

Implementing Energy Saving Algorithms for Ethernet Link Aggregates with ONOS

Pablo Fondo-Ferreiro, Miguel Rodríguez-Pérez, Manuel Fernández-Veiga

atlanTTic Research Center

University of Vigo

36310 Vigo, Spain

Tel.:+34 986 813459; fax:+34 986 812116; email: pfondo@det.uvigo.es

Abstract—During the last few years, there has been plenty of research for reducing energy consumption in telecommunication infrastructure. However, many of the proposals remain unimplemented due to the lack of flexibility in legacy networks. In this paper we demonstrate how the software defined networking (SDN) capabilities of current networking equipment can be used to implement some of these energy saving algorithms. In particular, we developed an ONOS application to realize an energy-aware traffic scheduler to a bundle link made up of Energy Efficient Ethernet (EEE) links between two SDN switches. We show how our application is able to dynamically adapt to the traffic characteristics and save energy by concentrating the traffic on as few ports as possible. This way, unused ports remain in Low Power Idle (LPI) state most of the time, saving energy.

Index Terms—SDN, ONOS, IEEE 802.3az, Energy Efficiency

I. INTRODUCTION

Two recent trends have been shaking the networking landscape during the last few years. On the one hand, the seemingly unstoppable deployment of SDN equipment and solutions in datacenters, has provided network operators with unprecedented flexibility. On the other hand, growing concerns about both environmental and monetary costs of ICT infrastructure has led to the proposal of a plethora of solutions to augment the energy efficiency of networking equipment.

However, a great deal of interesting proposals for reducing energy usage remain unimplemented because they either require significant changes to fundamental networking protocols or to the actual networking equipment, cf. [1]–[4]. We believe that software defined networking (SDN) can be used to overcome these limitations. In particular, having a comprehensive view of the network status and the ability to precisely control individual flow forwarding, can be used to define energy-optimum paths for traversing flows, either through the whole network domain [1]–[3] or when crossing a bundle link between two directly connected switches [4].

Open Network Operating System (ONOS) [5] is an open source project, designed for high availability, performance and scalability. ONOS abstracts the particular details of the actual SDN forwarding devices in the network, permitting the development of technology-agnostic network applications. These applications can obtain a global view of the network and they can also modify the actual flow tables in real time.

In this paper, we describe an ONOS application for the optimal distribution of network traffic among a bundle of energy efficient Ethernet (EEE) [6] links managed by a SDN network so that energy savings are maximized. Our application dynamically identifies the network flows traversing the bundle and adjusts the forwarding tables to achieve the energy optimal share of traffic.

Although there exists an optimal way to share the traffic as a whole among the links [4], that is the theoretical base of our implementation, the actual assignment of individual flows to the EEE links remains an open problem. Thus, in this paper we have compared three different flow scheduling algorithms. All three algorithms produce a similar share of traffic among the links, so they achieve energy saving results very near those predicted by the model. However, as the particular flows assigned to the links are different, their characteristics regarding traffic delay and loss rate differ.

The rest of this paper is structured as follows: Section II presents the related work. Then, we proceed with the problem statement in Section III. Section IV describes our SDN implementation, while Section V describes some alternatives for the implementation. Section VI summarizes the results obtained and finally Section VII exposes the conclusions.

II. RELATED WORK

The usage of SDN networks to diminish energy usage in computer networks has already been explored by several authors.

A survey on energy efficiency in SDNs is presented in [7]. This survey analyzes the different components of the SDN structure which can be dynamically configured to reduce power consumption. The approaches analyzed include reorganizing the flows in the network to have a small number of active devices in the network so that the unused devices can be put into sleep mode. When there is a low traffic load, they also mention the possibility of putting certain ports rather than whole devices into sleep mode. Those kind of approaches which set devices in a low-power mode based on the current load of the system, are usually referred to as traffic-aware.

GreenSDN, a SDN emulation environment based on Mininet and the python based POX SDN controller, has been proposed in [8], where they report on the difficulties they faced building a SDN environment with capabilities of emulating the energy

saving protocols operating at different levels of the network. They propose a mechanism which operates at the node level, exploiting the Low Power Idle (LPI) mode defined by IEEE 802.3az that is especially relevant to this paper. However, they only consider turning on and off whole switches and not individual interfaces when the traffic load is behind a bundle.

Another contributions in the literature about energy-efficiency leveraging the power of OpenFlow include ElasticTree [9] and ECODANE [10] which are data-center based proposals. Both proposals traffic-aware mechanisms consistent in dynamically turning links and devices on and off based on the current traffic load of the system.

The authors in [4] have shown that, under suitable conditions, the traffic load allocation that minimizes the energy consumption in a bundle of EEE links is achieved using a water filling algorithm. However, the proposed algorithm cannot be directly ported to SDN. First, the proposed water filling algorithm operates at the packet level, that is, when a switch receives a new packet that must be sent through the bundle, it has to decide the port used to transmit the packet based on the current backlog of the port. However, SDN operates at the flow level, therefore, all packets belonging to the same flow will be forwarded via the same port. Secondly, the algorithm needs to obtain the queue occupation of each port to classify incoming packets, but unfortunately SDN does not provide access to such information, to the best of our knowledge.

