Virtual Aggregation in OpenFlow Networks

BELGIS CHIAL SÁNCHEZ

Master’s Thesis at Acreo AB
Supervisor: Pontus Sköldström
Examiner: Markus Hidell

TRITA xxx 2013-nn
It is not the critic who counts; not the man who points out how the strong man stumbles, or where the doer of deeds could have done them better. The credit belongs to the man who is actually in the arena, whose face is marred by dust and sweat and blood; who strives valiantly; who errs, who comes short again and again, because there is no effort without error and shortcoming; but who does actually strive to do the deeds; who knows great enthusiasms, the great devotions; who spends himself in a worthy cause; who at the best knows in the end the triumph of high achievement, and who at the worst, if he fails, at least fails while daring greatly, so that his place shall never be with those cold and timid souls who neither know victory nor defeat.

Theodore Roosevelt
To God and my beloved Family, whom made possible the realization of this dream.  
I love you.
Acknowledgements

To Pontus Sköldström for being my mentor and helping me out in every step of the way. Thank you very much!
Abstract

The formulation of a scalable addressing and routing systems has been one the biggest priorities of the Internet Architecture Board (IAB) in recent years [1]. Virtual Aggregation (ViAggre) is a methodology which approaches this problem by allowing an ISP to modify its internal routing such that each router on the ISP’s network will only keep part of the global routing table [2]. This is a ‘configure only’ methodology that has proven to shrink the routing table on routers by an order of magnitude.

Recent developments on Software Defined Networks (SDN) aim to implement them on carrier networks [3]. Some limitations on hardware design are the same for SDN components, which are based on the same type of technology used for traditional routers. In the same spirit as the IAB suggestions in [1], if SDNs are to be deployed on large networks, a scaling method should be visualized too. We propose in this work, the adoption of the ViAggre methodology for SDNs.

In this work we design a ViAggre methodology module for a SDN, taking in consideration the differences between traditional networks and SDNs. Furthermore, we developed an algorithm to split the routing table using a radix trie implementation and we demonstrate that the ViAggre methodology can be ported from its traditional design to a new centralized design by implementing it on a SDN as a NOX controller module.
## Contents

**Contents** vii

**List of Figures** viii

### I Background Study 1

1 Introduction 3
   1.1 Question .................................................. 4
   1.2 Goals ..................................................... 4
   1.3 Outline .................................................. 4

2 Software Defined Networks 7
   2.1 OpenFlow ................................................ 8
      2.1.1 OpenFlow Tables ..................................... 9
      2.1.2 OpenFlow Secure Channel .......................... 12
      2.1.3 Encryption .......................................... 14
      2.1.4 Message Handling .................................. 14
   2.2 NOX controller ........................................... 15
      2.2.1 Developing in NOX ................................. 16

3 Virtual Aggregation methodology 19
   3.1 Control Plane Deployment methods ..................... 20
      3.1.1 Deployment Method I ............................... 20
      3.1.2 Deployment Method II ............................. 20
   3.2 Forwarding Packets ................................. 21

### II Design and Implementation 23

4 Design 25
   4.1 Requirements ........................................... 25
   4.2 Architecture ........................................... 25
5 User space - kernel space communication
5.1 RAM based file systems ........................................ 29
5.1.1 Procfs .................................................. 29
5.1.2 Sysfs ................................................... 30
5.1.3 Configfs ............................................... 30
5.1.4 Debugfs ............................................... 30
5.1.5 Sysctl ............................................... 30
5.1.6 Character devices ....................................... 30
5.2 Sockets .................................................... 31
5.2.1 UDP sockets ........................................... 31
5.2.2 NETLINK sockets ..................................... 31
5.3 Kernel System calls ......................................... 31

6 Implementation ................................................. 33
6.1 Obtaining the forwarding information base .................. 33
6.2 Saving the Forwarding Information Database ................. 37
6.3 Implementing the Virtual Aggregation Methodology ......... 39
6.4 Installing rules to the switches ............................. 42

III Evaluation of the Implementation ................................ 47
6.5 Composition of routes ....................................... 49
6.6 Performance of the algorithm .................................. 51

IV Conclusion and Future work .................................. 53
6.7 Conclusion ............................................... 55
6.8 Future work ............................................... 55

Bibliography ................................................................ 57

List of Figures

2.1 Traditional networks to SDN ................................... 8
2.2 OpenFlow architecture [4] .................................... 9
2.3 Flow chart of matchfield parsing [4] ......................... 10
2.4 OpenFlow matching process[4] ............................... 12
List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Viaggre basic concept [2]</td>
<td>19</td>
</tr>
<tr>
<td>3.2</td>
<td>Deployement method I [2]</td>
<td>20</td>
</tr>
<tr>
<td>3.3</td>
<td>Deployement method II [2]</td>
<td>21</td>
</tr>
<tr>
<td>3.4</td>
<td>Forwarding packets in ViAggre [2]</td>
<td>21</td>
</tr>
<tr>
<td>4.1</td>
<td>Viaggre module architecture</td>
<td>26</td>
</tr>
<tr>
<td>4.2</td>
<td>Viaggre module internal architecture</td>
<td>27</td>
</tr>
<tr>
<td>6.1</td>
<td>Viaggre module communication with routing daemon</td>
<td>34</td>
</tr>
<tr>
<td>6.2</td>
<td>NETLINK event</td>
<td>34</td>
</tr>
<tr>
<td>6.3</td>
<td>Original radix trie implementation</td>
<td>38</td>
</tr>
<tr>
<td>6.4</td>
<td>Updated radix trie implementation</td>
<td>39</td>
</tr>
<tr>
<td>6.5</td>
<td>ViAggre balanced spread of rules [5]</td>
<td>40</td>
</tr>
<tr>
<td>6.6</td>
<td>Radix tree example</td>
<td>41</td>
</tr>
<tr>
<td>6.7</td>
<td>Vector of list installation of routes</td>
<td>43</td>
</tr>
<tr>
<td>6.8</td>
<td>OpenFlow test topology</td>
<td>45</td>
</tr>
<tr>
<td>6.9</td>
<td>Vector of lists</td>
<td>49</td>
</tr>
<tr>
<td>6.10</td>
<td>Vector of lists with optimization</td>
<td>50</td>
</tr>
<tr>
<td>6.11</td>
<td>Aggregation Routes vs Regular Routes</td>
<td>50</td>
</tr>
<tr>
<td>6.12</td>
<td>Difference between MAX and MIN value</td>
<td>51</td>
</tr>
<tr>
<td>6.13</td>
<td>Splitting time vs number of routes</td>
<td>51</td>
</tr>
</tbody>
</table>
Part I

Background Study
Chapter 1

Introduction

The Information Communication Technology revolution driven by the high demand of users, is pushing the limits of modern networks. Furthermore, the Internet’s default-free zone (DFZ) routing table has been growing at a rapid pace over the last years [6] due to factors such as multihoming, traffic engineering, non aggregatable address alocations and business events, such as mergers and acquisitions [1].

A direct consequence of this growth is the harmful growth of the Routing Information Base (RIB) [2]. The RIB’s growth rate surpasses the current 1.3x/year baseline, exceeding Moore’s Law [1]. Due to the growth of the RIB, periodic upgrades of router memory holding the RIB and FIB will be necessary in the future. For this reason and the fact that high-speed routers are based on low volume technology [1], is expected a continuous increase of the cost on technology upgrades, which in return makes networks less cost-effective by increasing the cost of forwarding packet [7]. While memory and processing speeds could scale with the RIB and FIB growth, power and heat dissipation capabilities may not [2], leading routing technology upgrades into various engineering limits.

