packet Header Description," *Computer Networks*, vol. 50, no. 5, pp. 688-706, 2006.

[50] Wikipedia. [Online]. http://en.wikipedia.org/wiki/Hardware_description_language

[51] M. D. Ciletti, *Advanced Digital Design with Verilog HDL*.: Prentice Hall, 2010.

[52] P. P. Chu, *FPGA Prototyping by VHDL Examples: Xilinx Spartan 3 Version*.: John Wiley & Sons, 2008.

[53] Verilog. [Online]. http://www.verilog.com/

[54] P. P. Chu, *FPGA Prototyping by Verilog Examples: Xilinx Spartan 3 Version*.: John Wiley & Sons, 2008.

[55] IEEE Standard VHDL Reference Manual. [Online]. http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=2589

[56] IEEE Standard VHDL Language Reference Manual. [Online]. http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=4772738

[57] S. Kilts, *Advanced FPGA Design: Architecture, Implementation, and Optimization*.: John Wiley & Sons, 2007.

[58] P. P. Chu, *RTL Hardware Design Using VHDL*.: John Wiley & Sons, 2006.

[59] Xilinx. Xilinx ISE 10.1 Quick Start Tutorial. [Online]. http://www.xilinx.com/itp/xilinx10/books/docs/qst/qst.pdf

[60] S. Muchnic, *Advanced Compiler Design and Implementation*.: Morgan Kaufmann Publishers, 1997.

[61] W. Brown. The Architecture of Open Source Application. [Online]. lulu.com

[62] The LLVM Compiler Infrastructure. [Online]. http://llvm.org/

[63] A. Sen, "Create a Working Compiler with the LLVM Framework," 2012.

[64] Nvidia. [Online]. http://www.nvidia.com/object/cuda_home_new.html

[65] NVIDIA Corp., "Parallel Thread Execution ISA Version 3.1," September, 2012.

[66] NVIDIA Corp., "The CUDA Compiler Driver NVCC," January, 2008.

[67] "Standard fot the C Programming Language - C99," ISO/IEC 9899:1999, May, 2005.

[68] A. Munshi et al. The OpenCL Specification. [Online]. http://www.khronos.org/registry/cl/specs/opencl-1.0.48.pdf

[69] T. Mattson, A. Munshi O. Rosenberg. OpenCL Overview. [Online]. http://haifux.org/lectures/212/OpenCL_for_Halifux_new.pdf

[70] Java SE Platform. [Online]. http://www.oracle.com/technetwork/java/javase/tech/index.html

[71] V. Bill, *Inside the Java 2 Virtual Machine*.: McGraw-Hill Companies, 2000.

[72] The JVM Specification. [Online]. http://docs.oracle.com/javase/specs/jvms/se5.0/html/VMSpecTOC.doc.html

[73] "The Java HotSpot Performance Engine Architecture,".

[74] K. Shimura , S. Matsuoka ,F. Maruyama, Y. Sohda, Y. Kimura H. Ogawa, "OpenJIT Frontend System: An Implementation of the Reflective JIT Compiler Frontend," 2000.

[75] PacketC Language. [Online]. http://www.packetc.org/

[76] CloudShield Technologies. [Online]. http://www.cloudshield.com/platform/CS4000.asp

[77] R. Duncan, D. Mulcahy P. Jungck, *packetC Programming*.: Apress, 2011.

[78] P. Jungck R. Duncan, "packetC Language for High Performance Packet Processing," in *11th IEEE International Conference on High Performance Computing and Communications*, 2009.

[79] P. Jungck, K. Ross R. Duncan, "Managing Heterogeneous Architectures for High-speed Packet Processing," 2011.

[80] R. Boutaba N. M. M. K. Chowdhury, "Network Virtualization: State of the Art and Research Challenges," vol. 47, pp. 20-26, 2009.

[81] M. Iyer, R. Dutta, G. Rouskas, I. Baldine A. Wang, "Network Virtualization: Technologies, Perspectives, and Frontiers," *Journal of Lightwave Technology*, vol. PP, no. 99, 2012.