III. PROBLEM STATEMENT

This section describes the algorithm that minimizes the energy consumption in a bundle of EEE links. We will briefly summarize the results from [4] describing the optimum traffic allocation in bundled EEE links. The authors there demonstrate that, for certain common class of functions that characterize the energy-consumption profile of the links, the solution to the optimum allocation for a given offered traffic load is a simple sequential water-filling algorithm: each link capacity is fully used before sending traffic through a new otherwise idle link.

Luckily, the energy-consumption profile of both major modes used to govern the use of the LPI mode (namely the *frame transmission* and *packet coalescing* algorithms) belong to this class of functions. In summary, to achieve the optimum energy savings links will be in the following states: some links used to its full capacity, some links completely idle and at most one link transmitting packets at less than its full capacity.

Nevertheless, the optimum allocation in terms of maximum energy savings can easily lead to packet delays growing uncontrolled, if proper care is not taken. This issue has also been carefully analyzed in [4], where the authors propose modifications of the simple water-filling algorithm to control the average delay of the packets with a bounded cost in the energy savings.

IV. SDN BASED IMPLEMENTATION

We now proceed to describe the implementation of the algorithm using the facilities provided by SDN switches.

The first challenge resides in the fact that while the theoretical solution described in [4] assumes a packet level operation our implementation will have to operate at the flow level.

A. Flow Selection

Recall that SDN works at the flow level (e.g., OpenFlow [11] switches are composed of flow tables), where a flow is defined by a set of fields of the incoming packets. Therefore, the first step is to define which fields will identify our flows. SDN allows us to define very different levels of flow granularity: a very coarse level of granularity will only match on the destination MAC address or the physical input port of the packets, for example. However, this kind of definition of the flows does not seem to be suitable for our purpose because if deployed in a transit network, flows will share a common small set of exiting routers, thus limiting the variability of MAC addresses.

Since such a coarse granularity does not seem to be suitable for our algorithm, we have to identify a finer level of granularity that may allow us to split the traffic among the links. Moreover, we strive to aggregate non correlated flows, so their aggregated behavior is more stable. We have explored several levels of granularity to identify different flows among the packets so that we can send some flows to one port and another flows to other port, and as a result we will be able to utilize the full capacity of the bundle. We have explored two alternatives to achieve this: *flow tagging* and *field matching*.

First, we have considered the possibility of having a process at the input of our SDN network which smartly tags the packets with a flow label, imposing the tag assigned to each packet in a field directly matchable by ONOS (e.g., the DSCP field in the IP header). This way we could have the packets evenly distributed into a fixed number of flows, which would ease the allocation of the flows to the bundle. The main drawback of this approach is that it needs a dedicated tagger at input nodes in the network, so we have also tried to select flows in a distributed manner.

We would like the aggregated flows to be later assigned to a given port to show a predictable, ideally constant, demand. To this end, we try to aggregate independent end-to-end network layer flows, thus defined by the source and destination IP addresses pair of each packet. Note that, at layer 2 we could have an insufficient number of identifiable flows (e.g., in a transit network). On the contrary, the number of transport layer flows can be excessively high. Consequently, the aggregation of layer 3 flows is expected to result in a low variance in the rate of the aggregated flows, hence being the rate of these flows more predictable.

However, a direct mapping between a pair of source and destination IP addresses to a flow will produce 2^{64} different flows, which is clearly an unacceptable huge number of flows and obviously not scalable. Even considering only the destination address will produce 2^{32} different flows, which is also unacceptably large. Thus we have chosen use only some bits of the destination IP address to identify subflows inside the packets destined to the same MAC address.

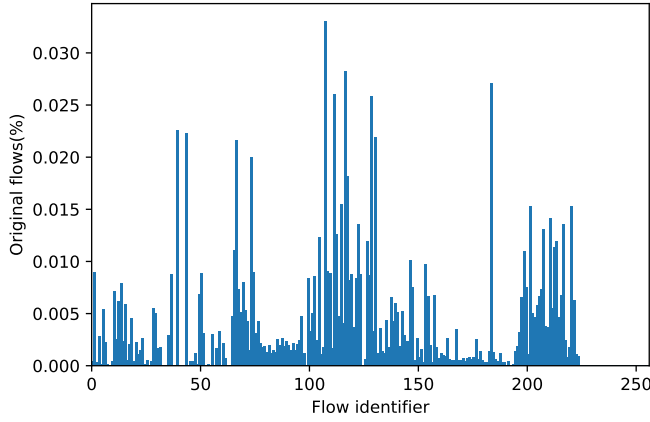


Figure 1. Flow distribution for the 8 first bits of the destination address.

