ENERGY-EFFICIENT CARRIER SDN NETWORKS

Pablo Fondo Ferreiro

Master’s Thesis presented to the
Telecommunications Engineering School
Master’s Degree in Telecommunications Engineering

Supervisors
Manuel Fernández Veiga
Miguel Rodríguez Pérez

2018
Entities should not be multiplied beyond necessity.
— William of Ockham

To my parents, for their unconditional love and support
Acknowledgements

I would like to express my gratitude to my supervisors, Prof. Manuel Fernández Veiga and Prof. Miguel Rodríguez Pérez, who in addition to guiding me throughout the development of this master’s thesis, have given me the opportunity to present the work at international conferences, as first steps in my research career.

This project was started under a departmental collaboration grant awarded by the Ministry of Education, Culture and Sports in the 2016-2017 academic year. This fellowship allowed me to gain background in the topics needed for the development of the thesis.

Vigo, June 20, 2018
Abstract

In the last years, the economic and environmental costs related to energy consumption led to efforts for enhancing the energy efficiency of networking equipment. Unfortunately, most of the proposals have not been implemented due to the limited flexibility of current networks. On the other hand, the Software-Defined Networking (SDN) paradigm introduces flexibility and programmability through the separation of control and data planes. In this thesis, we focus on minimizing the energy consumption in bundles of Energy Efficient Ethernet links leveraging SDN. We first analyze the problem from the point of view of SDN. Then, we design a complete application to minimize the energy consumption by periodically reallocating the traffic flows in the ports of the aggregate, dynamically adapting to variations in the traffic demand. We validate the proposed algorithms through simulations and also by implementing the application in a real SDN controller. The results show that our algorithms are capable of concentrating the traffic in few ports, saving up to a 50% of energy when there is a low traffic load. Nevertheless, the results show a trade-off between energy consumption and packet delay. Accordingly, we extend our solution to give support to traffic with low latency QoS requirements while keeping energy consumption minimized. Our results confirm that the latency of time-sensitive traffic can be reduced some orders of magnitude without increasing the energy consumption.

Key words: SDN, Energy Efficiency, IEEE 802.3az, ONOS, Low Latency QoS
Contents

Acknowledgements

Abstract

List of figures

1 Introduction
  1.1 Context
  1.2 Objectives
  1.3 State of the art
    1.3.1 Software-Defined Networking
    1.3.2 IEEE 802.3az
    1.3.3 Energy efficiency in SDNs
  1.4 Contributions

2 Energy-efficient algorithms for Ethernet link aggregates
  2.1 Optimum allocation
  2.2 SDN application design
    2.2.1 Flow definition
    2.2.2 Application workflow
    2.2.3 Flow rate estimation
  2.3 Allocation algorithms
    2.3.1 Greedy algorithm
    2.3.2 Bounded-Greedy algorithm
    2.3.3 Conservative algorithm
  2.4 Experimental results
    2.4.1 Metrics
    2.4.2 Description of the experiments
    2.4.3 Energy consumption: simulation results
    2.4.4 Packet loss: simulation results
    2.4.5 Average delay of the packets: simulation results
    2.4.6 Comparing the algorithms
    2.4.7 Implementation
  2.5 Conclusions
# Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3 QoS-aware algorithms</td>
<td>35</td>
</tr>
<tr>
<td>3.1 Problem statement</td>
<td>35</td>
</tr>
<tr>
<td>3.2 QoS-aware energy-efficient algorithms</td>
<td>35</td>
</tr>
<tr>
<td>3.2.1 Spare Port algorithm</td>
<td>36</td>
</tr>
<tr>
<td>3.2.2 Two Queues algorithm</td>
<td>37</td>
</tr>
<tr>
<td>3.3 Experimental results</td>
<td>39</td>
</tr>
<tr>
<td>3.3.1 QoS-aware energy-efficient algorithms: simulation results</td>
<td>39</td>
</tr>
<tr>
<td>3.3.2 ONOS implementation</td>
<td>43</td>
</tr>
<tr>
<td>3.4 Conclusions</td>
<td>45</td>
</tr>
<tr>
<td>4 Conclusions and future work</td>
<td>47</td>
</tr>
<tr>
<td>4.1 Contributions</td>
<td>47</td>
</tr>
<tr>
<td>4.2 Future work</td>
<td>48</td>
</tr>
</tbody>
</table>

Bibliography                                                             | 53   |
List of Figures

1.1 SDN architecture. Retrieved from [1]. ............................................ 3
1.2 Energy-Efficient Ethernet model. Retrieved from [2]. ...................... 4

2.1 Flow distributions for the first and last bits of the destination address. ... 12
2.2 Flow distribution of the destination address using different number of bits. 13
2.3 Flowchart with the logic of the application. ..................................... 15
2.4 Average error in the estimation of the flow rate for different periods. ... 17
2.5 Pseudocode for the Greedy Algorithms. ......................................... 19
2.6 Consumption of a 10 Gbit/s IEEE 802.3az port using frame transmission. 21
2.7 Pseudocode for the Conservative Algorithm. .................................. 22
2.8 Basic experimental topology. .......................................................... 23
2.9 Energy consumption variation with the duration of the sampling period. .. 25
2.10 Energy consumption for different traffic traces. ............................ 26
2.11 Packet loss percentage variation with the duration of the sampling period. 28
2.12 Packet loss percentage variation with the buffer size. ..................... 29
2.13 Packet loss for different traffic traces. .......................................... 29
2.14 Packet loss percentage variation with the duration of the sampling period. 30
2.15 Average packet delay for different traffic traces. ............................ 31

3.1 Example of the QoS-aware algorithms working in a 4-link aggregate. ... 38
3.2 Average delay of the low-latency packets for the 32.5 Gbit/s best-effort traffic trace. ................................................................. 40
3.3 Average delay of the low-latency packets for the 45.5 Gbit/s best-effort traffic trace. ................................................................. 41
3.4 Average delay of the normal packets for the 32.5 Gbit/s best-effort traffic trace. ................................................................. 42
3.5 Normalized energy consumption for the 32.5 Gbit/s best-effort traffic trace. 43
3.6 Experimental topology used for QoS-aware algorithms validation. ....... 44
3.7 Round-trip time for the different algorithms. ................................. 45
1 Introduction

1.1 Context

Nowadays, public concern about energy consumption of networking equipment is increasing due to not only environmental reasons but also economic ones. Inside datacenters the network consumes up to a 20% of its total power [3]. If we focus on mobile networks, the reduction in the energy consumption is also a Key Performance Indicator (KPI) according to The 5G Infrastructure Public Private Partnership (5G-PPP) [4].

As a result, a wide range of solutions have been proposed to reduce the energy consumption of networking equipment. However, most of these proposals remain unimplemented due to the lack of flexibility in current networks: some of these proposals require changes to basic protocols, others require changes to the networking equipment, etc. We can find several examples of these proposals in the literature, cf. [5–8].

On the other hand, the Software-Defined Networking (SDN) paradigm is gaining momentum in the networking community. The flexibility that it introduces led to its extensive adoption in datacenters. Besides, it is envisioned as a key enabler technology for 5G networks. The adoption of SDN in these networks is seen as an opportunity to improve the energy efficiency of the communications, surpassing the limitations of current networks thanks to the programmability and flexibility introduced.

1.2 Objectives

In this work, our goal is to address the problem of minimizing energy consumption when the traffic traverses a bundle of Energy-Efficient Ethernet links from a Software Defined Networking perspective. In detail, the specific objectives of this master’s thesis are the following:

- Get a deep understanding in Software-Defined Networking (SDN), both from a
conceptual perspective and also from a practical point of view.

- Understand how energy consumption works in aggregates of Energy-Efficient Ethernet (EEE) links between two switches.

- Design an SDN solution that minimizes the energy consumption in a bundle of EEE links among two switches.

- Since the energy optimization can lead to an increase in the delay of the packets, consider traffic with different QoS requirements in terms of latency.

- Create a simulator to validate the proper operation of the proposed solutions.

- Implement the proposed solutions as applications on top of the ONOS SDN controller.

1.3 State of the art

1.3.1 Software-Defined Networking

Software-Defined Networking (SDN) [1] is an emerging paradigm that proposes the decoupling of the data plane and the control plane in networking. This breaks the vertical integration of current networks enabling flexibility and programmability thanks to the separation of the control logic (i.e., the forwarding decisions) from the forwarding actions taken by the devices. Consequently, the devices become simple packet forwarding fabrics that do not need to execute complex protocols to determine how to forward the traffic. Instead, the decisions on how traffic should be forwarded are taken in a logically centralized entity named the SDN controller or network operating system (NOS). It is important to bear in mind that the SDN controller is not physically centralized, to allow for scalability and resilience as stated in [1]. Besides the separation of the control and data planes and centralization of the network control, SDN is also characterized by flow-based forwarding decisions and the ability to program the network through software applications. A flow is defined by a set of fields of the packets that match a criterion and a set of actions. This programmability enables a wide range of new applications allowing innovation in data networks.

A simplified SDN architecture is depicted in Fig. 1.1. The network infrastructure is composed by the forwarding devices (i.e., the switches), which represent the data plane. These devices communicate with the controller through the southbound interface. The SDN controller, which has a complete view of the network topology, executes the control plane. On top of the controller, the network applications are in charge of controlling the policies of the network and decide how traffic should be forwarded in the network. These applications use the northbound API to communicate with the controller, which is the one in charge of instructing the devices to create, modify or delete flow rules. Furthermore, the
controller also monitors the devices and provides the network applications with real-time information about the topology and the underlying devices. Therefore, both southbound and northbound interfaces are bidirectional, allowing secure communications in both directions.

OpenFlow [9] is the *de facto* southbound interface, used to allow the communication between the SDN controller and the forwarding devices. Although OpenFlow started as a research experiment in the Stanford University has gained popularity in industry, being nowadays supported by most vendors of commercial switches. OpenFlow-enabled switches are composed of flow tables, each table is composed of a set of flow rules and a flow rule consists in a match criterion, a set of actions and counters indicating the number of packets and bytes that have matched this rule. The OpenFlow protocol provides a standard communication interface that allows to operate on the flow rules, without the need of exposing the actual internal working of the switches. OpenFlow runs on top of TCP, thus allowing to establish a secure control channel between the switch and the SDN controller.

