UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Escola d'Enginyeria de Telecomunicació
i Aeroespacial de Castelldefels

BACHELOR'S DEGREE THESIS

TITLE: Design, implementation and evaluation of Synchronous Ethernet in an SDN architecture

DEGREE: Bachelor's Degree in Network Engineering

AUTHOR: Raúl Suárez Marín

ADVISORS: Sebastià Sallent Ribes
David Rincón Rivera

DATE: April 30, 2015

Título: Diseño, implementación y evaluación de Synchronous Ethernet en una arquitectura SDN

Autor: Raúl Suárez Marín

Directores: Sebastià Sallent Ribes
David Rincón Rivera

Fecha: 30 de Abril de 2015

Resumen

Muchos servicios de telecomunicaciones tienen estrictos requisitos de sincronización. La telefonía celular (3G, 4G/LTE) es un claro ejemplo, en el que las estaciones base necesitan una sincronización en frecuencia estable y de alta precisión para obtener las frecuencias portadoras y gestionar el acceso al medio de los terminales, ya que los recursos se comparten tanto en tiempo como en frecuencia. De la misma forma, se debe coordinar el *handover* de terminales entre celdas adyacentes. El inminente despliegue de redes 5G, con una mayor densidad de celdas y tasas de transmision más elevadas requerirá una mayor precisión en la sincronía.

Dos tecnologías se perfilan como la solución a los retos mencionados. Por un lado, Synchronous Ethernet (SyncE) es una arquitectura basada en elementos Ethernet, caracterizados por su precio económico, madurez y escalabilidad, que proporciona la distribución de un árbol de sincronización sobre una red mediante un intercambio de mensajes de control. En cuanto a las características de la sincronía, las capacidades de los nodos SyncE son similares a los nodos SDH/SONET. Por otro lado, SDN es un nuevo paradigma en redes en el cual se abstraen las funciones de encaminamiento, desacoplándolas del plano de datos. El control de la red se mueve hacia el controlador, una entidad centralizada que maneja la configuración de la red mediante reglas a traves de una interfície *southbound* como lo es OpenFlow. La arquitectura SDN abre nuevas posibilidades como la virtualización de funciones de red, la optimización global de operaciones de red y la reducción de costes de operación.

En este trabajo se proponen extensiones a OpenFlow para poder desplegar una red SDN habilitada para utilizar SyncE. Para ello, se analizan las dos arquitecturas (SyncE y SDN) y, teniendo en cuenta las reglas de diseño de ambas, se ha realizado una integración de SyncE en SDN. Además, se ha creado una aplicación para el controlador que implementa toda la inteligencia de SyncE, nuevas características (ahora posibles gracias al modelo centralizado) y una API que permite visualizar y controlar parámetros relacionados con SyncE. Las extensiones y modificaciones propuestas se han desplegado y testeado sobre un entorno real de laboratorio con unos resultados positivos.

Title: Design, implementation and evaluation of Synchronous Ethernet in an SDN architecture

Author: Raúl Suárez Marín

Advisors: Sebastià Sallent Ribes
David Rincón Rivera

Date: April 30, 2015

Overview

Many of today's telecommunications systems rely on strict timing and synchronization requirements. This is the case of cellular telephony (3G, 4G/LTE), where base stations need accurate and stable frequency clocks in order to obtain their carrier radio frequencies, arbitrate the frequency- and time-shared access of terminals, and coordinate the handover of terminals between adjacent cells. The imminent deployment of 5G networks, with a much higher density of cells and speed, will further increase the need for synchronization.

On one hand, Synchronous Ethernet (SyncE) is a well-known, cheap, scalable Ethernet data plane, with the addition of special messages that convey synchronization information. Regarding timing, SyncE nodes' capabilities are similar to those of SDH/SONET. On the other hand, SDN is a new networking paradigm that provides an abstraction of the forwarding function, decoupling the data plane from the control plane. The control of the network is moved to the controller, an external centralized entity that manages the network configuration based on policies through a southbound interface, for example OpenFlow. SDN architecture opens the way to the virtualization of network functions, the global optimization of network operations, and reduces operational costs.

This degree thesis proposes extensions of the OpenFlow protocol in order to deploy SDN-enabled Synchronous Ethernet networks. For that, SyncE and SDN architectures have been analysed, and the design rules of both technologies have been maintained. Moreover, an application has been developed for the controller so that it implements the intelligence of the SyncE architecture, new features (now possible thanks to the centralized model) and an API that allows to know and manage information regarding SyncE. The proposed extensions and modifications have been deployed and tested in a real environment.

Este trabajo va dedicado principalmente a mis padres Francisco Suárez y Conchi Marín, a mi hermana Inés y a mis abuelos, que siempre han estado junto a mi y que me han dado siempre todo y más, pero sobretodo porque han permitido que esto sea posible.

A Sebastià Sallent y a David Rincón, por ser unos excelentes docentes y mis mentores, que me han aconsejado y ayudado durante el proyecto.

A Xavier Gómez, por ser un gran amigo que siempre ha estado a mi lado.

A todos vosotros, ¡Muchas gracias!

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF CODES

CHAPTER 1. INTRODUCTION

Many of today's telecommunications systems rely on strict timing and synchronization requirements. This is the case of cellular telephony (3G, 4G/LTE), where base stations need accurate, stable frequency clocks in order to obtain their carrier radio frequencies, arbitrate the frequency- and time-shared access of terminals, and coordinate the handover of terminals between adjacent cells [1]. The imminent deployment of 5G networks, with a much higher density of cells, will further increase the need for synchronization.

Currently, circuit-based, time-division multiplexing (TDM) transmission technologies such as SDH/SONET provide clock distribution over the data transmission plane, by defining a tree-based hierarchy of clocks. Clocks differ in frequency and phase accuracy, and holdover stability, and can be classified in different strata or quality levels. The Primary Reference Clock (PRC), at the top of the synchronization network tree (stratum 1), is the master reference to which other, lower-quality clocks (stratum 2, 3, ...) lock and correct their inherent frequency drift. By designing a synchronization network that guarantees the traceability of any equipment's clock to the PRC, the line clock of TDM signals that feed the telecommunications equipment (usually E1 or T1 interfaces) ensures the required synchronization at the edge equipment.

However, telecommunications operators are migrating from circuit-based, time-division multiplexing (TDM) transmission systems to packet-switching technologies, due to the inherent advantages of the latter approach (higher flexibility, lower operation costs, economies of scale, and better integration with higher layer IP-based services, among others). Again, this is the case of 4G/LTE/5G networks, with their "all-IP" architecture. This raises a question: is there any technological solution able to integrate both packet switching and synchronization distribution capabilities?

Synchronous Ethernet (SyncE) is such a technology: the well-known, cheap, scalable Ethernet data plane, with the addition of special messages that convey synchronization information. Regarding timing, SyncE nodes are similar to SDH/SONET nodes: the reference clock is obtained from the signal received from a specific input port and it is used to correct the local clock. The regenerated timing is applied in the signals transmitted over the output ports [2]. Nodes exchange Synchronization Status Messages (SSM) in order to identify the quality of the clocks, and thus deciding (in a distributed way) the best topology for the tree-like clock chain.

In the networking landscape, Software-Defined Networking (SDN) has recently emerged as a new network management paradigm. SDN separates the control and data planes and introduces a centralized controller, an external centralized entity that manages the network configuration and forwarding functions based on policies defined by the network operator. [3]. The controller communicates with simple, cheap switching nodes through standardized interfaces, being OpenFlow (OF) the most popular [4]. Then, SDN turns networks into programmable networks where switching decisions are based on flows and not on the destination addresses [3]. It is now possible to dynamically or automatically configure or reconfigure the orchestration of IT infrastructures from the network up to applications combining the SDN with network function virtualization (NFV) [5]. This centralized architecture opens the way to the virtualization of network functions, the global optimization of network operations and reduces operational costs. The convergence of wired, wireless and cellular technologies is enabling the emergence of fully programmable IT infrastructures, but

in order to meet these challenges, SDN still has to solve some open issues. A critical characteristic in the new 5G architecture technologies is its synchronous nature, and that poses some challenges in the design of packet-switched access backbone networks.

Currently all the procedures and functions performed in the SDN architecture are asynchronous. Therefore, there is a need to extend SDN and Openflow, so that the data plane can handle synchronous forwarding services, and the control plane can manage synchronization-related parameters and statistics. While the former is currently a challenge in synchronous, time-slotted technologies (such as some access networks), the latter is the challenge to solve in SDN-enabled SyncE networks.

Given the growing presence of SyncE equipment, and the possibilities that SDN opens, we propose extensions of OpenFlow with additional capabilities to manage the synchronization plane of SyncE networks. Vendors already offer equipment that is both SyncE- and OpenFlow-compliant (for an example see [6]), but the control plane is still unable to manage the synchronization plane in an integrated way. We envision a centralized control plane that is not only capable of managing the forwarding operations and the routing, but also the clock distribution tree, reacting in a coordinated way when the traceability of the PRC is lost or when rearrangements of the synchronization plane are needed. This proposal not only covers the requirements of the SyncE architecture, but also constraints such as the maximum number of devices in a synchronization chain, as well as alerts for downtimes and automatic configuration of synchronization trees. Additionally, it is compatible with non-SDN networks and legacy SDH equipment, thus ensuring its practical applicability in real, incremental network deployments.

There are few works in the literature that tackle goals similar to this. We are aware of other efforts regarding clock synchronization in the SDN scenario [7] and a proposal of OpenFlow extension from the same authors [8], but the goal of this project is different: while they intend to design a centralized implementation of the (inherently distributed) Precision Time Protocol (PTP) [10] in SDN networks by using the controller to perform the complex calculations that nodes should do, the goal here is to extend the Operation, Administration and Management (OAM) features of OpenFlow. To the best of our knowledge, this is the first work to propose SyncE extensions for OpenFlow.

This project discusses how SDN and SyncE can interoperate, proposes OpenFlow SyncE extensions, and presents results from an implementation. The rest of the work is organized as follows. Chapters 2 and 3 review Synchronous Ethernet and Software-Defined Networking architectures respectively. Chapter 4 presents a high-level functional description of the SDN-related operations in SyncE networks, followed by the details. Chapter 5 presents the implementation and results. The document ends with conclusions and future lines of research in chapter 6.

CHAPTER 2. SYNCHRONOUS ETHERNET

This chapter describes the architecture, messages and operations of Synchronous Ethernet (SyncE), in order to provide the reader with the background to understand how SyncE can be integrated in the SDN architecture.

2.1. Basics of SyncE

Synchronous Ethernet has been defined and standardized by the ITU in order to extend the Ethernet layer to add frequency synchronization without loosing compatibility with native Ethernet devices. SyncE is described in ITU-T G.8010 [11] where two different layers, ETH and ETY, are distinguished. The ETY layer represents the physical layer as defined in IEEE 802.3 [9], while the ETH layer corresponds to the pure packet layer. In other words, the ETY represents the PHY (physical) layer and the ETH the MAC (Medium Access Control) layer of the OSI model. ITU defines three ways for transmitting synchronization in a network as illustrated in figure 2.1:

1. MESSAGE timing flow: The synchronization is obtained from the messages exchanged between nodes. These messages belong to the application layer. Some examples are NTP and PTP protocols.

2. SERVICE timing flow: The synchronization relates to a certain service, for example PDH. PDH uses stuffing bits in order to maintain synchronization in a given stream. This synchronization relates to the PDH service because even if there are several PDH streams in an SDH STM container, each one is synchronized independently.

3. PHYSICAL timing flow: It is the lowest level synchronization and can be used for synchronizing devices. The synchronization is carried in the transmitted signal (bit transitions).

SyncE and SDH use physical timing flows to synchronize their own clocks, and then they can synchronize their ports. This synchronization is traceable to other devices so a master device can synchronize the whole network.

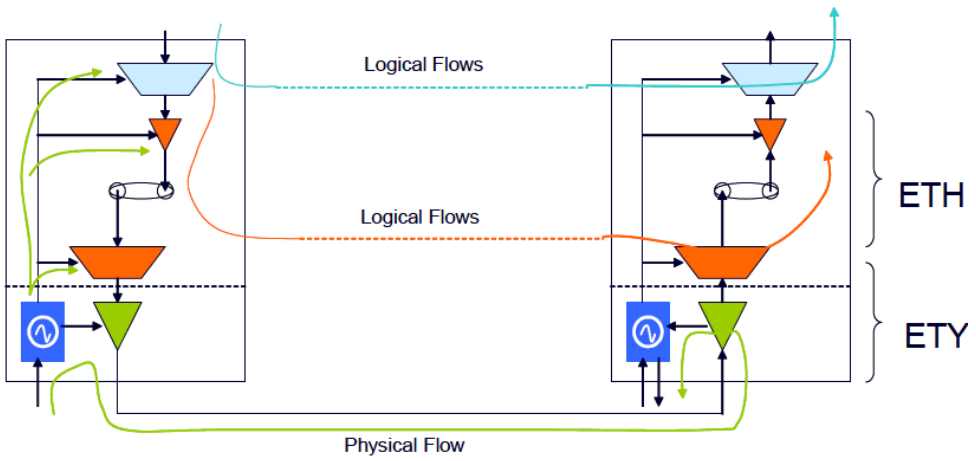

Figure 2.1: ITU G.8264 Figure I.1 Synchronization flow types [12].

A key issue for SyncE is inter-networking between SDH and SyncE equipment in order to manage an unified synchronization network. The mechanisms to ensure that are found fundamentally in three ITU-T recommendations: the SyncE architecture in G.8261 [13], performance and clock characteristics in G.8262 [14] and the messages between SyncE nodes in G.8264 [12].

2.2. Architecture of Synchronous Ethernet

Synchronous Ethernet is raised as an SDH evolution of packet-switched networks, and as it is based on SDH technology, both must maintain compatibility. In a Synchronous Ethernet network some nodes defined as synchronous, and others are defined as asynchronous (with a clock accuracy of $\pm$ 100 ppm and do not take part in the synchronization network). The synchronous devices rely on a local clock called Ethernet Equipment Clock (EEC) with an accuracy better or equal than $\pm$ 4.6 ppm. This quality is defined with a quality level (QL) level of QL-EEC1 (for E1 interfaces) or QL-EEC2 (for T1 interfaces). If the device has a better clock attached, its quality level is improved; for example, if it is attached to a Primary Reference Clock (PRC), its quality will be QL-PRC, or if it is attached to a Synchronization Supply Unit (SSU), it quality will be QL-SSU-A/B. The list of possible QL can be found in Annex A.1. Synchronous and non-synchronous SyncE devices must be compatible in order to maintain continuity of data. This means that, although SyncE elements nominally have an accuracy of $\pm$ 4.6 ppm, receivers must also be able to operate at $\pm$ 100 ppm. It is also a requirement that SDH and SyncE networks are compatible, so there can exist an unified synchronization network. For this reason, SyncE equipment has to support STM-N communications as well for reading clock quality readings.

When a device is synchronized, this synchronization can be traced to other devices in the network, enabling the creation of a synchronization network (figure 2.2). For that, a device detects the QL of the signals coming from adjacent nodes and decides through a selection process which is the best signal to be synchronized with. If a SyncE device is not attached to any clock, it relies on a $\pm$ 4.6 ppm local clock that can be synchronized by an external clock when the incoming quality is better.

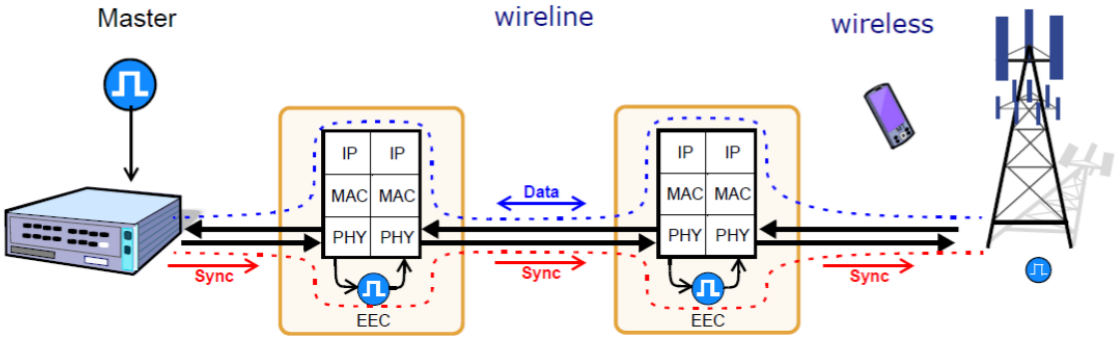

Figure 2.2: Synchronization and traceability of signals in SyncE networks.

A logical synchronization tree composed only by synchronous devices is formed being the root the best quality device (usually QL-PRC), and following a chain of synchronized devices till the end of the tree (branches). This tree has to be formed following certain constraints defined in recommendation ITU-T G.803 [15]:

1. **Maximum chain of 20 consecutive EECs**: SyncE equipment has a default EEC clock (Ethernet Equipment Clock, G.8262). This clock has a defined accuracy of $\pm$ 4.6 ppm in free-run mode. In order to maintain jitter and wander[1] values within the limits, SDH requires that over the synchronization network it is maintained a maximum of 20 consecutive devices with G.8262 based clock. In case that the chain should be extended, the next clock must have a better quality (based on an SSU-A/B or a PRC).

2. **Maximum of 60 EECs in a single chain**: SDH mandates not to have more than 60 EEC clocks in the same chain, no matter if there are G.812 quality clocks in between or not.

3. **Maximum of 10 SSUs in a single chain**: SDH mandates not to have more than 10 SSU clocks in the same chain.

Although the data plane topology can follow ring, tree or mixed topologies, it must be ensured that the logical topology of the synchronization network is always a tree (figure 2.3). Otherwise, there can be synchronization loops, leading to instability. Some messages and operations have been defined by the ITU-T in order to enforce this requirement. It is a common practice for network operators to have a backup PRC in case the main PRC fails. In this case, the network operator must engineer the possible synchronization trees and configure carefully every device so a synchronization loop never appears.

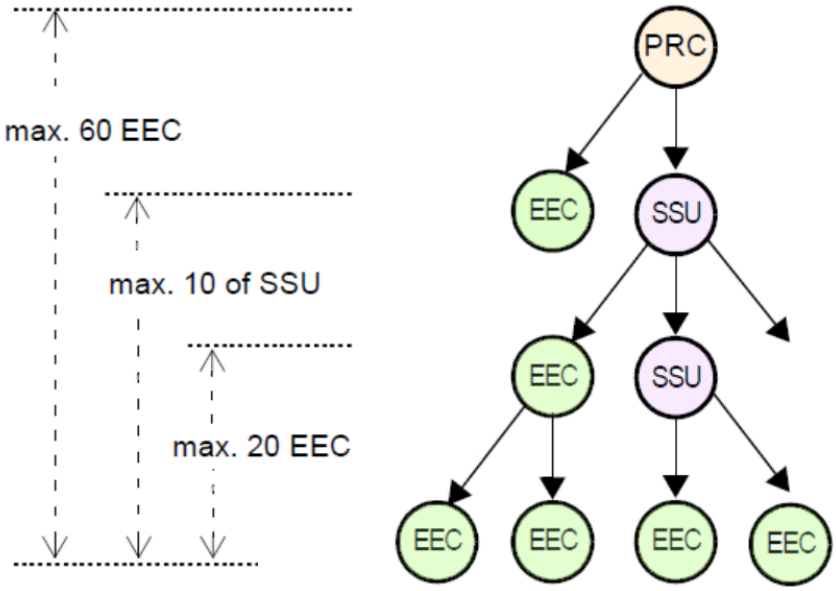

Figure 2.3: Synchronization network model for SyncE [16].

[1] A phase fluctuation of a signal is an oscillating movement with an amplitude and a frequency. If this frequency is more than 10 Hz, it is known as jitter, and when it is less than that, it is called wander.

2.3. Distribution of messages

As explained in subsection 2.1, a SyncE device can synchronize its own clock with incoming signals from adjacent nodes. However, nodes need to know what is the QL of their adjacent nodes in order to compare it with its own QL and if they are allowed to use it (i.e. for avoiding synchronization loops).

For that, an Ethernet Synchronization Message Channel (ESMC) is established between adjacent synchronous devices. This channel relies on the Organization Specific Slow Protocol (OSSP) specified in IEEE 802.3ay and is used to exchange Synchronization Status Messages (SSM) between nodes. ESMC PDUs are generated as a heart-beat (usually at a rate of 1 packet/second) but they can also be generated as result of an asynchronous event (loss of synchronization signal, change of QL level, ...).

The ESMC PDU has several fields; nevertheless, most of them carry constant SyncE-related (but not QL-dependent) values. The QL-dependent fields are: event-flag, which determines if the ESMC PDU has been event generated or not; and QL TLV, which contains the SSM code of the current QL of the node that generates this message.

Figure 2.4 shows the ESMC PDU. The first 14 bytes are the MAC header (in yellow), followed by the OSSP header (4 bytes, in green). The next 6 bytes are the ESMC fields (in blue) followed by 4 bytes of the ESMC QL TLV fields (in orange). Note that are some fields tagged as reserved; these fields are filled with zeros. Note also that at the end of the ESMC PDU there are 36 padding bytes in order to fulfil a minimum packet size of 64 bytes.

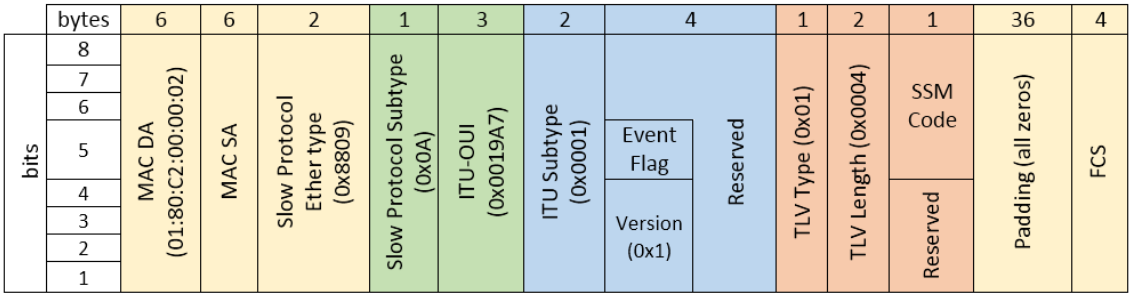

Figure 2.4: ESMC PDU frames between adjacent nodes.