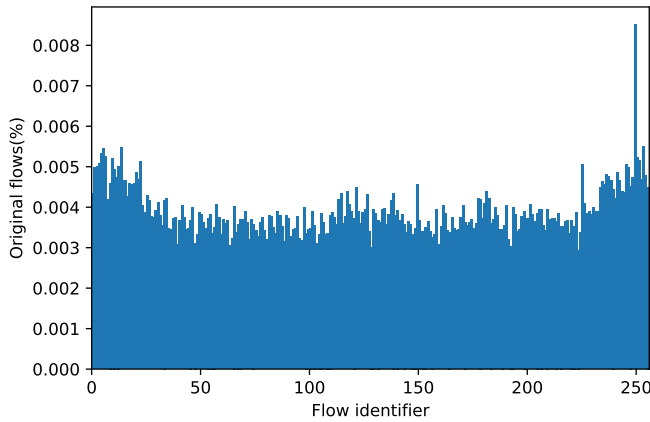


Figure 2. Flow distribution for the 8 last bits of the destination address.

Since we want to have a flatter distribution of the original flows (pair of source and destination IP addresses) to the aggregated flows, we have explored the usage of arbitrary bitmasks over IP addresses and analyzed how the flows were distributed.¹ Specifically, we have counted how many of the original layer 3 end-to-end flows of our trace will be contained in each aggregated flow. The flatter results were obtained for the last bits of the destination IP address, as shown in the histograms of Figs. 1 and 2 (other combinations have been tested and show similar results, thus they are omitted for the sake of brevity).

However, although the current version of OpenFlow allows arbitrarily bitmasking IP addresses, the ONOS controller does not, and restricts us to mask the first bits of the IP address. Although using the largest number of bits yields to lower variance in the distribution of the original end-to-end flows as shown in Fig. 3 and Tab. I, it also implies a too large number of aggregated flows being handled by the switch. Consequently, we will use just the 8 first bits of the destination IP address to

Table I
VARIANCE IN THE NUMBER OF ORIGINAL FLOWS PER AGGREGATED USING THE LAST BITS OF THE DESTINATION IP AS FLOW IDENTIFIER.

Number of bits	Variance
4	7985223
6	954055
8	210447
10	26446
12	4207

define subflows inside the packets destined to a given MAC address, yielding a maximum of 256 flows. We see this value as a good trade-off between a small number of flows to be manageable by the switches and a minimum value of granularity to be able to spread the traffic among the links.

B. ONOS Application

Once we have clearly specified how the flows are defined, we proceed to describe the SDN application that we developed. This application has been implemented using ONOS due to it being one of the most supported open source network operating systems. Besides, thanks to the usage of ONOS, our algorithm is agnostic of the SDN forwarding devices and can be directly deployed on any SDN network.

We would like to recall that the application does not work at the packet level but at the flow level. Consequently, when a packet is received in a switch it will first look up the installed flow rules and it will execute its associated functions whenever a match is found: i.e., the specific forwarding port will be selected. Only in the case where a packet does not yet match any rule among those installed (because it does not belong to an active flow) it will be sent to the controller. The controller will then be responsible for installing the corresponding flow rule in the switch. This behavior is the classical *reactive* forwarding application. Indeed, in our application switches are initialized without flow rules and the first packet of each flow is sent to the controller, which knows how to forward that packet and instructs the switch to install a flow rule for that flow, so that the following packets of that flow are forwarded directly by the switch at line rate, without being sent to the controller.

When a new packet which is to be forwarded by a bundle of EEE links is received at the controller, the application selects a random port of the bundle. This is done since the controller lacks any previous information about the rate of this flow.²

To overcome the limitation of not being able to individually send each packet to the adequate port as we could do with a water-filling algorithm implemented at the packet level, our application queries periodically the flow rules installed in the switches and reorganizes them attempting to minimize the energy consumption. The flow rules that send traffic to a bundle are carefully analyzed by our algorithm. First, the application attempts to estimate the load that each flow will transmit in the

¹The traffic traces analyzed come from the publicly available passive monitoring CAIDA dataset [12].

²Note that if a packet is received at the controller necessarily this is the first packet of a new flow, hence the controller does not have previous information about the traffic load of this flow.