To proactively avoid the emerging scalability problem, several solutions to reduce the RIB/FIB size have been proposed in [8, 9, 10]. However, the ViAggre methodology has proven to be the most used due to its ‘configure only’ scheme. This methodology proposes to split the FIB and/or RIB and spread the pieces onto different routers in the ISP network. The ViAggre methodology is transparent to external networks, avoiding the impact on networks’ interaction with its peers [2]. However, one major drawback with the Viaggre methodology is that it is quiet configuration heavy and lots of work to setup and maintain.

Another recent development in the networking area are the Software-Defined Networks (SDN). SDN proposes to have a split architecture, where the control plane resides in a central server and the data plane is spread on SDN switches. On SDNs a global view of the network is obtained and used to calculate the traffic flow at the central server, without the interaction of network protocols between the SDN switches (data plane). Recent efforts have been made to implement SDNs on large scale networks [11], where analogously to traditional networks, scalability is a
major concern. Therefore, applying the ViAggre methodology to SDN networks in order to attack the RIB/FIB growth problem is a valid solution, and quiet suitable. Because they are based on an architecture where the control plane is separated from the forwarding plane, the splitting of the FIB would be the main concern, as the RIB will be stored on the central server (control plane); Furthermore, the setup and maintenance would be simpler than the original methodology as it will be all done by an automated process in the central server, this will be further explained on next chapters. In this work we propose the porting of the ViAggre methodology to SDNs. More specifically, we focus on OpenFlow-enabled switches networks, which is one of the available open implementations of an SDN interface.

1.1 Question

The ViAggre methodology has been conceived for traditional networks to alliviate the continuous growth of the DFZ routing table, extending the useful life of the network components. If SDNs are to be implemented as a large scale networks, would it be possible to implement the ViAggre methodology on OpenFlow networks to save resources, even though architectural differences that exist between these two types of networks?

1.2 Goals

The goals of this master thesis can be compiled into the following:

- Investigate how could be implemented the ViAggre Methodology on OpenFlow networks.
- Implement a ViAggre module for a NOX OpenFlow controller.
- Draw conclusions about performance and efficiency from the implementation.

1.3 Outline

This work is divided into the following three big parts:

- Chapter 1 consists of a background study. This is a brief introduction to current Software Defined Networks architecture and also an in-depth study of the OpenFlow protocol as well as the Virtual Aggregation methodology. It also contains a brief explanation of the NOX controller API and what have been done in this domain.
- Chapter 2 contains the requirements and the design of the Virtual Aggregation module. It also describes the steps taken to implement the software this project.
1.3. OUTLINE

• Chapter 3 will have an evaluation of the ViAggre methodology implementation.

• And finally in Chapter 4, draws some conclusions of the work done and suggests some future work.
Chapter 2

Software Defined Networks

In traditional networks routers have both, control plane and data plane on the same box [12]. Once a router has decided what actions to take through routing protocols on the control plane, it will communicate with its data plane hardware through a proprietary API in order to forward the packets[3]. Furthermore, traditional networks are managed through low-level configurations of individual components [13]. The tight coupling of control plane and data plane on the same device leads to overly complicated network management[12]. Vendors and network operators are often reluctant to deploy new technologies and updates due to the fragility and complexity of networks nowadays. This slows the rate at which new network technologies are deployed or even tested compared to the rapid improvement on line speeds, port densities, and hardware performance [12].

Software Defined Networks offer a split architecture where the control plane and data plane are separated which enables the creation of a standardized API between the control plane and the data plane. The control plane in this architecture corresponds to a centralized server with a Network Operating System (NOS). The NOS enables to have a 'global view' of the network, so that modular network applications running on top of it are able to modify and add functionalities to the network, as shown in 2.1. Comparing to traditional networks one could say that the RIB resides in the central server, where it can be updated by applications. The SDN system then transfers the FIB into the connected devices.

The possibility of creating applications to add and test new features to the network without compromising the networks normal functionality, promises a faster adoption and development of new technologies in the network industry.

The possibility of creating applications to add and test new features to the network without affecting the normal function of the network promises a faster adoption and development of new technologies in the network industry. Recent work has been done to adapt all the current network functions to SDNs. However, few have focused on tackling the possible scaling problem if SDNs are to be implemented to large scale networks. Nonetheless, several efforts have been made to attack this problem on traditional networks that could be adapted to SDNs. As mentioned
before, some efforts have been made by dividing the edge networks and ISPs into separate address spaces [8]. In [9] is proposed to reduce the FIB tables by using tunnels and even though is a common practice; it still has the peculiarity that edge routers have to keep the whole FIB. Additionally, approaches to reduce the impact of the RIB growth by compression, route caching and selectively direct traffic through intermediate nodes have been proposed in [9, 14, 15].

The only projects to our knowledge that treats the size of the FIB in SDNs is the DIFANE project [16] and Routeflow. The DIFANE project developed an OpenFlow controller with the same name and implements flow based networking, directing traffic through intermediate nodes by using wildcard rules over the entire packet header structure. The big difference with this work is that they require modifications to the switches’control plane software making it incompatible with existing standards.

2.1 OpenFlow

The Open Network Foundation (ONF) attempts to standardize parts of SDNs through its two standards, the OpenFlow protocol and the OF-Config protocol. The OpenFlow protocol provides an interface between control plane and data plane that can be used to transfer rules to the data plane and network events to the central controller. The control plane is moved from the router to a centralized service called the OpenFlow controller. These networks consist basically of a controller which connects to OpenFlow switches over a secure connection using the OpenFlow protocol. The controller through this protocol can add, update, delete flow entries from the switches. The OpenFlow switches store the flow entries into flow tables or a group tables. Each flow entry first matches an incoming packet, and then it
2.1. OPENFLOW

may forward or modify it for further processing. The processing might be done as a simple scheme group table processing or pipeline processing[4]. The OpenFlow protocol version we used in this work was version 1.1.0, which we will explain in detail in following sections.

Figure 2.2. OpenFlow architecture [4]

2.1.1 OpenFlow Tables

The OpenFlow tables hold the flow entries. In the OpenFlow version used, several flow tables can be used in a same pipeline. Each entry consists of matching fields, instructions and a counter. Each part of the flow entries is explained next [4]:

Matchfield

This field is used to match each incoming packet to see which flow entry should be applied to it. Each field of the flow entry can contain a specific value to be matched and some of them may be wildcarded. We enumerate the fields of a packet used to match an entry as follows:

Ingress port, Metadata, Ethernet source and destination, Ether type, Vlan id, Vlan priority, MPLS label, MPLS traffic class, IPv4 source and destination address, IPv4 protocol/ARP code, IPv4 ToS bits, TCP/UDP/SCTP source and destination ports, ICMP type, ICMP code.

In Fig.3.2 is the flow diagram of how these fields are parsed.

