



University of
Zurich^{UZH}

Routing for Energy Efficiency in Software-defined Networks

*Alexander Hofmann
Zurich, Switzerland
Student ID: 11-916-863*

Supervisor: Bruno Rodrigues, Dr. Corinna Schmitt
Date of Submission: XX, 2017

Abstract

Introduction

Current data centers exhibit poor power efficiency, running network devices at full capacity regardless of traffic demands and distribution over the network [1]. A significant number of protocols and approach have been proposed in recent years in response to the demand for reducing the amount of energy consumed by the network infrastructure. However research on this topic, typically focused on making the single nodes, routers and switches, inside a network more efficient, essentially treating them as isolated machines, instead of trying to make the network more efficient as a whole. Several approaches have addressed this problem by introducing holistic methods - approaches that view the network as one and try to optimize the efficiency at the network level [1, 5, 6, 7]. Unfortunately none of these approaches is either available as open source code or tailored for SDNs (Software Defined Network). SDNs are particularly important as experimental environments, since they support rapid development and testing of new approaches and methods and thus enables the discussion between researchers.

Thus the main contribution of this work is establishing the ground software mechanisms to deploy complex traffic engineering models, such as but not limited to, GreenTE.

Goals

This software project has two main high level goals:

- Designing a system that would make it easier for future work to implement new policies inside an SDN environment designed specifically for experimentation, GreenSDN.
- Implement Zhang *et al.*'s GreenTE model using the designed system of policies.

In order to achieve these goals this project needs to extend existing controller functionality in order to build such algorithm, as well as improvements on essential capabilities towards the development of traffic engineering capabilities.

Results

The results of this project are thus an architectural design that allows the future developers to easily implement new policies, and a working prototype, consisting out of the integration of the designed and implemented elements in GreenSDN. Regarding the former, the designed policy system works as intended, as demonstrated from the policies implemented: one load balancing policy, and one merging policy, based on GreenTE [1]. Based on a simple clock-based system they are able to switch between the two and give the controller the information needed to control traffic. The latter, the merging policy, is able to merge traffic coming from multiple hosts into fewer paths, based on configurable parameters the developer can define when the controller starts up.

Thanks to GreenSDN's energy monitoring capabilities, It was possible to validate the implementation, showing that by merging paths the power efficiency of the network as a whole increases. The emulation results showed that is possible to achieve power savings in between 23% and 30% during low traffic periods, which falls right into the experimental results achieved by Zhang *et al.* [1], thus further validating this work.

Further Work

Despite providing a baseline environment and policy management system, GreenSDN requires improvements, which would allow it to exit its *Proof of Concept stage*. Some potential future works worth being mentioned are:

- Refactoring of the controller to use a newer controller library, e.g. *Ryu* [12] instead of *POX*, which is unfortunately not actively maintained any more.
- Implementation of more complex modelling methods: the implemented GreenTE model uses fixed values for certain parameters, which could be dynamically computed by a topology-aware controller.
- Testing new experimental approaches.
- Further analyzing the impact on *QoS* when using green capabilities.

Contents

Abstract	i
1 Introduction	1
1.1 Motivation	1
1.2 Description of Work	2
1.3 Thesis Outline	3
2 Related Work	5
2.1 GreenSDN	5
2.2 GreenTE	5
2.3 ElasticTree	6
3 Implementation of the GreenTE model	9
3.1 Architecture	9
3.2 Full Architecture & Development Details	10
3.2.1 Policies and Policy Manager	11
3.2.2 GreenTE	12
3.2.3 Caching Mechanism	12
4 Evaluation	15
4.1 Testing Environment	15
4.1.1 Mininet and POX	17
4.2 Policies Evaluation	17

4.3	Routing Mechanisms Evaluation	17
4.3.1	Impact on Energy Consumption	18
4.3.2	Random Traffic - Overall network consumption	20
4.3.3	Constant Traffic - Overall network consumption	20
4.3.4	Comparison with GreenTE [1]	22
5	Summary and Conclusions	25
5.1	Future Works	25
	Bibliography	27
	Abbreviations	29
	Glossary	31
	List of Figures	31
	List of Tables	33
A	Installation Guidelines	37
A.1	Reproducing the experiment	37
B	Contents of the CD	39

Chapter 1

Introduction

The Internet is an indispensable communication system. As society continues to shift towards an increased digital information, the requirements for scalability and Quality of Service (QoS) grow. This translates into an increase of the number and density of data centers, highly redundant network topologies and aggressive over-provisioning of bandwidth to accommodate potential link failures and traffic bursts. Meeting these requirements requires power-hungry machines and support systems (e.g. cooling) which leads to significant Green House Gas (GHG) Emissions.

Research on this topic, typically focused on making the single routers and switches, inside a network more efficient, essentially treating them as isolated machines, instead of trying to make the network more efficient as a whole. Several approaches try to address this problem [1, 5, 6, 7], however none of these solutions is readily available as open source. Green Traffic Engineering (GreenTE) [1] is one such approach that tries to tackle this issue, by trying to put as many links to sleep as possible during low utilization periods, for example, during the night.

This work tries to implement this approach as open source software. It uses GreenSDN [8], a virtualization environment to emulate a Software Defined Network (SDN) environment that can emulate energy saving protocols. GreenSDN is based on the Mininet environment [9] and the OpenFlow Controller (POX) [10]. It uses the extended monitoring capabilities of GreenSDN to validate and evaluate the implementation.

Thus, building a network infrastructure that supports sustainability features is of utmost importance to lower the carbon footprint of ICT systems in the information age.

1.1 Motivation

The 2016 United States Data Center Energy Usage Report [2] estimated, that in 2014, data centers across the US consumed 70 TWh, representing about 1.8% of the energy consumption in the country with an increase of 4% since 2010. Based on current trend estimates, data centers, in the U.S. alone, are projected to consume approximately 73

TWh in 2020. To put these numbers into context, the energy consumption of data centers in the U.S., is roughly equals to that of countries such as Austria and Venezuela.

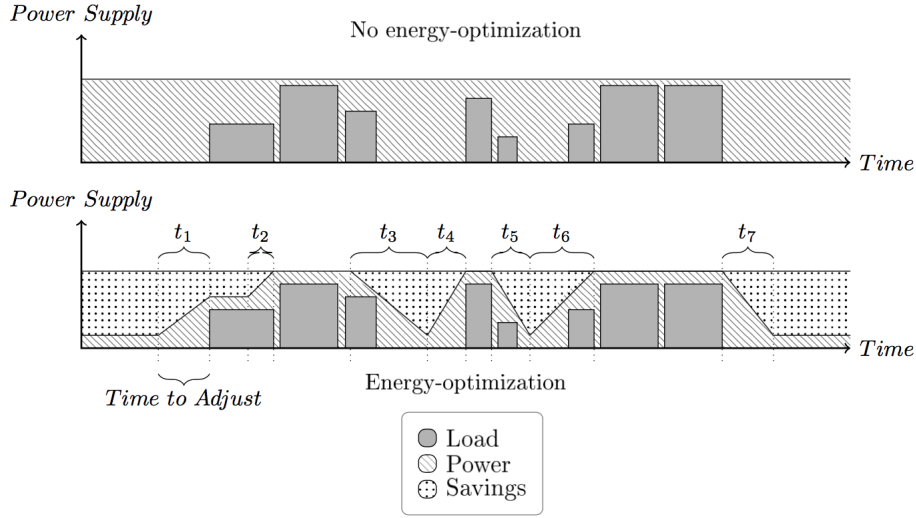


Figure 1.1: Energy-optimization vs no optimization [14].