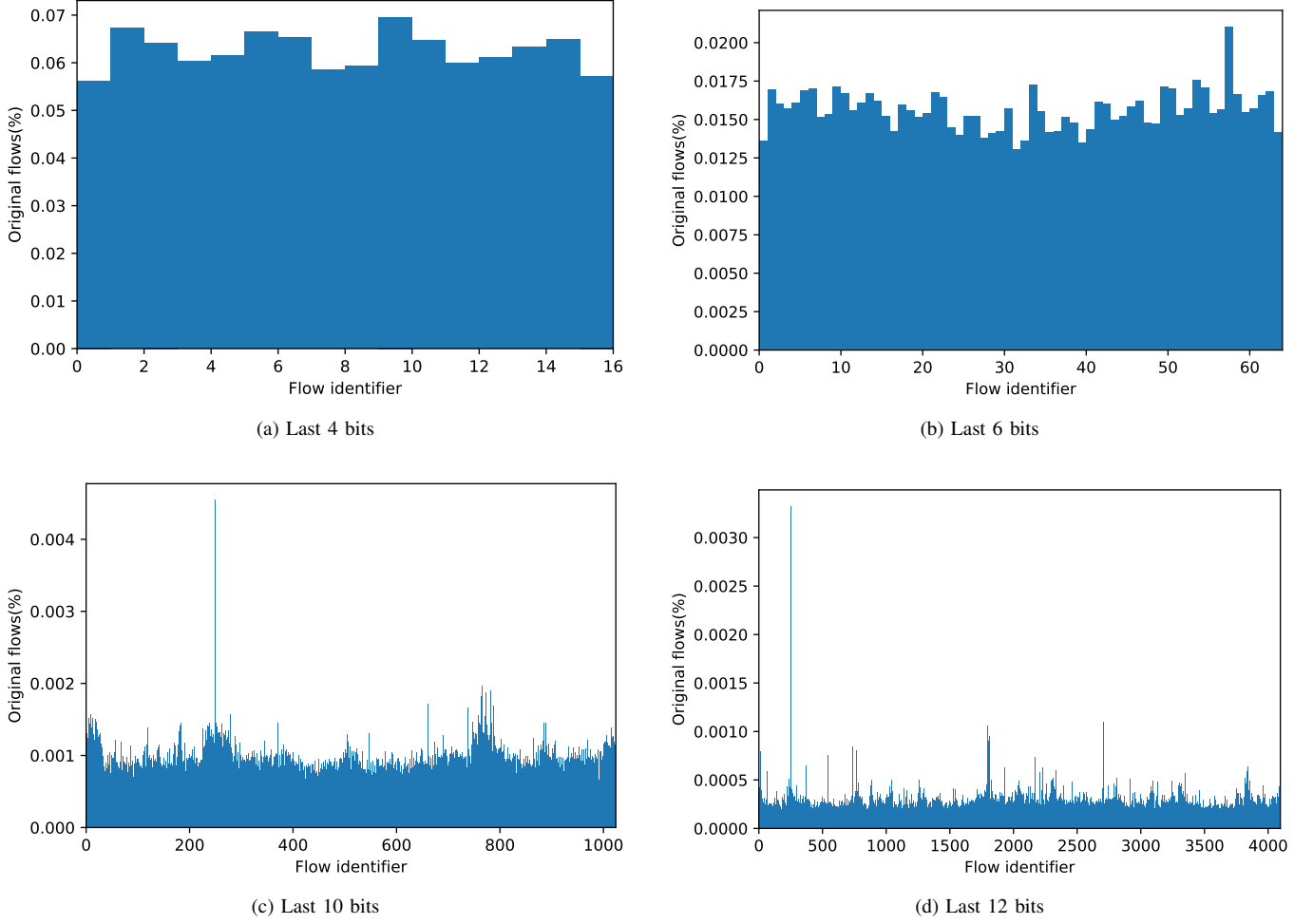


Figure 3. Flow distribution for the last bits of the destination address.

next interval. Clearly, without any other source of information, this prediction must be performed using the information of the bytes transmitted by this flow in the previous intervals. This information is provided by the counters that the switch stores along with each flow rule.³ If our application stores the number of bytes that each flow has transmitted up to that point, in the next sampling interval we can calculate the number of bytes transmitted in that interval as the difference between the total number of bytes at this point and the previous stored value for that flow. This value of bytes will be our measure of the traffic of each flow in that interval. A scaling factor is used for newborn flows (i.e., flows that were not present in the previous interval) considering the fraction of the interval that each flow has been active.

Using this information, the algorithm decides which flows will be assigned to each port, attempting to minimize the overall energy consumption of each switch, hence minimizing the energy consumption of the whole network. Finally, this modifications are instructed to the switch, which updates the

³Actually, SDN devices store counters with the number of bytes that have matched with each flow.

flow rules in accordance.

Accordingly, the main tasks of the algorithm are two-fold: first, estimate the load that each flow will request and secondly, compute an energy-efficient assignation of the flows to the ports. We will now describe the algorithms that we have implemented.

V. ASSIGNMENT ALGORITHMS

Once the application has selected the flows, the next task is to decide the best criteria to assign the identified flows to the set of ports belonging to the bundle.

In every interval, this algorithm estimates the traffic that each flow will transmit in the next interval strictly based on the bytes that have been transmitted by this flow in the previous interval.

A. Greedy Algorithm

A straightforward way to assign the flows consists on assigning them in order of decreasing demand, using a new port if the flow does not fit in any of the already used flows. The main advantage of such a simple approach is that it draws few computation resources at the controller. This algorithm is

```

1 allocate_greedy(flows, ports, bound=0) {
2   // Hold assigned port for each flow
3   flow_allocation[1..|flows|] = ∅
4
5   /* Sort flows by decreasing load value */
6   ordered_flows = sort(load(flows), DECREASING)
7
8   // Initialize occupation of the ports to 0
9   port_load[1..|ports|] = 0
10  port_flows[1..|ports|] = 0
11
12  for flow ∈ ordered_flows {
13    for port ∈ ports {
14      if ((port_flows[port] == 0) ||
15          (port_load[port] + load(flows)[flow]
16           ≤ 1 - bound/port_flows[port])) {
17        // Update port with the load of this flow
18        port_load[port] += load(flows)[flow]
19        port_flows[port] += 1
20        flow_allocation[flow] = port
21        break
22      }
23    }
24  }
25
26  return flow_allocation

```

Figure 4. Pseudocode for the *Greedy* Algorithms.

reminiscent of the classical water-filling approach but the unit of filling is the flow rather than the packet.