If the switch supports arbitrary bitmasks on the Ethernet source or destination fields, IP source or destination fields, these masks can more precisely specify matches. In addition to packet headers, matches can also be performed against the ingress port or metadata fields. If a flow table field has a value of ANY it matches all possible values in the header.
Counters

The counters have as objective to maintain a log of all events for the: tables, ports, queue, groups, buckets and received errors. The counters wrap around with no overflow indicator. If specific numeric counter is not available in the switch its value should be set to -1 in order to function properly.
Instructions

After matching an incoming packet and updating the counter, an instruction or instruction set is applied to the packet. A set of instructions can only contain one instruction of each type. A switch may reject a flow entry it is unable to execute the instructions associated with the flow entry. The supported types of instructions are:

- **Apply-Actions action(s):** This action can be used to apply the specified action(s) immediately, modify the packet between two tables or to execute multiple actions of the same type.

- **Clear-Actions:** Clears all the actions of the action set

- **Write-Actions action(s):** Adds or overwrite actions in the existing action set

- **Goto-Table next-table-id:** Specifies the next table in the pipeline.

Action Set

Each action set is associated with each packet that matches a flow entry. It is by default empty and modifiable using the instruction Write-Actions, Clear-Actions, and Apply-Actions. When using Write-Actions instruction a maximum of one actions may be applied. When multiple actions need to be applied is the Apply-Actions instructions must be used. The following actions are supported by this version of OpenFlow:

- copy TTL inwards, copy TTL outwards, pop, push, decrement TTL, set, qos, group, output

The output action is executed last. If no output action and no group action are specified in an action set, the packet will be dropped. The execution of a group is recursive.

Action list

An action list executes immediately a list of actions in the order specified by the list. The effect of this sequence of actions is cumulative. After the execution of the action list with the Apply-Action instruction the packet is processed in the following pipeline instruction.

Actions

It exists two types of actions. Required actions and optional actions, where the optional ones are not required to be supported by the switches as standard. The
required actions to be supported are the *Output*, *Drop* and *Group* actions. Some of the optional Actions are the *Set-queue*, *Push-Tag* and *Pop-Tag* actions.

OpenFlow standards supports packet forwarding to physical ports and switch-defined virtual port. The OpenFlow switch must also support the following reserved virtual ports as standard:

- **All**: Broadcasts packets to all ports but the ingress port or the ports configured as OFPC_NO_FWD.

- **CONTROLLER**: Encapsulates and send the packet to the controller.

- **TABLE**: Send the packet in the first flow table of the pipeline process.

- **IN_PORT**: Send the packet through the ingress port.

The optional ports to be supported by an OpenFlow switch are the following:

- **LOCAL**: Sends the packet to the default layer2 networking stack.

- **FLOOD**: Flood the packet through the default layer2 mechanisms.

![OpenFlow matching process](image)

**Figure 2.4.** OpenFlow matching process

### 2.1.2 OpenFlow Secure Channel

The OpenFlow protocol is used for the communication between the controller and the switches in a SDN. It supports three type of messages: controller-to-switch, asynchronous and symmetric messages.
Controller to switch messages

The controller to switch message are sent by the controller to inspect the state of the switches. The different type of controller switch messages are the following:

**Features**:
This message is sent to request the capabilities of the switch, usually on OpenFlow channel establishment. The switch in return will respond with a features reply message specifying the capabilities of the switch.

**Configuration**:
This message is sent to change the configuration parameters of the OpenFlow switches.

**Modify-State**:
The Modify-State message can add/delete and modify flows/groups in the OpenFlow tables and set the port properties.

**Read-State**:
The Read-State message is used by the controller to collect statistics from the switches.

**Packet-out**:
The Packet-out message is used by the controller to forward packets received by Packet-in messages as well as send packets through a specific port on a switch.

**Barrier**:
They are two types of Barrier messages: request/reply Barrier messages which are used to ensure message dependencies or receive notifications for complete operations.

**Asynchronous messages**
Asynchronous messages are sent by a switch in order to notify the controller of packet arrivals, switch states changes or errors.

**Packet-in**:
Packet-in messages can be sent by the switch to the controller when a packet is received that cannot be matched to an existing flow entry. It contains either part or the whole of the unmatched packet.

**Flow-Removed**:
This message is sent to controller when a flow is removed from the switches when its timer expires.

**Port-status**:
This message is sent by a switch when port configuration state changes.
Error:
With this message OpenFlow switches are able to notify the controller of errors.

Symmetric messages
Symmetric messages are exchange between controller and switches and can be started by either one of them.

Hello:
This message is exchanged upon the connection startup.

Echo:
This message are used to measure the latency or bandwidth of a controller-switch connection as well as verify its lifeness.

Experimenter:
This message provides a standard way for OpenFlow switches to add additional functionalities.

OpenFlow connection setup
The controller-switch connection must be to a user’s configured IP address, using user-specified port. When the connection first is established both the switch and controller send OFTP_HELLO messages with the highest version of the OpenFlow protocol supported by the sender. Following the reception of the message the lowest version of OpenFlow protocol is selected. If that version of OpenFlow protocol is supported by both, the connection is established. Otherwise, an OFTP_ERROR message will be sent back to the sender.

2.1.3 Encryption
The default TCP port of the controller is 6633. The switch and controller can communicate through an encrypted channel using TLS encryption. The switch and controller can mutually authenticate signed certificates.

2.1.4 Message Handling
The OpenFlow protocol provides reliable message transmission and processing.

Message delivery:
The messages are guaranteed to be delivered, unless the connection fails.

Message processing:
Messages that arrive to the switches from the controller must be processed. If the switch cannot completely process a message should send an error message back to the controller. Furthermore the switches must send asynchronous
messages generated by the change of state of the switch to the controller. Nonetheless, the switch could drop some packages received on the data ports due to congestion or QoS policies. This last mentioned should trigger the creation of no packet-in messages. The controller is also free to drop packets, but must respond to the hello and echo messages of the switches to prevent the connection dropping.

**Message ordering:**

Barrier messages as stated before can ensure ordering of messages. Switches may arbitrarily reorder messages to maximize performance in absence of barrier messages. When barrier messages are available, messages should be processed before the barrier. Consequently, a barrier reply must be sent and the barrier processed. Finally, the messages after the barrier could be processed.

## 2.2 NOX controller

In OpenFlow networks the signalling of traffic is separated from the data plane packet forwarding. This is done by a centralized controller, which is a centralized programmed model for an entire network. NOX is an open source implementation of a controller or network operating system (NOS)\[13\] for OpenFlow networks. NOX will run on commodity hardware and provide a software environment that allows programs to control networks through the OpenFlow protocol. This particular implementation of an OpenFlow controller has as its goals to provide an innovation platform with real hardware and usable network software for operators. Some of the principal features the NOX controller provides are:

- network management
- control module applications
- sophisticated network functionality on cheap switches
- abstract view of the network resources such as network topology and location of all hosts

As mentioned before, the NOX controller provides a software environment on top of which programs can control the OpenFlow network. These programs can be hooked to events generated by the NOX core or network, enabling the management of the traffic. Moreover, the NOX controller can scale by operating the network flows in a distributed manner. For example, running in a reactive mode the first packet of each flow would be sent to the NOX controller which will pass it to the application(s) registered to receive packet-in events. Then the application(s) can determine whether to:

- forward the flow
• collect statistics
• modify packets in the flow
• view more packets within the same flow

Some of these features mentioned above are provided by built in apps that have been developed for NOX. These built in applications are found in the src/nox/-coreapps/ or src/nox/netapps/ folders. They can be used to discover the network topology, track hosts as the move around the network, provide fine-grained network access controls, and manage the network history.