As mentioned previously, networks tend to build many redundant links and aggressively over-provisions link bandwidth to accommodate potential link failures and traffic bursts during peak periods. While this increases the overall QoS during peak periods, during low utilization periods it consumes more power than needed, creating opportunities to apply optimization strategies, as shown in Figure 1.1.

Works such as GreenTe [1], and ElasticTree [5] have both proposed methods based on the idea of finding a minimal subset network that can still handle incoming traffic while maintaining a reasonable level of *QoS*. Unfortunately, there is no open source software implementing this approach readily available for Software-Defined Networks.

Thus this work, tries to fill in the gap, by taking Zhang *et al.*'s work and applying it to a SDN environment, such as GreenSDN.

1.2 Description of Work

In Software-Defined Networks (SDN) based on the OpenFlow protocol, routing capabilities are implemented as an application running in a controller that uses Link Layer Discovery Protocol (LLDP) to find connections between nodes in the network. Therefore, the controller is able to maintain a network view as a graph and manipulates forwarding routes based on conditions or policies defined by a network administrator. In this project, routes should be defined in order to maximize the energy efficiency of the network. Therefore, due to the characteristics of SDNs in which the intelligence of the switches is abstracted off the data plane to the control plane and its applications, it is necessary to develop an SDN application that allows for computation and definition of such routes. Thus, this software project tackles new and unsolved problems related to the design and development of such

an algorithm taking in consideration existing algorithms, such as [6] or [7] to develop a viable and efficient solution.

Thus the main goals of this project are:

- Designing a policy system that allows developers to easily extend GreenSDN and thus enable experimentation.
- Implement the Zhang *et al.*'s GreenTE model using the designed system of policies.

To attain this goal, this work was developed taking as a basis different steps of a methodology, the main contributions of which are listed below:

- Literature Review and Analysis: included in Chapter 2 provided the theoretical basis for the Design and Development stage.
- Design and Development: this step included the planning and implementation of this software project in order to attain its goals, as described in Chapter 3.
- Evaluation: the experimental evaluation of the GreenTE implementation inside the GreenSDN environment, presented in Chapter 4.

1.3 Thesis Outline

This is a report on the development of a GreenTE approach as open source software. It is structured as follows: Section 2 discusses related work in the field of GreenTE. Section 3 describes the design decision and development of the tool. Section 4 evaluates the energy saving capabilities and compares them to the related work on the subject. Section 5 discusses lesson learned and future work.

Chapter 2

Related Work

In this section, the selected approach in green capabilities GreenTE, ElasticTree, and GreenSDN environment are briefly reviewed.

2.1 GreenSDN

In order to increase energy efficiency of modern datacentres, multiple energy saving mechanisms have been proposed. Unfortunately emulation environments have not been discussed as much. Rodrigues *et. al* presented GreenSDN [8], an emulation environment based on mininet and POX. GreenSDN was developed with mechanisms to: monitor traffic statistics; switching of power profiles; and implementing three different green capabilities approaches operating on different network layers [3, 4, 7]. For this software project, the most important feature of GreenSDN was the traffic monitoring capabilities. Thanks to this it was possible to observe the energy consumption and workload information for each node of the topology. Using this information the centralized controller can effectively compute new flow tables to implement the GreenTE model. Network statistics were obtained by polling the nodes of the topology. The nodes in turn respond with flows and ports statistics. This polling mechanism has the potential of increasing network traffic and consumption. To obviate this problem, only the edge nodes, those connected directly to the hosts, are queried. If a workload is detected, only the nodes along the active path will be polled for new information.

All host, nodes, and path information are managed inside the *informationManager* class. This class also implements the capability to compute path consumption and workload which is central to this software project.

2.2 GreenTE

The Internet has its share in energy consumption. The underlying network infrastructure, namely routers, switches and other network devices, still lacks effective energy manage-

ment solutions. Existing research treats the nodes in the network as isolated devices and focuses on reducing power consumption at a hardware component level [1]. GreenTE proposes to tackle the problem at the network level by routing traffic through different paths to adjust the workload on individual nodes. Modern network infrastructure exhibit poor energy efficiency, running devices at full capacity 24/7 and building redundant links in case of failure, regardless of traffic demand.

The high path redundancy and low link utilization provide unique opportunities for energy aware traffic engineering. Indeed, given multiple paths between the same source and destination and an overall low link utilization, it should be possible to move traffic from one path to the other and put the devices on the free path to sleep to consume less energy.

Zhang *et al.* [1] propose a green traffic engineering mechanism to increase network energy efficiency while maintaining an acceptable QoS level. It is formulated as a Mixed Integer Programming (MIP) problem with power savings as the value to maximize and Maximum Link Utilization (MLU) and network delay as its constraints. MIPs belong to the NP-Hard class of problems thus solving it for big topologies is not feasible. In order to solve it Zhang *et al.* [1] proposed a simple heuristic based on two constraints. Candidate paths and MLU.

Instead of searching among the set of all possible paths between the same source and destination, they search only among candidate paths. Candidates are selected from the k -shortest paths. By doing this, the MIP is computed using at most k -candidate paths. The second constraint is MLU. The problem was reformulated using MLU as a constraint. At any given moment, the utilization of a link must smaller then a certain threshold U_T .

While Zhang *et al.* proposed a mechanism to increase energy efficiency of a network by considering the network as a whole, the authors did not make their solution available as open source code. Additionally GreenTE was tested against a simulated environment and not an emulated one, like Mininet. Thus the main contribution of this work is to establish the ground software mechanisms to deploy, complex traffic engineering models, such as, but not limited to GreenTeE in an emulated SDN environment, namely mininet, using the GreenSDN environment.

2.3 ElasticTree

ElasticTree [5] is a network level capability, focusing on data center networks. *ElasticTree* optimizes energy consumption by turning off unnecessary links and switches during off-peak hours. The problem was formulated and modeled likewise to GreenTE [1], but focuses primarily on Fat-Tree topologies. It consists of three modules: optimizer, routing, and power control. The optimizer solves the traffic problem under constraints such as *MLU*, *shortest paths*, finding the minimum power network subset which still satisfies the current network situation and passes the information regarding which components to maintain active to the routing and power control modules.

This modular composition allowed Heller *et al.* to develop three different optimizers to compute the minimum-power network subset. The first method is based on solving the

formal model, which gives the best results but is also the slowest approach of the three and as such not scalable. The second method is greedy bin-packing which offers a good balance between scalability and quality of results. The last method is a heuristic based on knowledge of the concrete topology in use, which offers the fastest results and is thus the most scalable, but lowest quality in regard of optimal network subset.

ElasticTree was also implemented using OpenFlow and tested on actual hardware instead of through a simulation, as was the case with GreenTe. Unfortunately, the authors have not provided any source code of their implementation.

Chapter 3

Implementation of the GreenTE model

This chapter describes the technical details involved in the design and implementation of the GreenTE model and how it interfaces with GreenSDN. In this regard I first introduce the architecture and its main components detailing how they relate to each other in Section 3.1. I then present the full architecture and the development details in Section 3.2.