Many open source SDN controllers have been developed. In the literature, most of the works we have found use some of the following ones: NOX [10], POX [11], Ryu [12], FloodLight [13], OpenDayLight [14] or Open Network Operating System (ONOS) [15]. Among these, only the last two of them, OpenDayLight and ONOS, present a distributed architecture design, and so they are the most used nowadays. They are indeed very similar in many aspects, being OpenDayLight aligned with the networking vendors and telcos whereas ONOS has the support of vendors and service providers. Both are compatible with many southbound interfaces in addition to OpenFlow. The good documentation provided by the ONOS wiki and the high activity level of its developer community, makes
Chapter 1. Introduction

Figure 1.2: Energy-Efficient Ethernet model. Retrieved from [2].

it an appealing candidate for SDN beginners.

1.3.2 IEEE 802.3az

The growing concerns about energy consumption in networking led the IEEE 802.3 Working Group to start an effort to increase the energy efficiency of Ethernet networks in 2006. The result of this effort is the IEEE 802.3az [16] amendment to the standard, informally known as Energy-Efficient Ethernet (EEE) [17], which was first approved in 2010.

This amendment to the standard defines mechanisms to stop the transmission when there are no packets to be transmitted and also the ability to rapidly start it again when there is new data to send. For this reason EEE presents the Low Power Idle (LPI) state, which substitutes the continuous IDLE signal when there are no packets to send. This LPI defines long periods of inactivity (i.e., without transmitting anything) and short periods of refresh transmission to keep the state synchronization among the transmitter and the receiver. The energy consumption while the interface is in LPI state can be reduced to just the 10% with respect to the active mode. Thus, the longer an interface stays in this LPI state the lower the energy consumption in a period of time will be.

Nevertheless, the transition between LPI state and active state is not instantaneous: there is a fraction of time involved in turning off an interface (i.e., changing from active to LPI), $T_S$ and turning on an interface (i.e., changing from LPI to active), $T_W$. The actual energy consumption during the transitions can be up to the 100%, since most of the components of the interface must be active. The values for $T_S$ and $T_W$ are respectively 2.88 $\mu$s and 4.48 $\mu$s for a 10GBASE-T interface, which are greater than the transmission time of a 1500 B packet, which is 1.2 $\mu$s. The states of an Energy Efficient Ethernet interface are depicted in 1.2.

Given that the times involved in the transition from states are not negligible with respect to the transmission of a frame, different modes of managing when to enter and exit the LPI state have been proposed. The two main modes are the frame transmission and the packet coalescing (or burst transmission). The former is the simplest behavior which as
soon as a packet arrives to an interface in LPI state it wakes up the interface to transmit the packet. On the other hand, the *packet coalescing* mode allows for the storing of the packets in the queue of the transmitter until a number of packets is gathered. Then, all the packets which are in the queue are transmitted in a burst. Packet coalescing reduces the number of transitions between states and therefore maximizes energy savings, and the delay introduced in the packets can be bounded using a maximum timeout since the first packet of the burst arrived to the queue, as stated in [17].

### 1.3.3 Energy efficiency in SDNs

We present here the related work regarding energy consumption minimization in software-defined networks that we have found in the literature.

The authors in [18] carried out a survey on energy efficiency. They identify which components involved in the SDN networks can be configured in a dynamic way in order to reduce energy usage. Most of the solutions analyzed rely on re-routing the flows in the network so that the number of active switches is minimized. This way, these devices can be put in a low-energy state or even turned off, reducing the power consumption of the whole network. They refer to these proposals as *traffic-aware*, since they know the traffic load which is currently passing through the network. The use of the centralized view of the topology provided by the SDN controller is a key point needed by these solutions.

GreenSDN [19] is an emulation environment built using the Mininet and POX, which is a Python-based SDN controller. The authors summarize the difficulties that they have found implementing a SDN environment with *green* capabilities. They present an integration of three energy saving protocols operating at different layers: Adaptive Link Rate operating at the chip level, Synchronized Coalescing working at the node level and Sustainability-oriented Network Management System, which considers the whole topology to maintain the balance between QoS and energy savings. Particularly relevant to our work is the algorithm mechanism proposed in the node level which also exploits the Low Power Idle (LPI) state defined in the IEEE 802.3az standard. Nevertheless, they do not consider setting individual ports in low power idle mode when the traffic traverses an aggregate between switches.

Another interesting proposal is presented in [20], where the energy-efficient flow routing is formulated as an optimization problem. They present an heuristic, named EMMA, which aims to concentrate the traffic on the smallest possible set of nodes in the whole topology, so that the number of idle switches increases. They have implemented EMMA as an ONOS application and evaluated the proposal in a network emulated through Mininet. Particularly, this proposal considers that the flows have previously declared their demanded rates. Thereby, when a new flow arises, EMMA tries to allocate all the active flows in the subset of the network topology which is currently active. If they cannot be
allocated in the active topology, it computes the allocation in the whole network topology. Analogously, when a flow is removed, EMMA will try to determine whether existing flows can be re-routed so that energy consumption of the network diminishes.

ElasticTree [3] and ECODANE [21] present solutions focused on datacenter networks, where they exploit the redundancy in this type of networks and the wide variety in the workload that the datacenter must support over time. On the one hand, ElasticTree is an heuristic that adjusts the set of active devices to support the changes in the traffic load. This proposal has been validated over a testbed composed of production OpenFlow switches, using real traffic traces from a e-commerce website, saving up to a 50% of energy. On the other hand, the authors in ECODANE built an emulation framework using Mininet and the NOX SDN controller. Their system is composed of 5 modules: the optimizer, which is in charge of determining the minimum subset of the topology that needs to be active, the power control module that manages the power states of the switches, the forwarding module that manages the flow rules installed in the switches to forward the traffic, the traffic generator which generates the traffic to perform simulations and the data center network itself. Their results obtain between a 10% and 35% of energy reduction, depending on the source and destination of the traffic.

1.4 Contributions

In this master’s thesis, we have analyzed the energy consumption problem in bundles of Energy Efficient Ethernet links from a Software-Defined Networking perspective. We have designed a solution which leverages SDN to reduce the energy consumption in bundles of Energy Efficient Ethernet links. We have proposed a complete SDN application which is capable of adapting to the real traffic demand and concentrate the traffic into few ports of the bundle so that energy savings are maximized. We have explored different ways of estimating the rate of the traffic flows and proposed three energy-efficient allocation algorithms with different characteristics. Then we have conducted simulations to evaluate their performance not only in terms of energy consumption, but also in terms of packet losses and delay. Moreover, we have also implemented the SDN application on top of the ONOS controller and validated its operation in a network emulated with Mininet.

Since reducing the energy consumption increases the traffic delay, we have then considered traffic with different QoS requirements in terms of latency. We have proposed two different solutions to handle the low-latency traffic while at the same time reducing the energy consumption. Again, we validated the solutions both through simulations and implementing the application on top of the ONOS SDN controller. Our results show that it is possible to provide a low latency service to desired flows while at the same time minimizing the energy consumption using SDN.

The network simulator developed in this work has been released as open-source software...
1.4. Contributions

in [22], licensed under the GNU General Public License, version 3 (GPL-3.0). Thus, it is a platform to conduct research on SDN applications which periodically reallocate the flows in bundles of links between switches. Even though in this work we have focused on minimizing energy consumption, the simulator provides different measurements and features, such as QoS identification and priority queues on the ports, making room for the study of algorithms with a wide range of purposes.

Part of the work described in Chapter 2 has been presented in The 5th International Conference on Software Defined Systems (SDS) [23]. Likewise, the work presented in Chapter 3 has been submitted to The 9th Symposium on Green Networking and Computing (SGNC) which will be held in the frame of the 26th International Conference on Software, Telecommunications, and Computer Networks (SoftCOM).
In this chapter, we address the energy-efficient allocation of traffic flowing between two switches through an aggregate of EEE links, from the point of view of a software-defined network. We first recall the optimum solution from an analytical point of view and then analyze the limitations to implement this solution in SDN. We continue with the description of our solution explaining the different algorithms we propose to achieve energy savings. Finally, we validate our solution not only through simulations but also with an operative implementation using the ONOS SDN controller and a network emulated with Mininet [24].

2.1 Optimum allocation

We present here some preliminary results derived in [8], explaining the optimum allocation of traffic in a bundle of EEE links in a way to minimize energy consumption. This paper shows that the minimum energy usage can be achieved using a water filling algorithm, that is, a new port will only be used to transmit a packet if the packet cannot be transmitted by any of the ports already being used, since they are operating at its full capacity.

This result holds for the main classes of functions used to manage the power state of the IEEE 802.3az ports, i.e., frame transmission and burst transmission modes. In addition, they present an algorithm to show these results, which operates on a per-packet basis, deciding the port which will be used for each packet based on the occupation of the ports. Following a naive water-filling algorithm and only diverting traffic to a new idle port when the previous ones are completely used at its full capacity will lead to delay increasing boundless. Hence, they propose a simple modification to maintain the average delay bounded to a target value, by using the average delay of the already queued packets to determine the output port of a packet.
2.2 SDN application design

The SDN paradigm moves the forwarding logic from the devices itself to the SDN applications which run on top of the logically centralized controller. The switches are now pure forwarding fabrics instructed by the SDN controller. Thereby, instead of having to modify the firmware of the devices we will devise a SDN application to perform this energy-efficient allocation of the traffic.

Nevertheless, the reasons why the algorithm described in [8] cannot be directly implemented as an SDN application are twofold:

- In their algorithms, the switch decides individually for each packet which port will be used to forward it, based on the instantaneous occupation of the ports, what we will refer to as packet-level operation. On the contrary, SDN does not allow us to forward each packet individually, but it works at the flow level applying the same actions to the packets of a flow (i.e., forwarding the packets to the same port).

- Their proposal uses the current occupation of the queue of each port to determine which port a packet should be forwarded to. Sadly, this metric is not provided by typical SDN switches (e.g., it is not considered in OpenFlow).