2.2.1 Developing in NOX

NOX has an event driven programming model used when developing extensions. NOX consists of 4 modules: NOX Events, Components, API, and Utilities modules. Events created by the NOX event handling system inherit from a main Event parent class. They represent high-level and low-level events in the network.

The Event class gives a convenient way to hook into any kind of events. The processing of the events then is deferred to the handlers in registered applications. There are various predefined events to which applications can register, some of these are:

• Bootstrap_complete_event
• Datapath_join_event
• Datapath_leave_event
• Openflow_msg_event
• Packet_in_event
• Flow_in_event
• JSONMessage_event
• Link_event
• Ofp_msg_event
• Shutdown_event

The components modules encapsulate specific functionalities made available to NOX. Any type of behaviour wanted on the network has to be specified as a component module for the NOX controller. In version 1.1 used in this project the Discovery components was used:
In order to test different capabilities on the NOX controller, it can be used an applications such as dpctl, that uses the NOX API to send commands to the controller as a group of strings, as follows:

```
./dpctl tcp:<switch-host>:<switch-port> stats-flow table=0
```

**Code 2.1. dpctl command**
Chapter 3

Virtual Aggregation methodology

Many efforts have been made to tackle the routing scalability problem, due to the rapid routing table growth [2]. Larger routing tables create the need for more memory space to store the forwarding information base (FIB) on routers. Typically FIBs are stored on low volume, off-chip SRAMs that could easily have power or heat dissipation issues when scaling. Moreover, off-chip SRAMs do not track Moore’s Law, thus the cost of FIB memory upgrades represent a large cost when upgrading routers.

As a consequence of the increasing cost of FIB storage memory, networks will become less cost effective since the prices per forwarded byte will increase. Virtual Aggregation (ViAggre) is a configuration-only methodology (i.e. you only need to configure your network differently) to reduce the size of FIBs on routers. ViAggre allows each router to store only a fraction of the global FIB by using virtual prefixes of aggregated routes. These virtual prefixes or virtual networks can be of different subnet sizes as long as together they cover the complete address space.

![Figure 3.1. ViAggre basic concept][2]

Nevertheless, the operation and maintenance process to keep the configuration up to date, in Figure 3.1 an example of a basic concept of ViAggre in a network is shown. The address space is divided into 4 pieces using a virtual subnet prefix of
/'2'. The virtual prefixes are then assigned to different routers (which we will refer to as "aggregation routers") which will store the part of the RIB belonging to the assigned virtual prefix.

### 3.1 Control Plane Deployment methods

In order to distribute the routes into the aggregation routers two deployment methods are proposed [2]:

#### 3.1.1 Deployment Method I

The design goal of this method is to have no changes on the router software or routing protocols by using a feature in BGP/OSPF called "FIB suppression". FIB suppression allows us to fully populate the RIB on a router and then selectively install only a part of the full RIB into the FIB of the router. In this case we can reduce the size of the FIB per router but still need to maintain a full RIB on each router.

![Figure 3.2. Deployment method I [2]](image)

#### 3.1.2 Deployment Method II

This design goal is to be transparent to external networks and have no impact on the interaction with neighbor networks. In this case we maintain the full RIB on devices that are off the data path. In this design we rely on peering of external routers to a route reflectors used most commonly on pop networks on TIER 1 and TIER 2 ISPs. The route reflectors selectively distribute the prefixes accordingly to the routes being advertised on the routers and advertise the routes to other route reflectors. In this case the route reflectors only reflect parts of the RIB to individual routers meaning that we reduce both the size of the RIB and the FIB stored by each route.
### 3.2. FORWARDING PACKETS

The main issue with the ViAggre concept is the forwarding of packets. When a packet arrives at a router two things are possible, either the destination of the packet is within the virtual prefix handled by the router or not. In the case that the packet does not belong to any of the virtual prefix handled by the router it will be forwarded to the nearest aggregation router that do aggregate that prefix. In another words, each router will contain routes to all the prefixes aggregated and one entry for every virtual prefix advertised by other aggregation routers. Once a packet arrives to the aggregation router, the router could search its FIB to find the route for the packet.

Nevertheless the router will not be able to forward the packet to its destination in a hop by hop fashion, as intermediate routers could be aggregating other prefixes and will not be able to find a route for the packet, possibly resulting in a routing loop. Hence, the packet is sent from the aggregation point to its destination using MPLS tunnels as show in 3.4.
Part II

Design and Implementation
Chapter 4

Design

In this chapter we explain the requirements and architecture design of the ViAggre module for a NOX controller.

4.1 Requirements

The goal of this project was to design and implement a ViAggre module for use in OpenFlow networks. As the ViAggre configuration methodology was intended for IP networks with BGP routing we had to tweak the original design of the ViAggre methodology to better suit the SDN design.

Unlike the original ViAggre methodology were the partitioning of routes is done by equally dividing the address space regardless the amount of routes in each partition, we intend to have a more balanced spread of rules through the OpenFlow switches, giving a more or less even memory space usage on each switch. And finally the module should be able to translate all of the partitioned RIB into flow entries and install them correctly on to the switches.

4.2 Architecture

The overall architecture of an OpenFlow network consist of several OpenFlow switches connected to an OpenFlow controller, NOX in our case, through a secure channel.

In the original design of ViAggre explained in [2], two implementation methods are suggested. The first one using the FIB suppression feature of routers and the second one, which is more invasive that uses route reflectors to spread the different route partitions to the correspondent routers.

In order to implement this splitting methodology and receive all the benefits of the partitioning of routes we take advantage of the architecture and capabilities of the NOX controller. The basic forwarding scheme of packets will be kept as the original design, as it can be easily translated into rules to be applied on the OpenFlow switches. Nevertheless the method by which the routes/rules are spread
throughout the network had to be changed due to the basic difference of architecture between traditional networks and SDN networks.

The first implementation method for the original ViAggre methodology would not be feasible, as the aggregation routes advertisement is all done at the same level (the router). However, the most apt scheme to adopt is the second methodology were the split routes are installed by the route reflectors, which normally are off the forwarding path.

The second methodology is very similar to the scheme that could be implemented on OpenFlow networks. Nevertheless the use of this scheme connot the employment of several route reflectors depending of the infrastructure of the network, whereas in OpenFlow networks, the routes will have to be distributed by the NOX controller.

Since we are applying ViAggre on an SDN network with a centralized controller, we have no need to distribute the RIB or parts of the RIB to our network nodes. Instead, we can collect the full RIB in the controller, partition it, and install only the relevant FIB into the OpenFlow switches. When installing the FIB we keep the packet forwarding scheme from the original design as it can be easily translated into rules to be installed on the OpenFlow switches. Nevertheless the method by which the routing flow entries (i.e. the FIB) are distributed throughout the network had to be changed due to the basic difference of architecture between traditional networks and SDN networks. This method allows us to avoid all the tedious configuration work by simply letting the application take care of it. Furthermore, control policies implemented in a centralized way, will be much easier to understand and deploy than a distributed protocol[17].

The NOX controller has a global view of the underlying network, which enables a centralized ViAggre module to efficiently distribute routes, instead of using several route reflectors as in the original methodology. This greatly reduces the complexity of deploying ViAggre forwarding since it can be automated.