3.1 Architecture

The architecture is based on the GreenSDN which in turn is based on the SDN reference architecture [13] and is presented in Figure 3.1. It uses four planes of abstraction: i) data, ii) control, iii) application, iv) management:

- i) **data plane:** includes mininet with Open vSwitches (OVS) running in kernel mode to switch packets across the interfaces, and parallel links interconnecting each pair of nodes.
- ii) **control plane:** includes the worker modules of GreenSDN. These modules are responsible for obtaining and preparing data for the management layer.
- iii) **application plane:** concerns itself with presenting the data through Graphical User Interface (GUI) elements, such as a topology viewer, and charts representing network utilization.
- iv) **management plane:** contains the modules responsible for taking decisions. In this plane inside the control plane, I implemented the energy policies for the GreenTE model.

The GreenTE model was implemented inside the POX controller at the management plane. It uses the data fetched and prepared from the management utilities of GreenSDN. In particular it fetches the energy consumption and workload information of each node.

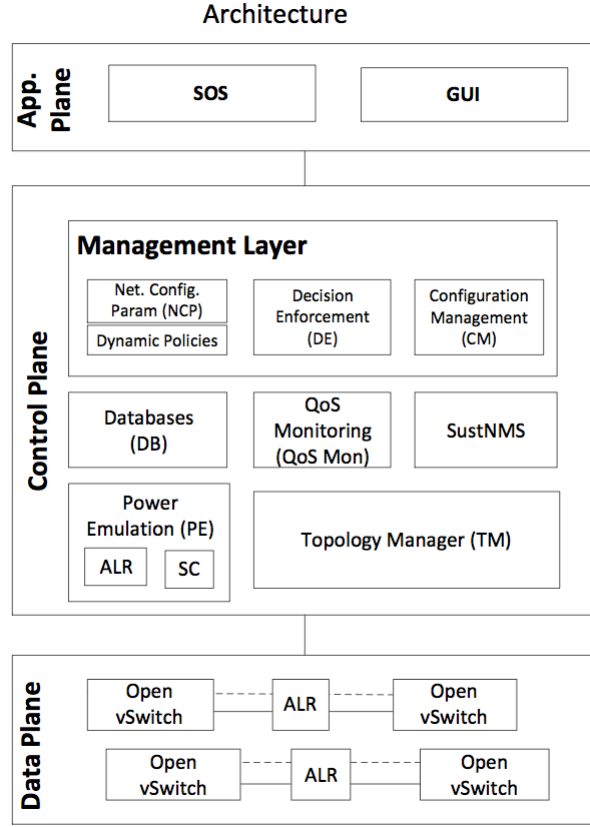


Figure 3.1: High-level view of the GreenSDN architecture [14]

3.2 Full Architecture & Development Details

This section presents in detail how the GreenTE model was implemented and how it interacts with the GreenSDN environment. To better demonstrate the effects of the GreenTE model in a before-after fashion, two policies were introduced: i) a Load Balancing policy, ii) and a Energy Savings policy:

- **Load Balancing:** The load balancing policy represents the traditional network traffic engineering methodology. The controller will try to balance all traffic in order to minimize MLU. In this policy any unused links are still put to sleep, so consumption is less than one would have in a realistic scenario.
- **Energy Savings:** The controller tries to move traffic from one path to the other. By doing this we can put the now unused links in sleep mode and increase energy efficiency. To avoid overloading any given path and to account for any possible traffic spike, a configurable MLU threshold U_t is introduced as in the GreenTE [1] paper.

In order to manage these and future policies a *PolicyManager* component was added to easily manage and define when to use which policies. The controller then delegates any decision regarding network traffic to these policies.

3.2.1 Policies and Policy Manager

The controller delegates its routing decisions to policies. Each policy defines a set of rules or a certain behavior for the system. Two policies were written for this project: *Merging* and *LoadBalancing*. The *LoadBalancing* policy makes the controller behave in a traditional network engineering way, by trying to minimize MLU. The *Merging* follows the GreenTE model and tries to save energy.

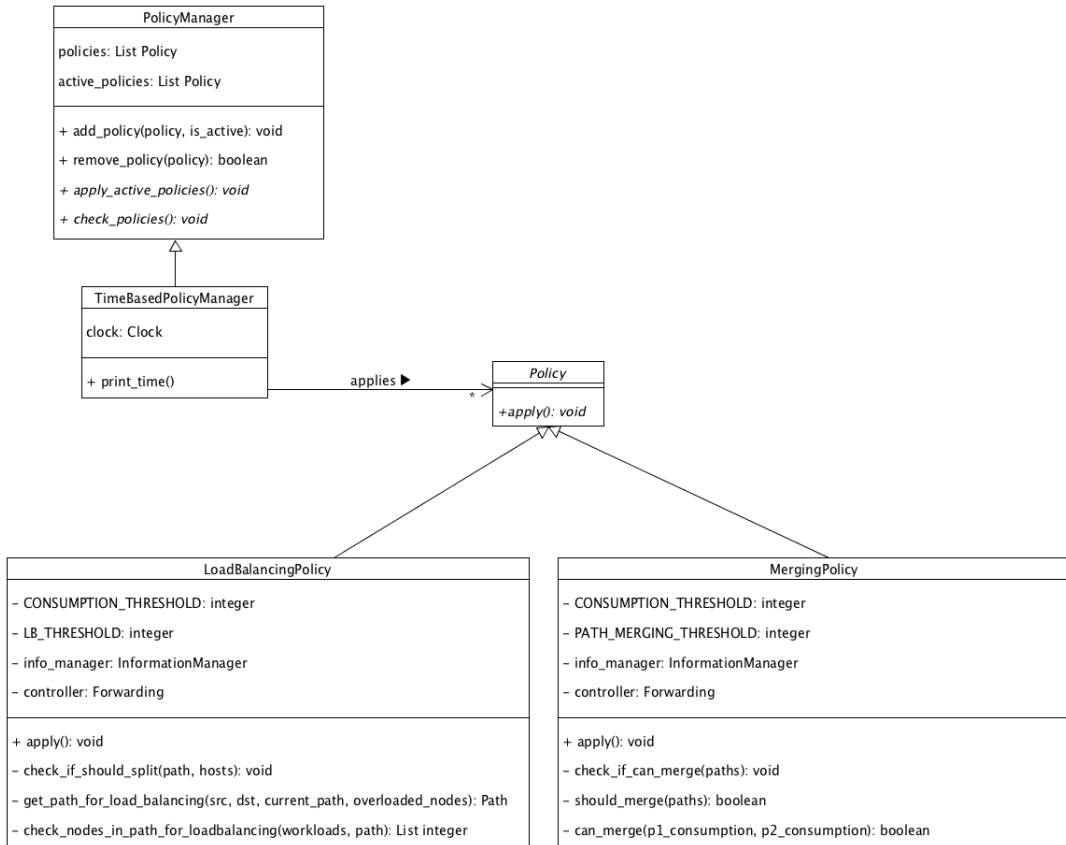


Figure 3.2: Class diagram of policies and policy manager.

The *PolicyManager* can be further extended by introducing new policies. Policies are handled in a way similar to the chain of responsibilities pattern. A collection of active policies is kept inside a *PolicyManager*. When a decision is required of the controller, it delegates any action to the *PolicyManager*. The *PolicyManager* in turn iterates over the list of active policies, and applies each one of them in order until there are no more policies left. This way it is possible to use multiple complementary policies if needed. It is also possible to define a pipeline for decisions, in which the data is manipulated before being passed on to the next active policy. A simple *check_policies()* method is added to the *PolicyManager* to check whether the active policies have to be changed and how. An abstract base *PolicyManager* class and an abstract base *Policy* class are provided with the project and act essentially as mocks, in order to have an idea of the basic methods and properties all *Policy* classes must have. Figure 3.2 gives an overview of policies and policy manager.