2.2.1 Flow definition

Since SDN operates at the flow level, the first point in the design of the SDN application is to define the flows that we are going to work with. Recall that a flow is identified by a set of fields of the packets. For example, a flow can be defined considering just one field of the packets (e.g., the destination MAC address) or considering several fields of the packets (e.g., MAC addresses, IP addresses, transport ports, DSCP field, etc.)

Our goal is to be able to spread the traffic among the links of the bundle, knowing that all the packets of a flow will be forwarded equally, being each flow associated with a port. Focused on that goal, we can establish the desired features for the flows:

- Enough number of flows in order to be able to split the traffic among the links of the bundle.

- Low number of flows so that flow tables keep small, resulting a high performance of the switches.

- Predictable demand in the flows. Since we do not know the rate of each flow beforehand, we would like each flow to have a low variance in the binary rate that it demands along time.
2.2. SDN application design

We have explored two different ways of defining the flows, namely flow tagging and field matching.

The flow tagging approach consists in implementing a non-SDN process, which will run at the network edge. This process will inspect the packets and tag them in a field which can be later used to identify the flows (e.g., DSCP). The criteria used for this non-SDN process to decide which tag will be imposed to each packet could perfectly emulate the behavior of the aforementioned packet-level algorithm described in [8], thus achieving optimum results. However, this proposal would introduce an unacceptable overhead in the data plane traffic because all the packets have to pass through a centralized tagger process outside the SDN network. Actually, these proposals would not exploit the SDN capabilities, being the real forwarding logic located at the non-SDN tagger process rather than in the SDN controller. Although it would be possible to implement this tagger process at the SDN controller, this solution is not valid at all, since it would mean that all the data plane traffic would be forwarded through the SDN controller, which is not scalable.¹

Since the flow tagging proposal is not scalable, we devise a flow definition using the fields of the packets, which is purely SDN-based. We have named this proposal field matching. It is important to study now which the adequate level of granularity for the solution is, based on the desired features of the flows. Using just MAC addresses could be too coarse granularity, for example in a transit network where the traffic travels between a small set of routers. On the other hand, using transport layer flows could result in too many flows, deteriorating the performance of the switches due to huge flow tables. Even defining the flows at the network layer, that is the pair of source and destination IP addresses, would result in too many flows: since each IP address consists in 32 bits, we can have up to $2^{32}$ different flows, which cannot be manageable by the switches.

Instead of using the source and destination IP addresses to identify our flows, we will use only some bits of the IP addresses to reduce the number of flows. This way, the original end-to-end layer 3 flows would be aggregated into coarse flows. We have explored now how to perform this aggregation (i.e., which bits we will use to define the aggregated flows) so that the aggregated final flows show a stable rate. To this end, we have explored different possibilities for the selection of the bits that will define the aggregated flows. We have computed how many of the original flows were bucketed into each aggregated flow using real traffic traces retrieved from the publicly available CAIDA dataset [25]. In particular we have analyzed more than 33 million packets belonging to almost 800 thousand end-to-end network flows. Note that the ideal results would be that all the aggregated flows contain the same number of original flows so that the flows are as balanced as possible, depicting a flat histogram.

¹Note that the SDN controller communicates with the switches through the control channel, which cannot handle all the data plane traffic.
Fig. 2.1 shows the distribution of the original flows into the aggregated ones comparing the usage of the first and last bits of the destination IP address. Although we have carried out the experiment with different values for the number of bits, we only show the graphs for 8 bits since the results are analogous. At a glance, we can see that the last bits produce a considerably flatter histogram than the first ones. This is due to the hierarchical structure of the IP addresses, since the last bits correspond to the host identifier (which introduces a great variability in the bits allowing for a balanced distribution) whereas the first bits correspond to the network itself.

As a result, we would be interested in using the last bits of the destination IP address to identify our flows. However, although the OpenFlow protocol allows the possibility of using arbitrary bitmasks over IP addresses, not every SDN controller implements that feature (e.g., ONOS does not), as a matter of fact. Actually, many SDN controllers do only allow us to match on the first bits of the IP addresses so we decided to use the first bits of the destination IP address. We have also explored the usage of the first bits of the source IP address, and also a combination of the first bits of both source and destination IP addresses, but the results did not improve.

We have also analyzed which one is the appropriate number of bits to use. This step is actually important, since it will determine the number of flows that the switches will have to handle. Fig. 2.2 represents the distribution of the original flows in the aggregated ones using different number of bits of the last part of the destination IP address. The histograms are all relatively flat, although we can notice the presence of a negligible number of spikes for the larger number of bits.

In order to perform a quantitative comparison we have computed the variance in the number of original flows contained in each aggregated one. The results of the variances are collected in Table 2.1. The first we must notice is that the variance of last bits is some orders of magnitude lower than the first bits, for every single value of the number of
### 2.2. SDN application design

![Flow distribution graphs](image)

**Figure 2.2:** Flow distribution of the destination address using different number of bits.

<table>
<thead>
<tr>
<th>Number of bits</th>
<th>Variance (number of original flows)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>First bits</td>
</tr>
<tr>
<td>4</td>
<td>1199855326</td>
</tr>
<tr>
<td>6</td>
<td>129835835</td>
</tr>
<tr>
<td>8</td>
<td>18666837</td>
</tr>
<tr>
<td>10</td>
<td>2987168</td>
</tr>
<tr>
<td>12</td>
<td>580385</td>
</tr>
</tbody>
</table>

**Table 2.1:** Variance in the distribution of the original flows for the destination address.
bits, reinforcing the idea that the use of the last bits yields a more uniform distribution. The next we can observe is a strong inverse relationship between the number of bits used and the variance of the distribution, both for the first and last bits of the destination IP address. That is, the more bits used the more balanced the distribution of the flows, so we will try to use as many bits as possible according to this criterion. Nevertheless, the number of bits will determine the number of flows the switches will have to handle: using \( b \) bits yields to \( 2^b \) flows, so we should use as few bits as possible regarding this criterion. Based on these results, we see that a value of 8 bits is a good trade-off between both considerations, resulting in a maximum of 256 flows. This provides a enough level of granularity so that the traffic can be distributed among the links of the bundle while at the same time keeps the flow tables small.

To sum up, taking into account the results just described, our flows will be defined by the destination MAC address and also the first 8 bits of the destination IP address.

2.2.2 Application workflow

After explaining how we are going to define the flows, the next step in the design of the SDN application is to define its logic.

It is important to bear in mind that the SDN works at the flow level: Each time a packet reaches a SDN forwarding device (i.e., a switch), it will look up in its flow tables searching for a flow rule that matches with the packet to be forwarded. When a match is found, the actions in that flow rule will be applied to the packet (i.e., the packet will be forwarded to a port) and the counters stored in that flow rule will be updated. Therefore we are able to treat each packet individually in each port, but we will operate the flow as a whole.

In our application, we will implement a reactive forwarding behavior. That is, a low-priority rule will be installed in the switches to send the packets to the SDN controller. This way, the packets that do not match any flow rule with higher priority will be sent to the SDN controller, which has a centralized view of the topology and will act in response. The SDN controller will transfer this packet to our application, which runs on top of the controller. Next, our application will determine which port the packet should be forwarded to and install a medium-priority flow rule, according to the criteria defined in section 2.2.1, in the switch that sent the packet. Therefore, future packets belonging to this same flow will be directly forwarded by the switch at the line rate without traversing the controller.

The SDN controller provides our SDN application with a view of the whole topology, enabling the computation of shortest paths to the packet destination. However, if the SDN controller does not know the destination yet, it will flood the packet out of all ports except the input port and using only one port of each bundle, without installing a flow rule for this packet yet. This way, when a packet is received at our application, it will
2.2. SDN application design

determine the next hop switch the packet should be forwarded to. This way when the
next hop is behind a bundle of links, our application selects at random a port of the
bundle to forward the packet and installs the flow in the switch accordingly. The random
selection is performed since the application does not have prior information about the
transmission rate of this new flow. Allocating the flow to the most used port would
introduce excessive losses if the flow demands a high rate. On the contrary, using an idle
port would activate a port, drastically increasing the energy consumption needlessly if
the flow demands a rate that can be handled without problem by the already used ones.
Furthermore, it is important to note that this is just an initial transient when a new flow
appears on the network as we will explain below.

What we have described so far is the part of the SDN application that manages the packet
forwarding in the network providing connectivity. It uses our custom flow definition but
spreads flows at random in the bundles between switches since no \textit{a priori} information
is had about the rate of the flows, thus not performing any energy-aware optimization.
Now, we proceed to describe how we have managed to reduce energy consumption in the
bundles. Fig. 2.3. describes the actions of the application in each interval. In detail, our
application will periodically perform the following tasks:

1. Retrieve the list of switches.

2. For each switch, identify the neighbors of the switch (i.e., the switches which a link
to it).

Figure 2.3: Flowchart with the logic of the application.
3. For each neighbor retrieve the ports in the switch that are connected with the neighbor.

4. If there is more than one port (i.e., there is a bundle between the two switches), retrieve the flows installed in the switch that forward packets to a port of this bundle.

5. Estimate the rate of each flow. That is to say, the amount of traffic that the flow will transmit in the next interval.

6. Compute a new allocation for these flows to the ports of the bundle in a way that energy consumption is minimized.

7. Instruct the switch to modify the flow rules that have changed their allocation.

There are two key challenges that remain open in this design:

- How to estimate the rate of each flow?
- How to allocate the flows to the ports of the bundle so that energy consumption is minimized?

On the one hand, the estimation of the rate of each flow is presented in section 2.2.3. On the other hand, the allocation of the flows to the ports of the bundle will be the main task of the allocation algorithm, described in section 2.3.