In detail, the flows that forward packets to the bundle are sorted in a decreasing order based on this estimation of the traffic that will be transmitted. Then, these flows are sequentially allocated to the ports maximizing the port occupation: if a flow can be allocated on the port with the highest occupation (i.e., if the sum of the estimated load of that flow plus the estimated load of the flows already assigned to that port is less than the capacity of the port) the flow is assigned to that port; otherwise, the next ports are analogously evaluated until a port where this flow can be allocated is found. The pseudo-code for the algorithm is shown in Fig. 4.

This algorithm is expected to have a good behavior in terms of energy consumption. Nevertheless, the algorithm does not perform any kind of control over the amount of packets that need to be queued on each port, requiring very great buffers (consequently introducing a considerable delay) in order to have a low percentage of packet losses.

B. Bounded-Greedy Algorithm

This algorithm is a variation of the previous one with the goal of reducing the size of the buffer needed on each port for a given packet loss ratio. The basic operation of the algorithm is the same as in the previous, but instead of filling the ports to their maximum capacity, we have constrained the maximum traffic load on any link to a function of the number of flows already allocated to it.

This way, we allow to reach higher aggregated loads on ports with a high number of allocated flows, as the aggregated variance of their demand is, in general, lower than that of ports with only a few flows. The pseudo-code is also shown in Fig. 4,

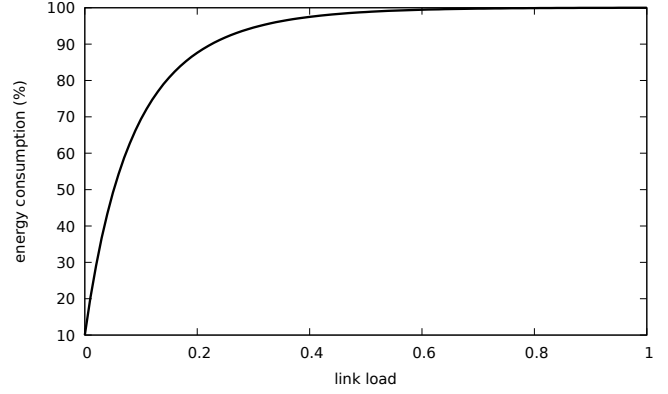


Figure 5. Individual consumption of a 10Gbit/s IEEE 802.3az interface.

where *bound* is the maximum amount of reserved space that must be left in a port with just one flow allocated to it.

C. Conservative Algorithm

Unfortunately, the previous algorithms cannot obtain acceptable results in terms of packet loss for a given buffer size. We have implemented another energy-efficient algorithm which, as an added benefit, minimizes the length of the transmission queues.

This algorithm, computes the total estimated traffic load that will be transmitted through the bundle in the next interval. This value puts a lower bound on the number of active ports for the next interval. Then, flows are spread evenly among all the active links. This clearly minimizes individual link occupation, but does not follow the water-filling algorithm. However, this does not matter much, as will be shown later in the results section. For a port governed by the *frame transmission* algorithm, the energy usage raises very rapidly with traffic load, see Fig. 5. So, it does not matter that much the load transmitted by each link once it is higher than about 20 % of its nominal capacity.

To further avoid packet losses, we do not directly use the estimated load to calculate the number of used ports, but we first add a safety margin load 20 % to avoid cases where all ports would be used too close to their nominal capacity.

Once we have calculated the number of ports of the bundle that we will be using, we proceed to minimize the occupation of each port in order to obtain an homogeneous occupation of all the used ports: not only similar rate but also similar number of flows. This is easily achieved sorting the flows in a decreasing order based on the rate estimation and then sequentially assigning each flow to the port with the lowest occupation among the ones that will be used for this interval. The algorithm is shown in detail in Fig. 6.

VI. EXPERIMENTAL RESULTS

The validation of the previous algorithms has been carried out on a scenario composed of two switches interconnected by a 5-link bundle of 10GBASE-T interfaces. Fig. 7 shows a snapshot of the ONOS Web GUI with the analyzed scenario, which has been deployed with Mininet [13].

```

1  safety_margin = 20%
2
3  allocate_conservative(flows, ports) {
4    // Hold assigned port for each flow
5    flow_allocation[1..|flows|] = ∅
6
7    expected_load = sum(load(flows)) + safety_margin
8    minimum_ports = ceil(expected_load)
9
10   // Only use the minimum number of ports
11   used_ports = ports[1..minimum_ports]
12
13   /* Sort flows by decreasing load value */
14   ordered_flows = sort(load(flows), DECREASING)
15
16   // Initialize occupation of the ports to 0
17   port_occupation[1..|used_ports|] = 0
18
19   for flow ∈ ordered_flows {
20     port = get_port_min_occupation(port_occupation)
21     // Update port with the load of this flow
22     port_occupation[port] += load(flows)[flow]
23     flow_allocation[flow] = port
24   }
25
26   return flow_allocation

```

Figure 6. Pseudocode for the *Conservative Algorithm*.