3.2.2 GreenTE

The main goal of traffic engineering mechanisms is to minimize Maximum Link Utilization to increase Quality of Service, which puts it in direct contraposition of Green traffic engineering where the goal is to maximize MLU, while maintaining acceptable QoS levels.

The idea behind GreenTE is surprisingly simple, Zhang *et al.* proposed to free some link, moving the traffic from one path to the other, so that the freed links can be put to sleep. By doing this we can increase overall network efficiency.

In order to implement this model inside the GreenSDN environment a GreenTE policy was introduced, the *Merging* policy. Additionally a number of changes are needed to the control and management planes. First and foremost there needs to be a way to compute overall workload and consumption for any given path. This is achieved by leveraging the GreenSDN capabilities to get per node workload and consumption. With this in place, it is simply a matter of iterating over the nodes of any given path and compute the workload.

Secondly it needs to be possible to easily modify the flow rules. For this, a method was added to the controller: *modify_path_rules(new_path, src, dst)*. Given a new path it iterates over each nodes and send a *OPENFLOW.OFPFC_MODIFY* message to modify the previously installed flow rules. If no rule exists *OPENFLOW.OFPFC_MODIFY* will add a new one.

To illustrate the GreenTE implementation Algorithm 1 presents the basic algorithm.

Algorithm 1 Merge method

```

1: function MERGE(paths) ▷ paths is the list of all paths between the same src and dst
   switches
2:   new_path ← paths0;
3:   for path ∈ paths do
4:     if new_path.workload + path.workload < Ut then
5:       new_path.workload ← new_path.workload + path.workload;
6:       modify_path_rules(new_path, path.src, path.dst);

```

3.2.3 Caching Mechanism

The controller needs to have an overview of all paths being used at the moment in order to take decisions. The existing *InformationManager* class already keeps track of all paths for each user. But in order for the *Merging* policy to work effectively, a different view of this data is needed. Specifically it needs to have the data as a function of source-destination pairs. Unfortunately computing this view each time it is needed tends to be computationally expensive even for small networks. Even with the relatively small test network, comprising of 28 switches and 4 hosts, this often led to timeouts and high CPU utilization in the controller. While some of the timeouts are caused by the evaluation environment (see 4.1), similar effects should also appear when working with very big topologies.

In order to solve this problem, a path lookup table, the *PathTable* class, was introduced to keep track of paths being actively used. This lookup table, is a simple map between source and destination and a list of paths. The table is populated once when the controller starts up and is kept updated anytime any paths are changed, or new links come up. Algorithm 2 shows the simple method for computing paths, *generate_paths*(*G*, *src*, *dst*) computes paths between *src* and *dst* using *NetworkX* [15]. For big topologies it might be unfeasible to compute all possible paths between a given source and destination. To solve this problem a threshold for the paths computed, *PATH_LIMIT*, was introduced.

Algorithm 2 Caching method

```

1: function PRE_COMPUTE_PATHS(graph, hosts)  ▷ graph is the network graph
2:   host_combinations ← generate_src_dst_combinations(hosts);      ▷ List of tuples
   representing all possible combinations of hosts
3:   for src, dst ∈ host_combinations do
4:     for PATH_LIMIT do  ▷ compute only upto PATH_LIMIT paths in case of
       big topologies
5:       new_path ← generate_path(G, src, dst)  ▷ generates a new shortest path
       between src and dst
6:       insert_into_lookup_table(new_path)

```

Chapter 4

Evaluation

This chapter presents an evaluation of the ease of extensibility of the code, and an evaluation of the GreenTE model implemented, especially drawing comparison with Zhang et al.'s [1] results. Regarding this last point, readers should keep in mind that while Zhang's work was tested in a simulated environment, this implementation is running inside *Mininet*, an emulated environment. This means that results could drastically differ to those of the paper this work is based on.

In order to evaluate the results, it is necessary to remember what the goals of this work were, thus this project set out to solve two problems:

- Designing a policy system that would make it easier for future work to implement new policies inside GreenSDN
- Implement the Zhang *et al.*'s GreenTE model using the designed system of policies.

Section 4.1 describes the environment configuration, including the specifications of the physical machine, the topology, and configured flows. Section 4.2 reviews the extensibility of the code. Section 4.3 presents results from the GreenTE policy.

4.1 Testing Environment

The machine used was a Macbook Pro (Early 2015), with an i5 @ 2.7 GHz and 8 GB RAM. The host operating system is macOS Sierra (10.12.5). The simulation code ran inside VirtualBox Version 5.1.18 r114002. The guest OS (Operating System) is Ubuntu 15.10 and was given 2 cores and 3 GB RAM. The SDN network was emulated in Mininet, with GreenSDN based on the POX controller running OpenFlow 1.0. Network traffic is generated by the Iperf tool. The topology implemented is the *facebook topology* shown in Figure 4.1. Each pair of nodes was connected using links with a maximum traffic of 1000 Mbps.

Iperf [16] was chosen to generate traffic. It is a tool for performing network throughput measurements. Iperf was chosen since it enables users to easily generate traffic by simply establishing a server (to discard traffic) and a client (to generate traffic). With Iperf it is also possible to specify the amount of traffic to generate but only in UDP (User Datagram Protocol) mode. Because of this all traffic is sent as UDP. Four hosts are used for testing, 3 as clients, generating traffic and one acting as the server (sink). Specifically *H18* is set as the sink, running *iperf -s -u* to accept incoming UDP traffic, while the other hosts run *iperf -c 10.0.0.6 -u -b \$bandwidth -t \$time*.

Network consumption and workload data was written to InfluxDB, a datastore for real-time analytics [17], while the time series were generated using Grafana, a software for time series analytics[18].

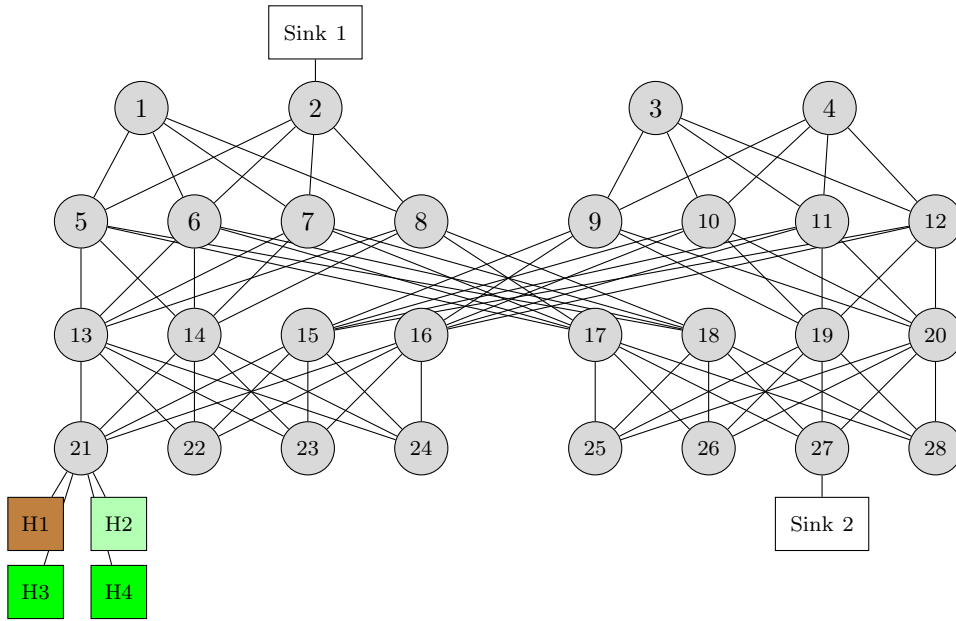


Figure 4.1: The Facebook topology. In the emulation only hosts *H1*, *H2*, *H3* are sending traffic to *Sink1*.