2.2.3 Flow rate estimation

The rate that each flow will demand in the following interval is estimated leveraging the counters that each flow rule stores. The counters include the number of packets and bytes that have matched with this flow along with the duration of the flow (i.e., the time that the flow has been active). When the application queries the flow rules installed in the switch, it will store the value of the bytes counter of each the flow. Thus, we can calculate the bytes that this flow has transmitted in the previous measurement interval calculating the difference between the current value of the bytes counter and the value stored in the previous interval. We calculate the rate during the previous interval as the division of the bytes transmitted in that interval and the duration of the interval (i.e., the sampling period of the application). If the flow was not present in the previous interval, we use the value of the duration of the flow instead of the sampling period, for an accurate measure (note that in this case the duration of the flow will be no longer than the sampling period). These values of the measured rate of the flow throughout the periods will be the information we will use to predict the rate of the flow in the next interval.
Since this estimation is a challenge, we have explored two simple estimators:

1. Using directly the measured value of just the previous interval.

2. Using an exponentially weighted moving average (EWMA) with the measured rates of the flow in the previous intervals. Particularly, the rate of the flow in the next interval is given by equation (2.1)

\[
R_n = \begin{cases} 
M_n, & n = 0 \\
\alpha \cdot M_n + (1 - \alpha) \cdot R_{n-1}, & n > 0
\end{cases}
\]  

(2.1)

where \( R_n \) represents the value of the EWMA at time \( n \) (thus the rate estimated for the interval \( n + 1 \)), \( M_n \) represents the measured rate in the interval \( n \) and the constant \( \alpha \in (0, 1] \) is a parameter that tunes the relevance of the samples as time goes by (higher values of \( \alpha \) reduce the importance of older rates).

We have analyzed the quality of the estimation using these two proposals, calculating the error in the estimation as the absolute value of the difference between the estimated value and the real value, for each flow in each interval. The results for a 32.5 Gbit/s trace are shown in Fig. 2.4. As we can see, the direct usage of the previous interval performs very similar to the usage of the exponentially weighted moving average for the different \( \alpha \) values and sampling periods studied. We can also perceive considerable high errors in
the estimated rate for sampling periods lower than 0.1 seconds. This is due to the high variability of the flows when using such a small time window. Therefore, we will directly use the rate of each flow in the previous interval to forecast its rate in the next interval, since it is a simpler method than the EWMA providing almost the same accuracy in the estimation.

2.3 Allocation algorithms

In this section we describe the allocation algorithms that we have implemented in order to reduce the energy consumption. The main task of this algorithm is to allocate a set of flows to a set of ports so that energy consumption is minimized. The input of this algorithm will be the set of flows to be allocated along with their estimated rates and also the set of ports that make up the aggregate.

2.3.1 Greedy algorithm

This solution attempts to fill the ports to the maximum of their capacity, only allocating a flow in an empty port if it does not fit in any of the already used ones. In this way, we attempt to use only the minimum number of ports that are needed, using them the closest to their nominal capacity and leaving the maximum number of ports completely empty so that energy savings are maximized. The optimum solution to this problem is actually a combinatorial NP-hard problem, since it is necessary to try all the possible combinations of assigning each flow to each port and determining which combination uses less ports. Accordingly, we will propose an heuristic solution, since the optimum is not scalable because it requires evaluating $|\text{ports}|^{|\text{flows}|}$ combinations. Instead of computing all the possibilities as in the optimum solution, our algorithm will analyze the flows in a decreasing order based on their estimated rates and allocate them to the most used port. Note that although it somehow follows a water-filling approach, it does not operate at packet level but at the flow level.

The pseudo-code of the algorithm is shown in Fig. 2.5, where bound is always set to 0. Firstly, the algorithm sorts the flows to be allocated decreasingly on their estimated rate. Next, the flows are allocated in a sequential order so that the port occupation is maximized: The ports are evaluated in a predefined order (e.g., ordered by port identifier) and the flow is allocated to the first port it fits. We consider that a flow fits in a port when the sum of all the flows assigned to the port is less than the nominal capacity of the port. This algorithm is akin to the classical First Fit Decreasing (FFD) heuristic solution of the bin packing problem [26].

We expect this algorithm to yield low values of energy consumption. Nevertheless, since this algorithm attempts to use the full capacity of the ports, the delay can grow
2.3. Allocation algorithms

allocate_greedy(flows, ports, bound=0) {
    // Hold assigned port for each flow
    flow_allocation[1..|flows|] = ∅

    // Sort flows by decreasing load value
    ordered_flows = sort(load(flows), DECREASING)

    // Initialize occupation of the ports to 0
    port_load[1..|ports|] = 0
    port_flows[1..|ports|] = 0

    for flow ∈ ordered_flows {
        for port ∈ ports {
            if ((port_flows[port] == 0) ||
                (port_load[port] + load(flows)[flow] ≤ 1 - bound/port_flows[port]))) {
                // Update port with the load of this flow
                port_load[port] += load(flows)[flow]
                port_flows[port] += 1
                flow_allocation[flow] = port
                break
            }
        }
    }
    return flow_allocation
}

Figure 2.5: Pseudocode for the Greedy Algorithms.
uncontrollably as advertised in [8]. Besides, since we are working with rate estimations, any fluctuation could lead to a non negligible level of packet losses for almost any buffer size.

2.3.2 Bounded-Greedy algorithm

This algorithm is a modification of the Greedy one, which attempts to control the delay of the packets and also reduce the packet losses that the Greedy algorithm introduces for a given buffer size. One source of losses is the fact that using of the ports very close to its nominal capacity is easily affected by fluctuations in the rate demanded by the flows. Therefore, this algorithm will try to avoid using the links to the maximum of their capacity by setting a threshold in the maximum load that can be allocated on a port. Specifically, we will limit the fraction of the port capacity that can be used to an increasing function in the number of flows already allocated to the port. We have used the following function

\[ 1 - \frac{B}{N} \]

where \( N \) is the number of flows already allocated to the port and \( B \) represents the fraction of space that cannot be used in a port when there is only one flow allocated to it. For the rest, the algorithm operates in the same way as the Greedy.

In this way, we try to avoid that ports which are used very close to their nominal capacity only contain a few coarse flows which can introduce greater variability, while at the same time we allow for ports with many flows can be used closer to its full rate. The pseudo-code is also shown in Fig. 2.5, where \( \text{bound} \) is the fraction of space that cannot be used in a port when there is only one flow allocated to it.

2.3.3 Conservative algorithm

Despite the effort of the Bounded-Greedy algorithm to mitigate packet losses and control the delay of the packets, the results may not be acceptable yet, as we will later show. Hence, we have designed another algorithm that does not only minimize energy consumption but also reduces packet losses.

The idea behind this algorithm is to first compute the minimum number of needed ports for the next interval. This value is lower bounded by the sum of the estimated rates of all the flows that will be transmitted through the bundle. Then, the flows are evenly distributed among the needed ports previously determined. Although this behavior minimizes the individual occupation of each link, it does not follow a water-filling approach. However, as we will show later, this does not really degrade energy consumption. The reason is that the individual energy consumption raises very quickly with the occupation of the link, as we can see in Fig. 2.6. As a consequence, when a port has an occupation higher
than about 20\% its impact in the consumption is minimum. Thus, this algorithm prefers to have its used ports with a balanced traffic occupation avoiding the need of using the ports very close to their nominal capacity in many situations as would happen with the Greedy algorithms.

In addition, with the aim of further reducing the likelihood of packet losses, rather than using the total estimated load to compute the number of needed ports we add to this value a safety margin, which we have experimentally established equal to 20\%. The number of ports is calculated as the ceiling function of the sum of the normalized total load and the safety margin. In this way, we safely avoid the situations where the ports would be used excessively close to their nominal capacity.

After determining the ports that will be used in the next interval, the Conservative algorithm proceeds with the allocation of the flows attempting to achieve a balanced distribution of the flows to ports, both in terms of rate and number of flows. To accomplish this, the algorithm performs a minimization of the occupation of the ports. As shown in the pseudo-code in Fig. 2.7, we first sort the flows decreasingly in the estimated rate and then sequentially allocate the flows to the port with the lowest occupation among those that will be used in the interval. Note that this algorithm is also capable of maintaining some ports completely idle, reducing energy consumption.

2.4 Experimental results

We have evaluated the allocation algorithms in a scenario composed of two switches. The switches are connected through an aggregate composed of 5 10GBASE-T interfaces.
Chapter 2. Energy-efficient algorithms for Ethernet link aggregates

```plaintext
safety_margin = 20%

allocate_conservative(flows, ports) {
    // Hold assigned port for each flow
    flow_allocation[1..|flows|] = ∅
    expected_load = sum(load(flows)) + safety_margin
    minimum_ports = ceil(expected_load)

    // Only use the minimum number of ports
    used_ports = ports[1..minimum_ports]

    // Sort flows by decreasing load value
    ordered_flows = sort(load(flows), DECREASING)

    // Initialize occupation of the ports to 0
    port_occupation[1..|used_ports|] = 0

    for flow ∈ ordered_flows {
        port = get_port_min_occupation(port_occupation)
        // Update port with the load of this flow
        port_occupation[port] += load(flows)[flow]
        flow_allocation[flow] = port
    }

    return flow_allocation
}
```

Figure 2.7: Pseudocode for the Conservative Algorithm.
2.4. Experimental results

The traffic is generated from one host which is connected to a switch and it is sent to another host connected to the other switch, forcing the traffic to go through the bundle. A screenshot of the ONOS Web interface with the topology of this scenario is depicted in Fig. 2.8.

Real traffic traces have been used for the different experiments. The traffic traces have been obtained from the public CAIDA dataset [25]. The particular trace that we have used comes from a 10 Gbit/s and the average rate is about 3.25 Gbit/s. This value is relatively low for our 50 Gbit/s aggregate, so we have reduced the inter-arrival times by a constant factor in order to increase the rate in the same amount.

2.4.1 Metrics

We are interested mainly in three performance metrics. Firstly, the overall normalized energy consumption is the main metric that we have used to validate the energy saving capabilities of our algorithms. Secondly, we have measured the packet losses that occur using our algorithm for a given buffer size, i.e., packets are discarded when they are forwarded to a port and the buffer of the port is full. Lastly, we have also measured the average delay experienced by the packets, taking into account the queue waiting time, the transmission time and the transitions times to enter (T_S) and exit (T_W) the low power mode (LPI) of the individual EEE ports. The times to enter LPI and wake up an interface are set to 2.28 µs and 4.48 µs, respectively, as defined in the IEEE 802.3az standard [16].

The energy consumption is calculated in two complementary ways. On the one hand we have calculated the energy consumption using an IEEE 802.3az simulator, available for download at [27]. To this end, we have calculated the individual consumption of
each port, feeding the same traffic that we sent by the port to instances of the IEEE 802.3az simulator. Then, the total consumption is given by the average of the individual consumption of each port.