2.4. Operations

Before describing the operations that a SyncE device must implement, the available configuration parameters are described:

- Synchronous mode: ports can be configured to work in either non-synchronous or synchronous operation mode. A non-synchronous port is not a candidate port for synchronizing the node and does not process the ESMC.
- Port priority: ports can be prioritized over others. Priority is set in a value range of [0-65535] being 0 the lowest and 65535 the highest priority [17].

- QL-ENABLED flag: this flag is set to true if the device should consider the QL of the adjacent nodes and false if not.
- signal-fail flag: this flag is defined for each port and is set when the port is not connected, or in case of problems detected in the upper layers. However, if QL-ENABLED is set to true, instead of using signal-failed flag a QL value of QL-FAILED is achieved. In order to avoid intermittent signal fail information, a hold-off timer must expire before applying any operation to that port.

The operations explained below are only applied to synchronous ports.

2.4.1. Generation of ESMC PDU

A node must generate an ESMC PDU periodically as a heart-beat. The time interval between messages is set by the OSSP with a maximum of 10 frames per second, but it is usually configured to 1 frame/second [12]. In this message, the device announces its neighbours its own QL. However, if node A is synchronized through node B, node A will report a quality level of QL-DNU (Do Not Use) to node B in order to avoid synchronization loops.

In case of a sudden QL change, an event-generated ESMC PDU is transmitted. Moreover, when the QL of a port changes, the selection process is triggered and depending on its result, different time constraints are defined for the generation of ESMC PDUs[17]:

1. No port is available: the device announces its native QL in a time T_{HM} = [300, 2000] ms.
2. Synchronization port is maintained: the device announces its QL in less than T_{NSM} = 200 ms. This ensures that other nodes can decide whether to change their synchronization port or not.
3. Synchronization port is changed: the device announces its QL in a time T_{SM} = [180, 500] ms. This ensures that other nodes can decide whether to change their synchronization port or not.

2.4.2. Reception of ESMC PDU

Upon reception of an ESMC PDU, if the flag QL-ENABLED is set, the SSM code is checked. If the QL does not change (is the same as the previous message), nothing happens and the node waits for the next packet. However, if the QL changes, the node goes through a selection process and a new synchronization port could be chosen. Afterwards, an event-generated ESMC PDU must be transmitted to notify this change to adjacent nodes. Otherwise, if an ESMC PDU is not received for more than 5 seconds (configured in the Information Timer, IT) on a certain port, the QL value of the adjacent node is to be considered QL-DNU. In this case, this port is removed from the candidates ports list of the selection process during a wait-to-restore (WTR) time. The wait-to-restore timer can be configured in a range value of 0 to 12 minutes, but it is usually configured at 5 minutes. Afterwards, the port is added again to the candidates ports list of the selection

process. Figure 2.5 shows the machine state diagram of the operations triggered by the reception of an ESMC PDU.

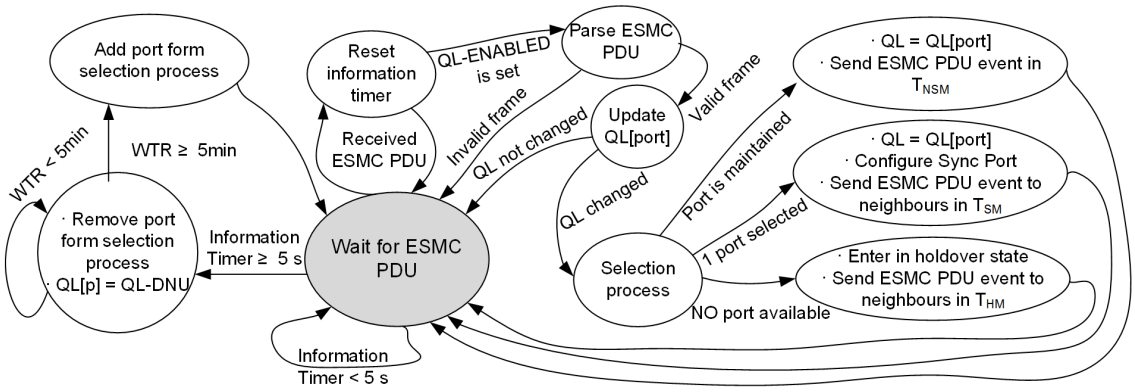

Figure 2.5: Machine state diagram of ESMC PDU reception.

2.4.3. Defect in a synchronization source

Synchronous Ethernet must be able to detect failures in the device, for example detecting ports that cannot be used anymore to synchronize the main clock because something happened (i.e. defects detected in upper layers, or the signal gets disconnected). In that case, the port reaches a QL-FAILED/signal-fail state if the defect is maintained during a hold-off time, which is a fixed value between 300 ms and 1800 ms [17]. Afterwards, this information is notified to the selection process for banning the affected port(s) from the candidates list. After WTR time, if the failure has been fixed, it is notified to the selection process so this port can now be a candidate for synchronizing.

2.4.4. Selection process

The selection process is triggered when a change of QL is detected. This algorithm decides which port is the best in terms of priority and quality. For that, the next steps are followed:

1. If QL-ENABLED is set, order valid ports in terms of QL (from high to low).
2. Delete ports in QL-FAILED or signal-failed state.
3. Order ports by priority.
4. Get best port(s).
    - If there are no ports available, the device gets in holdover state (synchronized with its own EEC clock).
    - If there is only one port available, that port is chosen.
    - If there is more than one port available, maintain the port that is currently being used if it is possible; otherwise, choose randomly. If there is the need to synchronize an external SSU clock attached to the device, synchronize it with one of the remaining ports.

2.5. Summary

This chapter has described how Synchronous Ethernet works. To summarize, a SyncE network is formed by asynchronous and synchronous devices. Synchronous devices are able to synchronize their own clocks by means of physical timing flows (bit transitions of data) and with that, synchronize their ports as well. This synchronization is traceable to other devices in the network so at the end the whole network is synchronized by a master device (usually a PRC). There is a permanent exchange of messages between nodes announcing their QL, and allowing other nodes to execute the selection process and choose the best synchronization source. The resulting topology is a tree with no synchronization loops because of the operations performed by each node. Synchronization loops can appear while reconfiguring the network; that is why network operators have to engineer what are the different possible synchronization trees and configure carefully every device (QL-ENABLED flag and port priority) so that synchronization loops never appear. This exercise can be costly for networks of thousands of devices.

CHAPTER 3. SOFTWARE-DEFINED NETWORKING ARCHITECTURE

This chapter describes the basics of Software-Defined Networking (SDN) and its singular components: Open vSwitch as an infrastructure layer, OpenDaylight as a control and applications layer, and OpenFlow as the communication interface between both planes.

3.1. Software-Defined Networking

Software-Defined Networking (SDN) is a novel network architecture that is dynamic, manageable, cost-effective and adaptable, seeking to be suitable for the dynamic high-bandwidth nature of today's applications [18]. SDN decouples network control and forwarding functions, allowing the underlying infrastructure to be abstracted from application and network devices and let these resources be (programmatically) managed by a centralized entity, hence letting the network operators to dynamically adjust network-wide traffic flows to meet changing levels. SDN is open standard-based and vendor-neutral simplifying network design and operation because the instructions are provided by a control plane instead of multiple vendor-specific devices and protocols. SDN is based on 3 layers (fig. 3.1):

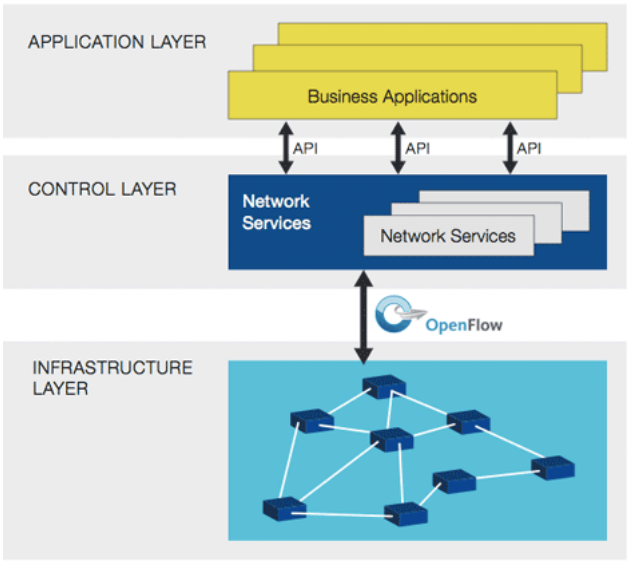

Figure 3.1: SDN Architecture

- Infrastructure layer: It is contained in the data plane and has network elements that will follow the rules of the control layer. The network elements are simple packet forwarding nodes without embedded control and decision-making capabilities.
- Control layer: It is contained in the control plane and involves the hypervisors[1] and the controller, that is in charge of configuring the devices in the forwarding plane in

[1] A hypervisor or virtual machine monitor (VMM) is a piece of computer software, firmware or hardware that runs and manages virtual machines.

order to implement network services that are defined in this same layer. This configuration is done through standardized protocols (i.e. OpenFlow) which are designed for communicating controllers and network devices' forwarding planes.

- Application layer: It contains the applications that control the network as if the network itself was a computer program. These applications communicate with the control layer through APIs defined in the controller, and the controller will manage the reconfiguration of the network to follow the application's rules.

A well-defined programming interface between the network elements and the SDN controller separates the control plane and the data plane. This southbound interface conveys the communication and management protocols, and the instruction set of the forwarding devices between both entities, providing interoperability between heterogeneous network devices.

3.2. OpenFlow

OpenFlow (OF) [19] is a communications protocol developed by the Open Networking Foundation (ONF) that works as a southbound interface in an SDN environment. The SDN/OpenFlow forwarding device is based on a pipeline of flow tables that handle rules, execute actions on matching packets, and keep statistics of matched flows using counters. Each rule matches a subset of packets and performs an action like, forwarding to an assigned output port, drop packets or modify header fields, among other typical operations. The OF protocol is currently divided in two parts: a wire protocol (currently version 1.5) and a configuration and management protocol OF-config (currently version 1.2). OF current versions can match the Ethernet, IPv4, IPv6, TCP/UDP, MPLS, VLAN tagging, and PBB fields of Ethernet packet. OF 1.4 is also able to control optical port parameters and statistics.

The control plane and the forwarding plane can communicate in three different ways [20]:

1. **Controller-to-Switch**: This communication is initiated by the controller and may require response from the device. Messages in this category are for initialization and configuration purposes. Some examples are OFPT_FEATURES_REQUEST (requests the features of the switch), OFPT_FEATURES_REPLY (reply from the previous request), OFPT_SET_CONFIG (sets configuration parameters in the switch), OFPT_FLOW_MOD (modifications to the flow table), OFPST_DESC (statistics regarding the device), OFPT_PACKET_OUT (sends a packet from the controller to the switch), among others.

2. **Asynchronous**: This communication is generated by the device in the forwarding plane without the controller querying. Messages in this category are intended for events. There are four types of events: 1) a flow has been removed or has expired (OFPT_FLOW_REMOVED); 2) a port has changed its status (i.e. the port is down now); 3) a packet does not match any flow entry and the devices does not know what to do, so it sends this packet encapsulated in a OFPT_PACKET_IN packet to the controller so it can decide; and 4) any error message.

3. **Symmetric**: This communication can be started by any device without solicitation. This category comprises *Hello*, *Echo*, *Error* and *Experimenter* messages.

These communications are held in TCP/TLS connections through port 6633, so delivery and order are guaranteed for main connections. The forwarding plane is also able to create auxiliary connections with the controller to improve its processing performance by exploiting parallelism. In these auxiliary connections, the transport protocol can be either TLS, TCP, DTLS or UDP. In case of UDP and DTLS, the delivery and ordering of packets is not guaranteed so other mechanisms should be implemented.

For more details about OpenFlow and its messages, refer to the OpenFlow Switch Specification 1.0.0 [20] and the OpenFlow overview in Appendix C.

3.3. Open vSwitch

Open Virtual Switch (Open vSwitch, OVS) [21] is a virtualized switch located in the data plane, licensed under Apache 2.0. Open vSwitch enables effective network automation through programmatic extensions, while still supporting standard management interfaces and protocols. It is also designed to support distribution across multiple physical servers in a way that makes the underlying architecture transparent, in a similar way that a hypervisor does with the operating system and the hardware. It supports OF and it is typically used with hypervisors to interconnect virtual machines within a host or virtual machines between different hosts across a network. It is a critical piece in a SDN solution. The current stable version is the Open vSwitch 2.3.0 released on August 14th, 2014 [22].

An Open vSwitch includes multiple flow tables that contain a set of flow entries, each of them comprising match fields, priority, counters, and a set of instructions to apply to matching packets. Instructions associated with each flow entry either contain actions or modify the processing of the pipeline (jumping from one flow table to another, in sequence). When the processing pipeline does not specify any next table, the packet is usually modified and forwarded, ash shown in figure 3.2. The forwarding options are diverse, from forwarding the packet to a physical and/or logical port, to sending the packet to the controller or flooding the packet through several ports. There is also the possibility of not using 'OpenFlow forwarding' but legacy switch forwarding rules.

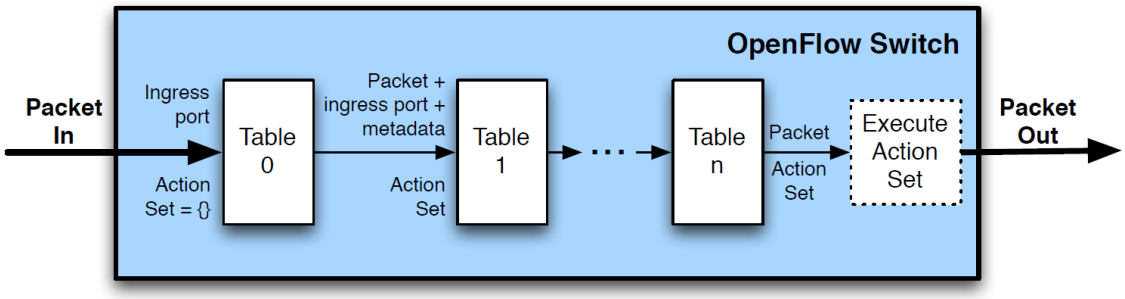

Figure 3.2: Conceptual view of the forwarding plane of an OpenFlow Switch.

Regarding performance, Open vSwitch has a flexible ovs-controller in user-space and a fast datapath in the kernel (figure 3.3). Incoming packets are managed as flows and are processed as follows:

- First, the packet arrives to the datapath (1).
- As the Open vSwitch has no flow entry that matches the packet, the packet is sent to the controller using an OFPT_PACKET_IN message (2).
- As a response, an OFPT_PACKET_OUT message is received which tells to the OVS which flow entries should be installed matching that packet so the packet can be forwarded (3). Till now only the user-space has been used.
- Later, when other packets matching the same flow entry arrive to the switch, as the flow entry is already installed (if not expired), the forwarding is managed by the datapath in the kernel, performing operations (1) and (4) faster.

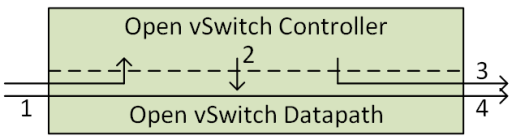

Figure 3.3: Treatment of incoming packets in Open vSwitch.

The Open vSwitch is composed of three different modules, two of them in user-space and one of them in kernel space:

1. **ovsdb-server**: This is the database that makes persistent the configuration of the system.
2. **ovs-vswitchd**: This is the core component of the device. It can establish communications with: the controller by using OpenFlow, the ovsdb-server by means of a management protocol, and with the kernel through netlink.
3. **openvswitch_mod.ko**: This module manages forwarding and tunnelling. It stores the exact-match cache of flows and has been designed to be fast and simple. This module does not manage any information related with OpenFlow so, although flow tables are stored here, the flow-entry expiration is not managed.

The key feature of Open vSwitch is not only that it supports OpenFlow, but also that it allows multiple switches to be running at the same time within the same host machine. This enables the ability of deploying any topology virtually.

3.4. OpenDaylight Controller

OpenDaylight (ODL) is a controller in continuous development by its community, and is one of the most supported within the industry. Companies in the support list include Cisco, Brocade, HP, IBM, Dell, Juniper, Microsoft and Huawei, among others [23]. The vision of ODL is to build a **modular** open source controller with a well published northbound API for network applications while utilizing southbound protocols such as OpenFlow for SDN to communicate with network elements. The OpenDaylight controller is pure software and programmed in Java (so it can run in any operating system with a Java Virtual Machine). The structure of ODL is the one shown in figure 3.4. All the modules or functionalities of OpenDaylight are programmed as plugins and are installed automatically by the OSGi framework when it is started. This controller supports running in an environment with backup controllers in case there are incidences [24].

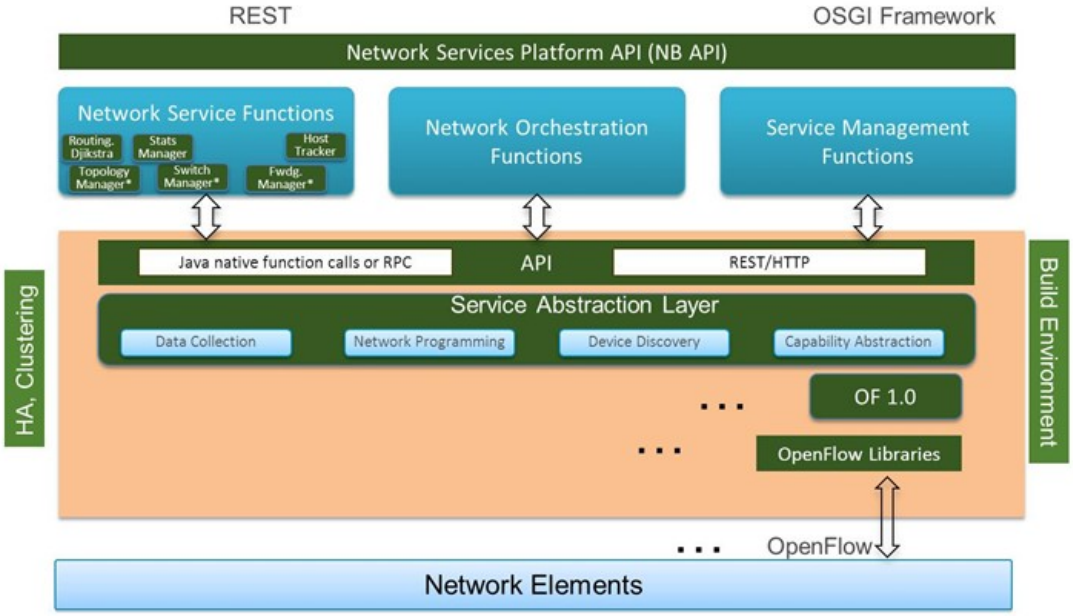

Figure 3.4: OpenDaylight structure for OpenFlow 1.0.

On the southbound interface, the controller can support multiple protocols such as OpenFlow 1.0 (as well as OpenFlow 1.3), BGP-LS and others; each one independent from the other. As of April 2015, OpenFlow 1.0 and 1.3 are implemented, but version 1.0 is the one fully functional within the controller. The southbound interface is dynamically linked into a Service Abstraction Layer (SAL) so it can be used by the application layer. In computing, the function of an abstraction layer is to hide the implementation details of a particular set of functionalities. In this case, the SAL exposes those services that have been implemented in the northbound interface to the southbound interface, and vice-versa. The SAL manages the fulfilment of the service that is requested from the application, independently of the underlying protocols that communicate with network devices. This decoupling ensures of investment protection to applications developers. For the controller to manage the devices in its domain, it needs to know some details about them such as their capabilities, reachability, topology and state, among others. This information is managed and stored by the plugins in the Network Service Functions (NSF) block, such as the Topology Manager, the Switch Manager, the Forwarding Rules Manager... These modules expose their func-

tionalities and can trigger events to other plugins (for example, an event related with a new node in the network). This functionality has been widely used in this work.

The OpenDaylight controller exposes open northbound APIs which are used by network applications. Currently, in ODL version 1.0.2, the controller supports two types of communications: OSGi and REST. OSGi is a command line communication implemented for applications that runs in the same address space as the controller does, while REST (a web based communication interface) is used for applications outside the address space of the controller and allows to the network manager to use a control panel in the web service of OpenDaylight.

3.5. Summary

This chapter has described the elements of an SDN architecture. The main novelty of this architecture is that it decouples the data plane (forwarding of packets) and the control plane (decision-making capabilities). At the top of the SDN architecture, a set of applications are running on the controller, which communicate with the control layer through APIs. The control layer then configures the network devices (located in the data plane) through a southbound interface. There are several protocols that can be used for the southbound interface. Currently, Openflow is the most popular. In this case, the data plane is formed by OpenFlow switches, which treat packets as flows. If a packet matches a flow entry (rule installed by the controller), some actions are performed to that packet.

The main advantage of SDN is that the configuration and intelligence of the devices is moved to a single central entity, thus having a centralized view of the network. In legacy networks, the devices only had information of themselves and their neighbours; in SDN networks the controller knows everything, so the decisions are more effective and efficient, and the management of the network is cleaner and easier. Also, as devices in SDN networks do not have decision-making capabilities, they are cheaper.

CHAPTER 4. SDN-ENABLED SYNCE ARCHITECTURE

This chapter describes our proposal for adapting SyncE to the SDN architecture. First of all it is described the proposed architecture followed by the description of the environment used during the development. After the main ideas are presented, the implementation is described.