Inside the emulation, a clock was implemented which is used to switch between 'performance' (loadbalancing) and 'energy savings' (GreenTe) mode. The idea is to emulate a realistic scenario: during the day, when the workload is high it is not possible to try and apply the GreenTe model since this would greatly impact *QoS*. During the night, however, when the traffic is low, we can apply green policies while maintaining an acceptable level of service. In order to show the efficacy of this model, the same amount of workload is applied in both policies, to show the difference applying green policies would make during low traffic periods.

Additionally, results are compared to a third approach: instead of putting nodes to sleep only during the energy savings time, nodes are systematically put to sleep whenever they are idle. This leads to higher energy savings, but will probably negatively affect *QoS* and overall jitter.

Finally in the last section results are compared to the evaluation presented in the GreenTe model paper [1].

4.1.1 Mininet and POX

Central to the GreenSDN environment, are Mininet and POX. Mininet [19] allows the user to create a realistic virtual network, running real kernel, switch and application code, on a single machine, allowing to quickly create a software-defined network prototype and run proof of concepts. The Mininet distribution included in GreenSDN implements the interface and node-level green capabilities emulation on the SDN controller using the OpenFlow Controller.

POX [11] is one of the most popular controllers implementing the OpenFlow protocol. It is python based and open-source. Because of it's easiness of use, extensibility and documentation it was chosen as the controller framework for GreenSDN [8]. Unfortunately, it seems development on POX has stopped. As my work is based on GreenSDN I continued using POX but future work should consider using a different controller framework e.g. Ryu [12].

4.2 Policies Evaluation

This section presents an evaluation of the ease of extensibility of the policies and policy manager. One of the goals for this project was to set the ground work for implementing new policies. Thus this section reviews the extensibility of the system by demonstrating that the implemented system works.

Figure 4.2 shows the per node consumption for a subset of the network. While the merging policy is active, all hosts send traffic over the same path. This is shown by the fact that all active nodes consume more or less the same amount of energy.

During load balancing mode the traffic flows through more then four nodes. Because of the topology, all hosts share the same ingress (node 21) and egress (node 2) nodes, this means those nodes have the maximum consumption between all nodes while the middle nodes consume less since the workload is distributed among multiple paths. This combination of active nodes all consuming the same amount of energy and inactive nodes going to sleep creates this pattern, which looks as if something were missing, thus 'hole' pattern.

Figure 4.3 presents the per-node-workload for a subset of the network where the same observations as for the consumption can be made as for Figure 4.2 since the consumption is linearly proportional to the workload.

4.3 Routing Mechanisms Evaluation

As mentioned previously, to evaluate the efficacy of the GreenTE policy, an additional *LoadBalancing* policy was introduced. While the *LoadBalancing* is active, the controller will spread the traffic as much as possible to minimize MLU. This behavior, which is the standard Network Traffic Engineering approach, allows us to achieve the maximum

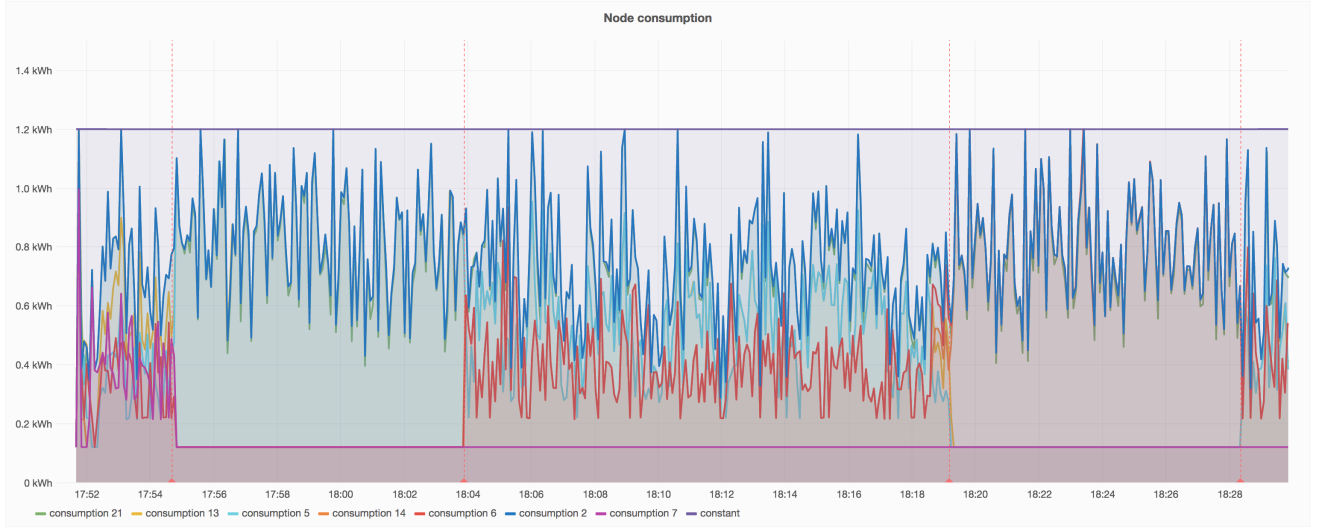


Figure 4.2: Per node consumption for a subset of the entire network. The constant value at 1.2 kWh is the max consumption for a switch.

amount of switches being used. This was chosen to better demonstrate the energy savings capabilities when running in GreenTE mode.

To test my implementation, two scenarios were used. In both scenarios test scripts were written where three hosts would generate traffic and a fourth host would act as sink. Specifically for the results here shown, hosts: $H1$, $H2$, $H3$ were used to send random amounts of traffic to $H18$, the sink:

- **Random Traffic:** A simple script was written where Iperf would be called at recurring but random time intervals with various amounts of traffic to simulate a realistic network scenario where both random spikes in traffic as well as idle moments could happen at any moment.
- **Constant Traffic:** For this scenario a script was written where Iperf would be called at constant intervals and with the same workload.

4.3.1 Impact on Energy Consumption

This section contains a brief explanation to better understand the graphs presented below (Figure 4.4 and Figure 4.5) and the impact of the aforementioned policies and routing mechanisms on energy consumption.