On the other hand, we have analytically calculated the consumption using models from the literature. The global consumption is calculated as the average of the individual consumption of the ports of the bundle. The individual energy consumption of a port is calculated as the time average of its instantaneous consumption. Since our application already slots the time in fixed-duration intervals, we compute the overall energy consumption of a port directly as the average of the consumption on each interval. Finally, we can calculate the consumption on each interval using well tested models already presented in the literature \[28,29\]. Particularly, we have used the following formula, taken from the model for IEEE 802.3az interfaces described in \[28\]

\[
\sigma(\rho_i) = 1 - (1 - \sigma_{off})(1 - \rho_i) \frac{E[T_{off}(\rho_i)]}{E[T_{off}(\rho_i)] + T_S + T_W}, \tag{2.2}
\]

where \(\sigma(\cdot)\) is the energy consumption, \(\rho_i\) is the traffic load on link \(i\), being normalized both quantities (i.e., \(\sigma(\cdot), \rho_i \in [0, 1]\)). The value of \(\sigma_{off}\) has been set to 0.1 according to different estimates provided by several manufacturers, and \(T_S = 2.28 \mu s\) and \(T_W = 4.48 \mu s\) according to the IEEE 802.3az standard \[16\]. Moreover, assuming Poisson traffic and the frame transmission mode is used in the ports, we have that

\[
E[T_{off}(\rho)] = e^{-\mu \rho T_S} \frac{T_S}{\rho}, \tag{2.3}
\]

where \(\mu^{-1}\) is the average duration of the packet transmission.

Note that the first alternative used to calculate the energy consumption provides exact results but needs to know the exact transmission instant of each packet by the ports. Therefore, it is valid for the simulations, but it is difficult to be measured in a real ONOS application. Nevertheless, as we have compared in the simulations the differences are very minor, validating the accuracy of the models.

### 2.4.2 Description of the experiments

Our experiments are divided in two categories: simulations and real implementations. First, we conduct simulations to analyze the performance of the algorithms in terms of the different metrics previously mentioned. We start analyzing the impact of the sampling period in the results for a given rate of 32.5 Gbit/s and then further validating our results with different rates 6.5 Gbit/s, 13 Gbit/s, 19.5 Gbit/s, 26 Gbit/s and 32.5 Gbit/s. Finally, we implemented the application on top of the ONOS SDN controller and tested it in a network emulated with Mininet \[24\].

24
2.4. Experimental results

Even though there exist some network simulators, we have decided to develop our custom Java-based network simulator, available for download at [22]. This way, most of the code of the algorithms can be shared with the real ONOS application. Note that this simulator will allow us to evaluate some type of metrics using our algorithms that will not be possible in the ONOS application, such as an accurate energy consumption calculation (determining the time that each port have been idle, considering the times to power on and off an interface), packet losses for different buffer sizes and delay of the packets.

We have discarded the results of the first interval, since during the first interval flows are just allocated at random (due to the lack of a priori information about the rate of the flows). This interval can be regarded as transient state where packet losses are likely to occur and no energy-aware allocation is performed.

In order to be used as a baseline to compare the results of the proposed energy-aware algorithms with, we have designed another algorithm which distributes the flows among all the ports of the bundle. We have named this algorithm the equitable.

2.4.3 Energy consumption: simulation results

The first experiment evaluates the variation of the energy consumption with the duration of sampling period of the algorithm for a rate of 32.5 Gbit/s, which is a tenfold increase in the rate of the original traffic. Fig. 2.9 shows the results of that experiment for a buffer

![Figure 2.9: Energy consumption variation with the duration of the sampling period.](image-url)
Chapter 2. Energy-efficient algorithms for Ethernet link aggregates

The formulas previously described for the energy consumption model yield the following theoretical lower bound for the global normalized energy consumption as the 78.5% for a packet size of 1500 bytes. This is achieved when the 32.5 Gbit/s trace is allocated in the bundle in the following way: 3 ports fully utilized at 10 Gbit/s, one with 2.5 Gbit/s and the last one with no traffic. This configuration yields the following energy consumption, respectively: 3 ports consuming the 100%, one consuming 83.25% and the last one 10%. Averaging these values yields the final consumption of 78.5%.

Looking at the results depicted in Fig. 2.9, the three proposed algorithms consume about an 80%, which is nearly a 20% less than the baseline equitable algorithm. We can also notice that the results of the three algorithms are very close to the analytical optimum of 78.5% validating the energy saving operation of the algorithm. We can also notice that the energy consumptions attained by the three algorithms are practically the same. Besides, we can also observe from Fig. 2.9 that very low values (e.g., lower than 0.1 seconds) present higher consumption than values greater than 0.1 seconds, hence lower energy savings.

The results in terms of energy consumption for the different traffic traces are shown in Fig. 2.10 using a sampling period of 0.5 seconds and a buffer size of 10000 packets. We can observe that the energy consumption is almost identical for the three proposed algorithms, being considerably lower than the non-energy-efficient equitable algorithm,
Table 2.2: Comparison between energy consumption results.

<table>
<thead>
<tr>
<th>Rate (Gbit/s)</th>
<th>Greedy real</th>
<th>Greedy model</th>
<th>B-Greedy real</th>
<th>B-Greedy model</th>
<th>Conservative real</th>
<th>Conservative model</th>
<th>Equitable real</th>
<th>Equitable model</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.5</td>
<td>27.94</td>
<td>27.96</td>
<td>27.94</td>
<td>27.96</td>
<td>27.94</td>
<td>27.96</td>
<td>77.82</td>
<td>79.41</td>
</tr>
<tr>
<td>13.0</td>
<td>45.37</td>
<td>45.13</td>
<td>45.26</td>
<td>45.18</td>
<td>45.77</td>
<td>45.92</td>
<td>90.23</td>
<td>93.86</td>
</tr>
<tr>
<td>19.5</td>
<td>48.16</td>
<td>47.86</td>
<td>49.63</td>
<td>49.04</td>
<td>62.42</td>
<td>63.09</td>
<td>94.10</td>
<td>97.94</td>
</tr>
<tr>
<td>26.0</td>
<td>63.80</td>
<td>63.92</td>
<td>62.71</td>
<td>64.82</td>
<td>63.67</td>
<td>63.99</td>
<td>95.79</td>
<td>99.26</td>
</tr>
<tr>
<td>32.5</td>
<td>79.59</td>
<td>79.26</td>
<td>79.32</td>
<td>81.94</td>
<td>79.52</td>
<td>81.97</td>
<td>96.62</td>
<td>99.73</td>
</tr>
</tbody>
</table>

used as baseline. There is just a slight difference in the case of the 19.5 Gbit/s, where the Conservative algorithm consumes a bit more than the Greedy algorithms, since it uses 3 ports while the Greedy algorithms will try to allocate the flows using just two ports. Nevertheless, the consumption attained by this Conservative algorithm is indeed much lower than the equitable one. Note that for low load traffic rates (such as 6.5 or 13 Gbit/s), our algorithms are capable of reducing the energy consumption about a 50%.

Overall, these results depict the real energy consumption, calculated from the times that the interface has been in low power idle mode. However, in the real implementation we will use the model previously described to calculate the energy consumption since the LPI times cannot be easily calculated. For this reason, we have also computed the consumption in this way using our developed simulator and compared both measures. The results of this comparison for a sampling period of 0.5 seconds are collected in Table 2.2.

The results show that the consumption estimated by the model are very close to the real values. The deviation of the model from the real value is never higher than a 4%. We can also notice that the highest deviations are produced for elevated energy consumptions (e.g., in the equitable algorithm). Moreover, we can notice that in this situations the real value is always lower than the one estimated by the model. This is due to the model assumes that the traffic is Poisson. However, although the analyzed traffic roughly follows a Poisson distribution, the actual behavior of the algorithms concentrates the flows in the ports in a deterministic manner. For this reason, the actual traffic send by each port slightly differs from a Poisson distribution. It happens that in some ports there are few flows assigned which are actually transmitting traffic in bursts. This allows the interface in question to spend more time in LPI mode than would occur if the traffic were purely Poisson, so the real energy consumption is lower than in the model. Even so, the energy consumption calculated by the model is very close to the real value.
2.4.4 Packet loss: simulation results

This experiment evaluates the packet losses introduced by each algorithm. Fig. 2.11 presents the variation of the packet losses with the sampling period, using a buffer size of 10,000 packets. Fig. 2.12 explores the packet losses introduced for different buffer sizes, using a sampling period of 0.5 seconds. The Greedy algorithm is the one which introduces more packet losses, followed by the Bounded-Greedy, then the Conservative one and finally the baseline equitable algorithm, which introduces no losses for every sampling period and buffer size. These results confirm our hypothesis, validating the operation of the Conservative algorithm in terms of packet losses for buffer sizes higher than 1000 packets. Unfortunately, the Greedy algorithms introduce a non-negligible amount of packet losses (higher than a 1 %) being unacceptable for some applications. Besides, note that for very low values of the sampling period (e.g., lower than 0.1 s), the packet losses are lower for all the algorithms.

The results for the different traffic traces are shown in Fig. 2.13, where the sampling period is set to 0.5 seconds and the buffer size to 10,000 packets. As expected, the Greedy algorithm is the one which introduces more losses. Then, the Bounded-Greedy is the next one, introducing also a non-negligible amount of packet losses, even for low rates of traffic. On the other hand, the Conservative algorithm does not even introduce packet losses, almost at the same level of the equitable algorithm which produces practically zero packet loss.
2.4. Experimental results

Figure 2.12: Packet loss percentage variation with the buffer size.

Figure 2.13: Packet loss for different traffic traces.
2.4.5 Average delay of the packets: simulation results

This experiment evaluates the impact on the packet delay introduced by our algorithms. Fig. 2.14 shows the average packet delay variation with the sampling period for a given buffer size of 10,000 packets. The first we can see in the figure is that the average delay introduced in the packets by the Greedy algorithm is about 4 ms which is considerably higher than any of the others, whereas the delay introduced by the Bounded-Greedy is about 1.5 ms, which is still a high value. The Conservative one is an order of magnitude lower, being about 250 µs. This delay is even lower in the equitable algorithm, as expected, being about 50 µs.