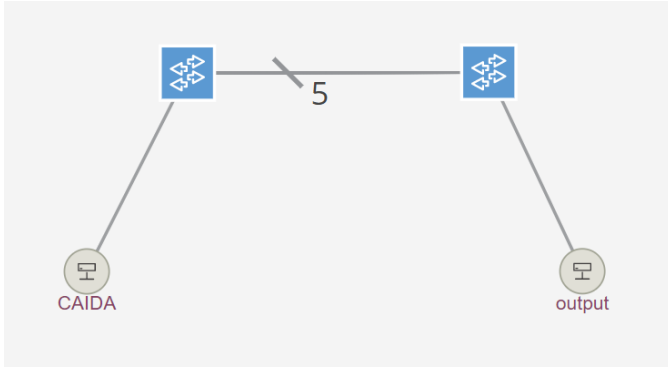


Figure 7. ONOS Web GUI with the setup of the experiment.

We have employed real traffic traces retrieved from the publicly available passive monitoring CAIDA dataset [12], feeding the data to the first switch so that it had to traverse the link bundle as shown in Fig. 7. The trace we have chosen has an average demand of about 3 Gbit/s, which is relatively low for our bundle of 50 Gbit/s,⁴ so we increased the rate tenfold to about 30 Gbit/s by reducing the inter-arrival times by a constant factor of ten.

To obtain the energy consumption results, we have proceeded in two complementary ways. The first one consisted on analytically calculating the *expected* consumption as the average expected consumption of its constituting ports. Then, the individual consumption of each port is calculated as the time average of its instantaneous consumption. As our algorithms already divide the time in constant intervals, we average

⁴The available traces provided by CAIDA are captured on a 10 Gbit/s interface.

over the consumption in each interval. Finally, the energy consumption on each interval can be calculated with several well tested models already known in the literature [14], [15]. In particular, we have employed (1), from the model presented in [14].

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

where $\sigma(\cdot)$ is the normalized energy usage, ρ_i is the normalized traffic load on link i . We have set $\sigma_{\text{off}} = 0.1$ according to several estimates provided by different manufacturers, and $T_S = 2.28 \mu\text{s}$ and $T_W = 4.48 \mu\text{s}$ as per the standard [16]. Besides, assuming frame transmission mode is used in the IEEE 802.3az interfaces, for Poisson arrivals, we have

$$E[T_{\text{off}}(\rho)] = \frac{e^{-\mu\rho T_S}}{\mu\rho} \quad (2)$$

where μ^{-1} is the average packet transmission duration.

We have further verified the results with a IEEE 802.3az simulator, available for download at [17]. To this end we have fed the exact same traffic that we sent via each port to five instances of the simulator, to then average the results, obtaining the global consumption. This later result is the one used in the following figures, as it does not depend on the veracity of the mathematical models. In any case, the differences were very minor, further confirming the validity of the models.

Using the above formulas, the theoretical lower bound for the energy consumption is 78.5 % when considering a packet size of 1500 bytes. This is achieved when the traffic in the trace, which has an average rate of 32.5 Gbit/s, is split in the bundle as follows: 3 ports with 10 Gbit/s, one with 2.5 Gbit/s and the remaining one completely idle, which yield 3 ports consuming the 100 %, one consuming 83.25 % and another consuming 10 %, respectively. We obtain the global consumption of 78.5 % as the average of these five values.

A. Experimental Setup

Although there exist several simulators such as ns-2 network simulator, we have decided to implement our custom network simulator in Java, available for download at [18], so that we can share most of the relevant code with the ONOS application.

We are interested mainly in two performance metrics. Firstly, the overall normalized energy consumption is the main metric that we have used to validate our algorithms. On the other hand, we have also measured the packet losses induced by our algorithm for a given buffer size (in number of packets), i.e., when a buffer in a port is full of packets, new packets forwarded to that port are discarded. Moreover, since our algorithm takes effect after the first interval (during the first interval flows are allocated randomly since we do not have any *a priori* information about the flows) we have decided not to consider the values of consumption and packet losses of this first interval, which can be considered as a brief transient state.

To use as a baseline of performance to compare the results of our algorithms with, we have implemented an *equitable*

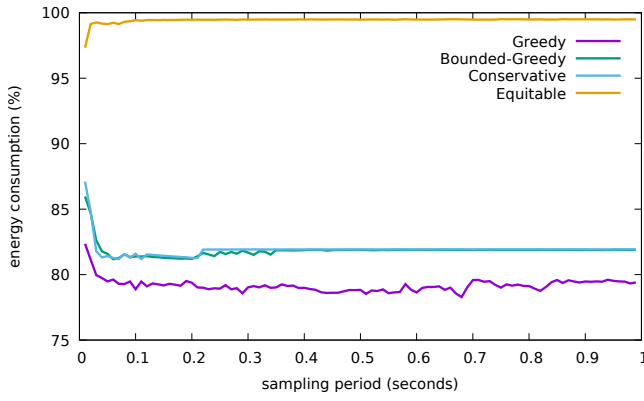


Figure 8. Energy consumption variation with the duration of the sampling period.