The red vertical dotted line marks the moment when there is a policy change. The yellow 'constant' value is the compounded value of all switches if they were running at 100% performance all the time, in which each switch would consume $1.2kWh$ of energy. The blue 'Baseline' value is the sum of consumption of all switches, in which the consumption of each switch is proportional to the workload. In case of no workload switches go idle

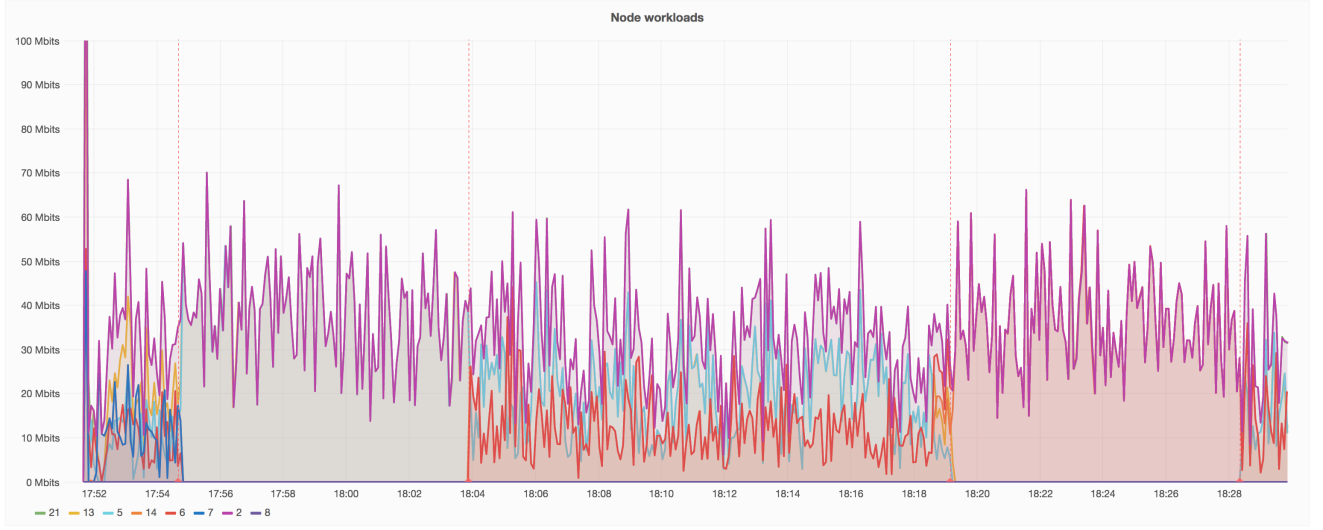


Figure 4.3: Per node workload for a subset of the entire network. It can clearly be seen that while the merging policy is active only 4 nodes are doing any work, while the others are in sleep modes.

consuming a fixed value of $0.2kWh$. Finally the green value is, similarly to the baseline, the sum of the consumptions of all switches in the network, but where switches are allowed to go into sleep mode when not used, consuming only $0.12kWh$. All power consumption reference values are as described in GreenSDN [8]. The following equations present how the energy consumption was computed inside GreenSDN:

$$PP = 200 + \left(\frac{500}{30} \right) * workload \quad (4.1)$$

$$PP_{sleeping} = 120 \quad (4.2)$$

$$PP_{ALR} = 200 + \left(\frac{500}{30} \right) * workload - 15\% * ALR \quad (4.3)$$

$$PP_{SC} = PP_{sleeping} * tOff + PP * tOn \quad (4.4)$$

Equation 4.1 presents the power consumption per node under normal workload conditions. Equation 4.2 the consumption of a switch with no workload. Equation 4.3 shows the computation when using ALR [3], which is a method to dynamically change data rates in response to traffic levels. Finally Equation 4.4 represents the per-port consumption when using SC, an approach based on Low Power Idle (LPI) mode [4]. For more information regarding power consumption computations, please refer to the GreenSDN paper [8].

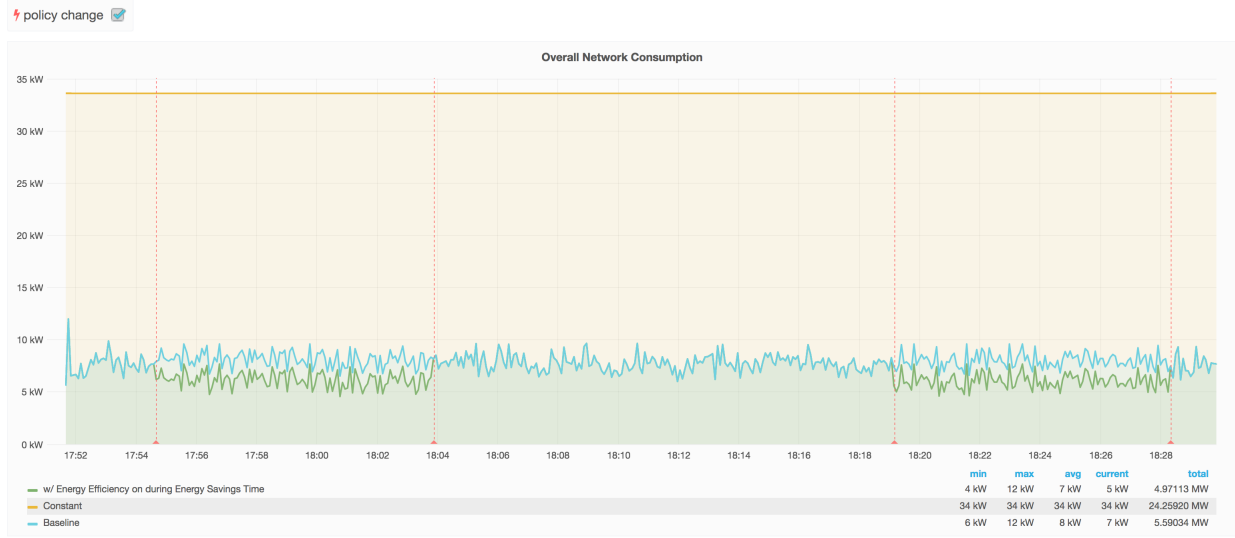


Figure 4.4: Overall network energy consumption with random traffic: while inside the red dotted interval (energy savings), it is possible to save almost 24% of the energy while the network traffic is low.

4.3.2 Random Traffic - Overall network consumption

Using the random traffic script, the three hosts $H1$, $H2$, $H3$ all send random amounts of workload at random time intervals. Essentially each host will, 1) Send a random amount of traffic (between 15 mbps and 35 mbps), and 2) sleep for a random amount of time (between 1 and 6 seconds), then repeat.

As can be seen in Figure 4.4, the moment the controller switches from performance mode to energy savings mode is quite clear. Figure 4.4 shows the graph of this run while Table 4.1 contains the computed mean consumption values from the overall network consumption. The data is from a run containing 2 phases where the controller switches from load balancing to energy savings. As can be seen from the table, there is a considerable opportunity to save energy. Comparing only the energy consumption between the baseline and the green policy during the energy savings mode, the average energy that can be saved by merging paths and putting switches to sleep is approximately $\approx 24\%$. Comparing the average consumption over an entire period composed of both energy savings and load balancing, the savings are approximately $\approx 11.8\%$, which is less than the energy savings achievable by constantly using the already built-in green policies. Further analysis would be needed to see the impact on QoS when always using green policies, but considering switches would constantly be put to sleep or woken up, the overall network jitter should increase as well as its average delay.

4.3.3 Constant Traffic - Overall network consumption

Using the constant traffic script, the three hosts $H1$, $H2$, $H3$ all send a fixed amount of traffic at host $H18$. For the results presented below, the traffic sent was 5 mbps for each

Table 4.1: Period between '17:51:40Z' and '18:29:51Z' in emulated time. All values approximated to six decimal places. Contains two phases, *Period 1* and *Period 2*, where the controller applies the energy saving policy. The overall average contains values from the entire period as shown in Figure 4.4.