The results in terms of average delay of the packets are shown in Fig. 2.15. We observe that the delay attained by the Greedy algorithm is the highest, followed by the Bounded-Greedy one. The Conservative algorithm shows a lower delay, but it is high in comparison to the baseline equitable algorithm. However, even the delay achieved by the Conservative algorithm could be unacceptable for critical applications, especially when the traffic load is high, reaching delays in the order of hundreds of microseconds.

In the particular case of the 6.5 Gbit/s trace the three algorithms behave identically, using just one port for all the traffic. Furthermore, in the Conservative algorithm the delay of the packets using the 26 Gbit/s trace is higher than using the 32.5 Gbit/s one. This is explained due to the principle of operation of the Conservative algorithm which employs, in average, four ports for the latter and just three for the former.
2.4. Experimental results

![Average packet delay for different traffic traces](image)

**Figure 2.15**: Average packet delay for different traffic traces.

2.4.6 Comparing the algorithms

The above results of the simulations exhibit a trade-off between, on the one side energy consumption and on the other side traffic delay and packet losses. As a consequence of reducing the energy consumption, the delay of the traffic is increased, which can even lead to packet losses. This result was expected, since we are reducing the number of ports so the individual occupation of the ports that are used will unfailingly increase.

The analysis of the three algorithms regarding the different metrics shows that the best algorithm is the Conservative one. Although the Greedy algorithms can slightly outperform the Conservative one in some situations, their energy consumption is in most scenarios very similar. Nevertheless, the increase in the delay of the packets and especially the packet losses introduced can be unacceptable. Besides, the computational complexity of the three algorithms is roughly the same for the three proposed algorithms. In addition, the Conservative algorithm allows to fine tune the trade-off between delay and energy savings through the *safety margin*. Increasing the safety margin will contribute to reduce the delay and the likelihood of losses, at the cost of increasing the energy consumption in many situations.

It is also important to ponder on the adequate value for the sampling period. Although the use of low values of the sampling period (e.g., 0.01 seconds) exhibit low delays and packet losses (with the subsequent increased energy consumption), these low values are hardly implementable in practice, since such low sampling periods result in a huge
overhead of control traffic (sampling the flow tables of the switches and instructing up to 256 flow modifications to every switch in each interval). This is hardly manageable by the switches for periods lower than 0.5 seconds and could even lead to instability in the network. What is more, for low sampling periods, this frequent rerouting can harm the performance of TCP as studied in [30]. Notwithstanding, they also show in their results that the impact is noticeable when the new path of the flow presents a longer round-trip time so that packet re-ordering is likely to occur. This situation can happen in our algorithms when moving flows to a more congested port than previously. Yet, it is important to note that by the nature of the Conservative algorithm, the probability of this situation to happen is minimized due to a balanced occupation of the active ports. Accordingly, for a negligible performance degradation, a real implementation should not use sampling periods lower than 0.5 seconds. Also note that for this value onwards the Conservative algorithm is indeed in steady state: there is no penalization in using even greater values for the sampling period, such as in the order of seconds.

2.4.7 Implementation

We have implemented the proposed SDN application on top of the Open Network Operating System (ONOS). We have emulated the same topology with Mininet in order to evaluate the proper operation of the application. Besides, we have also implemented another application on top of ONOS in order to measure the occupation of each port and then estimate the energy consumption as per equation (2.2), as stated above. This application will periodically query the counters stored in the switches regarding the ports and determine the number of bytes that each port has sent in the last interval. The transmitted rate by the port during the previous interval is calculated as the quotient of the number bytes and the duration of the interval. Then, the occupation of the port in the interval is computed dividing this transmitted rate by the nominal capacity of the port. Finally, using this value of occupation we can derive the energy consumption in that interval using the aforementioned model.

After properly implementing and debugging the Java applications, we have installed and activated the applications in the ONOS instance. Then, we have performed several tests using the ping tool to evaluate the connectivity among the between the emulated hosts and also iperf3 [31] to generate synthetic TCP and UDP traffic with specific rates. This way, we validated the proper operation in simple scenarios that helped us with the debugging of the applications.

Once validated the operation of the algorithms with synthetic traffic, we evaluated our application with the same real trace as in the simulations which we have transmitted using tcpreplay [32] at a rate about 330 Mbit/s since the computer used for the experiments is not capable of transmitting this traffic trace at higher rates.\(^2\) Accordingly, the nominal

\(^2\)We have used an Intel® Core™ i7-4710HQ (4th Gen) at 2.5 GHz
2.4. Experimental results

Table 2.3: Average port occupation of the ports of the bundle.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Port 1</th>
<th>Port 2</th>
<th>Port 3</th>
<th>Port 4</th>
<th>Port 5</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greedy</td>
<td>92.57</td>
<td>97.83</td>
<td>97.05</td>
<td>30.36</td>
<td>0.02</td>
<td>63.57</td>
</tr>
<tr>
<td>Bounded-Greedy</td>
<td>83.46</td>
<td>81.16</td>
<td>95.08</td>
<td>61.27</td>
<td>0.02</td>
<td>64.20</td>
</tr>
<tr>
<td>Conservative</td>
<td>84.17</td>
<td>83.60</td>
<td>80.78</td>
<td>79.76</td>
<td>0.02</td>
<td>65.67</td>
</tr>
<tr>
<td>Equitable</td>
<td>83.89</td>
<td>80.52</td>
<td>54.13</td>
<td>53.63</td>
<td>0.02</td>
<td>65.88</td>
</tr>
</tbody>
</table>

Table 2.4: Average energy consumption of the ports of the bundle.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Port 1</th>
<th>Port 2</th>
<th>Port 3</th>
<th>Port 4</th>
<th>Port 5</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greedy</td>
<td>99.89</td>
<td>99.99</td>
<td>99.99</td>
<td>92.36</td>
<td>10.24</td>
<td>80.49</td>
</tr>
<tr>
<td>Conservative</td>
<td>99.77</td>
<td>99.92</td>
<td>99.88</td>
<td>99.89</td>
<td>10.24</td>
<td>81.94</td>
</tr>
<tr>
<td>Equitable</td>
<td>99.78</td>
<td>99.90</td>
<td>99.04</td>
<td>98.97</td>
<td>99.27</td>
<td>99.39</td>
</tr>
</tbody>
</table>

capacity of the interfaces of the bundle has been set to 100 Mbit/s and we have used a sampling period of 10 seconds. The occupation of each port of the bundle averaged throughout 12 intervals in 10 independent executions is shown in Tab. 2.3.

These results of the real implementation totally match with the results of the previous simulations. First, we can see that the Greedy algorithm uses three ports to more than the 90% of their nominal capacity, one about a 30% and leaves the other one empty. These values describe the behavior of a water-filling algorithm, as desired per design. Secondly, the Bounded-Greedy algorithm, avoids having three ports very close to their nominal capacity, but still one port presents an occupation higher than a 95%. The Conservative algorithm behaves exactly as desired, using 4 ports around the 80% and leaving the other one empty. The equitable algorithm, spreads the traffic using all the ports of the bundle, thus each port has is used more than a 50%. We can notice that the last port is not completely idle but presents an occupation of 0.02% in the three energy-efficient algorithms, which represents the residual contribution of newborn flows which are randomly allocated during their first interval.

Moreover, analyzing the average occupation of all the ports we can perceive slight differences. This is because of packet losses, which occur when more than 100 Mbit/s are assigned to a port during an interval. This result, also coincides with our simulations, experiencing almost no packet losses in the Conservative algorithm, more than a 1% in the Bounded-Greedy and more than a 2% in the Greedy.

Analogously, Tab. 2.4 collects the average energy consumption averaged throughout the intervals for the same 10 independent executions. As we can observe, the differences
in the energy consumption among the three energy-efficient algorithms are minimum, and all of them consume about an 18% less than the baseline equitable algorithm, as expected. They only differ in the consumption in port 4, which consumes about a 7% less in the Greedy than in the cases of the Bounded-Greedy and the Conservative. These results match with the simulations, further validating the operation of our energy-efficient algorithms in an emulated environment.

Overall, the results of the implementation highlight that the three energy-efficient algorithms manage to reduce the energy consumption with respect to the baseline in a considerable way. Although the proposed Greedy algorithms can introduce packet losses, the Conservative algorithm is able to mitigate this effect attaining almost the same energy efficiency in most scenarios with negligible impact in packet losses.

2.5 Conclusions

In this chapter we demonstrate the design and implementation of an energy saving algorithm for link aggregates composed of IEEE 802.3az ports leveraging the SDN paradigm.

We have analyzed the allocation algorithms proposed through simulations and further validated the complete solution through its implementation in the ONOS SDN controller. The results show that the algorithms proposed are able to concentrate the traffic on few ports dynamically adapting to the variations in the demand, hence reducing energy consumption. Furthermore, these results confirmed that SDN can be used to reduce the energy consumption in bundles of EEE without the need of modifying the firmware of the devices.

However, the energy savings achieved here produce an increase in the delay of the packets, that may be unacceptable for critical traffic. The next chapter will explore this issue in more depth.
3 QoS-aware algorithms

3.1 Problem statement

In the previous chapter, we have tackled the problem of minimizing energy consumption in bundles of Energy-Efficient Ethernet links though Software-Defined Networking capabilities. The proposed application periodically retrieves the flow rules installed in the switches which transmit traffic to a bundle, forecasts the rate that will be demanded by each flow in the next interval and reallocates the flows in the bundle in order to reduce energy consumption. We have presented three allocation algorithms, all of them are able to reduce the energy consumption to near the optimum value by concentrating the traffic on a few ports. Among the proposed algorithms, the results revealed that the two greedy algorithms introduce non-negligible packet losses whereas the conservative one does not. Recall that the Conservative algorithm relies on computing the minimum number of needed ports and then balancing the flows among these ports. This way, the rest of the ports remain in LPI mode.

Nevertheless, the analysis of the delay that we have carried out in the previous chapter discloses that the delay of the packets can grow in excess, since none of the algorithms takes into account the latency requirements of the flows.

In this Chapter, we will introduce flows with different Quality of Service (QoS) requirements in terms of latency. We will propose modifications to the previous energy-efficient algorithms to handle traffic which demands a low-latency service, while at the same time keeping the substantial energy savings achieved.