[82] P. Szegedi et al., "Enabling Future Internet Research: the FEDERICA Case," vol. 49, no. 7, pp. 54-61, July 2011.

[83] The FEDERICA Project. [Online]. https://www.fp7-federica.eu

[84] Deliverable DSA1.1: FEDERICA Infrastructure. [Online]. http://www.fp7-federica.eu/documents/FEDERICA-DSA1.1.pdf

[85] MANTYCHORE. [Online]. http://www.mantychore.eu/

[86] MANTYCHORE: Description of Work. [Online]. http://jira.i2cat.net:8090/download/attachments/3211820/MANTYCHORE+FP7+-+DoW+-+Part+B+-+final+-+budget+removed.pdf

[87] GEYSERS. [Online]. http://www.geysers.eu

[88] GEYSERS Deliverable D2.1: Initial GEYSERS Architecture and Interfaces Specification. [Online]. http://www.geysers.eu/images/stories/deliverables/geysers-deliverable_2.1.pdf

[89] NOVI. [Online]. http://www.fp7-novi.eu

[90] NOVI Deliverable D3.1: State-of-the-Art Management Planes. [Online]. http://www.fp7-novi.eu/index.php/deliverables/doc_download/24-d31

[91] GENI Project. [Online]. http://www.geni.net

[92] VINI. [Online]. http://www.vini-veritas.net

[93] N. Feamster, M. Huang, L. Peterson, J. Rexford A. Bavier, "In VINI Veritas: Realistic and Controlled Network Experimentation," in *Proceedings of the 2006 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, Pisa, Italy, 2006.

[94] M. Grammatikou, M. Potts, P. Grosso, A. Fekete, B. Belter, M. Campanella, V. Maglaris L. Lymberopoulos, "NOVI Tools and Algorithms for Federating Virtualized Infrastructures," *to appear in Future Internet – From Technological Promises to Reality*, pp. 213-224, 2012.

[95] L. Cheng, R. Boutaba Q. Zhang, "Cloud Computing: State-of-the-Art and Research Challenges," *Journal of Internet Services and Applications*, vol. 1, no. 1, pp. 7-18, 2010.

[96] M. Sridharan et al. (2012, July) NVGRE: Network Virtualization using Generic Routing Encapsulation. Draft. [Online]. http://tools.ietf.org/html/draft-sridharan-virtualization-nvgre-01

[97] Ed. J. Davie B. Davie. (2012, August) A Stateless Transport Tunneling Protocol for Network Virtualization. Draft. [Online]. http://tools.ietf.org/html/draft-davie-stt-02

[98] M. Mahalingam et al. (2012, August) VXLAN: A Framework for Overlaying Virtualized Layer 2 Networks over Layer 3 Networks. Draft. [Online]. http://tools.ietf.org/html/draft-mahalingam-dutt-dcops-vxlan-02

[99] Nicira. [Online]. http://www.nicira.com

[100] It is time to Virtualize the Network. [Online]. http://nicira.com/sites/default/files/docs/It%20is%20Time%20To%20Virtualize%20the%20Network%20White%20Paper_5.pdf

[101] Vyatta. [Online]. http://www.vyatta.com

[102] Vyatta and SDN. [Online]. http://www.vyatta.com/learn/vyatta-and-software-defined-networks

[103] Midokura. [Online]. http://www.midokura.com

[104] G. Gibb, K.-K. Yap, G. Appenzeller, M. Casado, N. McKeown, G. Parulkar R. Sherwood, "FlowVisor: A Network Virtualization Layer," October, 2009.

[105] OFELIA Project. [Online]. http://fp7-ofelia.eu

[106] et al. S. Azodolmolky, "Optical FlowVisor: An OpenFlow-Based Optical Network Virtualization Approach," March, 012.

[107] G. Gibb, K.-K. Yap, G. Appenzeller, M. Casado, N. McKeown, G. Parulkar R. Sherwood, "FlowVisor: A Network Virtualization Layer," OPENFLOW-TR-2009-1, 2009.