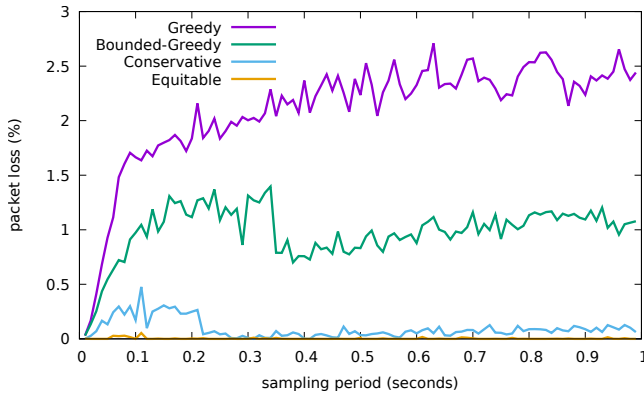


Figure 9. Packet loss percentage variation with the duration of the sampling period.

algorithm, which just homogeneously spreads the traffic among all the ports in the bundle.

Our first experiment evaluates how the energy consumption of our proposed algorithms varies with the duration of sampling period. Fig. 8 shows the results of that experiment for a buffer size of 10 000 packets.

As expected, the three algorithms outperform the equitable algorithm in terms of energy savings, consuming almost 20 % less. Besides, we can appreciate that the algorithms results are very close to the analytical minimum bound for the energy consumption. We can also appreciate that the energy savings obtained by the greedy algorithm are slightly higher than those of the other two algorithms, which consume almost the same. In addition, we can notice from Fig. 8 that values lower than 0.05 seconds yield noticeably worst results than values greater than 0.05 seconds in terms of energy consumption.

The next experiments evaluate the impact on the packet losses induced by our algorithm. Fig. 9 shows the packet loss percentage variation with the sampling period for a given buffer size of 10 000 packets. Fig. 10 represents the packet losses for different buffer sizes, for a given sampling period of 0.5 seconds.

The impact in packet loss depicted in the figures shows

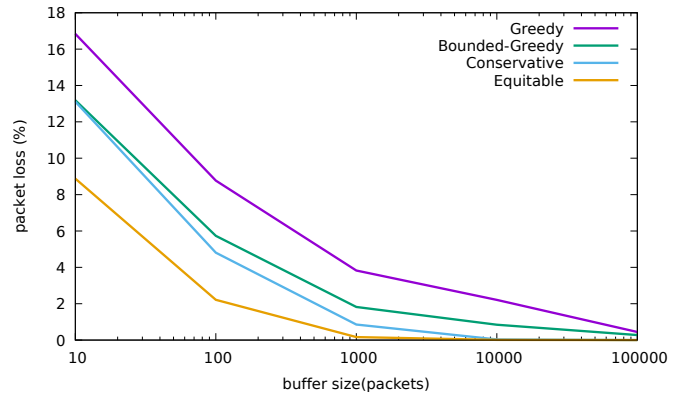


Figure 10. Packet loss percentage variation with the buffer size.

how the conservative algorithm outperforms the other two, but without reaching the level of the equitable one. In Fig. 9 we can also appreciate that this values exhibited by the conservative algorithm are indeed very close to 0 for almost any value of the sampling period (for a buffer of 10 000 packets).

Carefully analyzing this graphs we can appreciate that lowering the energy consumption implies incrementing the packet losses. Therefore, both metrics cannot be simultaneously optimized. Nevertheless, the improvement in the energy savings achieved by greedy algorithm with respect to the conservative one seems not to be worthy, due to the great impact in the packet loss rate. However, although in view of Fig. 10, low values of the sampling period such as 0.01 seconds report good values of packet losses, this is hardly implementable in practice: such a low sampling period would imply not only sampling the flow tables of every switch 100 times per second, but also sending up to 256 flow modifications every 10 ms. This is clearly a considerable amount of control traffic and imposes a huge number of flow modifications per second that would barely be manageable by the switches. Therefore, for a practical solution with a negligible performance degradation, we should employ a sampling period of at least 0.5 seconds. Finally, the energy consumption of the bounded-greedy algorithm is almost the same as the conservative, but the latter is clearly better in terms of packet loss percentage. Hence, there is no advantage in using the bounded-greedy rather than the conservative in any way, since the computational complexity of the three algorithms is equivalent.

To sum up, our energy savings do not come completely free: energy savings increase traffic delay. Consequently, if the delay must be bounded to a low value, a low buffer size has to be used, leading to appreciable packet losses due to the traffic rate variability of the transmitted flows. The conservative algorithm is able to obtain the minimum traffic delay and packet losses while obtaining almost identical energy savings. For instance, for the analyzed 30 Gbit/s traffic trace, the delay averages 270 μ s and the energy consumption is 82 % when using a flow sampling period of 0.5 s and a buffer size of 10 000 packets. For the conservative algorithms, this trade-off between delay and

energy savings can be tuned via the *safety margin* parameter (which we have set to 20 % up to this point). For instance, this same scenario using a safety margin of 70 % reduces the average delay to 150 μ s, although it elevates the consumption to nearly 91 % which is still a 9 % improvement in the energy consumption.

VII. CONCLUSIONS

This paper demonstrates the implementation of an energy saving algorithm using the facilities provided by SDN equipment. We have used the ONOS network operating system to reduce the energy consumption of an Ethernet link aggregate between to switches with IEEE 802.3az ports.