Approach	Period 1 Avg.	Period 2 Avg.	Overall Avg.
Baseline (kWh)	8.120864	8.008092	7.888783
Energy efficiency (kWh)	6.187908	6.087771	6.957096
Energy efficiency always on (kWh)	6.187908	6.071385	6.015651
Savings (%) 'Energy efficiency'	23.802341	23.979755	11.810269
Savings (%) 'Energy efficiency always on'	23.802341	24.184377	23.744249

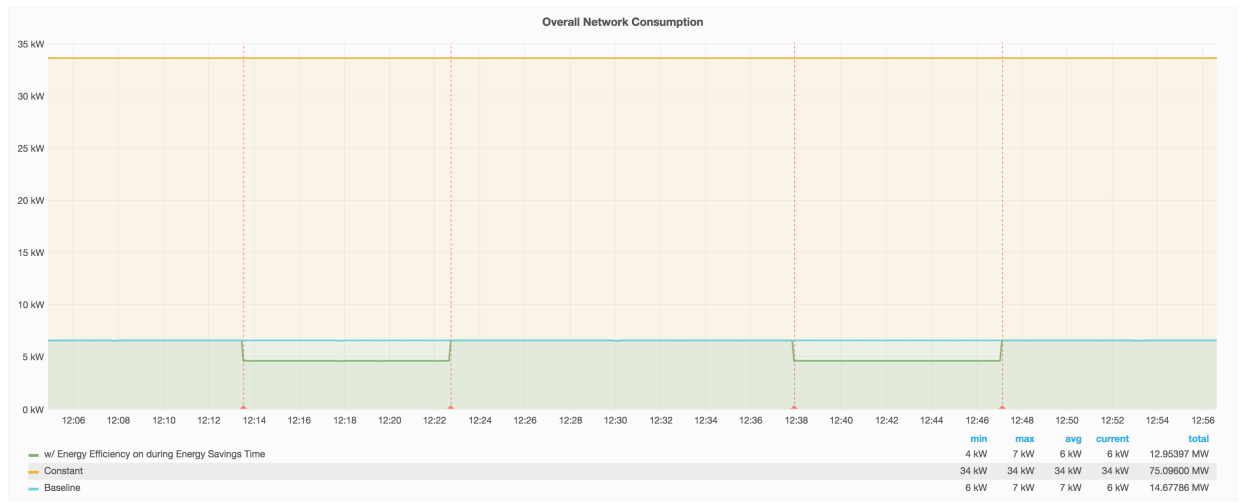


Figure 4.5: Overall network energy consumption while sending constant traffic:

host. All three hosts start more or less at the same time and as per *iperf* default, they send workload for 10s. Then it immediately starts again, without sleeping as in the case for the random traffic script.

As with the 'Random Traffic' section, Figure 4.5 shows the time series for the overall network consumption during the course of a run consisting of two energy savings phases. While Table 4.2 shows the mean consumption for the same run using the various profiles. In the case of constant traffic, the energy savings achieved, by putting nodes to sleep is even higher than with the random traffic 'profile', reaching almost 30%. The savings for the overall run are slightly lower then in the 'random traffic' test. With the 'random traffic' profile, both higher and lower traffic spikes are possible which probably explains the $\approx 1.4\%$ deviation. Here as well, energy savings are much higher when always putting idle nodes to sleep vs. just during energy savings time.

Table 4.2: Period between '12:04:30' and '12:56:53' in emulated time. All values approximated to six decimal places. Contains two phases, *Period 1* and *Period 2*, where the controller applies the energy saving policy. The overall average contains values from the entire period as shown in Figure 4.5.

Approach	Period 1 Avg.	Period 2 Avg.	Overall Avg.
Constant (kWh)	33.6	33.6	33.6
Baseline (kWh)	6.579125	6.580129	6.579459
Energy efficiency (kWh)	4.622529	4.623260	5.892887
Energy efficiency always on (kWh)	4.622529	4.623260	4.592750
Savings (%) 'Energy efficiency'	29.739456	29.739067	10.435083
Savings (%) 'Energy efficiency always on'	29.739456	29.739067	30.195629

4.3.4 Comparison with GreenTE [1]

In the evaluation section of Zhang *et al.*'s work, they presented the various results of applying the GreenTe model on different network topologies and using different traffic matrices. Results were computed by feeding existing traffic information inside a simulated environment. All results shown refer to periods using only the GreenTe approach, so I will compare them directly only to my 'energy savings' periods.

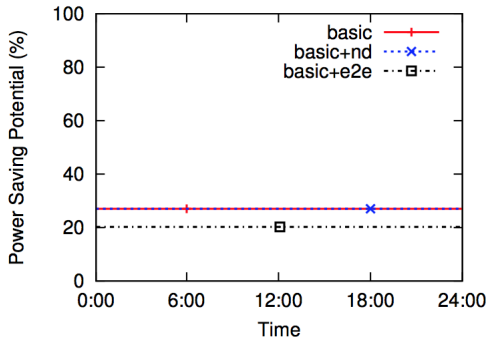


Figure 4.6: Power saving potential of the Abilene research network.

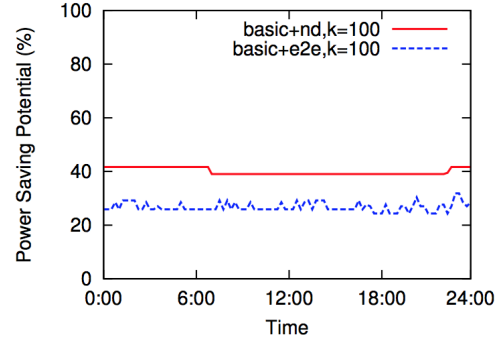


Figure 4.7: Power saving potential of the GEANTE research network

As shown in Figure 4.6 and Figure 4.7, depending on the constraints set, the number k of candidate path used and network topology, GreenTe achieved between 20% and 40% power savings.

Figure 4.8 shows the savings potential for my implementation of the GreenTe model, using the *facebook topology*. The results falls right inside this range. In the 'Random traffic' scenario we achieved approximately 24%, while in the 'Constant traffic' scenario 30%. I believe that while my results validate the results proposed from Zhang *et al.*, further testing with different topologies is needed in order to further prove the efficacy of the GreenTe approach.

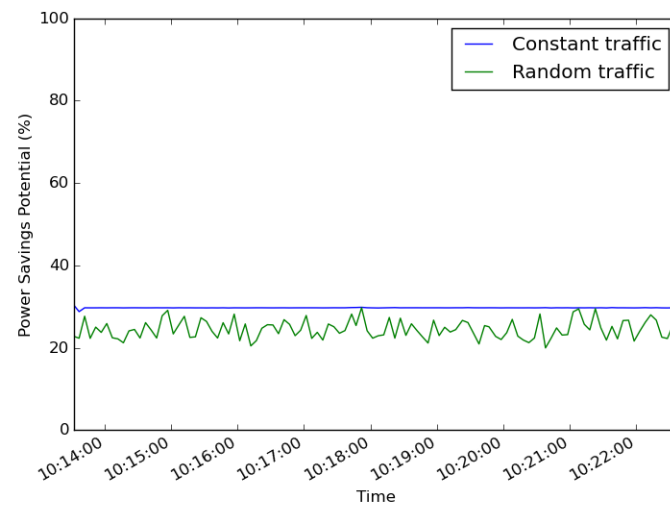


Figure 4.8: Savings potential.

Chapter 5

Summary and Conclusions

The primary requirements in network infrastructures are high-performance and high-availability. By introducing energy savings policies, it is possible to strike a balance between *QoS* and efficiency. Given the importance of these infrastructures it is necessary to be able to reliably emulate the effects of policies in realistic environments before the changes are rolled out to the actual data centers. This will make it easier for researchers to test new and innovative approaches without the need to allocate expensive hardware for testing purposes.

The goal of this software project was to implement Zhang et al.'s GreenTE model [1] and to set the ground work for more complex modeling in the field of Green Capabilities.