![ViAggre module architecture](image)

**Figure 4.1. ViAggre module architecture**

To implement ViAggre forwarding we created an application for the NOX con-
4.2. ARCHITECTURE

troller which performs four main tasks:

- Obtaining the FIB
- Saving the FIB
- Optimized division of rules from the data structure
- Installation of rules to OpenFlow switches

Figure 4.2. Viaggre module internal architecture

The module as mentioned before, will have the capability to interacting with neighboring networks, save the FIB into a data structure to consequently split the rules using a splitting algorithm. The aggregation routes and partition routes will be selected and translated into rules. Finally, according to the ViAggre methodology it will install the aggregation rules and partition rules into the corresponding switches.
Chapter 5

User space - kernel space communication

Most modern operating systems divide the memory of a device into user space and kernel space. The kernel space is restricted for running the kernel, and device drivers. In the user space all the users’ applications are run. The user-space applications are unable to interact with the kernel-space memory since they run in different memory spaces on the machine. This chapter presents various methods available in the Linux kernel for communication between these spaces. This will be useful in next chapters to comprehend the technology choice to communicate with the kernel in the ViAggre module design. Here we present a brief overview of these options:

5.1 RAM based file systems

The Linux kernel provides RAM based file systems that represent a single value or a set of values. These values can be accessed by a user space program by standard read and write Linux functions. These RAM based files created in the RAM of the machine result in callback functions in the Linux kernel.

5.1.1 Procs

The procs interface is located in the proc folder. It was designed to transport all kind of process information. It can provide information about running processes, system interruptions, version of the kernel, etc.. Information about the IDE, SCI devices and TTY’s can also be obtained through this interface. Furthermore, the procs interface also provides information about the network, such as ARP tables, socket used or even network statics. The procs interface has been widely used, however, is deprecated and should not be used in new implementations [18].
5.1.2 Sysfs

The sysfs interface was designed to represent a whole device model. It provides information about devices, drivers and buses. Interconnections hierarchy are represented by the sysfs structured directory [19]. Some of the most important top level directories in kernel 2.6.23 are the following:

- `sys/block/` Block devices such as ram and sda are described here.
- `sys/bus/` All buses used are described here with two subdirectories defining the device and the driver used.
- `sys/class/` Each device type is stored in its own file.
- `sys/device/` The devices organized by bus used.
- `sys/firmware/` Here are described the firmware for some hardware.
- `sys/kernel/` This file system have the directories for other filesystems.

5.1.3 Configfs

Configfs is somewhat similar to the sysfs, with the big difference that is based on kernel objects. These objects are created from the userspace with the call of `mkdir` command. The kernel in exchange creates the files that can be read or written by the user. In other words instead of having a well pre-created file structure, the files are created dynamically through the `mkdir` script and deleted with the `rmdir` command.

5.1.4 Debugfs

Debugfs is an interface specially developed for debugging purposes. It provides the possibility to set or get values with the help of only one line of code.

5.1.5 Sysctl

The Sysctl interface is designed to configure kernel parameters at run time. It is used mostly by Linux networking subsystems. The values can be obtained using the cat, echo or `sysctl` commands. The commands have no persistency unless the values are written directly to the `/etc/systl.conf` file [20].

5.1.6 Character devices

The Character devices interface is used for character device drivers. This is done to give a way for users with privileges to write directly to devices in the kernel.
5.2 Sockets

The Linux kernel also leaves the possibility of accessing it through socket interfaces. In contrast with file systems, it provides a way for the kernel to send notifications and not wait for the user space program choses to access it.

5.2.1 UDP sockets

A user program can send strings to the kernel through a UDP socket and binded to port 555 which is a specific port for these matters.

5.2.2 NETLINK sockets

NETLINK is a special Inter-Process Communication interface for transferring network related information between the user-space and the kernel. The communication between parts is full-duplex and makes use of the standard socket API for user-space processes and a special kernel API for kernel modules. NETLINK sockets provide a special implementation of the socket called "Generic NETLINK Family", which acts like a NETLINK multiplexer. Generic NETLINK communications are a series of specific communication channels which are multiplexed into a single NETLINK family. These channels are identified by channel numbers and dynamically allocated [21].

5.3 Kernel System calls

The Linux kernel provides several system calls that user-space programs can call in order to retrieve data and services from the kernel. When a user-space program invokes a system call the CPU switches to kernel mode and executes the appropriate kernel function.
Chapter 6

Implementation

This chapter will discuss the design implementation of each of the parts of the ViAggre module.

6.1 Obtaining the forwarding information base

As stated on the requirements, the module should have the capabilities to interact with external nodes. In traditional networks a node will be able to be part of a network when the same protocol is implemented on the node.

In order for an OpenFlow Network to interact with traditional protocols, it will have to be implemented a routing daemon at the controllers level, in [22]. The actual implementation of such module/daemon and the management of the protocols' signaling is out of the scope of this work. Nevertheless, because the ViAggre methodology is based on the FIB of a virtual node, obtaining the FIB generated from the routing daemon is an essential part of the project.

In a Linux environment, most of the routing daemons work the same way, but to narrow down our options we took as reference one of the most common routing daemons, 'Quagga' [23]. To keep track of changes on the FIB caused by network changes, two options were presented. The first method is to connect directly to Quagga using the Zebra protocol and the second method was using the kernel interface.

Communicating with the Zebra protocol would allow us to directly query the Quagga daemon about installed routes on the FIB. Nevertheless, this option will limit the deployment of the ViAggre module to the Zebra daemon. In counterpart, obtaining the FIB from the kernel interface would leave an open option of which routing daemon to be used as shown in 6.1.

In order to implement the communication to the kernel interface we had various options as explained in a previous section. Two options were available: RAM based file systems and NETLINK sockets. Due to the event driven nature of the NOX controller, RAM based file systems would be cumbersome to use. Using a RAM based file system would require us to implement a recursive search and reading
of the files, to spot changes on the FIB. This would decrease the reliability and potentially impose large delays on the convergence of the network. Therefore we decided to use NETLINK as it gives us more control over what type of information we desire to gather from the kernel, more specifically the forwarding table.

The NOX controller does not have support for NETLINK sockets event natively. We created an event within its core and create a socket within its main loop in order to listen to messages about changes in the forwarding table as shown in 6.2.

In order to create a NETLINK socket on Linux, one simply calls 'socket()' system call with some arguments. In order to specify that the NETLINK type of socket should be created we use 'AF_NETLINK' socket family type, the protocol type can be either a raw socket (SOCK_RAW) or datagram (SOCK_DGRAM). The protocol on the Linux socket implementation is the exact NETLINK capabilities we would like to use. These capabilities are an extended list but in our case is 'NETLINK_ROUTE'. The system call to open a NETLINK socket looks like this:

```c
fd = socket(AF_NETLINK, SOCK_RAW, NETLINK_ROUTE);
```

### Code 6.1. Socket function prototype
6.1. OBTAINING THE FORWARDING INFORMATION BASE

To bind the caller must provide a local address using the sockaddr_nl structure containing the current user process id when openning the socket, the socket family and the multicast group if any.

```c
struct sockaddr_nl {
    sa_family_t nl_family;
    unsigned short nl_pad;
    __u32 nl_pid;
    __u32 nl_groups;
};
```

**Code 6.2.** Sockaddr_nl structure

So to succesfully open a NETLINK socket and wait for updates on the forwarding table we did it in the following way:

```c
fd = socket(AF_NETLINK, SOCK_RAW, NETLINK_ROUTE);
bzero(&la, sizeof(la));
l.a.nl_family = AF_NETLINK;
l.a.nl_pid = getpid();
l.a.nl_groups = RTMGRP_IPV4_ROUTE | RTMGRP_NOTIFY;
bind(fd, (struct sockaddr *)&la, sizeof(la));
```