Implementing SyncE in SDN networks is important mainly because of two reasons:

1. It allows network operators to introduce synchronization in new SDN network deployments.

2. It eases network management in SyncE networks. Having a centralized entity that manages the network makes that the configuration of every device is located in the controller, and allows a more efficient and faster management and configuration of the synchronization tree. Moreover, there is no need that the synchronization trees are engineered in order to configure the switches adequately for avoiding synchronization loops, as well as there is no need on taking into account topology limitations. These requirements can be programmed in the application that controls the synchronization in the network.

4.1. Proposed architecture

In order to deploy SyncE in an SDN environment, the main design directives of SDN and SyncE must be respected. As seen in previous sections, in Synchronous Ethernet there are network elements (NEs) that get synchronization signals from traceable clocks. In order for the SyncE NE to decide which port to use, there is an exchange of ESMC PDU messages between nodes, and depending on the QL some algorithms are triggered. Also, in order to limit the convergence time of the network, there is a set of time requirements that determine when other elements in the network should be notified of QL changes.

When implementing an SDN-enabled SyncE network, it must be considered that switches are only able to perform OpenFlow operations (install flow entries, perform forwarding, ...), read statistics (i.e. port statistics) and configure switches (i.e. bring ports up or down). Hence, switches are only able to read quality level values and report them as statistics, to configure synchronization ports and to install flow entries. Other operations must be implemented by the controller. As the SyncE-related operations are implemented in the SDN controller (QL management, selection process, ...), there is no need to notify to other switches about QL updates, hence there is no need of ESMC PDU exchange anymore. **However, this communication is required when the nodes are in different, non-coordinated environments**[1] (SDN_{env1} and SDN_{env2} or SDN and non-SDN). For that, flow entries are used in order to automate the circulation of an ESMC PDU if there are no changes in the QL. Of course, someone must generate a very first ESMC PDU, and the

[1] Non-coordinated environments or environments is defined in this document as two networks managed by two network operators that may or may not use different technologies regarding synchronization.

SDN controller is in charge of that. This is explained in detail in further subsections. This architecture is shown in figure 4.1.

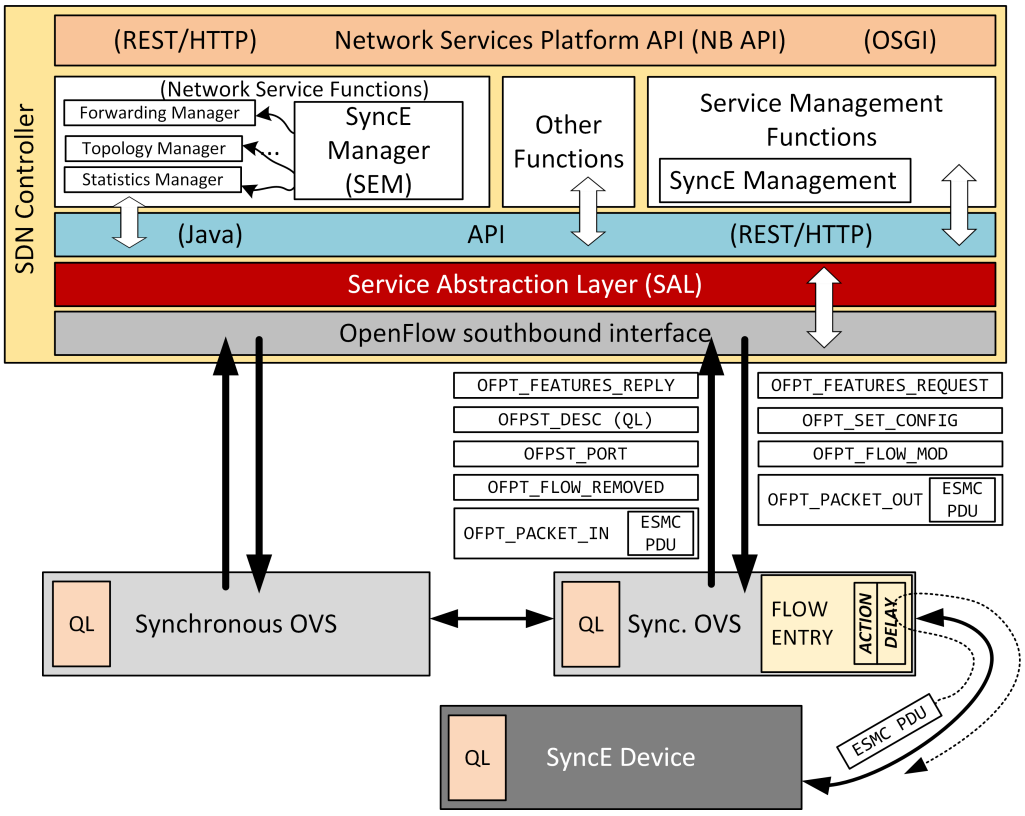

Figure 4.1: Proposed architecture.

The SDN controller must implement the following functionalities (and for some of those to be implemented, OpenFlow extensions are required):

- Ability to query whether a switch implements SyncE hardware or not. This can be solved by extending the OFPT_FEATURES_REPLY message which provides a description of the ports. This message has been extended with an extra descriptor for every port. The message also provides information regarding the matching and action set capabilities, which have also been extended.
- Ability to read and store the QL of every switch. For that, the OFPST_DESC message can be extended with an extra field for the QL.
- Ability to read and store signal-fail information of the switches' ports. For that, the OFPST_PORT is the adequate message to extend with an extra field where the signal-fail value is sent.
- Ability to configure the switch in order to use a specific port for synchronization. The OFPT_SET_CONFIG message, which is used for switch configuration, can be extended with an extra field indicating what port should be used for synchronizing.
- Ability to install more specific flow entries in the switch. The OFPT_FLOW_MOD message, which is used to install, update and delete flow entries can be extended in order to allow a match option for the event flag in the ESMC PDU, and an action set that allows QL re-writing and delay of packets to allow circulation of ESMC PDUs.

- Ability to know when a flow entry is removed. As the flow entries implement timers for flow entry expiration, it is possible to use those for implementing the IT and WTR timers. When a timer expires, the flow entry expires. If a flow entry expires or is removed, an OFPT_FLOW_REMOVED is generated. This message carries information regarding the match fields so it should be extended as well.
- Ability to generate ESMC PDUs with the purpose of communicating with other environments that may expect ESMC PDUs, so there can be an exchange of ESMC PDU messages between non-coordinated environments at the data plane. The OFPT_PACKET_OUT message is sent by the controller for carrying packets that will be transmitted in the data plane. This message does not have to be extended.
- Ability to receive ESMC PDUs (that are exchanged in the data plane) in the controller with the purpose of replying those messages as the sender will expect a response, and with the purpose making decisions upon QL changes of devices located in other environments. To send the ESMC PDU from the switch to the controller an OFPT_PACKET_IN message is used, but it does not have to be extended.

A network composed by an SDN environment with two switches (OVS1 and OVS2) and non-SDN SyncE NE_3 following a topology and a clock distribution as shown in figure 4.2. Conceptually, it should behave as follows:

1. The controller is switched on.
2. OVS1 is switched on and the communication with the controller is established. There is an exchange of OFPT_FEATURES_REQUEST and OFPT_FEATURES_REPLY messages and there is a very first polling of statistics.
3. OVS2 is switched on and there is an exchange of OFPT_FEATURES_REQUEST and OFPT_FEATURES_REPLY messages.
4. Another round of polling starts. When the controller receives the statistics of OVS2, the selection process is triggered as $QL_{OVS1} < QL_{OVS2}$.
5. OVS2 is configured to use OVS1 as a synchronization source as a result of the selection process.
6. Another round of polling starts. There is a periodic polling while the switches are connected to the controller. As there are no changes in any parameter, nothing happens.
7. A non-SDN SyncE NE_3 is connected to OVS2.
8. OVS2 receives an ESMC PDU from SyncE NE_3 (because it is in another environment), but as there is no matching of any flow entry (because such flow entry does not exist yet), this packet is sent to the controller through an OFPT_PACKET_IN message and the controller processes it.
9. As a result of step 8, the controller sends an ESMC PDU with a QL value of OVS2 using an OFPT_PACKET_OUT message, installs a flow entry in OVS2 controlled by an IT timer, and the selection process is triggered. As $QL_{SyncE\ NE_3} < QL_{OVS1}$ and $QL_{SyncE\ NE_3} < QL_{OVS2}$, OVS1 and OVS2 are configured to use SyncE NE_3 as a synchronization source.

10. OVS2 receives another ESMC PDU from SyncE NE_3. Now, this packet is matched with the flow entry. The ESMC PDU is rewritten with OVS2's QL and sent to SyncE NE_3.

11. SyncE NE_3 gets disconnected from OVS2.

12. No ESMC PDU are received at OVS2 and the flow entry expires. OVS2 generates an OFPT_FLOW_REMOVED message and it is processed by the controller.

13. As a result of the previous step, the controller installs a new flow entry in OVS2 controlled by a WTR timer, and runs the selection process. OVS1 and OVS2 are reconfigured as in step 5.

14. The WTR timer expires.

4.2. Environment

The environment chosen for implementing the design developed in this project is the following:

- Oracle VM VirtualBox: This software has been used in order to run two independent virtual machines (VM), one for the SDN controller and another one for the Open vSwitch. The VM runs Ubuntu 64-bit, 2 GB RAM. It has been used a single core for Open vSwitch and two cores for SDN controller for performance reasons. As the Open vSwitch can be virtualized, it is not needed to set up more VMs as it is possible to generate a topology of several switches in a single Open vSwitch instance.

- OpenDaylight 1.0.2 (version of October 24th, 2014) [25]: This was the latest stable version available of OpenDaylight when this project was started. The version used is the pre-compiled one. The code used as a base for modifications is the one commited on October 24th, 2014, 2:18 PM. This commit belongs to the stable/helium branch, and is completely stable. There have been taken actions in order to use a 'frozen' stable version of the controller as it is updated day by day by the OpenDaylight community. Using a frozen stable version makes easier to the developer to detect if further errors/warnings are caused by the developer or by the developing community.

- OpenFlow 1.0 (OF1.0): Although the ONF had already specified OpenFlow 1.4 when this project started, OpenDaylight only implemented completely OF1.0, while OF1.3 was partially implemented. OF1.4 was not implemented yet.

- Open vSwitch 2.3.0 (version of August 14th, 2014): This was the latest stable version available of Open vSwitch when this project was started, and can run either OF1.0 or OF1.3. The source-code has been downloaded to the VM and has been compiled. Any error/warning appearing in the development period is only due to developer's fault.

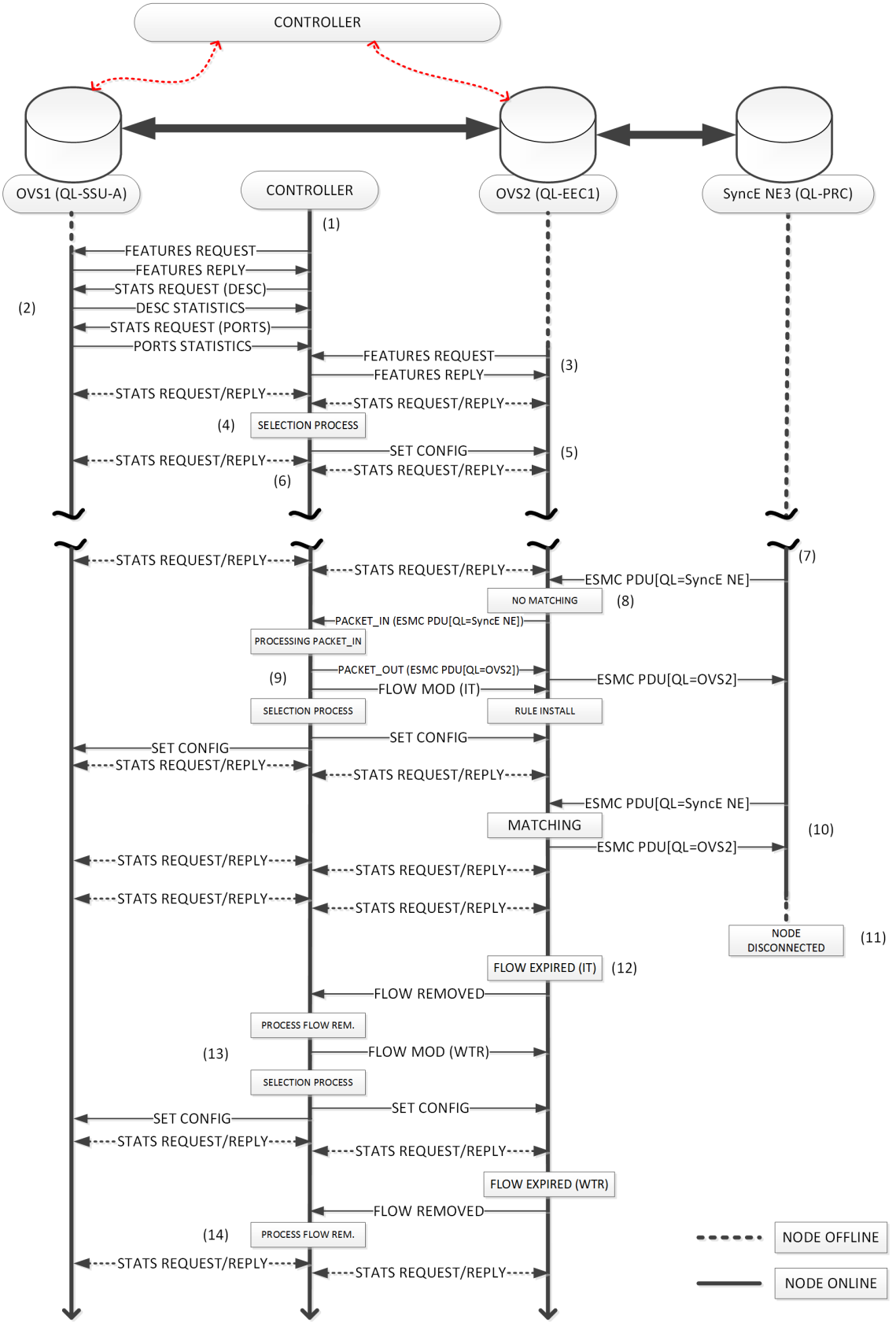

Figure 4.2: Interaction between nodes.

4.3. Implementation

This section describes in detail the different changes regarding OpenFlow, OpenDaylight and Open vSwitch for the purpose of enabling Synchronous Ethernet operations. For a better understanding of next subsections, the operations regarding SyncE have been installed in a new plugin called Synchronous Ethernet Manager (SEM), and extended OF functionalities have been applied to existing plugins in the OpenDaylight controller as well as in the OF1.0 implementation of Open vSwitch. Also, in *.java* files, names starting with *I* mean that it is an interface[2] (between the SAL and an API, and the SAL or the control layer), otherwise it is a class. Also note that in order to analyse results, the OpenFlow 1.0 Wireshark dissector[3] has been modified in order to implement extended parts as well as not yet supported messages.

This section is organized as follows. First, the switch features are described in subsection 4.3.1, followed by the statistics readings in subsection 4.3.2. Once the content of these messages is processed, it is possible to execute the selection process (subsection 4.3.3). Then, as a synchronization port is select, a configuration of the devices (subsection 4.3.4) and a flow entry installation (subsection 4.3.5) takes place. When a flow entry expires because of its timer, some actions have to be done (subsection 4.3.6). Also, the device has to be able to generate ESMC PDU frames by using OFPT_PACKET_OUT messages and to read them by using OFPT_PACKET_IN messages in order to communicate with other synchronous environments (subsection 4.3.7).

4.3.1. Switch features

Upon TLS session establishment between the controller and the switch, the controller sends and OFPT_FEATURES_REQUEST message (this is, a controller-to-switch communication type). This message does not contain any body, but only the OF header, hence no extension is required. Then, the switch responds with an OFPT_FEATURES_REPLY message (code 4.1). The modified parts are shown in bold red.

Code 4.1: OFPT_FEATURES_REPLY structure.

```
struct ofp_switch_features {
    ovs_be64 datapath_id;   /* Datapath unique ID.*/
    ovs_be32 n_buffers;     /* Max packets buffered at once. */
    uint8_t n_tables;       /* Number of tables supported */
    uint8_t pad[3];         /* Align to 64-bits. */

    /* Features. */
    ovs_be32 capabilities;  /* Bitmap of capabilities */
    ovs_be32 actions;       /* Bitmap of supported actions */
    struct ofp_phy_port ports[0]; /* Port definitions */
};
OFP_ASSERT(sizeof(struct ofp_switch_features) == 24);
```

[2] In computing, an interface is a shared boundary across which two separate components of a computer system exchange information. This interface defines what methods should be defined by classes that implement that interface.

[3] A dissector of a protocol decodes the information of that protocol in order to be shown by Wireshark.

The 32-bit action space maps the actions a switch can perform. In OF1.0, bits 0-11 are defined with actions, while bits 12-31 are undefined actions (figure 4.3). A new action has been defined at bit 12 for QL rewriting and another at bit 13 for delaying packets.

	bits													
14 - 31	13	12	11	10	9	8	7	6	5	4	3	2	1	0
[NOT DEFINED]	SET_DELAY	SET_ESMC_SSM	ENQUEUE	SET_TP_DST	SET_TP_SRC	SET_NW_TOS	SET_NW_DST	SET_NW_SRC	SET_DL_DST	SET_DL_SRC	STRIP_VLAN	SET_VLAN_PCP	SET_VLAN_VID	OUTPUT

Figure 4.3: Bitmap of actions. If the action is supported, the bit is set to 1.

The *ofp_phy_port* structure provides information of every port of the device such as: port number, MAC address, configuration and state flags. The 32-bit port configuration space indicates whether or not a port has been administratively brought down, options for handling STP packets, and how to handle incoming and outgoing packets. As only bits 0-6 are used, a new one can be defined for indicating if a port can be synchronous or not. Hence, bit 7 has been defined as OFPPC_SYNC_PORT.

The result is shown in figure 4.4. The capture shows a switch that has synchronous ports and is able to replace QL values from ESMC PDU packets.

```
98 0.151170000    172.16.0.1          172.16.0.100        OpenFlow    338 Type: OFPT_FEATURES_REPLY
Padding: 000000
▶ capabilities: 0x000000c7
▼ actions: 0x00003fff
    .... .... .... .... .... .... .... ...1 = Output to switch port: True
    .... .... .... .... .... .... .... ..1. = Set the 802.1q VLAN id: True
    .... .... .... .... .... .... .... .1.. = Set the 802.1q priority: True
    .... .... .... .... .... .... .... 1... = Strip the 802.1q header: True
    .... .... .... .... .... .... ...1 .... = Ethernet source address: True
    .... .... .... .... .... .... ..1. .... = Ethernet destination address: True
    .... .... .... .... .... .... .1.. .... = IP source address: True
    .... .... .... .... .... .... 1... .... = IP destination address: True
    .... .... .... .... .... ...1 .... .... = IP ToS (DSCP field, 6 bits): True
    .... .... .... .... .... ..1. .... .... = TCP/UDP source port: True
    .... .... .... .... .... .1.. .... .... = TCP/UDP destination port: True
    .... .... .... .... .... 1... .... .... = Output to queue: True
    .... .... .... .... ...1 .... .... .... = Set new QL TLV: True
    .... .... .... .... ..1. .... .... .... = Delay action set: True
▼ Port data 1
    Port number: 4
    HW Address: 3e:ae:d9:fc:af:02 (3e:ae:d9:fc:af:02)
    Name: ovs5portNO-SDN
  ▼ Config flags: 0x00000080
      .... .... .... .... .... .... .... ...0 = Port is administratively down: False
      .... .... .... .... .... .... .... ..0. = Disable 802.1D spanning tree on port: False
      .... .... .... .... .... .... .... .0.. = Drop all packets except 802.1D spanning tree packets: False
      .... .... .... .... .... .... .... 0... = Drop received 802.1D STP packets: False
      .... .... .... .... .... .... ...0 .... = Do not include this port when flooding: False
      .... .... .... .... .... .... ..0. .... = Drop packets forwarded to port: False
      .... .... .... .... .... .... .0.. .... = Do not send packet-in msgs for port: False
      .... .... .... .... .... .... 1... .... = Synchronous port: True
```

Figure 4.4: Wireshark capture of an extended OFPT_FEATURES_REPLY packet.

When the Synchronous Ethernet Manager (SEM) receives this message, it determines whether a node can be used in the synchronization network or not. The requirement for being included in the synchronization network is to have at least one synchronous port.

Being able or not to set a new QL TLV or delaying packets as part of the action set is not strictly a requirement, as it is only needed for edge equipment[4].

The network manager can see synchronous and asynchronous devices in the Graphical User Interface (GUI) that OpenDaylight provides looking at the extended switch properties 'SyncE compliant' and 'Sync source' (figure 4.5).

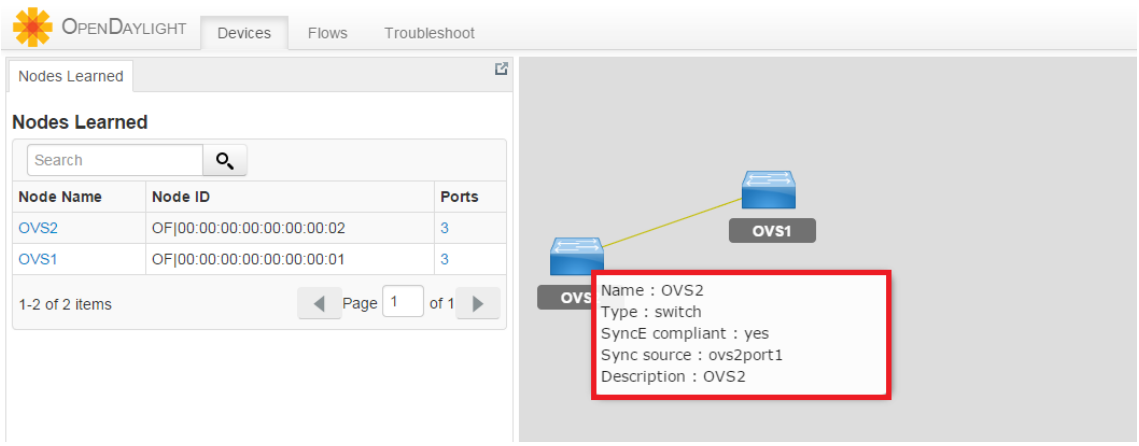

Figure 4.5: OpenDaylight's GUI showing information regarding SyncE devices.

