Department of Information Engineering and Computer Science

Master's Degree in
Computer Science

Final Dissertation

Implementing and running the Digital Twin of a network for energy consumption monitoring and optimization

Supervisor

Prof. Fabrizio Granelli

Candidate

Laura Scoccianti

Academic year 2021/2022

Acknowledgements

Thanks to professor Fabrizio Granelli for the original idea which gave birth to this project.

Thanks to my parents for their support throughout these university years, and sorry it took one too long.

Thanks to Christian for his patience and always being by my side, even in times when neither I was by my side.

No matter which way you ride,
it's always uphill and against the wind.

Contents

Introduction

The ever growing need for wider and more performing networks, in response to constantly increasing demands and traffic load, is exposing a number of issues regarding the costs of maintaining such widespread infrastructures, not just from an economical point of view, but also in terms of resources needed to power such energy-intensive infrastructure.

Studies report that in 2018, more than half of the world population had access to Internet, estimating the number to rise to 75% by the end of 2022 and eventually reaching 90% by the year 2030. These numbers only consider the individual people connecting to the Internet, which must also be added to a staggering quantity of Internet of Things (IoT) devices [1].

With traffic patterns reaching ever higher volumes and levels of unpredictability, a traditional approach to networking is gradually becoming inadequate: the lack of flexibility and scalability, typical of a static way of implementing a network infrastructure, is leading to more complex and difficult-to-manage architectures. Software Defined Networking (SDN) aims at overcoming such limitations, providing a holistic view of the activity and the configuration of a network. The main advantage of this approach lies in the fact that the network becomes centrally manageable and programmable, making it more flexible and responsive to sudden changes in traffic flows, link availability, and new design requirements.

The purpose of this thesis consists in proposing a way of implementing a digital twin of a network that allows for power consumption monitoring and optimization in relation to traffic volumes, by exploiting the instruments provided by SDN.

Several studies, which will be mentioned in Chapter 1, show that part of the waste in energy consumption is due to network devices, or some of their interfaces, lying in a powered-but-idle state, meaning that no traffic is effectively being routed through them. Moreover, it has also been demonstrated that the energy absorbed by an interface is proportional to its bit rate: this may turn into waste, in case the traffic volume is not enough to occupy all the available bandwidth.

A digital twin is the virtual representation of a real system, which is fed with real time data. This allows for the simulation of present and future scenarios based on a model which is constantly updated and validated by live information regarding the status of the real system.

The digital twin developed in this research provides a model of the network, on which it is possible to perform computations that allow for the implementation of mechanisms for the optimization of interfaces working parameters. This has immediate consequences on the energy needed to run the entire network.

In order to implement and test the functionality of the digital twin, some network architectures have been emulated in an open source testbed which supports software defined networking. The developed digital twin is intended to be a starting point for the exploitation of such technology in modern networks: its structure, here used in conjunction with naive optimization strategies, is scalable enough to be used in support of more complex algorithms.

Chapter 1 first gives an insight on the main concepts and advantages of software defined networking, especially in comparison with the traditional approach. The focus is then shifted to the concept of digital twin, presented through a short chronicle that shows how the idea evolved in the last decade and how it gradually made its way in various application fields, with telecommunication still being one of the least explored. Finally, an overview of the costs of running a telecommunication network is given, with particular attention to the aspects related to energy consumption of core network devices, such as switches and routers.

Chapter 2 is divided into two parts. The first one presents the tools used for emulating a software defined network and explains the setup of the testing environment. The remaining of the chapter gives an high level view of the reasoning behind the development of the digital twin: it is explained that data gathered from the real network is used to build a graph whose edges, that represent real links,

are enriched with information regarding the traffic flowing through them. All computations, regarding both routing and energy consumption aspects, are then performed on the graph.

Chapter 3 presents the pseudocode for the SDN application implementing the digital twin, so to put into practice the concepts explained in Chapter 2. This is followed by a detailed description of the parameters for the execution of the application which provide full control over the features of the digital twin.

Finally, the results obtained from various simulation runs are presented. They clearly show that, by using the digital twin to monitor the usage of the links, and to adjust their working bit rate accordingly, it is possible to significantly reduce the overall energy consumption needed by the network.

Eventually, Chapter 4 discusses the limits of the proposed optimization algorithm, which are mainly due to the its dependency on some properties of the target network, such as the topology and the pattern of the traffic. In general, meshed architectures with unpredictable or bursty traffic have reported a significant reduction in power consumption, compared to the same configurations running with a traditional networking approach. Ultimately, this suggests that the developed digital twin can be considered as a starting point for further implementations of optimization mechanisms, that may suit other particular application scenarios.