**Code 6.3.** Open NETLINK socket

This call will open a socket, but no information is recieved on the start up. RTNETLINK sockets, which are a special class of NETLINK sockets employs a request-response mechanism to send or receive information from the networking environment. A NETLINK socket request is composed by a stream of messages structures. Thus we had to send a message to the kernel in order to receive the startup configuration of the forwarding table. This is done by the standard Linux implementation of the "sendmsg()" function. like the following:

```c
ssize_t sendmsg(int fd, const struct msghdr *msg, int flags);
```

**Code 6.4.** Sendmsg() function and NETLINK

Msg is a pointer to a structure of type msghdr which has a pointer to a struct of type sockaddr_nl, which is the destination address of the sendmsg() function, in our case the kernel. The msg_iovlen is a pointer to a struct of type iovec that will contain all the NETLINK struct messages in order to request or manipulate information of the network environment in the kernel.

```c
struct msghdr
{
```
A NETLINK message will always be composed by the NETLINK header, which will provide information about the type of RTNETLINK type of message through the nlmsghdr structure.

```
struct nlmsghdr
{
    __u32 nlmsg_len;
    __u16 nlmsg_type;
    __u16 nlmsg_flags;
    __u32 nlmsg_seq;
    __u32 nlmsg_pid;
};
```

Code 6.6. NETLINK message header

The type of RTNETLINK message we have to send in order to obtain the routing information from the kernel is defined by a set of Macro expansions on linux/rt-netlink.h file, and in our case we had to defined to a RTM_GETROUTE type of message. Following the NETLINK message header is the RTNETLINK header message. There are different type of structures that could be added depending on the operation being requested. But, as we are requesting changes on the forwarding table we had to include the struct rtmsg which allowed us to retrieve and modify entries on the routing tables.

```
struct rtmsg {
    unsigned char rtm_family;
    unsigned char rtm_dst_len;
    unsigned char rtm_src_len;
    unsigned char rtm_tos;
    unsigned char rtm_table; /* Routing table id */
    unsigned char rtm_protocol;
    unsigned char rtm_scope;
    unsigned char rtm_type;
    unsigned rtm_flags;
};
```

Code 6.7. Rmsgs structure
The most critical part of the initialisation of the rtmsg structure is the rtm_table or reserved table identifier, this definitions can be found in the linux/rtnetlink.h file. As we were interested solely on the main forwarding table we have initiated the structure variable with RT_TABLE_MAIN. Immediately after the RTNETLINK operation header are the attributes related to the operation defined on the rattr structure. Once a the message to retrieve the forwarding table is sent, the kernel will respond with another stream of structures. In order to parse this structures we used predefined MACRO expansions to make the buffer positioning easier. some of the ones used are :

- NLMSG_NEXT(nlh, len) returns a pointer to the NETLINK header
- NLMSG_DATA(nlh) returns a pointer to RTNETLINK header
- RTM_RTA(r) returns a pointer to the start of attributes of RTNETLINK operation header

Once the kernel’s message containing the current forwarding table has been parsed it has to be saved into a data structure in order for us to manipulate it. This is explained in detail in the next section.

6.2 Saving the Forwarding Information Database

In the networking domain exists several types of algorithms to classify routing information and flow rules; Each having different performances. Some of the most relevant metrics to measure the performance of classification algorithms are the search speed, storage requirements and ability to handle large real-life classifiers. Furthermore, one of the most critical part of any device is the correct storage of information. The ability to update the data structures where information is saved, as classifiers change, depends mostly on the type of data structure been used. Data structures can be categorized into those which have to be reconstructed from scratch and those who can add or delete entries incrementally.

The need for fast update rates depends on the application. For instance applications such as firewalls, were entries are not regularly added, slow conversion or updates may be sufficient. However, applications such as routers with per-flow queues may require frequent and faster updates [24]. Due to the nature of the project we needed a fast convergent data structure capable to add or delete entries "on the fly", when the forwarding table changes.

Moreover, the implementation of the data structure should have the capability of sorting entries depending on the longest prefix. Nonetheless, implementing a proven data structure algorithm is a project on itself. Hence, we have utilized an implementation of a d-dimensional radix trie to keep the FIB. The implementation of the radix sort trie will allow us to have acceptable update speeds and the ability to delete and add entries without reconstructing the whole structure from scratch.
Even though we used a complete implementation of a radix trie, we still needed a way to keep track of how many routes existed per branch and sub-branches. This need of having information about the amount of routes on each of the branches is due to the heuristics proposed on the requirements section, where we will try to have an even spread of routes over the connected OpenFlow switches. An example of a piece of a radix trie and its nodes information when implemented without changes is shown in 6.3.

We had two approaches to add the number of routes information on each node. The first and simplest was to first create the trie and then walk through the whole trie counting and updating the nodes information. This method will have to be recursively done each time a new node had been inserted to the trie or any search had to be made. The second approach was to update the information of the node while inserting each node. Once inserted a node we will check for the total amount of routes the child node has and update the information on the node. Then, recursively will only update the values of the parent nodes. This method prove to be more effective and less computational demanding, considering that it will only update the direct parents of the node, and will not recursively walk the whole trie each time a node is inserted.

```c
Update()
    if right branch exists then
        get number of routes in branch;
        right routes += number of routes in branch;
    if left branch exists then
        get number of routes in branch;
```
left routes += number of routes in branch;
node routes = left routes + right routes;
if node != root then
    parent node Update();
}

Code 6.8. First number of routes update algorithm

Once the above mentioned algorithm was implemented we had a trie, where as explained before, each node will contain the total number of routes of their children, enabling us to have a global view of the trie from the top nodes. An example of a possible trie obtained after to the modifications of the code will be the following:

Figure 6.4. Updated radix trie implementation

6.3 Implementing the Virtual Aggregation Methodology

The balanced spread of rules through out the network suggested as initial requirement, implies the possible multiplication of aggregation rules on switches. In software based OpenFlow switches, augmenting the number of aggregation rules on each switch, increments the linear search for rules, decreasing performance. Thus we pro-
posed algorithms to have as equal as possible number of rules while maintaining the number of aggregation rules minimal.

![Figure 6.5. ViAggre balanced spread of rules][5]

We first proposed the simplest algorithm:

```c
node = root of tree;
number of routes per partition = total number of routes / switches;
while (counter <= switches) {
    while (n <= number of routes per partition) {
        search tree for next node with route;
        translate node to flow;
        install rule to switch [counter];
        n++;
    }
    counter++
}
```

**Code 6.9. 1st Split Algorithm**

This algorithm will calculate the total number of routes per switch needed. Then, searches the tree from top to bottom sequentially on each branch for a node with a valid route and translate the route into a flow rule. Finally installs the rule to a switch, maintaining a counter of all the installed routes. Each time the counter is bigger than the number of routes per switch needed, the algorithm will continue installing routes into the next switch.

After analyzing this algorithm we can easily see that its complexity is rather large due to the nested loops used for control, affecting greatly the performance of the module. This algorithm treats the radix tree as a large array, not using the
6.3. IMPLEMENTING THE VIRTUAL AGGREGATION METHODOLOGY

extra information of total amount of routes on branches of each node. Furthermore, it runs without having reference of the possible aggregation routes of each partition making more difficult the deployment of the ViAggre methodology.