When a node is connected to the controller, the process shown in figure 4.6 is executed. The SEM is in charge of attaching *synchronous* properties (table 4.1) to the node and port objects so the system can keep track of SyncE-related values. For nodes (*SyncENodeProp*), the property comprises information regarding whether it is a synchronous node or not, and the QL_ENABLED flag. For ports (*SyncEPortProp*), the property comprises information regarding the priority of the port, the signal_fail flag, the QL value and the *used* flag (if true, this port is being used by the node to get synchronization). These properties are installed in the SAL.

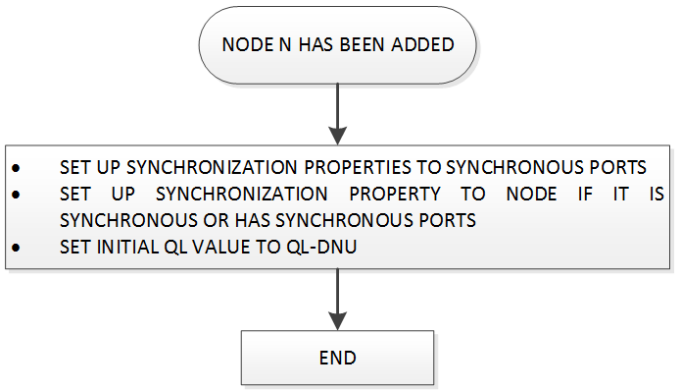

Figure 4.6: Flow chart that takes place when a new node is registered in the controller.

[4]Edge equipment is a node that is located at the edge of a network, and can be connected to another network that is managed by another network operator.

SyncENodeProp	**SyncEPortProp**
+name: String = 'SyncENodeProp' -modo_sincrono: bool = true -_QL_ENABLED: bool = true	+name: String = 'SyncEPortProp' -priority: int -signal_fail: bool = false -used: bool = false -qL: int = 0b1111

Table 4.1: Synchronous properties for nodes (left) and ports (right).

4.3.2. QL & signal-fail statistics

For the controller to request statistics from a switch, it starts a controller-to-switch communication by sending an OFPT_STATS_REQUEST indicating what kind of statistics it requests. The switch can offer the statistics described below:

- **OFPST_DESC (0x0000)**: These are description statistics which provide information regarding the switch manufacturer, hardware and software revision, serial number and datapath description if it is available. The information is switch-related.
- **OFPST_FLOW (0x0001)**: They report individual flow entry information such as duration, packet count, byte count, cookies and characteristics of the flow (match options, actions, etc.). The information is flow-related.
- **OFPST_AGGREGATE (0x0002)**: These statistics report about multiple flows and provide aggregated results for packet count, byte count and number of flows aggregated. The information is flow-related.
- **OFPST_TABLE (0x0003)**: These are flow table statistics and provide information such as number of active entries, look up count, matched flows count, and table characteristics. The information is flow-table-related.
- **OFPST_PORT (0x0004)**: They report information about physical ports such as received, transmitted and dropped packets or bytes, errors and collisions. The information is port-related.
- **OFPST_QUEUE (0x0005)**: The information request is queue related and it contains transmitted bytes/packets and dropped packets.
- **OFPST_VENDOR (0x0006)**: These are vendor-specific statistic messages. It is required to identify the vendor in the first four bytes. Next bytes are vendor-defined. Vendors should contact OF consortium to get a vendor ID.

In the case of QL readings, as the QL is defined for a device, the most suitable message to carry it is the OFPST_DESC as it refers to the switch as a device. As signal-fail is defined for ports, an OFPST_PORT message is used to carry it. These messages have been extended as shown in code 4.2 and code 4.3. In the case of the OFPST_PORT, a byte has been extended for signal-fail readings, and 7 extra bytes of padding have been added in order to match a 64-bit structure. Extended code is shown in italic blue and modified code in bold red.

Code 4.2: OFPST_DESC structure.

```
struct ofp_desc_stats {
    char mfr_desc[256];   /* Manufacturer description. */
    char hw_desc[256];    /* Hardware description. */
    char sw_desc[256];    /* Software description. */
    char serial_num[32];  /* Serial number. */
    char dp_desc[256];    /* Description of the datapath. */
    char sw_ql[32];       /* QL of the switch */
};
OFP_ASSERT(sizeof(struct ofp_desc_stats) == 1088);
```

Code 4.3: OFPST_PORT structure.

```
struct ofp_port_stats {
    ovs_be16 port_no;
    uint8_t pad[6];                  /* Align to 64-bits. */
    ovs_32aligned_be64 rx_packets;   /* received packets. */
    ovs_32aligned_be64 tx_packets;   /* transmitted packets. */
    ovs_32aligned_be64 rx_bytes;     /* received bytes. */
    ovs_32aligned_be64 tx_bytes;     /* transmitted bytes. */
    ovs_32aligned_be64 rx_dropped;   /* packets dropped by RX. */
    ovs_32aligned_be64 tx_dropped;   /* packets dropped by TX. */
    ovs_32aligned_be64 rx_errors;    /* received errors. */
    ovs_32aligned_be64 tx_errors;    /* transmitted errors. */
    ovs_32aligned_be64 rx_frame_err; /* frame alignment errors. */
    ovs_32aligned_be64 rx_over_err;  /* packets with RX overrun. */
    ovs_32aligned_be64 rx_crc_err;   /* CRC errors. */
    ovs_32aligned_be64 collisions;   /* collisions. */
    uint8_t signal_fail;             /* signal-fail. */
    uint8_t pad7[7];                 /* Align to 64-bits. */
};
OFP_ASSERT(sizeof(struct ofp_port_stats) == 112);
```

Given the unavailability of real SyncE devices during the development of the project, a dummy function has been created for simulating QL calculation and signal-fail condition. For that, every is assigned device a unique datapath descriptor[5] and *dummyQL* and *dummySignalFail* are shared files among all the virtual switches. These files contain the relation between the datapath descriptor and the QL of the node and the signal-fail state of the ports. This approach also allows debugging the application, as both parameters can be changed as needed. Then, when a description or port statistics is requested, the device operates as usual, reads the file, and replies. In figure 4.7 an OPFST_DESC message reporting a QL-PRC quality level is shown. Figure 4.8 shows an OPFST_PORT message reporting a signal-fail condition for port ovs1port2.

The same changes (figure 4.9) have been applied in the controller. These messages are received by the *OFStatisticsManager* (in the southbound interface) and stored into the *OFDescriptionStatistics* and *OFPortStatisticsReply* models (for OPFST_DESC and OPFST_PORT respectively). Then, for it to be accessible to any application, the SAL must expose the information through its *ReadService*. Then, the information is transferred to another model belonging to the SAL (*NodeDescriptor* model for QL readings, and *NodeConnectorStatistics* model for signal-fail readings), which have also been adapted.

[5]A datapath descriptor is a string that identifies the OVS and it is helpful for the network manager. For example, an important node can be configured with a datapath descriptor equal to 'important-node'.

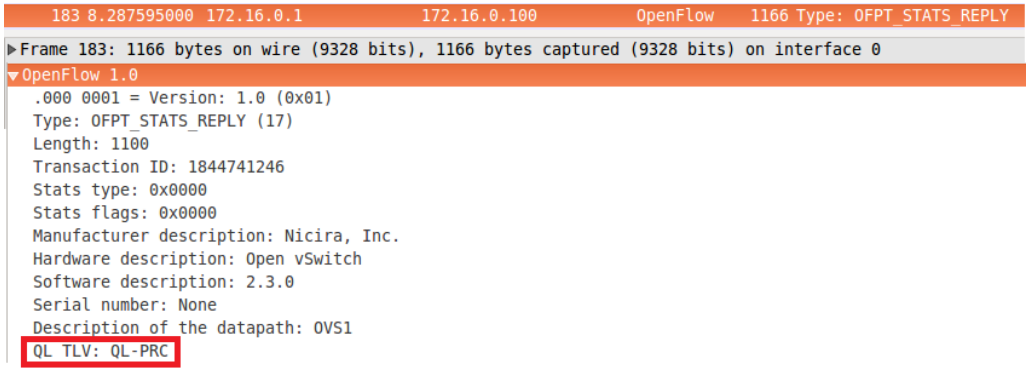

Figure 4.7: Wireshark capture of an extended OFPST_DESC packet.

Now, readings are available to any application and the GUI as seen in fig. 4.10 and 4.11.

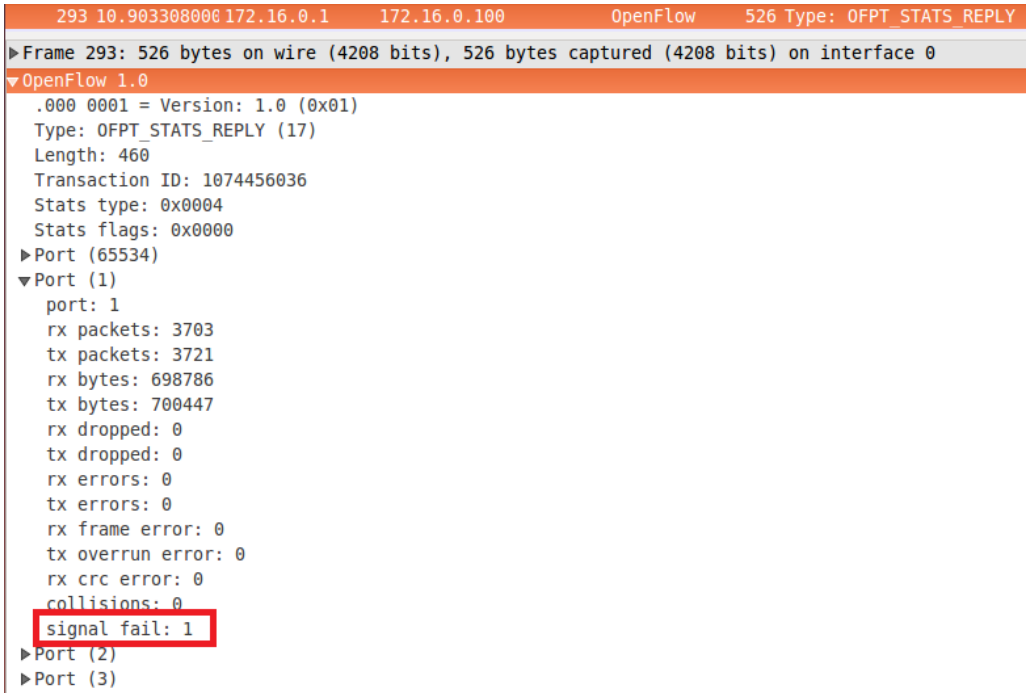

Figure 4.8: Wireshark capture of an extended OPFST_PORT packet.

When an update of the description or port statistics is received, the process described in figure 4.12 takes place. In case an OPFST_DESC is received, it is checked if a root for the synchronization tree has been chosen or if there is a better candidate. If the reported statistics has the best QL known by now, it is considered as the best QL of the network until a better QL is discovered or the node gets disconnected. Afterwards, it is checked if the QL has changed and if the current QL value is worse or equal than QL-EEC1/2. If any of these two requirements is met, the selection process is triggered. In case an OPFST_PORT is received, if the signal-fail value has changed, and the port is susceptible to be used or stopped being used for synchronization, the selection process is started.

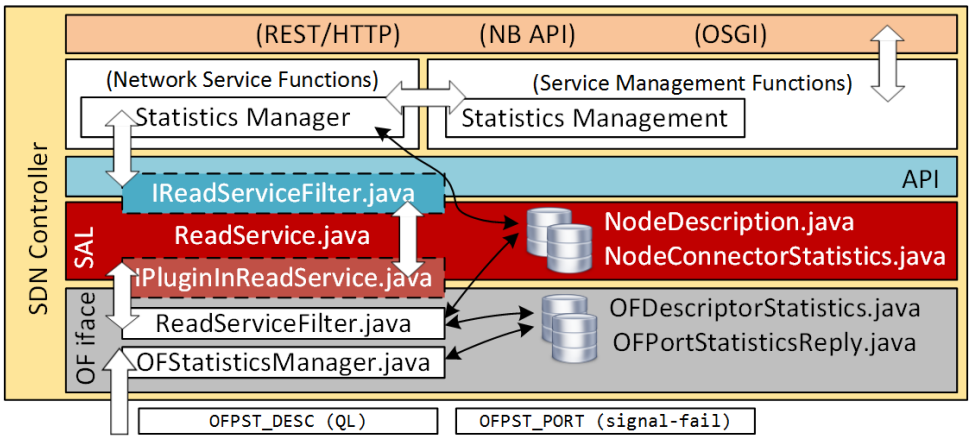

Figure 4.9: Modules and interfaces involved in QL and signal-fail readings in OpenDaylight.

Node Information

Description	Specification
manufacturer	Nicira, Inc.
hardware	Open vSwitch
software	2.3.0
serialNumber	
description	OVS1
synchrnoizationQualityLevel	QL-PRC

Figure 4.10: QL readings in the OpenDaylight GUI.

Ports

Port Details

Refresh

OF

Node Connector	Rx Pkts	Tx Pkts	Rx Bytes	Tx Bytes	Rx Drops	Tx Drops	Rx Errs	Tx Errs	Rx Frame Errs	Rx OverRun Errs	Rx CRC Errs	Collisions	signal fail
OF|1@OVS2	7525	19919	1414275	2497650	0	0	0	0	0	0	0	0	0
OF|2@OVS2	7444	19908	1398068	2497073	0	0	0	0	0	0	0	0	0
OF|3@OVS2	9541	9509	1791540	1787876	0	0	0	0	0	0	0	0	1

1-3 of 3 items

Page 1 of 1

Figure 4.11: signal-fail readings in the OpenDaylight GUI.

4.3.3. Selection process

The selection process is an algorithm that decides what port should a node use for synchronizing taking into account the QL of the node and its neighbours, as well as the state of the signal-fail flag of the ports. In order to obtain this information, the SEM has been connected with the *Topology Manager* application. This application can provide information regarding the topology; specifically: a set of edges in the current topology, a list of host objects that are attached to a given port, the list of nodes that have hosts connected and the list of edges of an specific node. For the purpose of the selection process, the latter information is used, since it allows the SEM to get the links between the current

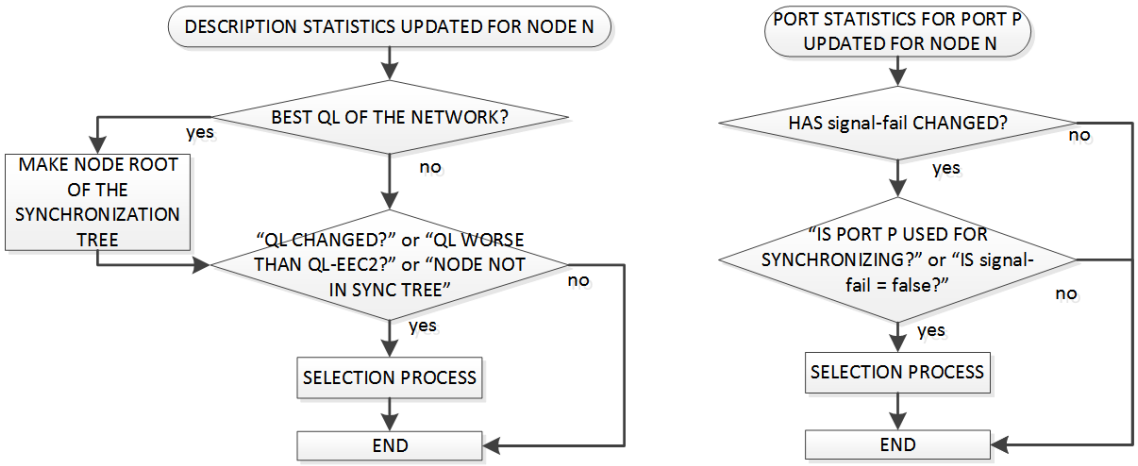

Figure 4.12: Flow charts of statistics updates processes.

node and neighbour nodes, retrieve the QL of each of these nodes and compute the best synchronization port. In order to obtain the QL of every adjacent node, a connection with the *Switch Manager* application (which allows the SEM to get node's and port's properties, as well as almost any information regarding a switch) is needed. All the operations of the selection process are controller-based and do not require any extra interaction with the data plane.

The selection process is triggered for a node N_k when one of these events occurs to N_k:

- Change in QL or in signal-fail state. If a QL worsens or improves, or if a port enters in signal-fail state or it is recovered from it, the synchronization tree might be reconfigured. These changes are notified to the controller periodically through statistics polling (triggered by the controller). As an exception, if the QL of the node has not changed, but that node it is still not synchronized, the selection process is also triggered.
- A SyncE-related flow entry is removed or expires. This means that the either the IT or WTR timers have expired, so the synchronization tree might be reconfigured.
- A synchronous device is connected to the controller. This device should be added to the synchronization tree. In case the device is disconnected from the controller, the affected part of the synchronization tree should be reconfigured.

When one of these events takes place on a node N_k, **the selection process is executed for the node N_k**:

1. Retrieve the QL from the adjacent synchronous nodes resulting in a list of possible ports. This step also checks if node N_k is synchronizing adjacent nodes. If it does and the QL of N_k has worsen, the synchronization branch is removed. For example, if a node N_k is synchronizing a node but now the QL of N_k has worsen, that branch might be invalid and should be removed. Another branch should be computed later.
2. Remove ports in QL-FAILED or signal-fail modes.

3. If QL-ENABLED is set, order the ports by QL and remove lower-QL ports from the list.

4. Remove ports with lower priority from the list.

5. Depending on the ports available after this process, three situations are possible:

   - No ports are available: the device is configured to go into holdover mode.
   - One port is available: that port is configured as a synchronizing source.
   - More than one port is available: check if the current used port is in the list. If so, use it; otherwise, choose randomly and configure the node.

6. Update port properties.

7. If there was a port able to be used for synchronization, check if adjacent nodes to N_k may need to be synchronized by N_k. If so, execute steps 1 to 6 for those nodes. This step improves the configuration time of the synchronization tree compared to the legacy SyncE selection process algorithm, as in this case more than one node is configured (N_k and its adjacent node(s)). For a bigger improvement, this step can be repeated till the whole network is synchronized.

When the result of the selection process is known, it is possible to proceed with port configuration and flow installation, explained in following subsections.

4.3.4. Switch configuration

Once a port has been selected for synchronizing a device, this configuration change must be reported to the affected device. For that, the controller is able to set (and query) configuration parameters of the switch with the OFPT_SET_CONFIG (and OFPT_GET_CONFIG_REQUEST) messages. This is a controller-to-switch communication and the controller only replies to the OFPT_GET_CONFIG_REQUEST message. The OFPT_SET_CONFIG structure (code 4.4) determines configuration parameters such as packet handling (i.e. whether or not IP fragments should be dropped) and OFPT_PACKET_IN configuration (i.e. maximum payload bytes). This message has been extended with a 32-bit space for synchronization port configuration (in italic blue). The total size of the struct has changed, shown in bold red.

Code 4.4: OFPT_SET_CONFIG structure.

```
struct ofp_switch_config {
    ovs_be16 flags;             /* OFPC_* flags. */
    ovs_be16 miss_send_len;     /* Max bytes of new flow that data-
                                   path sends to the controller. */
    ovs_be32 synchronization_port; /* SyncE sync port. */
};
OFP_ASSERT(sizeof(struct ofp_switch_config) == 8);
```

As OFPT_SET_CONFIG and OFPT_GET_CONFIG_REQUEST messages use this same structure, one only change is necessary. Given the unavailability of real SyncE devices during the development of the project, a dummy function simulating a port configuration of

the switch has been developed. For that, every device has a unique datapath description and a 'port configuration' shared file among the virtual switches. This file contains the relation between the datapath description and the port used to synchronize.

Figure 4.13 shows an OPFST_SET_CONFIG message configuring a switch to use port #3. Note that the controller has to make sure that this message is processed. To ensure this, the controller sends an OFPT_BARRIER_REQUEST. This message is sent by the controller (controller-to-switch communication) when the controller wants to ensure that message dependencies have been met and wants to receive a notification that the operation has been completed. The switch must respond with an OFPT_BARRIER_REPLY with the same *xid* (sequence number) as the request when the processes are completed.

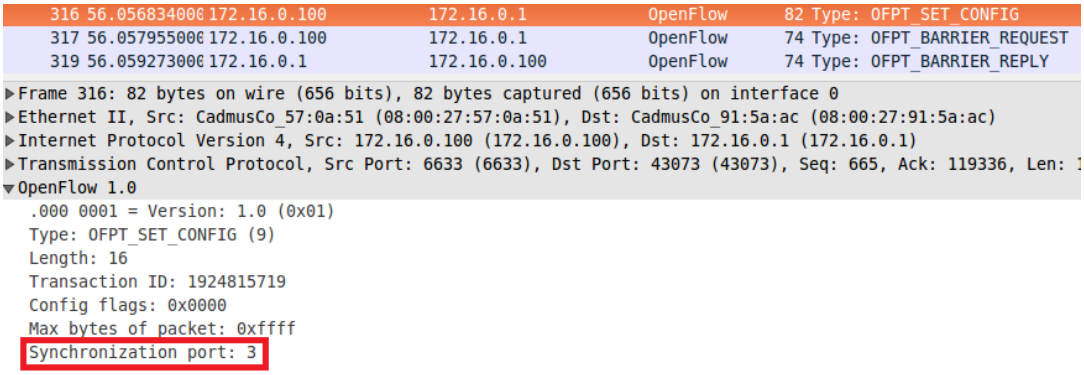

Figure 4.13: Wireshark capture of an extended OFPST_SET_CONFIG packet.