The obtained results match those predicted by the packet level model of the energy saving algorithm we have employed as the basis for our implementation. Thanks to the usage of ONOS, our algorithm is ready to be deployed in any SDN network, irrespective of the underlying SDN technology of the equipment manufactures.

We plan to extend the implementation to cover the case where several link aggregates are present in the SDN network, to harness the flow selection work already carried out by the switches for aggregates downstream to the flows, leveraging ONOS's centralized view of the topology.

ACKNOWLEDGEMENTS

This work was supported by the "Ministerio de Economía, Industria y Competitividad" through the project TEC2017-85587-R of the "Programa Estatal de Investigación, Desarrollo e Innovación Orientada a los Retos de la Sociedad," (partly financed with FEDER funds).

REFERENCES

- [1] L. Chiaraviglio, M. Mellia, and F. Neri, "Minimizing ISP Network Energy Cost: Formulation and Solutions," *IEEE/ACM Transactions on Networking*, vol. 20, no. 2, pp. 463–476, Apr. 2012.
- [2] D. Jung, R. Kim, and H. Lim, "Power-saving strategy for balancing energy and delay performance in WLANs," *Computer Communications*, vol. 50, pp. 3–9, Sep. 2014.
- [3] Y.-M. Kim, E.-J. Lee, H.-S. Park, J.-K. Choi, and H.-S. Park, "Ant colony based self-adaptive energy saving routing for energy efficient Internet," *Computer Networks*, vol. 56, no. 10, pp. 2343–2354, Jul. 2012.
- [4] M. Rodríguez Pérez, M. Fernández Veiga, S. Herrería Alonso, M. Hmila, and C. López García, "Optimum Traffic Allocation in Bundled Energy-Efficient Ethernet Links," *IEEE Systems Journal*, p. in press, 2015.
- [5] "ONOS - A new carrier-grade SDN network operating system designed for high availability, performance, scale-out." [Online]. Available: <https://onosproject.org/>
- [6] K. Christensen, P. Reviriego, B. Nordman, M. Bennett, M. Mostowfi, and J. A. Maestro, "IEEE 802.3az: the road to Energy Efficient Ethernet," *IEEE Communications Magazine*, vol. 48, no. 11, pp. 50–56, Nov. 2010.
- [7] M. F. Tuysuz, Z. K. Ankarali, and D. Gözüpek, "A survey on energy efficiency in software defined networks," *Computer Networks*, vol. 113, pp. 188–204, Feb. 2017.
- [8] B. B. Rodrigues, A. C. Riekstin, G. C. Januário, V. T. Nascimento, T. C. M. B. Carvalho, and C. Meirosu, "GreenSDN: Bringing energy efficiency to an SDN emulation environment," in *2015 IFIP/IEEE Int. Symp. Integr. Netw. Manag.* IEEE, May 2015, pp. 948–953.
- [9] B. Heller, S. Seetharaman, P. Mahadevan, Y. Yiakoumis, P. Sharma, S. Banerjee, and N. McKeown, "ElasticTree: Saving energy in data center networks," in *Nsdi*, vol. 10, 2010, pp. 249–264.
- [10] T. Huong, D. Schlosser, P. Nam, M. Jarschel, N. Thanh, and R. Pries, "Ecodane—reducing energy consumption in data center networks based on traffic engineering," in *11th Würzburg Workshop on IP: Joint ITG and Euro-NF Workshop Visions of Future Generation Networks (EuroView2011)*, 2011.
- [11] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: enabling innovation in campus networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 2, pp. 69–74, 2008.
- [12] "The CAIDA UCSD Anonymized 2013 Internet Traces — 2013/08/15 13:14:00 UTC." [Online]. Available: https://www.caida.org/data/passive/passive_2013_dataset.xml
- [13] "Mininet: An Instant Virtual Network on your Laptop (or other PC)." [Online]. Available: <http://mininet.org/>
- [14] S. Herrería Alonso, M. Rodríguez Pérez, M. Fernández Veiga, and C. López García, "A GI/G/1 Model for 10 Gb/s Energy Efficient Ethernet Links," *IEEE Transactions on Communications*, vol. 60, no. 11, pp. 3386–3395, Nov. 2012.
- [15] M. A. Marsan, A. F. Anta, V. Mancuso, B. Rengarajan, P. R. Vasallo, and G. Rizzo, "A Simple Analytical Model for Energy Efficient Ethernet," *IEEE Communications Letters*, vol. 15, no. 7, pp. 773–775, Jul. 2011.
- [16] "IEEE Standard for Information technology— Local and metropolitan area networks— Specific requirements— part 3: CSMA/CD Access Method and Physical Layer Specifications Amendment 5: Media Access Control Parameters, Physical Layers, and Management Parameters for Energy-Efficient Ethernet," *IEEE Std 802.3az-2010 (Amendment to IEEE Std 802.3-2008)*, pp. 1–302, Oct. 2010.
- [17] M. Rodríguez Pérez, "A Rustified Simulator for 10 Gb/s EEE with Configurable Hysteresis." [Online]. Available: <https://migrax.github.io/HystEEE/>
- [18] P. Fondo-Ferreiro, "SDN Bundle Network Simulator." [Online]. Available: <https://pfondo.github.io/sdn-bundle-simulator/>