The second algorithm was implemented and as the previous algorithm the number of switches is taken as reference to split the number of routes to be installed. In the first algorithm, routes would have been searched in order, from top to bottom, starting from the first left branch until the last right branch of the whole tree. This method, as explained before, would not take into account the aggregation rule(s) to be installed on each switch after installing the simple rules.

In the second algorithm, rather than partitioning in a sequential way, it recursively partitions the branches belonging to the aggregation node with the largest number of child routes. In other words, the second algorithm was designed to: First, search the branch with the largest amount of routes. Second, will subdivide this branch to obtain two 'new sub tries', where the top node unifying the two 'new sub tries' will be the aggregation node that will contain the aggregation route to be installed in the other switches. Third, the process starts over again with the two 'new sub tries', where the branch with the largest number of routes will be divided. This process will continue until it satisfies the number of switches, thus partitions, or the number of routes/rules are balanced throughout the partitions.

We will try to explain the algorithm through an example rather than with a pseudocode based on 6.6.

In this algorithm as stated before, The "root" node of the tree or sub-tree is divided recursively into branches. For example in 6.6, if we had 5 switches on the network, the algorithm will do as follows:

It will split the trie in two, filling in the first two places of a 'vector of 5 lists' which represent the 5 switches, with nodes 5 and 2, generating two new sub tries.

![Radix tree example](image)
Once the first split is done, the list/sub trie on the vector with the largest amount of routes is searched. Then, the node will become the root node and is divided. In our case, the sub trie with 5 routes is divided, obtaining two sub tries, one with two routes and the other with three routes. The new sub trie with the largest amount of routes is left in the same position in the vector of list, the second sub trie with the smallest amount of routes is pushed into the back of the vector of lists as seen as follows:

```
5 2 1
```

The same procedure will be followed until we obtain the following vector of lists:

```
3 2 2
```

With this algorithm we achieve a partitioning of routes, without having to walk through the whole tree as the first algorithm suggested. Additionally, Each position on the vector of list will be the equivalent of the routes to be installed into a specific switch.

Additionally, we propose an optimization to this algorithm. This recursive splitting could be done even though the vector of lists has been correctly filled and no more partitions need to be made. We propose to continue this division until a ratio between the list with the smallest number of routes and the list with the largest number of routes is achieved to obtain a more evenly partition of routes.

Furthermore, each node inserted into the list inside the vector is the pointer(s) to the aggregation node of a subtree into which the original tree has been divided, and gives us directly the aggregation route(s) to be spread to other switches.

For example, we could set the proportion of $1/2$. This would mean that the list with the maximum number of routes will have to be divided until it is at the most two times bigger than the list with the minimum number of routes. The recursive process of splitting will continue the same, with the only difference in where the new sub tries will be placed. When divided several times optimizing the partitions, the new nodes are to be inserted into the list with the least number of routes.

This optimization will have its trade-offs. Once the vector of lists has been filled in completely the first time, any further partitioning will provoke the creation of extra aggregation routes. This optimization will have as result a better distribution of the number of routes among the switches, but at the same time will consume more memory space by adding more aggregation routes in the flow tables to maintain a more evenly distributed number of flow entries on each switch.

### 6.4 Installing rules to the switches

Once the routes have been partitioned, it only remains to install the rules, aggregation rules and tunnels to the corresponding switches to have a functioning network.
As explained before, we will use the same forwarding scheme as the original ViAggre method explained in Chapter 4. Furthermore, as explained in a previous chapter, each list of the resultant vector of lists, will correspond to a partition of the routes to be installed in a switch and each node of every list, will be the aggregation node of a sub trie of routes.

In other words, one list in the vector of list will contain all the routes of one switch. Each node of a list in the vector of lists, will correspond to a sub trie of routes, and each 'root node' of each sub trie will correspond to an aggregation route, which will be installed in all the other switches minus the correspondant switch where its sub tries were installed. An example of the correspondence of the vector to switches is in the following figure:

![Figure 6.7. Vector of list installation of routes, based on [5]](image)

To send command messages and install rules to OpenFlow switches, we rely on the function send_openflow_msg() given by the NOX implementation we used. As the aggregation routes, are rules too it was used the same function. This function’s prototype is the following:

```c
send_openflow_msg ( const datapathid &dpid,
struct::ofl_msg_header *msg,
uint32_t xid,
bool block)
```

This function has several arguments; The first one is the dpid which is a reference to the value of the datapath id obtained from the switch to which the command is to be sent. The second argument is where is sent the actual command. It is a pointer to a structure of type 'ofl_msg_header'. The 'Ofl_msg_header' is just the header of all type of message structures to be used.

As we intend to install a route, the 'ofl_msg_flow_mod' was used and cast back to an "ofl_msg_header". This 'ofl_msg_flow_mod' structure will in return have a pointer to a structure of type struct "ofl_match_header" is used for the matching rules. Furthermore, the "ofl_msg_flow_mod" is also used to state the priority of the rule. As we are working with IP routes, it could be possible that two equal
rules with different prefixes be in the same switch. In this case, the priority of the rule is important to deduce, as the rules have to behave as a longest prefix match. We set the rule’s priority to the length of the network mask plus ten, to ensure this behavior. Using the 'ofl_match_header' we then describe the actual IP routes as a rule and create the MPLS rules (tunnels).

```c
struct ofl_match_header {
  OFPMT_STANDARD, /* One of OFPMT_* */
};
struct ofl_match_standard {
  struct ofl_match_header header;
  uint32_t in_port; /* Input switch port. */
  uint32_t wildcards; /* Wildcard fields. */
  uint8_t dl_src[OFP_ETH_ALEN]; /* Ethernet source address. */
  uint8_t dl_src_mask[OFP_ETH_ALEN]; /* Ethernet source address mask. */
  uint8_t dl_dst[OFP_ETH_ALEN]; /* Ethernet destination address. */
  uint8_t dl_dst_mask[OFP_ETH_ALEN]; /* Ethernet dest. address mask. */
  uint16_t dl_vlan; /* Input VLAN id. */
  uint16_t dl_vlan_pcp; /* Input VLAN priority. */
  uint16_t dl_type; /* Ethernet frame type. */
  uint8_t nw_tos; /* IP ToS (actually DSCP field, 6 bits). */
  uint8_t nw_proto; /* IP protocol or lower 8 bits of ARP opcode. */
  uint32_t nw_src; /* IP source address. */
  uint32_t nw_src_mask; /* IP source address mask. */
  uint32_t nw_dst; /* IP destination address. */
  uint32_t nw_dst_mask; /* IP destination address mask. */
  uint16_t tp_src; /* TCP/UDP/SCTP source port, or ICMP type. */
  uint16_t tp_dst; /* TCP/UDP/SCTP destination port, or ICMP code. */
  uint32_t MPLS_label; /* MPLS label. */
  uint8_t MPLS_tc; /* MPLS TC. */
  uint64_t metadata; /* Metadata passed between tables. */
  uint64_t metadata_mask; /* Mask for metadata. */
};
```

Code 6.11. ofl_match_standard & ofl_match_header structures

Due to the limitations of hop by hop forwarding when using the ViAgg methodology: The usage of tunneling from the aggregation switch to its destination is necessary. MPLS provides two ways to do an Label Switched Path (LSP) on traditional networks, independently or ordered [25]. We chose the independent approach. As mentioned before, SDNs and the NOX implementation more specifically, allows us to have a global description of the network, thus we applied a shortest path algorithm to determine the LSPs.