The SEM has to be able to send this type of messages, but this can only be done by the OF southbound interface. That is why it has been added an API interfacing the SEM plugin and the SAL (*IConfigProgrammerService.java*), an interface between the SAL and the OF southbound interface (*IPluginInConfigProgrammerService.java*), a SAL service translating between both interfaces (*ConfigProgrammerService.java*), and a new configuration service in the OF southbound interface (*ConfigurationService.java*). This is shown in figure 4.14.

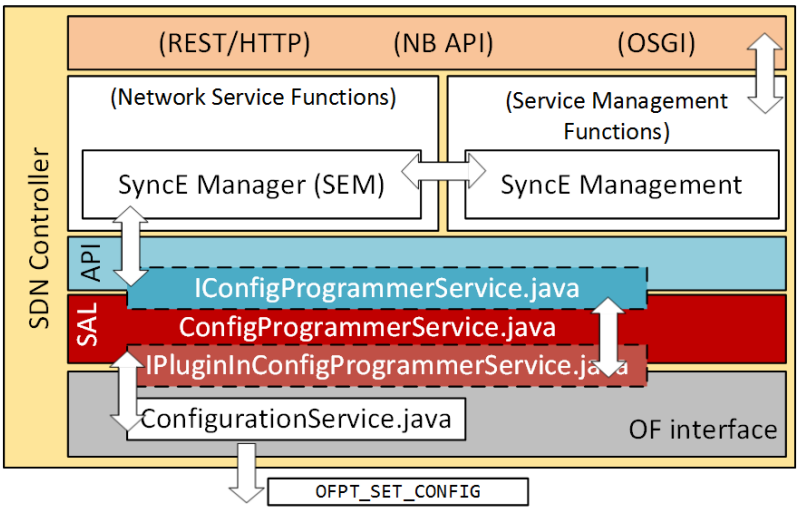

Figure 4.14: Modules and interfaces involved in OFPT_SET_CONFIG message generation in OpenDaylight.

The network manager can see in the GUI whether or not a device is synchronous, as shown in subsection 4.3.1, and what synchronization reference is using (figure 4.15). In the figure it is shown a linear topology formed by OVS1 and OVS2 with qualities QL-PRC and QL-EEC1 respectively. Then, OVS1 uses its own clock for synchronizing while OVS2 uses the port connected to OVS1 (ovs2port1). This information is not shown for asynchronous devices.

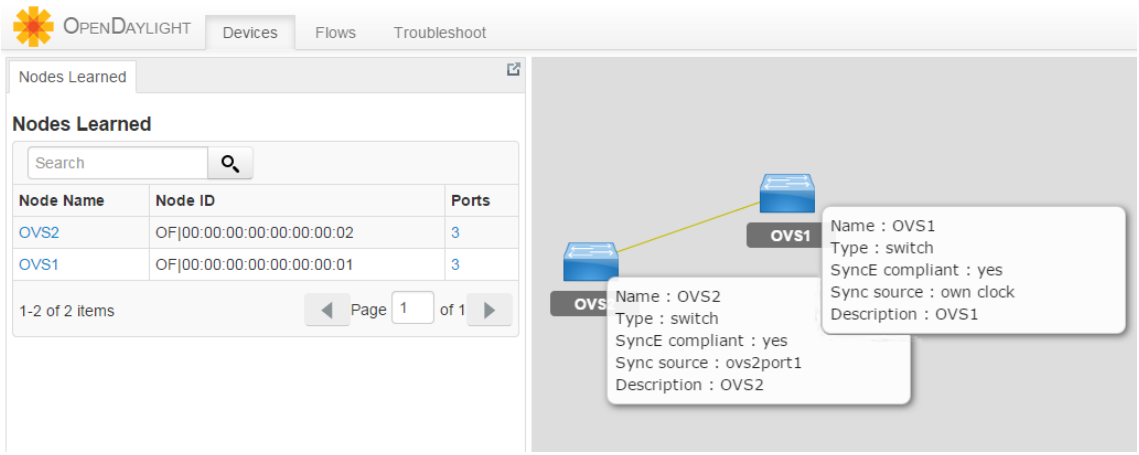

Figure 4.15: OpenDaylight's GUI showing information regarding SyncE devices.

4.3.5. Installation of flow entries

When a synchronous node is connected to another synchronous node in another network, flow entries must be installed in that node in order to manage the exchange of ESMC PDUs. These flow entries provide the switch the ability of rewriting the QL TLV of an ESMC PDU packet in case the event-flag is set to 0 (heart-beat generated), as the ESMC PDU has to be sent back to the sender. However, event-generated ESMC PDUs must be sent to the controller. The timer installed in the flow entries will be IT or WTR depending of the state of the system, as explained in following subsections. The requirements for the flow entry are shown in figure 4.16 and described below:

1. Identify an ESMC PDU: There is no need of an OF extension since a match for the Ethernet frame type is provided.

2. Identify an event-flag of an ESMC PDU: There is no match option that provides this function. Therefore, it is needed to modify the match structure in the OFPT_FLOW_MOD message (code 4.5; changes are shown in bold red). OF1.0 specifies a 1 byte padding space between dl_vlan_pcp and dl_type match fields. This space can be exploited for this use (synce_event_flag now), being easier to implement as the size of the structure is maintained.

3. Rewrite the QL TLV field from an ESMC PDU: If the frame matches the flow entry, the action to be executed is to rewrite the QL TLV value with its own device QL. A new action has been defined (code 4.6) and there is no need to change the structure of the OFPT_FLOW_MOD message as actions are not in a fixed position, unlike match options.

4. Define when the actions should be executed: If two devices exchange ESMC PDUs, the worst situation is when both devices belong to different SDN environments. In this case, as the flow tables are defined to be executed immediately, the exchange of message would be much higher than 10 frames per second (limit established by OSSP). Hence, the ability of configuring a flow entry to execute the actions after a given time is mandatory. In order to do that, a new action is defined in which the delay to be applied to the action set is specified(code 4.7).

5. Output the packet to the input port. This does not need any OF extension.

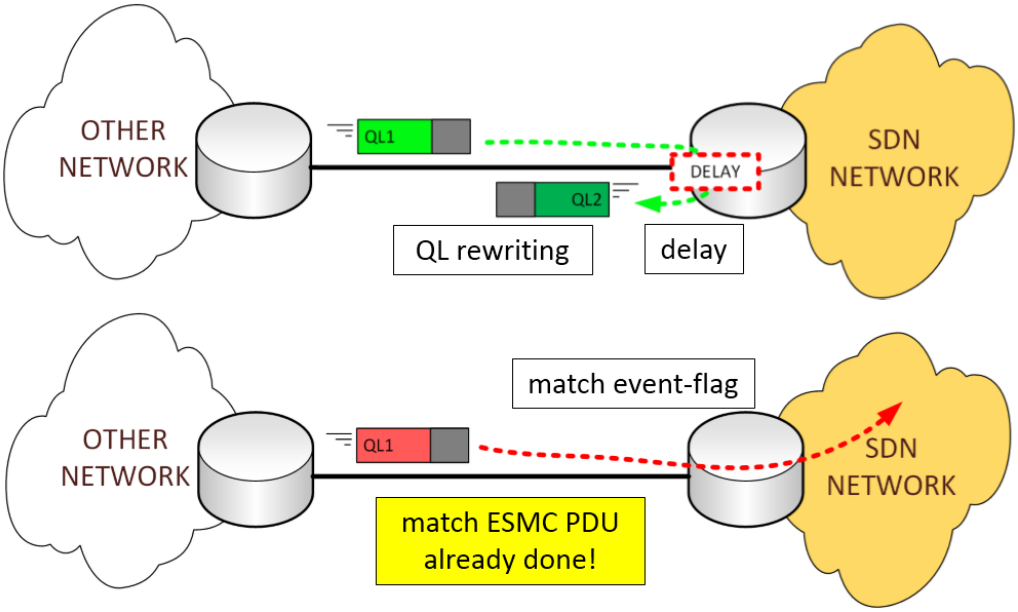

Figure 4.16: Scheme of the exchange of ESMC PDUs between non-coordinated networks.

Code 4.5: OFPT_FLOW_MOD ofp_match structure.

```
/* Fields to match against flows */
struct ofp_match {
    ovs_be32 wildcards;         /* Wildcard fields. */
    ovs_be16 in_port;           /* Input switch port. */
    uint8_t dl_src[6];          /* Eth source address. */
    uint8_t dl_dst[6];          /* Eth destination address. */
    ovs_be16 dl_vlan;           /* Input VLAN. */
    uint8_t dl_vlan_pcp;        /* Input VLAN priority. */
    uint8_t synce_event_flag;   /* SyncE event flag. */
    ovs_be16 dl_type;           /* Ethernet frame type. */
    uint8_t nw_tos;             /* IP ToS (DSCP field). */
    uint8_t nw_proto;           /* IP protocol or lower 8 bits of
                                   ARP opcode. */
    uint8_t pad2[2];            /* Align to 64-bits. */
    ovs_be32 nw_src;            /* IP source address. */
    ovs_be32 nw_dst;            /* IP destination address. */
    ovs_be16 tp_src;            /* TCP/UDP source port. */
    ovs_be16 tp_dst;            /* TCP/UDP destination port. */
};
OFP_ASSERT(sizeof(struct ofp_match) == 40);
```

Code 4.6: OFPST_FLOW_MOD ofp_action_nw_esmc_ssm structure.

```
struct ofp_action_nw_esmc_ssm {
    ovs_be16 type;      /* OFPAT_SET_ESMC_SSM. */
    ovs_be16 len;       /* Length is 8. */
    ovs_be16 qL;        /* QL to be announced. */
    ovs_be16 padding;   /* Padding to complete 64-bit struct. */
};
OFP_ASSERT(sizeof(struct ofp_action_nw_esmc_ssm) == 8);
```

Code 4.7: OFPST_FLOW_MOD ofp_action_set_delay structure.

```
struct ofp_action_set_delay {
    ovs_be16 type;      /* OFPAT_SET_DELAY. */
    ovs_be16 len;       /* Length is 8. */
    ovs_be32 delay;     /* Delay in ns. */
};
OFP_ASSERT(sizeof(struct ofp_action_set_delay) == 8);
```

These extensions have been implemented in the controller and in OVS. In order to allow the SEM to install flow entries, it has been connected with the *Forwarding Rules Manager* (FRM) application, which has a secure way of installing flow entries as it takes validation procedures. As new match fields and action sets have been created, the southbound interface as well as the SAL had to be adapted, keeping compatibility with other modules. The connections can be seen in figure 4.17.

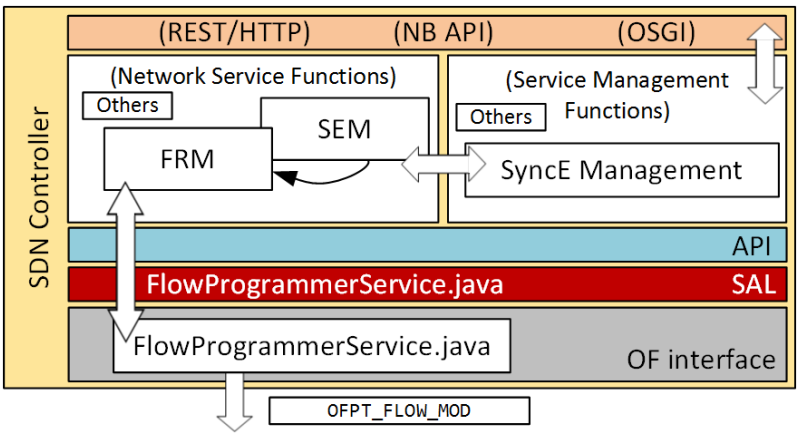

Figure 4.17: Modules involved in OFPT_FLOW_MOD generation in OpenDaylight.

The resulting OFPT_FLOW_MOD is shown in figure 4.18, where the installation of the flow entry is shown. The extensions are: match for the ESMC PDU event-flag, an action for rewriting the QL TLV field and an action for delaying the output of the packet. The event-flag is 0 (heart beat messages); the QL TLV value in this case is 2, which means a quality QL-PRC; and the delay applied is 0.5 seconds (500 x 10^6 ns).

The GUI of OpenDaylight has been adapted in order to show the new flow entry's parameters. Figure 4.19 shows the actions of setting a new QL value, the action of delaying the action execution and the matching for the event flag value in the ESMC PDU. This can also been seen from the Open vSwitch side (figure 4.20). In this case it shows *synce_event_flag* match values of 0x10 and 0x18. These match values are due to implementation reasons, in which the matching has to be for an entire byte, so this value is composed of version and event-flag registers (figure 2.4).

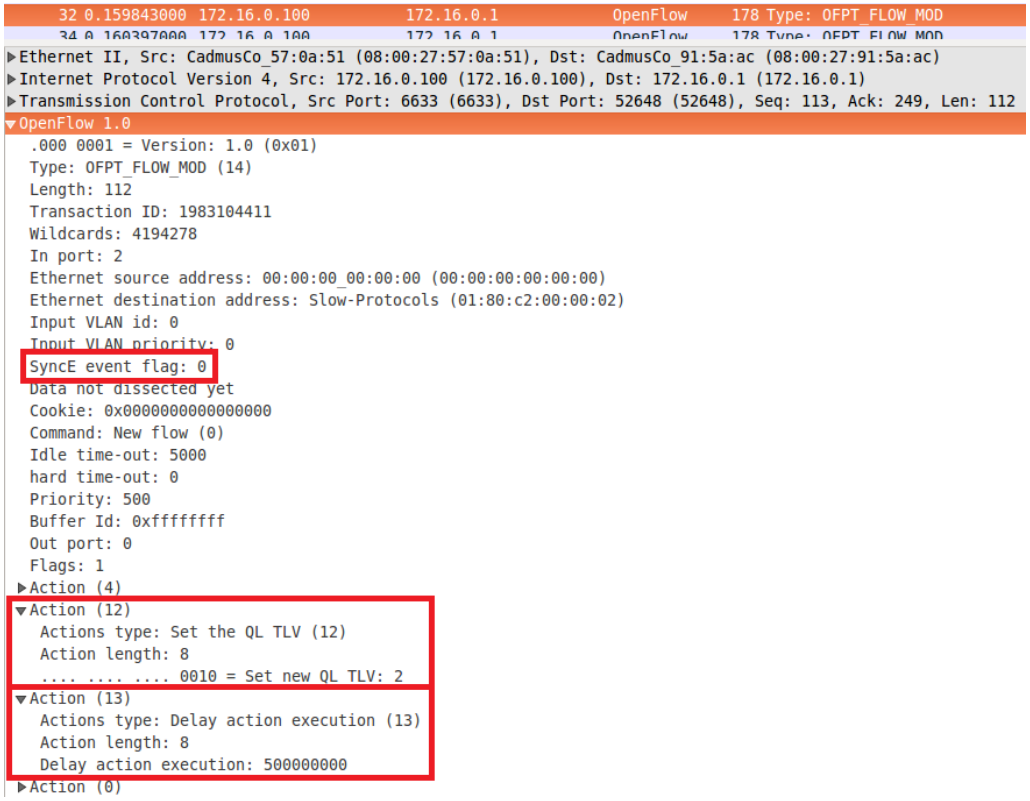

Figure 4.18: Wireshark capture of an extended OFPT_FLOW_MOD message.

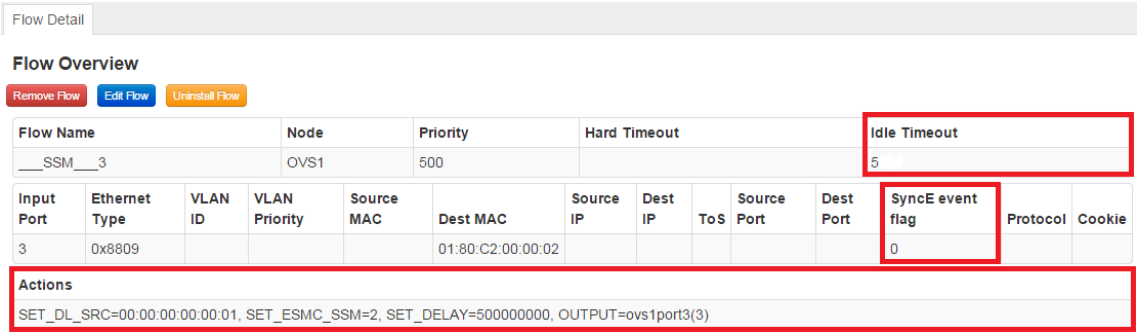

Figure 4.19: OpenDaylight's GUI showing the flow entry's installation parameters.

```
NXST_FLOW reply (xid=0x4):
 cookie=0x0, duration=0.336s, table=0, n_packets=0, n_bytes=0, idle_timeout=5, idle_age=0, priorit
y=2,ossp,in_port=3,dl_dst=01:80:c2:00:00:02,dl_type=0x8809,synce_event_flag=0x18 actions=mod_dl_sr
c:00:00:00:00:00:01,mod_ssm:11,delay:500000000,IN_PORT
 cookie=0x0, duration=0.274s, table=0, n_packets=0, n_bytes=0, idle_timeout=5, idle_age=0, priorit
y=1,ossp,in_port=3,dl_dst=01:80:c2:00:00:02,dl_type=0x8809,synce_event_flag=0x10 actions=mod_dl_sr
c:00:00:00:00:00:01,mod_ssm:11,delay:500000000,IN_PORT
```

Figure 4.20: Flows installed in the OVS, in the OVS console.

Matching for *synce_event_flag* and action QL rewriting have been implemented in the OVS. However, due to time limitations in the development of this work, action set delay is implemented but not performed. This is, the delay rule can be installed but it is ignored when the rule is executed.

4.3.6. Expiration of flow entries

When a flow entry is installed, it is configured with an IT or WTR timer. When any of these timers expires, some actions have to be performed. Then, it is necessary for the SEM to know about flow expirations. OF provides an asynchronous message that fulfils this requirement: OFPT_FLOW_REMOVED. This message is generated when disabling a flow entry due to removal or expiration. The *FlowProgrammer* service in the SAL has been extended to support notification of these events to applications within the controller. Interfaces and other services were already implemented by default. The connections between modules and interfaces is shown in figure 4.21.

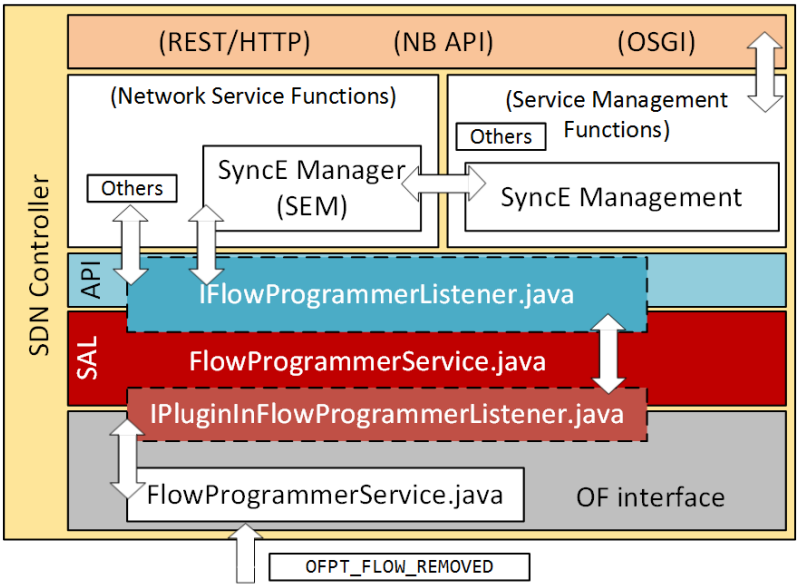

Figure 4.21: Modules and interfaces involved in OFPT_FLOW_REMOVED event message in OpenDaylight.

The network manager is able to detect in which nodes the timers are activated. Flow entries for timers are identified by its name as seen on figure 4.22. The network manager is able to interact with these flow entries as well, being able to uninstall or edit them. This means forcing a timer to expire or to set up a configuration manually.

When a flow is removed or expires, the process shown in figure 4.23 takes place. First of all, it is checked that the flow handled is SyncE-related. If it is, then we should remove this flow from the *installed_flows* list. Then, it is checked whether it has an IT timer or an WTR timer. If the timer is IT, it means that no ESMC PDU has been received for 5 seconds, so the port should be set with signal-fail to true and a new flow entry should be installed for dropping ESMC PDU messages, as they must not be processed by the controller while the WTR timer is running. If it has an WTR timer, it means that the WTR timer has expired so this port can be used for synchronizing again, so the signal-fail is set to false. In both cases (for IT or WTR timer), the selection process takes place.

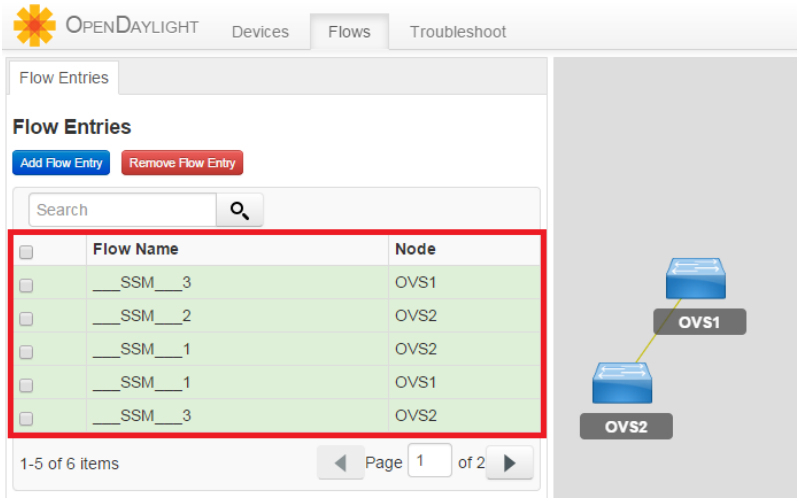

Figure 4.22: Flow entries installed in every device in the GUI of OpenDaylight.