1 State of the art

This chapter will introduce the current state of the art as regards three basic topics which will be useful throughout this whole work:

1. Section 1.1 describes Software Defined Networking paradigm, its architecture, and its advantages compared to traditional networking;

2. Section 1.2 gives an insight on the concept of Digital Twin, from his origins to nowadays implementations and usage;

3. Section 1.3 provides some basic information on the compromise between energy saving and performance degradation, as it is mentioned in literature.

1.1 SDN: Software Defined Networking

The capacity of network technologies to handle rising traffic loads is rising in response to the ever increasing demands. Nowadays, most of the equipment, from domestic to office use is well into the gigabit per second working range [2].

In the years, also traffic patters have changed and have become more complex: it's not just a matter of supply and demand, rather it's a problem of optimally coping with the different shapes of traffic. The traditional architectures connecting large Ethernet LANs to WAN and Internet facilities are well suited to the client-server model, where traffic is either in one direction or another, with a fair dose of predictability that allows for a rather static configuration of the network [2].

Traditional networks come with a set of limitations, identified by the Open Networking Foundation (ONF) as [3][2]:

- complex architecture, needed to respond for demands regarding quality of service, security, high and variable traffic volumes. Such complexity make the networks difficult to manage and in time caused a number of protocols to be defined for very different requirements and particular portions of networks;
- inconsistent policies addressing security problems in large networks, since it is more and more difficult to configure ACLs as the architecture size grows;
- difficulty to scale because of the static architecture of networks, that leads to inefficiency in handling unpredictable traffic patterns;
- vendor dependence caused by the lack of open interfaces for network programming, meaning that each vendor autonomously develops its equipment to be suitable to a range of constantly rising needs. This results in enterprises usually being bound to a specific manufacturer.

On the contrary, a modern networking approach, that find expression in the concept of Software Defined Networking (SDN), should meet a number of requirements allowing for architectures to be highly maintainable, responsive to changes, and scalable. In addition to this, automation should be introduces in the network in order to reduce the possibility of manual errors when propagating new policies across a wide range of devices. This leads to the concept of model management: the network must be seen as an organic model, rather than a set of individual elements [4][2].

On a more technical plane, in a traditional network architecture, the control logic for routing and forwarding operations is embedded in devices such as routers and switches. Each device uses protocols (e.g. BGP, OSPF, etc.) to interact with its neighbors and to gather information needed for packet forwarding.

Software Defined Networking (SDN) is a network paradigm that decouples the network control and forwarding functions, introducing the possibility for the network to be programmable, while

abstracting the underlying infrastructure. In SDN, a device is no longer accountable for the population of its forwarding table, since there is a central entity which overlooks the network and fills the tables used by all devices (Figure 1.1) [5].

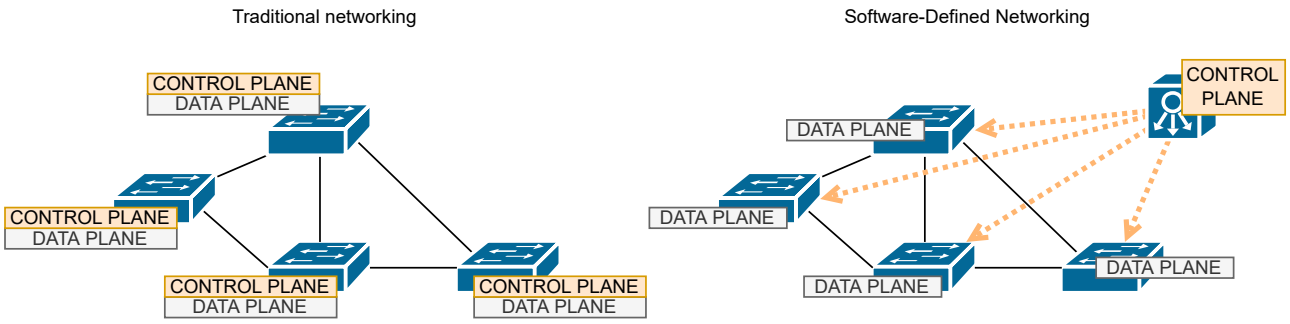


Figure 1.1: SDN allows for the decoupling of control and data planes

A SDN architecture is composed of three different layers (Figure 1.2): the bottom one contains the forwarding hardware (e.g. switches), the middle one is the SDN controller, sometimes called Network Operating System, and the topmost one hosts the running applications. Such applications define how the network works by communicating their requirements to the controller via the northbound interface, also called network control API [6].

The controller interacts with the switches in the hardware layer via the so-called southbound interface. Sometimes this is called hardware open interface because it is compatible with all devices of any vendor, provided that they support the SDN approach. Since SDN is a paradigm, it is possible to implement it with different tools, the most used and documented being the OpenFlow protocol. OpenFlow is a standard for the communication between data plane and control plane, and it is nowadays supported by a wide majority of the most important vendors of networking devices. This topic will be discussed more in details in Chapter 2 [6].