3.2 QoS-aware energy-efficient algorithms

In this section we will introduce two modifications to the energy-efficient algorithms described in Chapter 2 in order to consider the demands of low-latency flows while maintaining the reduction in the energy consumption of the bundle. The specific mechanism
used to identify low latency flows is not relevant for this work since the method actually employed does not affect the algorithm at all. Thus, we will assume that the low-latency flows are tagged with a well-known DSCP, which is carried in the IP header. As a result we will work with two types of flows: low-latency flows and best-effort flows.

It is important to say that these modifications are generic enough to be applied to any of the different energy-efficient algorithms previously presented. However, the Conservative algorithm outperforms the other two algorithms in packet losses and delay with a minimum degradation in energy savings. Consequently, for the rest of this chapter, we will only use the Conservative as the energy-efficient algorithm, for the sake of simplicity.

### 3.2.1 Spare Port algorithm

The Spare Port algorithm exploits the fact that, when the traffic load is not too high, the energy-efficient algorithms concentrate the traffic on few ports, leaving the rest of the ports completely idle. Particularly, it accepts an increase of energy consumption in unused ports so as to provide expedited service to low-latency flows. The modified algorithm works in two-phases, first allocating best-effort flows and then the low-latency ones.

1. In the first phase, the Conservative algorithm is directly applied without modifications but just to the best-effort flows.

2. In the second phase, the remaining low-latency flows are assigned to the least occupied port among those of the bundle.

This algorithm can perform well under the assumptions that low-latency traffic represents a small fraction of the total traffic and that there is the best-effort flows can be allocated leaving some unused port. This way, the application of energy-efficient algorithm to the best-effort flows will concentrate the traffic on few ports leaving at least one completely empty. Then, in the next step, some of the unused ports will be used to forward low-latency traffic, without increasing the delay of the best-effort traffic. However, it is important to analyze the limitations of this algorithm when some of these assumptions does not hold:

1. When there is a high traffic demand such that all the ports of the bundle must handle best-effort flows, low-latency traffic will not be forwarded through an exclusive port. As a result, both low-latency and best-effort traffic will be treated in the same way, without meeting the needs of premium traffic.

2. When the amount of low-latency traffic is significant, the energy consumption of the spare port used can drastically increase because of the energy profile of an EEE
3.2. QoS-aware energy-efficient algorithms

3.2.2 Two Queues algorithm

As we have previously advised, the Spare Port algorithm can increase the energy consumption if the amount of delay-sensitive traffic increases. Even more important, the solution will not manage to satisfy the demands of flows with low-latency requirements when there is a high amount of best-effort traffic. As a result, in this solution we leverage on the ability of most SDN switches to have multiple queues attached to a physical port. These queues can be defined with different priorities. In fact, this is the standard way of providing QoS in SDN devices as stated in the OpenFlow (OF) specification [33]. Although this capability is not required, it is provided by most of the devices, such as OpenvSwitch [34], which is presumably the most widely used OF-enabled switch.

In the Two Queues Algorithm, we will define two queues with different priorities inside each physical port of the switches: the queue with the highest priority will be used for the low-latency traffic and the other one for the best-effort traffic. The algorithm operates in two phases, determining first the port and then the queue of each port:

- The first phase consists in directly applying the unmodified energy-efficient algorithm described in Chapter 2 to the whole set of flows, both including low-latency and best-effort, without any distinction between them. This way, the whole set of flows is allocated in few ports.

- The second phase sets the adequate queue inside the assigned port for every flow. Low-latency flows are assigned to the high-priority queue of the ports whilst best-effort flows are assigned to the low-priority queue.

This way, the allocation of the flows to the ports is actually given by the energy-efficient algorithm. Thanks to the introduction of multiple FIFO queues inside the port, we can prioritize flows with stringent QoS requirements in terms of latency, thus providing an expedited service. The decision of the next packet to be served by a port is straightforward: each time the port ends the transmission of a packet it will pick the next packet to be transmitted from the non-empty queue with the highest priority. Consequently, a packet in the high-priority queue can only have to wait for the transmission of that arrived to the port earlier and at most one packet from the low-priority queue (i.e., if the transmission of the best-effort packet was in progress when the low-latency one arrived at the port).

This algorithm does not show any of the limitations of the Spare Port algorithm. On the one hand, the latency demands of high-priority traffic can be guaranteed irrespective of the amount of best-effort traffic load. On the other hand, this algorithm does not increase link, which raises very quickly with the port occupation, as previously shown in Fig. 2.6.
energy consumption for providing expedited service, since the allocation of all the flows to the ports is computed with the energy-efficient algorithm. As a result, the energy consumption will be exactly the same as using the original energy-efficient algorithm proposed in Chapter 2.

The main drawback of this algorithm is that it increases the latency of best-effort traffic, since low-priority packets will not be served on a port as long as there are low-latency packets waiting to be transmitted on that port. This effect can be more noticeable as the fraction of the traffic which demands an expedited service is more significant. However, since the use of the priority queues only implies a reordering of the packets in a port, the average delay of all the packets will not change. Thus the maximum delay of best-effort packets is bounded. Indeed, this algorithm introduces a trade-off between the delay of low-latency packets and the delay of normal packets, without altering energy consumption.

With the aim of illustrating the operation of the algorithms, Fig. 3.1 shows an example of their operation in a bundle of 4 links. As explained, the Spare Port algorithm uses a
single queue allocating low-latency traffic in the emptiest port, whereas the Two Queues algorithm uses the same ports for all the traffic but allocating low-latency traffic in a high priority queue inside each port.

3.3 Experimental results

We have performed a set of experiments focused on evaluating the performance of the proposed algorithms and comparing the results with the energy-efficient Conservative algorithm developed in Chapter 2. First, we have conducted simulations and further validated the operation by implementing the algorithms in an ONOS application.

3.3.1 QoS-aware energy-efficient algorithms: simulation results

The goal of this experiment is to conduct simulations to analyze the performance of the QoS-aware energy-efficient algorithms described in this Chapter. We are mainly interested in the following metrics:

- Delay of the low-latency packets.
- Delay of the best-effort packets.
- Energy consumption.

To this end, we first extended the Java-based simulator developed for this thesis, which is available for download at [22], with capabilities to identify two types of packets, namely those with low-latency QoS requirements and the rest. Then, we added two priority queues to each port, which is a feature needed by the Two Queues algorithm. In this way, the queue with the lowest priority will be the default for all the traffic unless specified the opposite. Finally, we implemented the QoS-aware algorithms in the simulator. As previously detailed, the Two Queues algorithm is the only one of our algorithms which uses the high-priority queue, whereas the Empty Port algorithm uses the low-priority queue for all the traffic.

The simulations have been conducted using the same topology as in the previous Chapter as depicted in Fig. 2.8: two switches with SDN capabilities, interconnected by a bundle of five 10 GBASE-T IEEE 802.3az interfaces.

Again, we have used a real traffic trace retrieved from [25] for the best-effort traffic in our simulations. This traffic trace, which has an average rate around 3.25 Gbit/s, has been scaled by modifying the inter-arrival times to generate traces with the following rates: 6.5 Gbit/s, 13 Gbit/s, 19.5 Gbit/s, 26 Gbit/s and 32.5 Gbit/s. We have set the algorithms to work with a sampling period of 500 ms and buffer size limited to 10,000 packets as shown in Chapter 2 to provide acceptable results in terms of packet losses.
Figure 3.2: Average delay of the low-latency packets for the 32.5 Gbit/s best-effort traffic trace.

The traffic with low latency requirements has been synthetically generated. These traces are made of relatively small packets (e.g., 100 bytes) with constant inter-arrival times, as a rough approximation to multimedia traffic with real-time requirements. Varying the inter-arrival times we have generated different rates. The final traces actually fed to the simulator correspond to the merge of the best-effort real traces and the low-latency synthetic traces. Note that the packets of the synthetic traces have been tagged as low-latency traffic, so that the simulator is able to distinguish them from normal traffic.

Figure 3.2 shows the average delay of the packets with low-latency requirements using the QoS-aware algorithms and also the delay of these packets using the baseline Conservative algorithm. The results in the figure correspond to the normal traffic trace of 32.5 Gbit/s while we vary the rate of the low-latency traffic. As we can see, the baseline Conservative algorithm yields considerably worst results than the QoS-aware algorithms, introducing a delay of more than 100 μs. Besides, the results show a high variability in the delay depending on the traffic itself, because the Conservative algorithm does not difference among the types of traffic and consequently it handles these packets in the same way as the rest of the traffic. For example, in this case, the fluctuations for the different rates of the low-latency traffic come from the fact that the low-latency flows will be allocated to a different port in each case, being forced to compete with a different amount of normal traffic.

We have omitted the results using lower rates for the normal traffic for the sake of brevity, since the results are analogous.
3.3. Experimental results

The QoS-aware algorithms achieve a significantly lower average delay which is two orders of magnitude below the baseline, as desired. The Spare Port algorithm shows around 5 µs and the Two Queues algorithm attains less than 2 µs, for any of the rates described. In the Spare Port algorithm, the main contribution of the delay is given by the time to wake up the interface ($T_W = 4.48$ µs), which will be most of the time idle when a low-latency packet arrives at the port. This is not the case in the Two Queues algorithm, where low-latency traffic shares the port with best-effort traffic. As a result, many times a high-priority packet arrives, the port will be active sending normal traffic and it will only have to wait for the current packet transmission to end. This takes less time than a transition from idle to active state. For example, transmitting a 1500 bytes long frame takes 1.2 µs which is less than waking up an interface, which takes $T_W = 4.48$ µs.

In addition to the above traces, we will explore the average delay of the low-latency packets when the system has to handle an extreme high load of best-effort traffic so that all ports need to be used for normal traffic. The results of this experiment, for a 45.5 Gbit/s normal traffic trace are shown in Fig. 3.3. As we can see in the figure, the Spare Port and the Conservative algorithm experiment an average delay higher than 200 µs fluctuating up to 1000 µs, depending on the actual packets. On the other hand, the Two Queues algorithm achieves a latency lower than 2 µs. These results verify our hypothesis that the Spare Port algorithm is not capable of providing a low latency service in high load scenarios whereas the Two Queues algorithm still provides the same level of delay despite the extreme load of best-effort traffic. In particular, since all the ports are
Chapter 3. QoS-aware algorithms

Figure 3.4: Average delay of the normal packets for the 32.5 Gbit/s best-effort traffic trace.

being used to forward normal traffic and thus low-latency packets will be allocated in the same port and queue than normal traffic.