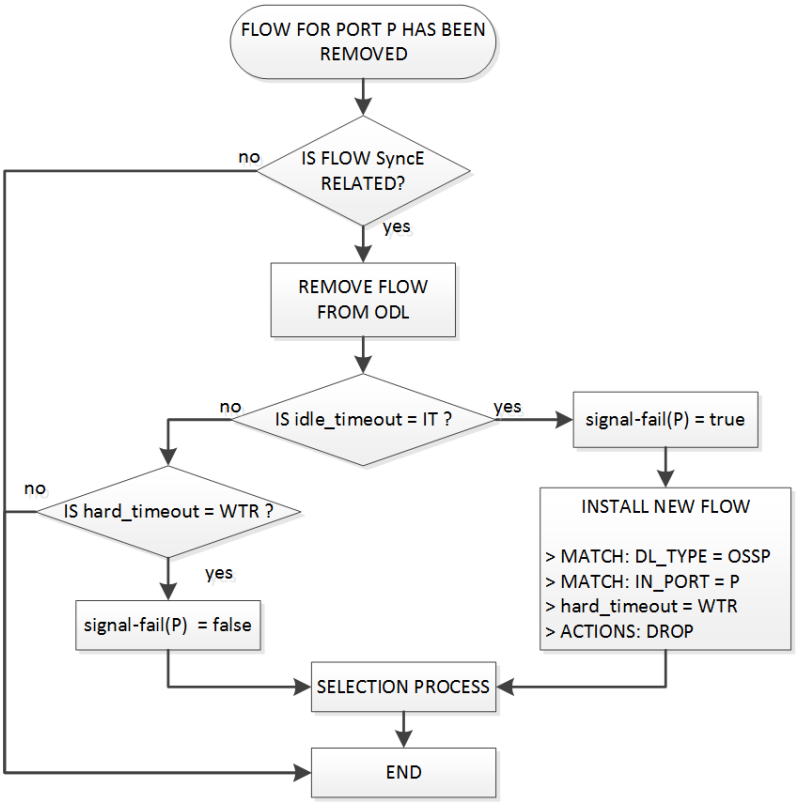

Figure 4.23: Flow chart of flow expiration process.

4.3.7. ESMC PDU processing

If there is a communication with a synchronous device in another environment, there is an exchange of ESMC PDU frames. None of the devices know if the other device is SDN or non-SDN, so upon connection with the controller, they should try to send an ESMC PDU message through every port that is connected to other environments. This means that the controller must be able to generate an ESMC PDU frame, encapsulate it in a OFPT_PACKET_OUT message, send it to the switch, and the switch should apply the

actions specified in that message. If the other device is non-SDN-enabled, it will process the message and when necessary, it will send an ESMC PDU back. If the other device is SDN-enabled, it will rewrite the ESMC PDU and send it back again after a certain delay. So, in case both devices are SDN, the rate is set by the rewrite delay (0.5 seconds); and in case it is a non-SDN device, the rate is set by the non-SDN device.

Also, if a change in the QL of any of both devices happens, a new ESMC PDU must be generated with the event-flag set to 1. Then, if an SDN device receives this ESMC PDU, it will encapsulate the ESMC PDU in a OFPT_PACKET_IN message and send it up to the controller. The SEM must be able to receive OFPT_PACKET_IN messages, identify if they are SyncE-related, and if so, process them.

To summarize, the SEM must be able to transmit and receive ESMC PDUs. For that, the DataPacket-service exposed by the SAL has been used. No modification regarding services or interfaces is needed. A model for the ESMC PDU called *ESMC.java* has been added in the SAL (all types of packets are defined there) with which the SEM is able to work with and parse incoming messages from other nodes.

When a OFPT_PACKET_IN is received, the process shown in figure 4.24 takes place. First, it is checked if it is a SyncE-related packet or not, and if the node is in QL-ENABLED mode (if not, it should not process ESMC PDUs). Then, if the QL value of the packet is better than the node's QL, a selection process takes place. Afterwards, a flow is installed in order to control that an ESMC PDU is being received periodically. Finally, a response ESMC PDU (with the QL value of the node) is sent to the switch through OFPT_PACKET_OUT so it can be sent back to the other node.

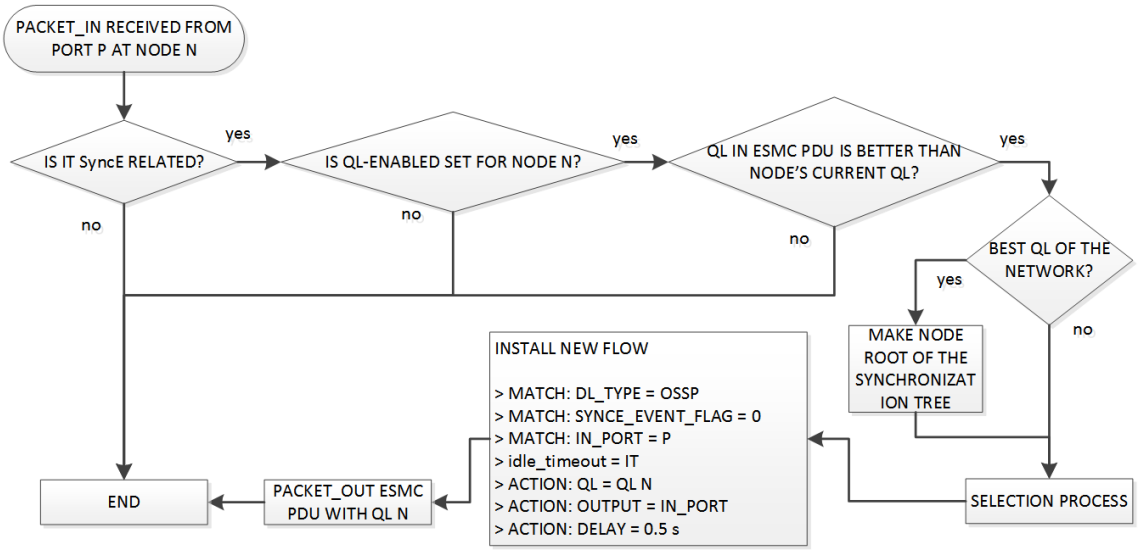

Figure 4.24: Flow chart process when a data-packet is sent to the controller.

Figure 4.25 shows an ESMC PDU heading to a switch. If the switch does not match the packet, it is sent to the controller through an OFPT_PACKET_IN message carrying an ESMC PDU, as shown in figure 4.26. Once the controller has processed the packet, it is sent back to the switch using an OFPT_PACKET_OUT message carrying an ESMC PDU (figure 4.27).

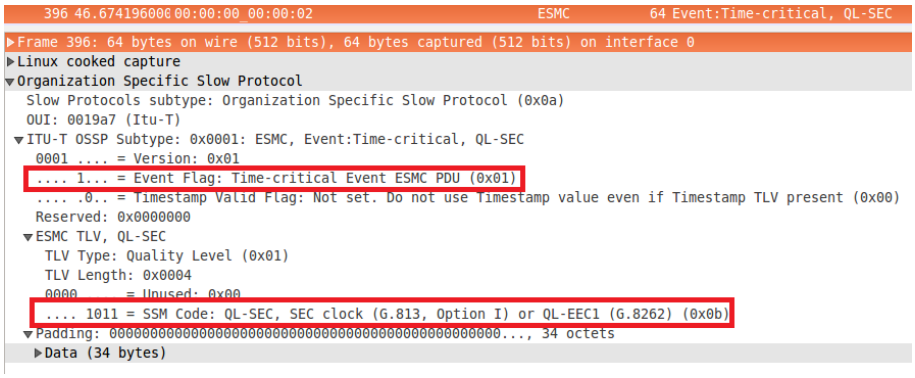

Figure 4.25: ESMC PDU.

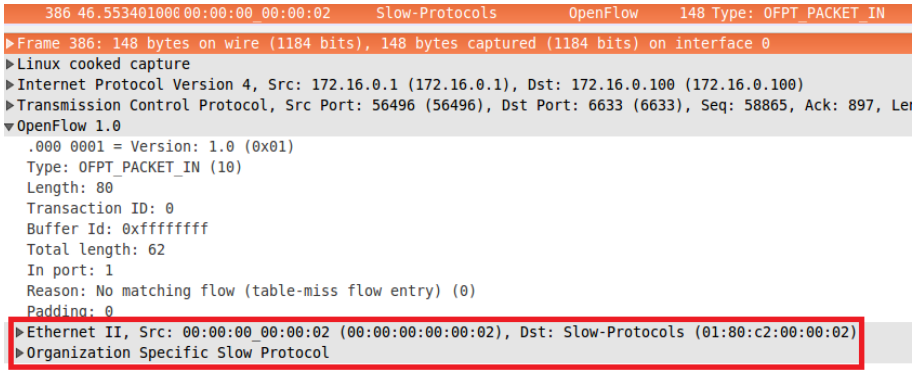

Figure 4.26: ESMC PDU carried by an OFPT_PACKET_IN message to the controller.

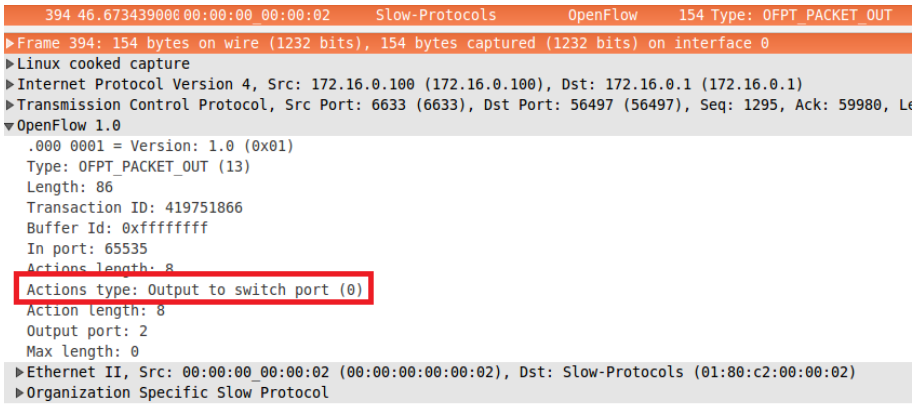

Figure 4.27: Generation of an ESMC PDU within an OFPT_PACKET_OUT message to the switch.

4.4. Summary

This chapter has described all the extensions required for OpenFlow, as well as the implementation of those in OpenDaylight and Open vSwitch, in order to create an SDN-enabled SyncE environment. The SEM application hosted in OpenDaylight is able to manage SyncE-related parameters and use them in order to create a synchronization tree in the network. The only extension that has not been fully implemented is the delay action because of time constraints during the development of this work. The rule installation is supported, but it is ignored when a packet matches the flow entry with that action. However, this is a minor issue as only affects to scenarios where two SDN synchronous devices belonging to different environments are connected. All the other scenarios have been studied and solved.

CHAPTER 5. RESULTS

This chapter presents how this solution has been implemented using the elements mentioned in section 4.2., as well as the tests performed on the different scenarios and the results obtained.

5.1. Description of the testbed

The testbed, ash shown in figure 5.1, is composed of two virtual machines: one with OpenDaylight, where the controller is hosted (control plane) and another one with Open vSwitch (data plane), where the virtual switches and the test topology is hosted. Between these two VM there is a link to allow communication. In order to connect the virtual switches to create a topology, several virtual Ethernet interfaces (or veth) must be set up. A veth is a virtual link between an input and an output point, so configuring an input point in one switch and an output point in another switch allows a full-duplex communication between them. In order to obtain connectivity between every virtual switch and the controller, the switches are connected to the interface that connects with the controller. The network manager is connected to OpenDaylight through a web service, and to the Open vSwitch VM in order to allow an easy and flexible configuration and debugging.

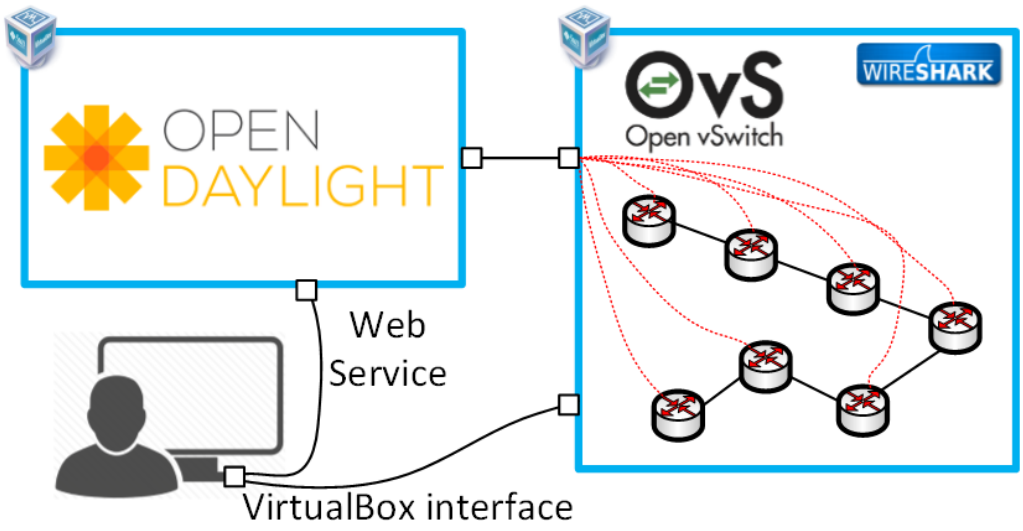

Figure 5.1: Configured scenario.

In order to test the SDN SyncE implementation, the following tests have been performed:

1. Calculation and establishment of the Synchronization tree.
2. Configuration time of a simple network when a PRC clock appears.
3. Unconfiguration time of a simple network when a PRC clock is lost.
4. Bandwidth consumption of the solution.
5. System response to the injection of background traffic.

5.2. Test #1: Calculation of the synchronization tree

In order to demonstrate that the implementation works as expected, an initial functional test has been performed. The test is characterized as follows (figure 5.2):

- **INITIAL STATE**: OpenDaylight is in a steady state with 2 switches connected to different PRCs and 9 switches with QL values of QL-EEC1 which are not synchronized. OpenDaylight is configured to request description statistics every 0.5 seconds.
- **EVENT**: The controller detects a switch with a QL of QL-PRC.
- **FINAL STATE**: The network is synchronized by a PRC.

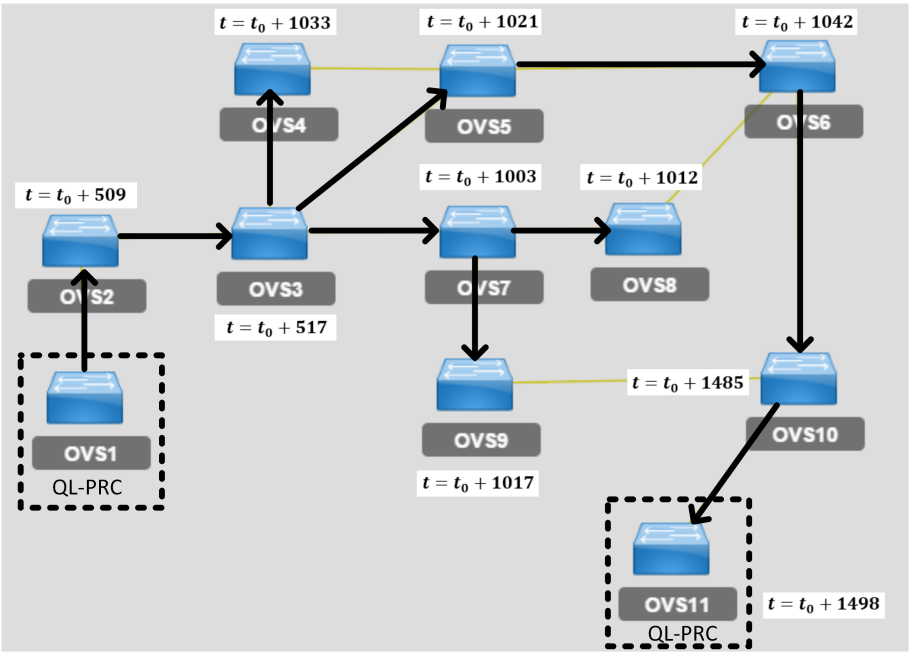

Figure 5.2: Synchronization tree of a complex network. Times (in ms) are referenced to t_0 which is the time on which OVS1 and OVS11 are connected to the network.

When the controller discovers the topology of the network, it starts polling each device about their statistics every 0.5 seconds. All the devices are polled at the same time and in the same order (the order might not be the same on which the nodes were connected to the controller, and can change if the topology changes). As the controller is receiving statistics, it keeps track of the best known QL at the moment so the root of the synchronization tree can be set. In this case, some devices with QL-EEC1 replied first but they were not established as the root of the synchronization tree as their QL was not good enough (QL-EEC1). When OVS1 replied with its statistics, the root of the synchronization tree was set. If OVS11 had replied before, OVS11 would have become the root. After the root has been chosen, the synchronization tree grows as explained in the selection process (section 4.3.3.), this is, two or more levels of the synchronization tree are configured. After 0.5 seconds, when the controller polls statistics again, two or more levels of the synchronization tree will be configured, and so on till the tree is fully configured. Keep in mind that when a node N_k is synchronized, the controller also checks if an adjacent

node can be synchronized by N_k. The synchronization tree grows till it reaches OVS11. The figure shows the time when each device was configured, and from that we can see how the synchronization tree was calculated. In the first polling round, OVS2 and OVS3 were configured. In the second one, OVS7, OVS8, OVS9, OVS5, OVS4 and OVS6 were configured. In the last round, OVS10 and OVS11 were configured. The total time that the network needed to synchronize is 1.5 seconds.

After some time, OVS1 looses its PRC and enters in holdover mode with QL-EEC1 (figure 5.3). This is detected by the controller when polls about the statistics. Then, the root of the synchronization tree is removed, the whole synchronization tree is unconfigured and as OVS11 has a QL-PRC quality, OVS11 is set as the root of the synchronization tree. The process of building the synchronization tree goes as explained before. The whole tree is configured in approximately 1 second.

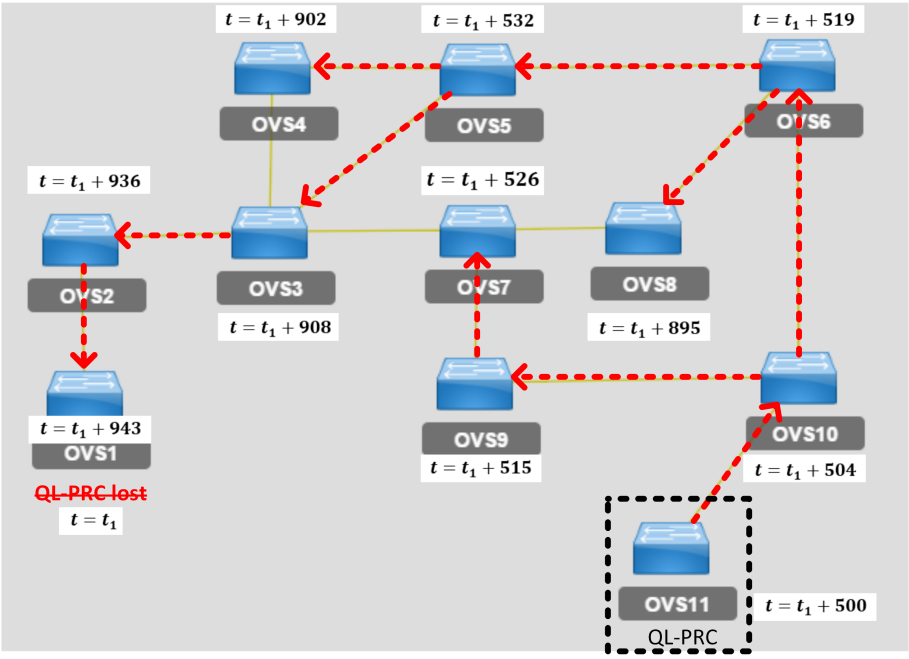

Figure 5.3: New synchronization tree of a complex network after a failure. Times (in ms) are referenced to t_1 which is the time of failure of the PRC.

Notice that it was not required to to manually pre-configure any device, proving that there is no need for engineering a main and a backup synchronization tree for avoiding possible timing loops. Time requirements as well as topology requirements can be programmed in the selection process to meet any requirement. The only possible configuration parameter that a network operator might expect is the possibility of configuring which is the main clock and which is the backup one. However, this feature has not been implemented in this solution.

5.3. Test #2: Configuration time of the synchronization tree in a linear topology

The goal of this test is to measure how much time does the system need to configure the test network. The network is formed by 10 nodes in linear topology (worst case). The first node is connected to a PRC and has a QL value of QL-PRC while others have QL-EEC1. The system should configure the nodes to get synchronization from the switch with the best QL value in the same way as in test #1.

The initial and final states of the network are the following (figure 5.4):

- **INITIAL STATE**: OpenDaylight is in a steady state with 9 switches with QL values of QL-EEC1 which are not getting any useful synchronization source. OpenDaylight is configured to request description statistics every 0.5 seconds.
- **EVENT**: OVS1, that connected to a PRC, is connected to the network.
- **FINAL STATE**: There are 10 switches in the network and all of them are running with a QL of QL-PRC. This QL is derived from the switch that is connected to a PRC.