Unlike traditional networks, the label distribution of the LSPs is not done by any of the traditional protocols, but rather by the controller and the OpenFlow protocol in the form of rules. To actually test and better comprehend the dynamics of the
network the ViAggre module we used a test topology that we created through a virtual implementations of OpenFlow 1.1 capable switches (6.8). This topology was implemented using a small Python script for the mininet application that allowed us to recreate the topology each time without configuring the virtual interfaces and connections between the switches each time we had to do a new test.

When analyzing the topology of possible cases to create tunnels from the aggregation node to the host networks we realized that there were three possible options to create them. In 6.8 we could imagine that each of the hosts are actually in different external networks. (1a, 1b, 3a, 3b, etc.).

![Figure 6.8. OpenFlow test topology](image)

Considered the following example on 6.8:

If a host in network 1a would like to send a package to a host in network 4a, and the aggregation switch for the routes in network 4a is switch 6. The packets will be pushed to switch 6 and will give us three options to create the tunnels:

The first option would be to create tunnels per port. This means that we would have to create in this example two rules to each of the external ports of the switch to which the external networks are connected. Moreover this would mean that the number of MPLS rules will augment by the number of connected hosts.

The second option to create MPLS tunnels, is creating tunnels per node using multiple labeling. This would mean that the packet when arrives to a switch 1 will be pushed an MPLS label that will represent the port number of switch 4 where the network 4a is connected. Then the second label will be pushed, and it will represent the switch to which it will be attached, in our case switch 4. It is important to point out that, when pushing the multiple labels a critical part was to remember that the stack bit on the label should be zero for the outer label representing the switch and 1 for the inner label representing the port, in order to set the end of the stack [25].

When pushing the packet through the intermediate switches, the switches will match the label that represents the switch where the destination is attached to and pushed it forward until it arrives switch 4. Switch 4 will then match the rule to pop out the first label, and then the second rule representing the port will be popped out to finally deliver the packet to its destiny.
The final option to create the tunnels was to create them per node but installing specific port rules on each switch. In this case, the packets will be pushed only one MPLS label referring to the switch where network 4a is connected and will be popped out once it arrives to switch \(4\). Then switch \(4\) will try to match the IP packet to a local rule which will represent the port where network 4a is connected.

The three options gives a reliable way to create the tunnels, however, each implementation has its pros and cons. The first option, created one tunnel per port connected, we discarded immediately as the number of rules to be created would be equal to the number of switches to the power of the number of ports connected, therefore incrementing greatly the amount of rules to be created on each switch, compared to the other two options to create the tunnels.

The second and third options seemed quiet similar but their implementations have also its trade-offs. The second option would have more latency on the package forwarding due to the double labeling but will save more space in the flow tables, as it will only represent a rule per node. In the other hand, the third option gives lower latency but higher space consumption in the flow tables, because it will represent one rule per node to create the tunnel and one rule per port connected on the switch. As we aim to decrease the size of the flow table, we opted to implement the second option on the test environment.
Part III

Evaluation of the Implementation
To better understand the dynamics of the network and have a performance measurement of the code, two experiments were run. The first, measured the composition of the routes installed in the switches. The second was done to measure the performance and efficiency of the ViAggre implementation.

6.5 Composition of routes

The first experiments were run, by simply running the ViAggre implementation and obtaining the number of aggregation routes, normal routes and MPLS tunnels. It was used a range of 1 to 50000 randomly selected routes to perform the experiment. The routes were obtained from a real router and selected randomly for each experiment. We experimented creating random routes, but we decided to use real routes to have a more realistic approach to the experiments.

Each node on each list of the vector of list, as explained in earlier chapters, will represent an aggregation route. Each node will represent the root node of a subtrie, which will contain part of the total number of routes to be installed. In 6.9 We can see an example of the vector of lists without an optimization ratio and in 6.10 the same vector of lists after applying the optimization ratio.
After obtaining the data, we can observe that the number of installed aggregation routes depends on the optimization ratio. The bigger the optimization ratio is, the more aggregation routes we will obtain. This is due to the fact that aggregated routes are being divided and installed in different switches where they cannot be aggregated. In Figure 6.11 we can see clearly the gap between the total number of aggregation routes to be installed per switch on each case, where the largest optimization ratio creates the largest amount of aggregation routes.

Furthermore, as expected, when applying a bigger optimization ratio, the difference between the maximum number of routes of the list and the smaller number of routes of the lists is reduced (Figure 6.12). This translates into a more even distribution of routes throughout the switches. Nevertheless, as shown in Figure 6.11, the use of the optimization ratio has its downside: The number of installed aggregation routes increases with bigger optimization ratios.
6.6 Performance of the algorithm

Finally, the second test was done by timing the ViAggre splitting module. The module was timed when not using any optimization ratio, and when using a 0.5 and 0.8 optimization ratio. We expected a different behavior in 6.13 if the first algorithm explained in section 6.3 was implemented. The first algorithm proposed in section 6.3 searched the data structure as large linked-list that would have as result a linearly increasing time when the number of rules installed increase. However, we can observe that the plots are constant and not linearly increasing. In 6.13 we can clearly see that the processing time increased when applying bigger ratios. We could conclude that the behavior in 6.13 is not due to the number of routes, but mostly to the optimization ratio and the trie composition. Furthermore, in a worst case scenario if the trie is totally unbalanced, the trie will turn into a long linked-list and to properly split this will take a large amount of time.
Part IV

Conclusion and Future work
6.7 Conclusion

In this project, we proposed the design and implementation of a Virtual Aggregation module for NOX controllers. This module was designed taking into account the differences in architecture between Software Defined Networks and traditional networks for which the Virtual Aggregation methodology was developed. Moreover, the possible future applications in large scale networks were a key point for the design. In addition, support for NETLINK socket events was created for the NOX controller implementation, in order to sense changes on the kernel’s forwarding table. Furthermore, an algorithm was developed by which a semi-balanced division of routes is done so that it could be utilized for the spreading of the routes/rules to the OpenFlow switches. After performing tests on the implementation, we can conclude that the algorithm will perform well with a large amount of routes, as the processing time will not depend on the number of routes, but on the optimization ratio used. Using an optimization ratio has its trade-off; Using optimization ratios improves the distribution of routes throughout the network, therefore not overloading only one switch, but at the same time increasing the amount of aggregation routes to be installed. In this work, it has been demonstrated that the Virtual Aggregation methodology can be ported to SDN networks by changing how routes distribution was done on the original methodology, creating support for new events and developing an algorithm to divide the routes.

6.8 Future work

We suggest that further work should be focused on testing the implementation in different application scenarios. In [26], it is suggested the possible creation of high throughput nodes and high availability nodes by connecting multiple low-performance nodes working as one abstract one. Improvements on widening the spectrum of rules supported by the module would be interesting. While we only tested on IP routes, the same methodology can be applied on Ethernet networks with different sets of rules. To be able to apply this methodology with different types of rules, a new type of data structure would be necessary. Different implementations of data structures with better performance and flexibility when classifying flows are suggested in [24]. Furthermore, we suggest to improve the implementation of the ViAggre module to improve performance in the network as suggested in [16, 27].
Bibliography