Figure 3.4 compares the average delay of the best-effort packets using the QoS-aware algorithms with the average delay of these packets using the baseline Conservative algorithm, for the 32.5 Gbit/s normal traffic trace, varying the rate of the low-latency traffic. As we can observe, when the amount of low-latency packets is very low (e.g., lower than 100 Mbit/s in this case), the average delay of normal packets is identical in the two algorithms and the baseline, being around 230 µs. Nevertheless, when this amount increases the delay exhibited by the baseline raises and also does so the delay of the Two Queues, being the latter slightly higher. On the other hand, the delay of the Spare Port algorithm is not affected by the rate of the low-latency traffic, since it is being forwarded through a different port than the normal traffic. These results confirm our hypothesis stated in the description of the algorithms.

Figure 3.5 shows the average energy consumption of the bundle using the different QoS-aware algorithms and also the baseline Conservative algorithm, for the normal traffic trace of 32.5 Gbit/s, varying the rate of the low-latency traffic. Again, when the amount of high-priority traffic is negligible (e.g., lower than 10 Mbit/s), the three algorithms attain the same energy consumption. As expected, the Two Queues algorithm achieves exactly the same consumption as the baseline Conservative algorithm irrespective of the rate of the time-sensitive traffic. However, for values higher than 10 Mbit/s we see that
the energy usage rapidly increases in the Spare Port algorithm, being nearly a 100% for rates above 100 Mbit/s. This confirms that energy consumption can raise quickly in the Spare Port algorithm as soon as the amount of high-priority traffic is not negligible. Yet, it is important to recall that in this case the low-latency traffic consist in packets which are separated by a constant amount of time, with the consequent implication that the interface devoted to handle this traffic will be constantly switching from active to LPI states. Therefore, drastically increasing the energy consumption of this port.

To sum up, these results validate our hypothesis showing that both proposed algorithms are able to offer a low-latency service to traffic with that kind of QoS requirements as long as the system is not congested. Under the situation of an extreme load of traffic without QoS requirements, the Two Queues algorithm is the only capable of providing expedited service. When the amount of low-latency traffic is negligible, there is not impact in the overall performance. Nevertheless, when the rate of this type of traffic is significant the Spare Port algorithm drastically increases the energy consumption whereas the Two Queues mildly increases the delay of normal traffic.

3.3.2 ONOS implementation

We have further validated the proposed algorithms which gives support to delay-sensitive traffic, implementing them in a real application on top of ONOS. We have used a network composed of OvS switches emulated with Mininet which are controlled by the ONOS
Instead of just using the same topology as in the simulations and the previous experiment, we decided to use a more complex one, involving more hosts and another switch. In particular, our experimental setup consisted of three switches (numbered from 1 to 3) and eight hosts (numbered from 1 to 8). Hosts 1 to 4 are connected to switch 1 and hosts 5 to 8 are connected to switch 3. These switches will be referred to as edge switches, since they are directly connected to end hosts, while switch 2, the inner switch, is connected to both edge switches through bundles made of 4 links. All the interfaces in this scenario have a nominal capacity of 1 Gbit/s. Since there are no direct links connecting the edge switches, the traffic originated in from hosts 1 to 4 with destination in hosts 5 to 8, and vice versa, have to go across the three switches, hence traversing the two bundles. Fig. 3.6 shows a screenshot of the ONOS web interface with this topology.

Using this topology we have assessed the correct operation of both QoS-aware algorithms with a simple demonstration scenario: three UDP elephant flows without latency requirements are originated in hosts 1, 2 and 3, with respective destinations in hosts 5, 6 and 7. We have created these flows using the iperf3 tool, acting hosts 5 to 7 as iperf3 servers whereas hosts 1 to 3 behave as clients. The first two clients send traffic at 700 Mbit/s and the third one at 600 Mbit/s. In this way, the flows will be allocated on the first three ports of each bundle, respectively. Then, we added three lightweight flows generated from host 4 with destination in host 8. The purpose of these mice flows is to measure the latency suffered by the packets, using the different algorithms. To this end, we have generated the flows using the ping tool, properly setting the DSCP field of the packets so that two flows are identified by the SDN application as having time-sensitive QoS requirements and the other not. Among the traffic with low-latency QoS requirements the application uses the Spare Port or the Two Queues algorithm depending on the value of the DSCP field. The execution of the ping command will provide us the measure of the round-trip time (RTT) of the packets.

Fig. 3.7 shows box and whisker plots with the RTT of 10000 packets of the lightweight flows using the different algorithms. The whiskers show the 99% of the samples and outliers have been removed for the sake of clarity. We can see that traffic without real-time
3.4. Conclusions

The energy saving algorithms presented in the Chapter 2 achieve substantial reduction in the energy consumption but introduce a delay in the packets that can be unacceptable for time-sensitive applications. In this Chapter we have considered two types of traffic with different QoS requirements in terms of latency. We have proposed two alternative modifications of those algorithms that can provide low delay to traffic with stringent latency requirements, while maintaining minimum energy consumption. One of the alternatives maintains unchanged the delay of normal traffic at the cost of slightly increasing the energy usage. On the contrary, the other one accepts a small increase in the delay of best-effort traffic to maintain energy consumption minimized. Unfortunately,
in the presence of extremely high loads, the former solution is not capable of providing the low-latency service.

We have analyzed the algorithms through simulations verifying its expected performance. Furthermore, we have implemented the proposed alternatives as an ONOS application and validated its desired operation in an emulated environment using Mininet.

Overall, these results verify that it is possible to give support to time-sensitive applications while at the same time minimizing energy consumption in bundles of Energy-Efficient Ethernet links, leveraging the flexibility and programmability introduced by SDN.
4 Conclusions and future work

Throughout the realization of this master’s thesis we have fully achieved the objectives initially set. Besides, this research work implied not only the design and development from the scratch of the network simulator, but also the installation and configuration of the ONOS framework which helped me in the understanding of the SDN paradigm. Although the simulator has been developed in Java and the ONOS applications are also Java-based, we have also employed scripting languages such as Python and Bash for post-processing the results and test automation, respectively. Besides, we have used C to develop low-level applications to parse the traffic traces and convert them to a simpler format which eases the manipulation. The representation of the data has been carried out using Gnuplot.

In the next sections we conclude this dissertation by summarizing the contributions of the present work and also the lines that we have identified to extend it.

4.1 Contributions

The main focus of this thesis has been the minimization of the energy consumption in networking equipment with SDN capabilities when the traffic traverses an aggregate of links between the two switches. We have first analyzed the previous work to understand the theoretical fundamentals and explores its limitations regarding its implementation in SDN. Then, we elaborated a solution in form of SDN application that efficiently concentrates the traffic flows in few ports of the bundle, dynamically adapting to the variations in the traffic demand. We proposed three allocation algorithms which we later analyzed in terms of energy consumption, delay of the packets and losses introduced. We validated the algorithms using real traffic traces through simulations and also in a real implementation on top of the ONOS SDN controller. The results verified the expected operation of the algorithms showing that the SDN capabilities of networking equipment can be used to reduce energy consumption in bundles of EEE links up to a 50%.
Chapter 4. Conclusions and future work

the need of modifying the firmware of the devices.

However, the previous solution introduces an increase in the delay as trade-off to reduce energy consumption. Therefore, we proposed two modifications to the previous algorithms to offer a low-latency service to traffic with stringent QoS requirements, while keeping energy consumption reduced. Then, we conducted simulations and further implemented the proposals as ONOS applications evaluated in a network emulated with Mininet. The results showed that the algorithms are able to provide a low-delay service to time-sensitive traffic achieving a reduction of some orders of magnitude. Even under the situation of normal traffic congestion, one of the proposals manages to continue offering an accelerated service.

In addition, the simulator we have designed and developed throughout this master’s thesis has been released under the GNU General Public License, version 3 (GPL-3.0) and it is available for download at [22]. It is therefore an open source platform for research on algorithms for aggregates of links in SDN applications, providing a rich set of measurements such as binary rate transmitted through each port, energy consumption, packet delay, losses, error in the rate estimated, time elapsed in computing the allocation, etc. Accordingly, it is a contribution to the SDN community which is not restricted to research on energy consumption optimization but can be used to analyze any type of algorithm based on periodic reallocation of the flows.

Finally, most of the work carried out in this master’s thesis has been directly contributed to the research community. On the one hand, the energy saving algorithms described in Chapter 2 has been presented in The 5th International Conference on Software Defined Systems (SDS). On the other hand, the QoS-aware algorithms introduced in Chapter 3 have been submitted to The 9th Symposium on Green Networking and Computing (SGNC) which will be held in the frame of the 26th International Conference on Software, Telecommunications, and Computer Networks (SoftCOM). The former has already been published in [23] whereas the latter is still under peer review.

4.2 Future work

We have identified some lines in which the present work could be extended. First, the centralized view of the whole topology that the SDN controller provides to the applications can be harnessed so that inner switches can reuse the flow allocations performed in the edge switches, in the case where multiple link bundles are present in the network. Consequently, reducing the computations that the application needs to perform.

Secondly, it will also be interesting to test our solutions in a testbed composed of hardware OF-enabled devices with IEEE 802.3az ports controlled by our ONOS application. This way, it will be possible to empirically measure to what extent our algorithms are capable
of reducing the energy consumption in a real testbed.

Finally, we see a research opportunity in reducing the control plane traffic required, since our work has been focused just on data plane traffic. Although we have given some clues about control plane traffic when studying the adequate values for the sampling period, a thorough analysis is still open for research. We envision that minimizing control plane traffic can be a perfect complement for this work, contributing to the overall energy reduction. For example, the proposed allocation algorithms can be modified to take into account the current allocation of the flows and then attempt to minimize the number of flow re-allocations performed. This would not only reduce the amount of control traffic but also increase the stability of the allocations and avoid unwanted effects on TCP performance.
Bibliography


Bibliography