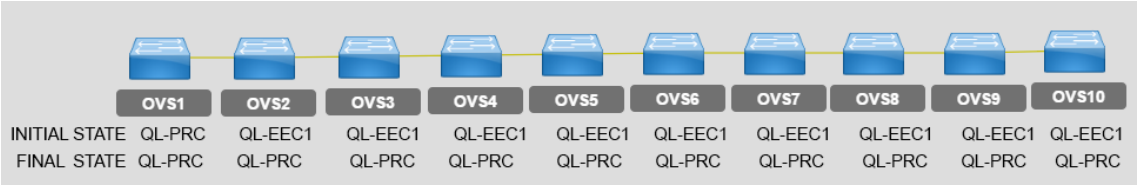

Figure 5.4: Initial and final states of test #2.

The results for the configuration time of the synchronization tree are shown in figure 5.5. The x-axis shows the switch that is configured, from OVS1 to OVS10, while the y-axis shows the relative time on which every switch was configured. Switches are considered to be configured upon reception of an OFPT_SET_CONFIG message. Times are referenced to when the controller detects that a switch is connected to a PRC (this is, the establishment of the root of the synchronization tree). The results have been compared with the results that should be obtained from a legacy SyncE network (denoted as SyncE limits in the figure) in the same conditions. The total synchronization time in that case would be determined by the SyncE time requirement T_{SM} = [180, 500] ms, which means a minimum total configuration time of T_{min} = 180 x *#tree-levels* and a maximum total configuration time of T_{max} = 500 x *#tree-levels*. In legacy SyncE networks, the total configuration time is dependent on the number of levels of the synchronization tree because the nodes do not have information of QL of nodes that are not their neighbours. In this case, there are 10 tree levels.

We have tested three different approaches for the calculation of the synchronization tree, including the extreme cases that require the longest and the shortest times:

- **Configuration #1**: Based only on the SyncE approach, it determines that the selection process is to be executed for a node N_k if its neighbour's QL

has changed, resulting in the configuration of node N_k. At the end, every time that the controller polls about statistics, one level of the synchronization tree will be configured leading to a line in the figure following the form Accumulated_time(tree-level) = 0.5 seconds x tree-level + 0.5 seconds. The offset of 0.5 seconds is the time that the controller needs to detect the root of the synchronization tree, determined by the polling time.

- **Configuration #2**: This approach is based in the 'improved SyncE' approach described in subsection 4.3.3. and used in test #1. That configuration leads to two possible increments of time: $increment_A$ of 500 ms and $increment_B$ of some milliseconds. $Increment_A$ is due to polling time (500 ms) and shows that the node was synchronized because there was an update of the QL and the node was not synchronized yet. $Increment_B$ is due to processing time of the selection process (some milliseconds) and shows that the node N_k was synchronized because when the selection process ran for an adjacent node, it looked for its neighbours to run the selection process on them.

- **Configuration #3**: This approach is based on the 'centralized view' of the SDN approach. This approach determines that upon detection of the root of the synchronization tree, it will manage to configure the whole network. The configuration time is determined mainly by the polling time: Accumulated_time(tree-level) = 0.5 seconds + processing_time x tree-level where the processing time is the combination of the time required to run the selection process for the given node and the transmission of the OFPT_SET_CONFIG message.

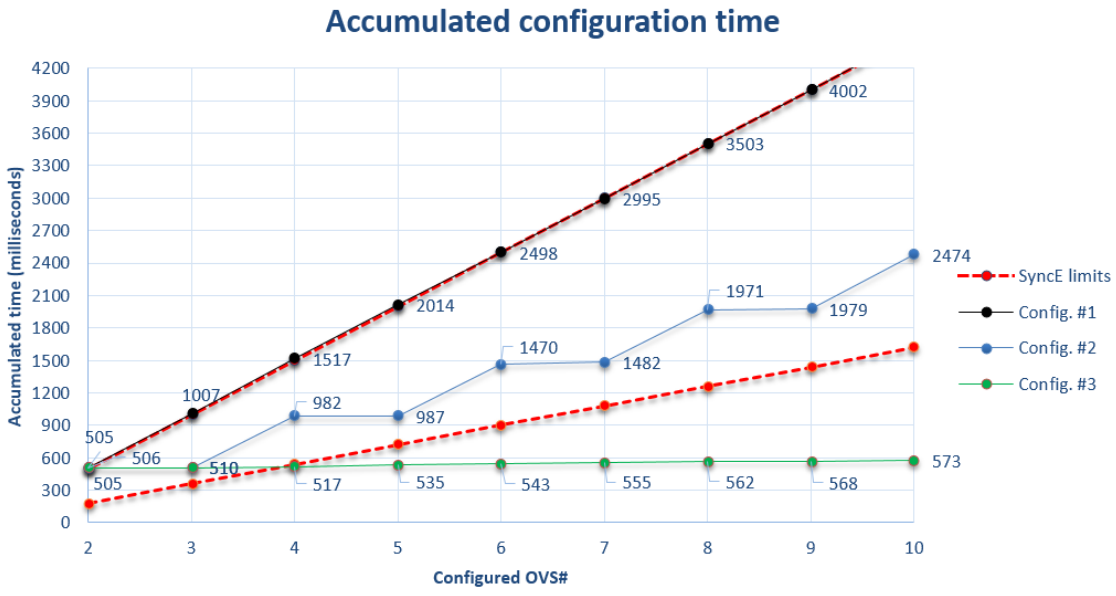

Figure 5.5: Configuration time of the synchronization tree.

In all cases the time requirements are met, as the results in the figure are below the SyncE limits. Moreover, the network can be configured below the minimum SyncE limit, and this is due to the centralized view of SDN. This feature of SDN allows that the centralized entity knows all the information of the network, so it is like all the devices knew (virtually)

about everyone's QL, thus performing better. However, as a polling time of 500 ms is very intensive (compared to the default configured polling time of 60 seconds of OpenDaylight), the bandwidth consumption is increased, as explained in further sections.

The configuration time of the synchronization tree and the increase in the bandwidth consumption could be improved if the statistics were sent asynchronously when an event occurs (change of QL or signal-fail), as the polling of statistics would not be needed anymore. This approach would require the controller to configure the switch to monitor some statistics, and to notify the controller if the statistics change. The switch should be able to understand this message, to monitor the required statistics and to trigger a statistics message asynchronously (following the idea of the OFPT_FLOW_REMOVED message). This feature has not been implemented in this project only due to time limitations.

5.4. Test #3: Unconfiguration time of the synchronization tree

The goal of this test is to measure how much time does the system need to unconfigure the test network upon a loss of PRC. The network is formed by 10 nodes following a linear topology. The first node is connected to a PRC and has a QL value of QL-PRC while others have QL-EEC1, but are configured to obtain synchronization from the first node. Upon detection of the loss of the PRC, the system should unconfigure **all** the nodes belonging to the 'broken' branch as there are not other possible clock sources. After unconfiguration, it is possible to start the process to build a new synchronization path.

The initial and final states of the network are the following (figure 5.6):

- **INITIAL STATE**: OpenDaylight is in a steady state with 10 switches with native QL values of QL-EEC1, but obtaining a QL value of QL-PRC from the first node, which is connected to a PRC. OpenDaylight is configured to request description statistics every 0.5 seconds.
- **EVENT**: The PRC is lost.
- **FINAL STATE**: There are 10 switches in the network and all of them are running with their native clocks (QL-EEC1).

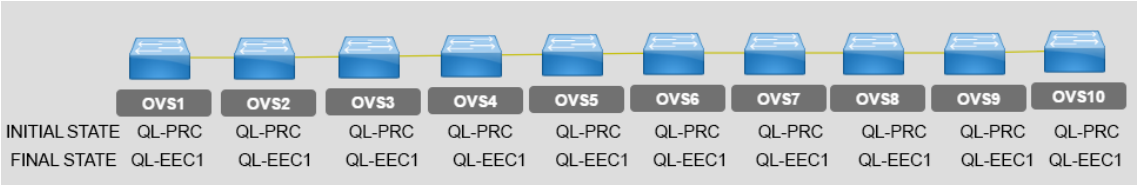

Figure 5.6: Initial and final states of test #3.

The results for the unconfiguration time of the synchronization tree are shown in figure 5.7. The x-axis shows the switch that is configured, from OVS1 to OVS10, while the y-axis shows the relative time on which every switch was unconfigured. Switches are considered to be unconfigured upon reception of an OFPT_SET_CONFIG message. Times

are referenced to the time on which the PRC was lost in the switch. The results have been compared with the results that should be obtained from a legacy SyncE network (denoted as SyncE limits in the figure) in the same conditions. The total unconfiguration time in that case would be determined by the SyncE time requirement T_{HM} = [300, 2000] ms, which means a minimum total unconfiguration time of T_{min} = 300 x *#tree-levels* and a maximum total unconfiguration time of T_{max} = 2000 x *#tree-levels*. The total unconfiguration time in legacy SyncE is dependent on the number of levels of the synchronization tree because the nodes do not have information of QL of nodes that are not their neighbours. In this case, there are 10 tree levels.

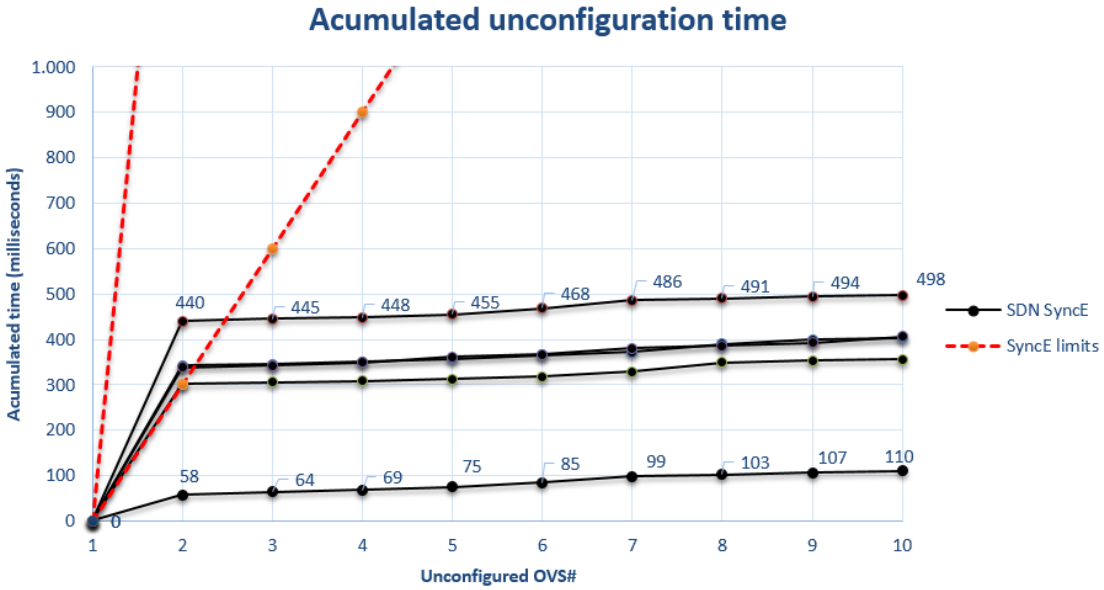

Figure 5.7: Unconfiguration time test.

The results show a family of curves labelled as 'SDN SyncE' and are a result of multiple tests of the same scenario. Depending on the time when the test starts, the unconfiguration time is different. This is because if the test started just after it is time for the polling of statistics, the detection time will be very short. However, if the controller polls statistics and right after the test starts, it will take approximately 500 ms to do another polling of statistics and the detection time will be longer that in the previous case. In all cases, the curves have two possible increments: $increment_A$ of 500 ms and $increment_B$ of some milliseconds. $Increment_A$ is due to polling time (500 ms) and shows that the node (OVS2 in this case) was unconfigured when the controller detected that the PRC source was lost. $Increment_A$ only appears once. $Increment_B$, that also appears only once, is due to processing time (some milliseconds) for sending unconfiguration messages to all the nodes in the affected branch (OVS3 to OVS10).

In all cases the time requirements are met, as the results in the figure are below the SyncE limits. Moreover, the network is configured below the minimum SyncE limit, and this is due to the centralized view of SDN. This feature of SDN allows that the centralized entity knows all the information of the network, so upon notification of a PRC failure, the affected branch(es) are unconfigured at once. However, as a polling time of 500 ms is very intensive (compared to the default configured polling time of 60 seconds), the bandwidth consumption is increased, as explained in the next subsection.

As described in subsection 5.3., the unconfiguration time of the synchronization tree could be improved as well if the statistics were sent asynchronously when an event occurs (change of QL or signal-fail), as the polling of statistics would not be needed anymore.

5.5. Test #4: Bandwidth consumption

The goal of this test is to measure how much bandwidth is used as a result of the intensive polling of statistics (SDN-enabled SyncE) compared to the default configuration of polling (SDN withouth SyncE). Table 5.1 compares the configuration parameters and bandwidth consumption for both cases, at the Ethernet frame level. Figure 5.8 shows measured bandwidth consumption for isolated messages (OFPST_DESC in black and OFPST_PORT in red).

Parameter	**Default value**	**Configured value**
OFPST_DESC polling time	60 seconds	0.5 seconds
OFPST_DESC frame size	1134 bytes	1166 bytes
OFPST_DESC BW	0.16 Kbps	18.66 Kbps
OFPST_PORT polling time	5 seconds	5 seconds
OFPST_PORT frame size	294 bytes	302 bytes
OFPST_PORT BW	0.47 Kbps	0.48 Kbps

Table 5.1: Theoretical bandwidth consumption.

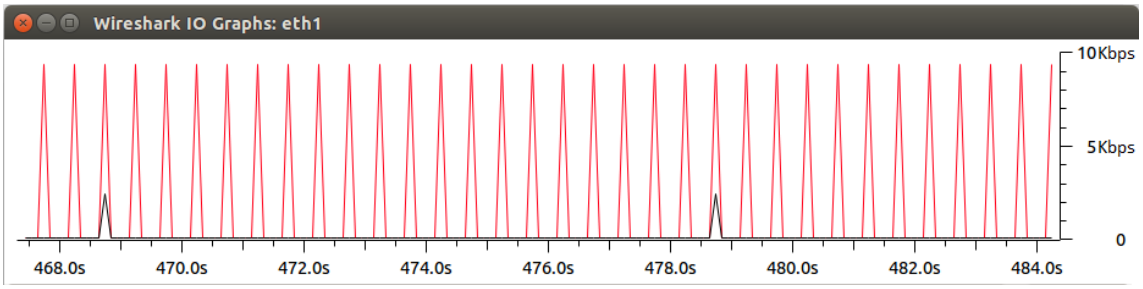

Figure 5.8: Bandwidth consumption for OFPST_DESC in red (high values) and OFPST_PORT in black (low values) during 16 seconds.

The average signalling bandwidth is increased in 18.5 Kbps per synchronous node as a result of the intensive statistics polling (as reported by Wireshark in figure 5.9 in a network of 10 synchronous switches), with a total average bandwidth of 24 Kbps per synchronous node, 1.7 Kbps in the downlink and 22.7 Kbps in the uplink[1]. Table 5.2 shows the uplink bandwidth consumption in a network with 10 switches. For networks of hundreds or thousands of nodes, this would mean an increase of the bandwidth consumption from 2.3 Mbps for 100 nodes to 22.7 Mbps for 1,000 nodes in the uplink. The downlink is not affected. If this was an issue, the network manager could double the time of polling statistics to 1 second and get a decrease of a 50 % in bandwidth, but the reaction time to failures would be doubled.

[1]The uplink is considered the communication channel between the switch and the controller, while the downlink is the communication channel between the controller and the switch.

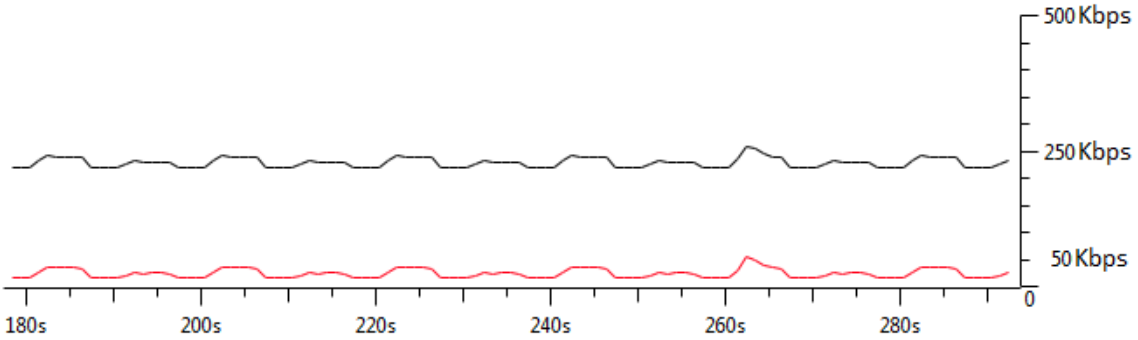

Figure 5.9: Total bandwidth consumption without SyncE nodes in red (low values) and with SyncE nodes in black (high values).

Source	Bandwidth
Statistics polling data (uplink)	205 Kbps
Non-statistics data (uplink)	22 Kbps
Total uplink	227 Kbps

Table 5.2: Bandwidth sources.

This increase in bandwidth consumption could be could be improved if the statistics were sent asynchronously when an event occurs, as described in subsection 5.3., or if an specific statistics message type is created for carrying synchronous information only. With this last solution, the size of the statistics message that carries that information (QL and signal-fail) would be lower than the OFPST_DESC and OFPST_PORT messages combined (so the bandwidth consumption due to statistics polling is improved), without having an impact on the configuration and reaction times.

5.6. Test #5: Injection of background traffic

The goal of this test is to measure how much time does the system need to configure the test network when it carries background UDP traffic that may produce delays and loss of packets. Since OpenFlow uses TCP for a reliable communication, losses produce retransmissions, which means a delay in the communication and an extra traffic. The used topology is the same as in test #2 (figure 5.4). The system should configure the nodes to obtain synchronization from the switch with the best QL value. The configuration #3 from test #2 (centralized view of SDN) is used for a better performance. The configuration of the whole system is shown in figure 5.10, where the link between virtual machines is limited through a traffic shaper provided by VirtualBox. This link from the Open vSwitch to the controller (uplink) has been limited to 1 Mbps and will be the bottleneck.

The initial and final states of the network are the following:

- **INITIAL STATE**: OpenDaylight is in a steady state with 9 switches with QL values of QL-EEC1 which are not getting any synchronization source, and a switch connected to a PRC. OpenDaylight is configured to request description statistics every

0.5 seconds. There is a background traffic that may produce delays and loss of packets.

- **EVENT**: A switch attached to a PRC is connected to the network.
- **FINAL STATE**: There are 10 switches (about 227 Kbps of signalling bandwidth consumption in the uplink) in the network and all of them are running with a QL of QL-PRC. This QL is derived from the switch that is connected to a PRC.

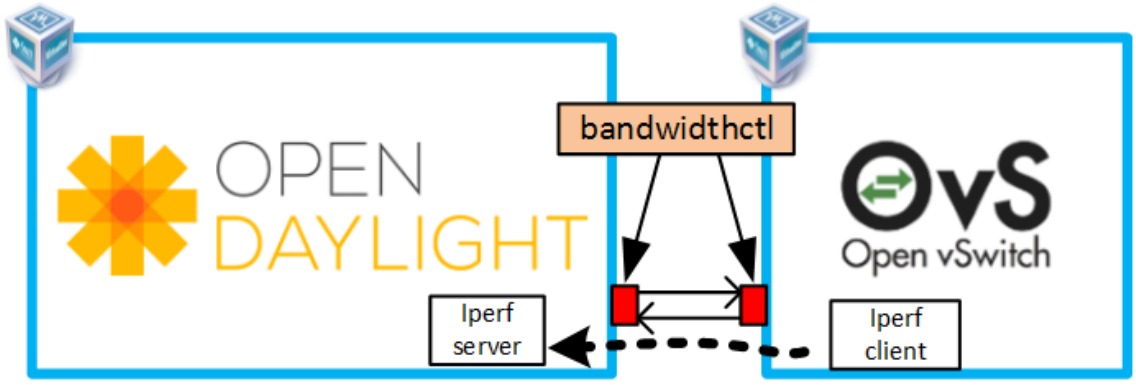

Figure 5.10: Scenario for test #5.

Test #2 showed that a network like the one object of this test can be configured in approximately half a second. In that case, the time to configure a node (initial delay) is determined by the polling time (0.5 seconds) plus the processing time (some milliseconds). Test #4 showed that the bandwidth consumption per synchronous node in the uplink is about 23 Kbps, which is 230 Kbps for a network of 10 nodes.

Table 5.3 shows the relation between the background traffic in the uplink, the initial delay and the processing time per tree level. Values are an average, and those between braces are the minimum and maximum values obtained from the several tests that have been performed. The processing time per tree level is the mean value of all the processing times obtained with the same background traffic. As the background traffic increases, the higher is the initial delay and the processing time. As explained in test #2, the total configuration time is set by the SyncE time requirement T_{SM} = [180, 500] ms. This requirement defines a maximum initial delay of 500 ms and a maximum processing time per tree level of 500 ms. From the measurements of table 5.3, the maximum background traffic that does not violate the aforementioned requirements is 760 Kbps, while higher values make the initial delay to be higher. Squeezing the available bandwidth to 200 Kbps provokes an increase on the initial delay to 700 ms, which does not meet the time requirements. Although the needed available bandwidth for the statistics is 200 Kbps, the switches also need to send keep-alive messages as well as OFPT_PACKET_IN messages, which generates an extra bandwidth consumption. Higher background traffics make that the initial delay and the average processing time are increased, therefore it does not meeting the time requirements.

For a background traffic higher than 850 Kbps (highlighted in orange), the controller does not receive some statistics messages due to the failure of several retransmissions of those messages. For a background traffic higher than 900 Kbps (highlighted in red), the controller is not able to receive enough *Hello* messages and switches are considered to time

out, thus destroying the synchronization tree. For those cases, the network is not able to work.