[108] (2012) OFELIA Deliverable 8.2 - Requirements and Specifications for the VeRTIGO Modules. [Online]. https://alpha.fp7-ofelia.eu/redmine/projects/ofelia/repository/changes/Work%20Packages/wp8/deliverables/D8.2/D8.2-final.pdf.

[109] Sonkoly et al, "OpenFlow Virtualization Framework with Advanced Capabilities," in *EWSDN*, October, 2012.

[110] A. Story, C. Schlesinger, N. Foster S. Gutz, "Splendid Isolation: A Slice Abstraction for Software-Defined Networks," in *Proceedings of the First Workshop on Hot Topics in Software Defined Networks, HotSDN '12*, New York, NY, USA, 2012.

[111] H. Yamanaka et al. (2012, April) Open Networking Summit. [Online]. http://opennetsummit.org/pdf/nict.pdf

[112] OFELIA Control Framework. [Online]. http://fp7-ofelia.github.com/ocf/

[113] "OFELIA basic use-case document: Deliverable (internal)," 2010.

[114] E-GENI. [Online]. http://www.openflow.org/wk/index.php/E-GENI

[115] S. Sevinc, J. Lepreau, R. Ricci, J. Wroclawki, T. Faber, S. Schwab, S. Baker L. Peterson, "Slice-Based Facility Architecture," Whitepaper, draft version 1.01, 2008.

[116] Panlab. [Online]. http://www.panlab.net/testbed-repository.html

[117] AMSoil. [Online]. https://bitbucket.org/motine/amsoil

[118] FOAM. [Online]. http://groups.geni.net/geni/wiki/OpenFlow/FOAM

[119] (2012) OFELIA D8.2 - Requirements and Specifications for the VeRTIGO Modules. Deliverable. [Online]. http://www.fp7-ofelia.eu/publications-and-presentations/public-deliverables

[120] S. Sorensen. (2012) Security Implications of Software-Defined Networks. [Online]. http://www.sdncentral.com/wp-content/uploads/2012/05/Security_Implications_of_SDN.pdf

[121] R. Recio. Enterprise Data Center Security with Software Defined Networking and OpenFlow. [Online]. https://amphionforum.com/wp-content/uploads_a/2012/05/Enterprise%20DC%20Security%20with%20SDN%20and%20OpenFlow%20-%20public.pdf

[122] W. Hu, M. Hapuarachchi, T. Deshane, D. Dimatos, G.y Hamilton, M. McCabe, J. Owens J. Neefe Matthews, "Quantifying the Performance Isolation Properties of Virtualization Systems," in *Proceedings of the 2007 Workshop on Experimental Computer Science, ExpCS '07*, New York, NY, USA, 2007.

[123] G. Kumar, M. Chowdhury, A. Krishnamurthy, S. Ratnasamy, I. Stoica L. Popa, "FairCloud: Sharing the Network in Cloud Computing," *SIGCOMM Comput. Commun. Rev.*, vol. 42, no. 4, pp. 187-198, August, 2012.

[124] W. Zhou, M. Caesar, P. Brighten Godfrey A. Khurshid, "Veriflow: Verifying Network-Wide Invariants in Real Time," *SIGCOMM Comput. Commun. Rev.*, vol. 42, no. 4, pp. 467-472, September 2012.

[125] S. Shin, V. Yegneswaran, M. Fong, M. Tyson, G. Gu P. Porras, "A Security Enforcement Kernel for OpenFlow Networks," in *Proceedings of the First Workshop on Hot Topics in Software Defined Networks, HotSDN '12*, New York, NY, USA, 2012, pp. 121-126.

[126] A. Guha, Ch. Liang, R. Fonseca, S. Krishnamurthi A. D. Ferguson, "Hierarchical Policies for Software Defined Networks," in *Proceedings of the First Workshop on Hot Topics in Software Defined Networks, HotSDN '12*, New York, NY, USA, 2012, pp. 37-42.

[127] P. Porras, V. Yegneswaran, M. Fong, G. Gu, M. Tyson S. Shin, "FRESCO: Modular Composable Security Services for Software-Defined Networks," in *Proceedings of the 20th Annual Network & Distributed System Security Symposium, NDSS*, San Diego, California, USA, 2012.