With the implemented policy system, it was simple to implement the load balancing and merging policies that made it possible to implement Zhang *et al.*'s work inside GreenSDN and achieve results similar to the ones, the authors presented in GreenTE [1].

5.1 Future Works

Despite providing a baseline environment and policy management system, GreenSDN requires improvements, which would allow it to exit its *Proof of Concept stage*. Some potential future works worth being mentioned are:

- Refactoring of the controller to use a newer controller library, e.g. *Ryu* [12] instead of *POX*, which is unfortunately not actively maintained any more.
- Implementation of more complex modelling methods: the implemented GreenTE model uses fixed values for certain parameters, which could be dynamically computed by a topology-aware controller.
- Testing new experimental approaches.
- Further analyzing the impact on *QoS* when using green capabilities.

Bibliography

- [1] Zhang, Mingui, et al.: "GreenTE: Power-aware traffic engineering." Network Protocols (ICNP), 2010 18th IEEE International Conference on. IEEE, 2010.
- [2] Shehabi, A., et al.: "United States Data Center Energy Usage Report", 2016.
- [3] Gunaratne, Chamara, et al. "Reducing the energy consumption of Ethernet with adaptive link rate (ALR)." IEEE Transactions on Computers 57.4 (2008): 448-461.
- [4] Mostowfi, Mehrgan, and Ken Christensen. "Saving energy in LAN switches: New methods of packet coalescing for Energy Efficient Ethernet." Green Computing Conference and Workshops (IGCC), 2011 International. IEEE, 2011.
- [5] Heller, Brandon, et al. "ElasticTree: Saving Energy in Data Center Networks." NSDI. Vol. 10. 2010.
- [6] Agarwal, Sugam, Murali Kodialam, and T. V. Lakshman. "Traffic engineering in software defined networks." INFOCOM, 2013 Proceedings IEEE. IEEE, 2013.
- [7] C. H. A. Costa, M. C. Amaral, G. C. Januario, T. C. M. B. Carvalho and C. Meirosu, "SustNMS: Towards service oriented policy-based network management for energy-efficiency," Sustainable Internet and ICT for Sustainability (SustainIT), 2012, Pisa, 2012, pp. 1-5.
- [8] B. B. Rodrigues, A. C. Riekstin, G. C. Januario, V. T. Nascimento, T. C. M. B. Carvalho and C. Meirosu, "GreenSDN: Bringing energy efficiency to an SDN emulation environment," 2015 IFIP/ IEEE International Symposium on Integrated Network Management (IM), Ottawa, ON, 2015, pp. 948-953.
- [9] Mininet Network Emulator. URL: <http://mininet.org/>. Visited on January, 2017.
- [10] OpenFlow, <https://www.opennetworking.org/sdn-resources/openflow>. Visited on March, 2017
- [11] POX, "Python-based openflow controller", url = <https://github.com/noxrepo/pox>. Visited on March, 2017
- [12] Ryu, "Ryu is a component-based software defined networking framework", <https://osrg.github.io/ryu/>. Visited on March, 2017

- [13] ONF, "SDN Reference Architecture", https://www.opennetworking.org/images/stories/downloads/sdn-resources/technical-reports/TR_SDN_ARCH_1.0_06062014.pdf. Visited on May, 2017
- [14] B. B. Rodrigues (2016). GreenSDN: energy efficiency in software defined networks. Master's Dissertation, Escola Polit cnica, University of S o Paulo, S o Paulo, 2016.
- [15] NetworkX, "NetworkX is a Python language software package for the creation, manipulation, and study of the structure, dynamics, and functions of complex networks.", <https://networkx.github.io>. Visited in March, 2017
- [16] Iperf, "iPerf3 is a tool for active measurements of the maximum achievable bandwidth on IP networks.", <https://iperf.fr>. Visited in March, 2017
- [17] InfluxDB, "Scalable datastore for metrics, events, and real-time analytics.", <https://influxdata.com>. Visited in June, 2017
- [18] Grafana, "The leading open source software for time series analytics", <https://grafana.com>. Visited in June, 2017
- [19] Lantz, Bob, Brandon Heller, and Nick McKeown. "A network in a laptop: rapid prototyping for software-defined networks." Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks. ACM, 2010.

Abbreviations

GHG	Green House Gas
GreenTE	Green Traffic Engineering
GUI	Graphical User Interface
ICT	Information and Communications Technology
LLDP	Link Layer Discovery Protocol
LPI	Low Power Idle mode
MIP	Mixed Integer Programming
MLU	Maximum Link Utilization
NP-Hard	Nondeterministic, Polynomial time
OS	Operating System
OVS	Open vSwitches
SDN	Software Defined Networks
QoS	Quality of Service
UDP	User Datagram Protocol

List of Figures

1.1	Energy-optimization vs no optimization [14].	2
3.1	High-level view of the GreenSDN architecture [14]	10
3.2	Class diagram of policies and policy manager.	11
4.1	The Facebook topology. In the emulation only hosts $H1$, $H2$, $H3$ are sending traffic to $Sink1$	16
4.2	Per node consumption for a subset of the entire network. The constant value at 1.2 kWh is the max consumption for a switch.	18
4.3	Per node workload for a subset of the entire network. It can clearly be seen that while the merging policy is active only 4 nodes are doing any work, while the others are in sleep modes.	19
4.4	Overall network energy consumption with random traffic: while inside the red dotted interval (energy savings), it is possible to save almost 24% of the energy while the network traffic is low.	20
4.5	Overall network energy consumption while sending constant traffic:	21
4.6	Power saving potential of the Abilene research network.	22
4.7	Power saving potential of the GEANTE research network	22
4.8	Savings potential.	23

List of Tables

- 4.1 Period between '17:51:40Z' and '18:29:51Z' in emulated time. All values approximated to six decimal places. Contains two phases, *Period 1* and *Period 2*, where the controller applies the energy saving policy. The overall average contains values from the entire period as shown in Figure 4.4. . . . 21
- 4.2 Period between '12:04:30' and '12:56:53' in emulated time. All values approximated to six decimal places. Contains two phases, *Period 1* and *Period 2*, where the controller applies the energy saving policy. The overall average contains values from the entire period as shown in Figure 4.5. . . . 22

Appendix A

Installation Guidelines

The following instructions use VirtualBox as virtualization technology, but they should work VMWare as well.

- Import the mininet VM in VirtualBox by clicking on *File, Import Appliance* and selecting the *mininet.ova* file.
- Copy and replace the contents of the *ext* folder provided on the CD inside the */home/mininet/pox/ext* folder.
- Follow the instructions of the *README.txt* file inside the *custom* folder.

Everything is now in place to start the experiment.

A.1 Reproducing the experiment

In order to reproduce the experiment the following steps are needed.

- Use SSH to connect to the guest machine: `ssh -Y mininet@MININET_IP_ADDR`
- When prompted for a password enter: 'mininet'
- Start mininet: `sudo mn -custom /mininet/custom/facebook_topology.py -topo my-topo -mac -controller remote`
- Once mininet has started and is ready, start hosts:
 - Open host sessions: `xterm h1 h2 h3 h18`
 - From *H18* (the sink): `iperf -s -u -p 5000`
 - From *H1*, *H2*, *H3*: run either `/home/mininet/pox/ext/custom/constant_traffic.sh` or `/home/mininet/pox/ext/custom/random_traffic.sh`

- Open an additional SSH session: `ssh -Y mininet@MININET_IP_ADDR`
- Run the pox controller specifying the *facebook topology*: `cd pox && ./pox.py fbcontroller -topo=fb`

Appendix B

Contents of the CD