Background traffic	Initial delay	Processing time per tree level
0 Kbps	500 ms	~5 ms
760 Kbps	500 ms	~67 ms {17, 118}
800 Kbps	~739 ms {242, 1300}	~86 ms {7, 238}
825 Kbps	~2522 ms {1421, 3400}	~504 ms {111, 1266}
850 Kbps	~3883 ms {230, 10107}	~267 ms {98, 478}
875 Kbps	~3025 ms {470, 4213}	~308 ms {100, 600}
900 Kbps	~3588 ms {1155, 7158}	~433 ms {278, 555}
925 Kbps	~8045 ms {5370, 10721}	~944 ms {799, 1089}

Table 5.3: Relation between the background traffic in the uplink, the configuration delay and the processing delay per tree level. Values are averages and those between braces are the minimum and the maximum values obtained from several test.

5.7. Test #6: Interoperability between SDN-enabled SyncE and non-SDN environments

In order to demonstrate that the implementation works as expected regarding the interoperability with other non-SDN environments, a functional test has been performed. The test is characterized as follows (figure 5.11):

- **INITIAL STATE**: OpenDaylight is in a steady state with 11 switches with QL values of QL-EEC1. There is no master clock. OpenDaylight is configured to request description statistics every 0.5 seconds.
- **EVENT**: OVS5 receives an ESMC PDU with QL value of QL-PRC from a non-SDN environment.
- **FINAL STATE**: The network is synchronized by an external PRC that belongs to another network through OVS5.

When OVS5 starts receiving ESMC PDUs (meaning that a synchronous device has been connected to OVS5), OVS5 notifies this to the controller by means of an OFPT_PACKET_IN message. As this is the best clock that the network can get, the controller makes OVS5 the root of the synchronization tree. Afterwards, when the controller polls about statistics, the network is configured in the same way as in test #1. While OVS5 keeps receiving ESMC PDUs, OVS5 is maintained as the root of the synchronization tree. If OVS5 stops receiving ESMC PDUs and the IT timer expires, the synchronization tree is unconfigured. The figure shows the time when each device was configured, and from that we can see how the synchronization tree was calculated. In the first polling round, OVS4, OVS3, OVS6, OVS8 and OVS10 were configured. In the second one, OVS9, OVS11 and OVS2 were configured. In the last round, OVS1 was configured. The total time that the network needed to synchronize is 1 second.

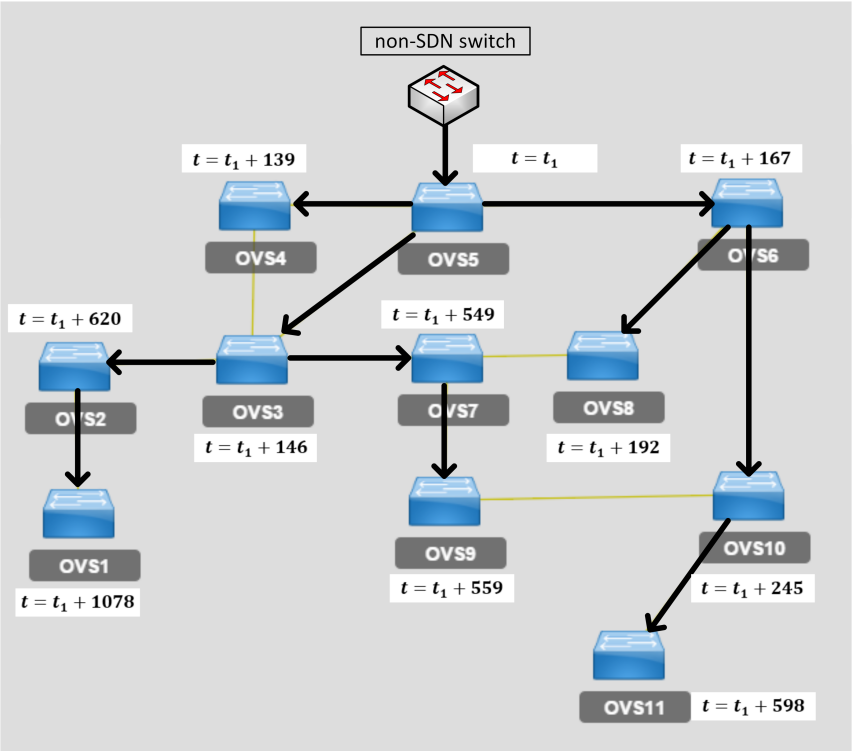

Figure 5.11: Synchronization tree of a complex network through a non-SDN switch. Note that times (in ms) are referenced to t_1 which is the time on which OVS5 received the first ESMC PDU.

5.8. Summary

This chapter has described functional and performance tests for the implemented SDN-enabled SyncE architecture.

Tests #1 and #6 are functional tests that have demonstrated that the system is capable of calculating and configuring a synchronization tree, as well as the interoperability with other non-SDN networks.

Tests #2 and #3 showed that depending on the approach for the calculation of the synchronization tree, the total configuration time of the network can the same as in legacy SyncE networks, or can be improved by taking advantage of the centralized view that SDN offers, obtaining in this case a total configuration time of approximately half a second. Both cases can be improved in performance if the statistics could be generated by the switches as asynchronous messages.

In test #4 the bandwidth consumption has been measured obtaining an important increase with respect to the legacy SyncE case, and the impact of it has been studied in test #5, which showed that while the required uplink bandwidth of the statistics is maintained, the total configuration time of the synchronization tree stays in the time requirements established by SyncE. The bandwidth could be improved if the statistics were generated by the switches as asynchronous messages. Alternatively, the creation of a custom bundle of statistics for carrying synchronous information parameters could also lower bandwidth usage.

CHAPTER 6. CONCLUSIONS AND FUTURE LINES OF STUDY

6.1. Conclusions

This project has demonstrated the feasibility of an SDN-enabled Synchronous Ethernet architecture, merging the advantages of both architectures: the centralized control of SDN and programmability of a network, and the synchronization of SyncE. This is a little step towards synchronization on SDN networks, that we expect to become a key issue in the deployment of carrier and mobile networks such as the new 5G.

By building a test implementation based on OpenDaylight, a well-known and used controller, and Open vSwitch, one of the most popular virtual switch software, we have shown that it is realistic and would not be far from an implementation in real networks, after some resource optimization. The extensions for OF1.0 that have been proposed provide to the controller all the information regarding synchronization, thus making it capable of deciding the best synchronization tree for the network, and the network manager being able to manage and configure the network in a centralized manner.

OpenFlow, the main pillar of this project, is developing new versions thus making some of the explained extensions obsolete. On the contrary, extensions are becoming easier to implement in newer implementations of OpenFlow such as OF1.5, in which custom matching options and custom action sets are supported by the standard protocol. However, the work behind the implementation of those extended features in Open vSwitch and OpenDaylight are not obsolete.

The results of the SDN-enabled SyncE architecture are more than satisfying. The SDN controller is able to create a synchronization tree meeting the time requirements established by Synchoronous Ethernet, and allowing a very fast unconfiguration time when a failure on a branch is detected. It is also able to extend synchronization trees from other SDN or non-SDN networks, allowing a shared synchronization tree and interoperability. Among the advantages of the new architecture, network managers can easily configure the network just by setting which are the priorities of the main clocks (or if they are not configured they are chosen randomly). In legacy SyncE/SDH networks, network operators had to engineer and configure the priorities of the ports of all the devices of the network (which is a problem when the network is composed of thousands of nodes) in order to avoid synchronization loops and maintain a backup synchronization tree. Now, this is automatically done by the controller thus easing network management. The only disadvantage of this solution is the bandwidth consumption which can be a problem for networks with thousands of synchronous devices.

To summarize, this project has achieved frequency synchronization using a SDN-enabled Synchronous Ethernet architecture solution by means of OpenFlow extensions.

6.2. Future lines of study

Several lines of study for future developments have been identified during the execution of this project, as we describe now.

The first one is to migrate the extensions and implementations from OF1.0 to OF1.3 (which is now fully supported by OpenDaylight) or OF1.4/1.5, and to other versions of OpenFlow as the implementations are released. Today, this migration might not be straightforward but it is possible: for example, the OFPT_FLOW_MOD message now supports custom match options which could be used for maintaining compatibility with legacy OpenFlow implementations. Other messages such as OFPST_DESC, OFPST_PORT, OFPT_FLOW_REMOVED and OFPT_FEATURES_REPLY are maintained without any change. However, the latest specifications do not allow (custom) asynchronous messages on statistics. This feature has been described as a solution to the intensive bandwidth consumption in this project, and could be an improvement not only for SDN-enabled SyncE networks, but for any other SDN network: why requesting every time for statistics if the device could notify configured events on statistics? If a future version of Openflow added such an asynchronous notification, the performance of our solution would improve noticeably.

The specification of OpenFlow 1.5 has defined a way of performing several operations that required multiple OF messages in previous versions, with just a single message, by creating a bundle. This idea could also be applied to statistics, so instead of requesting predefined sets of statistics (such as transmitted and received bytes in the case of ports, the datapath descriptor, or the serial number, to name a few examples) a bundle of statistics could be made for requesting specific parameters in a single statistics message (i.e. QL and the signal-fail of ports 1, 2 and 3).

In this project we have not been able to use real, hardware-based synchronous OpenFlow switches to test the SDN-enabled SyncE architecture. Such tests are the natural following step to validate our implementation. Currently we are contacting synchronous devices providers in order to do that in future projects.

Finally, an interesting path to explore is to add phase to the frequency synchronization capabilities of our architecture. Enabling of phase synchronization the current SDN-enabled SyncE architecture would allow a better management of time and frequency resources in next generation mobile networks like 5G [26]. This phase synchronization can be achieved by means of IEEE 1588 PTP standard [10], a protocol that uses message timing flows between devices to exchange timestamps that allow a phase correction of the clock of a device. A ‘Time Synchronous Ethernet’ has been proposed in [27] by combining PTP and SyncE. It would be interesting to extend our architecture in that direction, possibly by incorporating a recent Openflow extension proposal [8] that allows time-triggered configuration updates.

6.3. Environmental impact

During the development of this project it has only been used a computer. The environmental impact on that is just the power consumption of such computer.

Although one of the possibilities of SDN in the environmental impact field is the potential optimization of the energy consumption of computer networks (as is the case in [28]), the specific area of synchronization is not directly correlated with power usage.

6.4. Publications

A paper describing this project has been published in the 1st IEEE Conference on Network Softwarization (NetSoft 2015) and presented on April 17, 2015 in the University College of London (UCL), as a synchronization solution for SDN networks [29].

BIBLIOGRAPHY

[1] J. Aweya, “Implementing Synchronous Ethernet in Telecommunication Systems”, IEEE Communications Surveys & Tutorials, vol. 16, no 2, pp. 1080–1113, 2014. Page(s) 1

[2] J. Aweya, “Emerging Applications of Synchronous Ethernet in Telecommunication Systems”, IEEE Circuits and Systems Magazine, vol.12, no.2, pp. 56-72, 2012. Page(s) 1

[3] N. Feamster, J. Rexford, E. Zegura, “The Road to SDN, An intellectual history of programmable networks”, ACM SIGCOMM Computer Communication Review, vol 44, Issue 2, pp. 87-98, April 2014. Page(s) 1

[4] Open Networking Foundation. OpenFlow specifications. Available at https://www.opennetworking.org/sdn-resources/onf-specifications/openflow (last visit April 23, 2015) Page(s) 1

[5] ETSI Network Functions Virtualisation Industry Specification Group. http://www.etsi.org/technologies-clusters/technologies/nfv (last visit April 23, 2015). Page(s) 1

[6] Telco Systems T-Marc 3348 EthMPLS (datasheet). http://support.telco.com/index.php?page=download&file=a116&ref=9 (last visit April 23, 2015). Page(s) 2

[7] T. Mizrahi, Y. Moses, “ReversePTP: A Software Defined Networking Approach to Clock Synchronization”, Proceedings of the Third Workshop on Hot Topics in Software Defined Networking HotSDN’14, pp. 203-204, August 2014. Page(s) 2

[8] T. Mizrahi, Y. Moses, “Time-based Updates in OpenFlow: A Proposed Extension to the OpenFlow Protocol”, CCIT Report #835, July 2013, EE Pub No. 1792, Technion, Israel, July 2013. Available at http://tx.technion.ac.il/ dew/OFTimeTR.pdf. Page(s) 2, 54

[9] IEEE Standards Association, 802.3. https://standards.ieee.org/about/get/802/802.3.html (last visit April 23, 2015). Page(s) 3

[10] IEEE TC 9, “1588 IEEE Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems V2”, 2008. Page(s) 2, 54

[11] Recommendation ITU-T G.8010 / Y.1306: Architecture of Ethernet layer networks. Page(s) 3

[12] Recommendation ITU-T G.8264 / Y.1364: Distribution of timing information through packet networks. Page(s) xi, 3, 4, 7

[13] Recommendation ITU-T G.8261 / Y.1361: Timing and synchronization aspects in packet networks. Page(s) 4

[14] Recommendation ITU-T G.8262 / Y.1362: Timing characteristics of synchronous Ethernet equipment slave clock (EEC). Page(s) 4

[15] Recommendation ITU-T G.803: Architectures of transport networks based on the Synchronous Digital Hierarchy (SDH). Page(s) 4

[16] SyncE synchronization network model. http://en.wikipedia.org/wiki/Synchronous_Ethernet (February 23, 2015). Page(s) xi, 5

[17] Recommendation ITU-T G.781: Synchronization layer functions. Page(s) 6, 7, 8

[18] Software-Defined Networking (SDN) Definition. https://www.opennetworking.org/sdn-resources/sdn-definition (April 9, 2015). Page(s) 11

[19] Open Networking Foundation home page. https://www.opennetworking.org/sdn-resources/openflow (last visit April 23, 2015). Page(s) 12

[20] OpenFlow Switch Specification 1.0.0 (Wire Protocol), December 31st, 2009. https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.0.0.pdf (February 11, 2015). Page(s) 12, 13, 67

[21] Open vSwitch home page. http://openvswitch.org/ (April 23, 2015). Page(s) 13

[22] Open vSwitch Overview by Justin Pettit, March 3rd, 2011. http://openvswitch.org/slides/OVS-Overview-110303.pdf (February 11, 2015). Page(s) 13

[23] List of members supporting ODL project. http://www.opendaylight.org/project/members (February 11, 2015). Page(s) 15

[24] OpenDaylight Controller: Overview. https://wiki.opendaylight.org/view/OpenDaylight_Controller:Overview (February 11, 2015). Page(s) 15

[25] OpenDaylight repository for the code commited on October 24th, 2014, 2:18 PM. https://git.opendaylight.org/gerrit/#/c/12202/ (last visit February 11, 2015). Page(s) 20

[26] Synchronous Ethernet to transport frequency and phase/time, September 2012. http://www.comsoc.org/ctn/synchronous-ethernet-transport-frequency-and-phasetime (April 27, 2015). Page(s) 54

[27] K. Hann, S. Jobert, S. Rodrigues, “Synchronous Ethernet to transport frequency and phase/time”, IEEE Communications Society, vol. 50, issue 8, pp. 152-160, 2012. Page(s) 54

[28] Sergio Jiménez, Encaminament amb optimització de consum energètic en una Software-Defined Network, Degree thesis, EETAC-UPC, http://upcommons.upc.edu/pfc/handle/2099.1/14060 (last visit April 28, 2015). Page(s) 55

[29] Raúl Suárez, David Rincón and Sebastià Sallent, “Extending OpenFlow for SDN-enabled Synchronous Ethernet networks”, First IEEE Conference on Network Softwarization (Netsoft 2015), workshop on Management Issues in Software-defined networks, Software-defined infrastructure and network function virtualization (MISSION 2015). To be published. Page(s) 55

ACRONYMS

API	Application Programming Interface
DNU	Do Not Use
DTLS	Datagram Transport Layer Security
EEC	Ethernet Equipment Clock
ESMC	Ethernet Synchronization Message Channel
FRM	Forwarding Rules Manager
GB	Gigabyte
GUI	Graphical User Interface
IEEE	Institute of Electrical and Electronics Engineers
IT	Information Timer
ITU	International Telecommunication Union
NE	Network Equipment
NSF	Network Service Functions
NTP	Network Time Protocol
OAM	Operation, Administration and Management
ODL	OpenDaylight
OF	OpenFlow
ONF	Open Networking Foundation
Open vSwitch	Open Virtual Switch
OSGi	Open Services Gateway Initiative
OSSP	Organization Specific Slow Protocol
OVS	Open Virtual Switch
PDH	Plesiochronous Digital Hierarchy
PDU	Protocol Data Unit
PRC	Primary Reference Clock
PTP	Precision Time Protocol
QL	Quality Level
RAM	Random-Access Memory
REST	Representational State Transfer
RTP	Real-time Transport Protocol
SAL	Service Abstraction Layer
SDH	Synchronous Digital Hierarchy
SDN	Software Defined Networking
SEM	Synchronous Ethernet Manager
SSM	Synchronization Status Message
SSU	Secondary Synchronization Unit
STM-N	Synchronous Transport Module N
SyncE	Synchronous Ethernet
TCP	Transmission Control Protocol
TDM	Time-Division Multiplexing
TLS	Transport Layer Security
UDP	User Datagram Protocol
VM	Virtual Machine
WTR	Wait To Restore

APPENDICES

APPENDIX A. LIST OF SYNCHRONOUS ETHERNET QL

SyncE defines a list of possible QL values regarding the clock quality from QL-INV0 (best quality) to QL-DNU (worst quality).

Value	Quality Level
0000	QL-INV0
0001	QL-INV1
0010	QL-PRC
0011	QL-INV3
0100	QL-SSU-A
0101	QL-INV5
0110	QL-INV6
0111	QL-INV7
1000	QL-SSU-B
1001	QL-INV9
1010	QL-EEC2
1011	QL-EEC1
1100	QL-INV12
1101	QL-INV13
1110	QL-INV14
1111	QL-DNU

Table A.1: Quality Level values.

APPENDIX B. DUMMY FILES

What follows is an example of the state of the files where the QL and the configuration port parameters are stored for a network of 11 switches. OVS1 and OVS11 are attached to a PRC, as they are synchronized with nobody but still get a good QL, and OVS2 is synchronized with OVS1 through port 1.

Code B.1: Example of a shared QL file dummyqL.data.

```
OVS1=QL-PRC
OVS2=QL-PRC
OVS3=QL-EEC1
OVS4=QL-EEC1
OVS5=QL-EEC1
OVS6=QL-EEC1
OVS7=QL-EEC1
OVS8=QL-EEC1
OVS9=QL-EEC1
OVS10=QL-EEC1
OVS11=QL-PRC
```

Code B.2: Example of a shared configuration port file dummyqL_port.data.

```
OVS1=None
OVS2=1
OVS3=None
OVS4=None
OVS5=None
OVS6=None
OVS7=None
OVS8=None
OVS9=None
OVS10=None
OVS11=None
```

APPENDIX C. MESSAGES OF OPENFLOW 1.0.0

The following description of Openflow messages is an excerpt of Chapter 4 of the Openflow specification v1.0.0 [20].

4.1 OpenFlow Protocol Overview

The OpenFlow protocol 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 of network events and changes to the switch state. Symmetric messages are initiated by either the switch or the controller and sent without solicitation. The message types used by OpenFlow are described below.

4.1.1 Controller-to-Switch

Controller/switch messages are initiated by the controller and may or may not require a response from the switch.

Features: Upon Transport Layer Security (TLS) session establishment, the controller sends a features request message to the switch. The switch must reply with a features reply that specifies the capabilities supported by the switch.

Configuration: The controller is able to set and query configuration parameters in the switch. The switch only responds to a query from the controller.

Modify-State: Modify-State messages are sent by the controller to manage state on the switches. Their primary purpose is to add/delete and modify flows in the flow tables and to set switch port properties.

Read-State: Read-State messages are used by the controller to collect statistics from the switchs flow-tables, ports and the individual flow entries.

Send-Packet: These are used by the controller to send packets out of a specified port on the switch.

Barrier: Barrier request/reply messages are used by the controller to ensure message dependencies have been met or to receive notifications for completed operations.

4.1.2 Asynchronous

Asynchronous messages are sent without the controller soliciting them from a switch. Switches send asynchronous messages to the controller to denote a

packet arrival, switch state change, or error. The four main asynchronous message types are described below.

Packet-in: For all packets that do not have a matching flow entry, a packet-in event is sent to the controller (or if a packet matches an entry with a "send to controller" action). If the switch has sufficient memory to buffer packets that are sent to the controller, the packet-in events contain some fraction of the packet header (by default 128 bytes) and a buffer ID to be used by the controller when it is ready for the switch to forward the packet. Switches that do not support internal buffering (or have run out of internal buffering) must send the full packet to the controller as part of the event.

Flow-Removed: When a flow entry is added to the switch by a flow modify message, an idle timeout value indicates when the entry should be removed due to a lack of activity, as well as a hard timeout value that indicates when the entry should be removed, regardless of activity. The flow modify message also specifies whether the switch should send a flow removed message to the controller when the flow expires. Flow modify messages which delete flows may also cause flow removed messages.

Port-status: The switch is expected to send port-status messages to the controller as port configuration state changes. These events include change in port status (for example, if it was brought down directly by a user) or a change in port status as specified by 802.1D (see Section 4.5 for a description of 802.1D support requirements).

Error: The switch is able to notify the controller of problems using error messages.

4.1.3 Symmetric

Symmetric messages are sent without solicitation, in either direction.

Hello: Hello messages are exchanged between the switch and controller upon connection startup.

Echo: Echo request/reply messages can be sent from either the switch or the controller, and must return an echo reply. They can be used to indicate the latency, bandwidth, and/or liveness of a controller-switch connection.

Vendor: Vendor messages provide a standard way for OpenFlow switches to offer additional functionality within the OpenFlow message type space. This is a staging area for features meant for future OpenFlow revisions.