[128] E. Al-Shaer, Q. Duan J. H. Jafarian, "OpenFlow Random Host Mutation: Transparent Moving Target Defense Using Software Defined Networking," in *Proceedings of the First Workshop on Hot Topics in Software Defined Networks, HotSDN '12*, New York, NY USA, 2012, pp. 127-132.

[129] Fortinet. (2007) Securing Virtual Networks. [Online]. http://www.fortinet.com/doc/solutionbrief/VirtualNetworkSecurity_sol_brief.pdf

[130] J. Snyder. (2008) Virtual Machines, Networking Security, and the Data Center: Six Key Issues and Remediation Strategies. [Online]. http://www.opus1.com/www/whitepapers/virtualization.pdf

[131] Java SE Security. [Online]. http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136007.html

[132] M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inouye, T. Hama, S. Shenker T. Koponen, "Onix: A Distributed Control Platform for Large-Scale Production Networks," in *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, 2010, pp. 1-6.

[133] M. J. Freedman, J. Pettit, J. Luo, N. McKeown, S. Shenker M. Casado, "Ethane: Taking Control of the Enterprise," in *SIGCOM Comput. Commun. Rev.*, vol. 38, 2007, pp. 1-12.

[134] T. Nandagopal, R. Ramjee, K. Sabnani, T. Woo T. V. Lakshman, *The SoftRouter Architecture*. New York, NY, USA: in Proceedings of the Third ACM SIGCOM Workshop on Hot Topic in Networks, 2004.

[135] D. Caldwell, N. Feamster, J. Rexford, A. Shaikh, J. van der Merwe M. Caesar, "Design and Implementation of a Routing Control Platform," in *in Proceedings of the 2nd Symposion on Networked Systems Design and Implementation*, 2005, pp. 15-28.

[136] G. Hjalmtysson, D. Maltz, A. Myers, J. Rexford, G. Xie, H. Yan, J. Zhan, H. Zhang A. Greenberg, "A Clean Slate 4D Approach to Network Control and Management," *SIGCOM Comput. Commun. Rev.*, no. 35, pp. 43-54, October 2005.

10 Acronyms

AM	Aggregate Manager
AST	Abstract Syntax Tree
CE	Controller Engines
CIL	Common Intermediate Language
CIM	Common Information Model
CUDA	Compute Unified Device Architecture
DAG	Direct Acyclic Graphs
DPDK	Data Plane Development Kit
DSP	Digital Signal Processor
EAL	Environment Abstraction Layer
GPU	Graphics Processing Unit
HAL	Hardware Adaptation Layer
HDL	Hardware Description Language
HRM	Hardware Resource Manager
FE	Forwarding Engines
ForCES	Forwarding and Control Element Separation
FPGA	Field Programmable Gates Array
IaaS	Infrastructure as a Service
INDL	Infrastructure and Network Description Language
IPS	Intrusion Prevention Systems
IR	Intermediate Representation
ISA	Instruction Set Architecture
JIT	Just In Time
JVM	Java Virtual Machine
LFB	Logical Functional Block
LICL	Logical Infrastructure Composition Layer
LLVM	Low Level Virtual Machine
NCP	Network Control Plane
NDL	Network Description Language
NE	Network Element
NetPDL	Network Protocol Description Language
NML	Network Markup Language
NV	Network Vitrualization
OCCI	Open Cloud Computing Interface

OFV	Optical FlowVisor
OpenCL	Open Computing Language
OPN	Optical Private Network
PTX	Parallel Thread Execution
RDF	Resource Description Framework
RM	Resource Manager
RTL	Resistor-Transistor Logic
SDN	Software Defined Networking
SSA	Static Single Assignment
VHDL	Very High Speed Integrated Circuits Hardware Description Language
VM	Virtual Machine
VON	Virtual Optical Network
VXDL	Virtual Resources and Interconnection Networks Description Language
XML	eXtensible Markup Language