The controller is in charge of translating high level instructions coming from the applications into low level configurations for the network nodes. At the same time, it must also translate information coming from the hardware so that applications can use them to program the network. From a certain point of view, the controller allows for a filtering of the information coming from the physical network, so that the applications only have to deal with a simplified view of it [7].

The central concept, and main advantage, of the SDN paradigm is facing problems in a global manner, via applications that interacts with the controller through an interface. This northbound interface is hopefully standard, thus making every solution portable and usable by any SDN using it [6].

1.2 Digital Twins

A Digital Twin can be formally defined as the union of three elements:

1. A real space containing a physical object;

2. A virtual space containing a virtual object;

3. The link for data flow from real space to virtual space and vice versa.

The core idea consists in simulating a real system by gathering real time data from it, in order to verify and validate both the models used to develop the real system and the consequences of future modifications in various conditions. In fact, digital twins are used mainly for real time optimization and risk evaluation.

A digital twin may also include machine learning algorithms to elaborate data gathered from the physical twin.

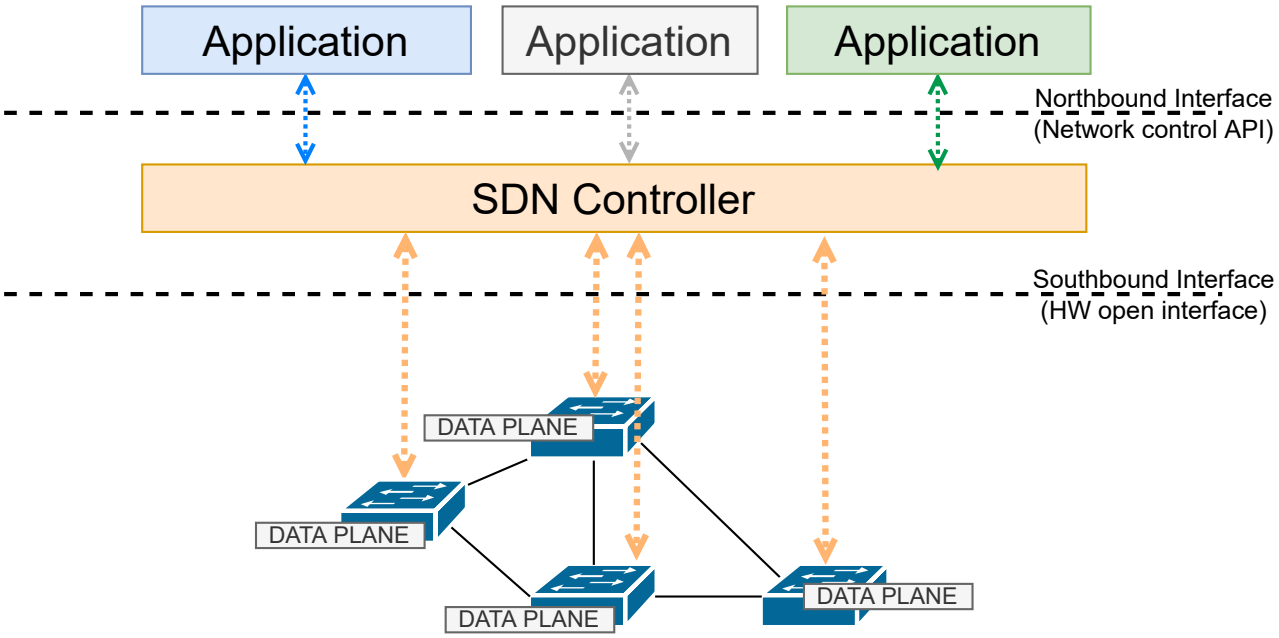


Figure 1.2: SDN architecture

1.2.1 A brief history

The concept of Digital Twin was mentioned for the first time at the beginning of the 2000s, but it took a decade before its first practical application was introduced by American Institute of Aeronautics and Astronautics for NASA and U.S. Air Force. At that time, the two American institutions needed to address the problem of computing some precise factors-of-safety: the classical predictive approach was insufficient, since it was based on similitude between the circumstances in which the statistics were obtained and the future unforeseeable conditions [8].

As reported in [9], Digital twins can be described as ultra-realistic simulations of real objects, investigating multiple interdependent aspects. They would allow for a simulation that takes into account not only the existing models and statistics, but also the real time data gathered during the whole life cycle of the real object, from the beginning of its production to the end of its life. The main goal for NASA and U.S. Air Force was to continuously forecast the health of their systems, along with the remaining life and, eventually, the success probability of the mission. On top of this, a digital twin could also allow for the prediction of a response to a critical event, with the purpose of mitigating damages and degradation through the activation of healing mechanisms or the suggestion of changes in the parameters' values.

At the time the concept of digital twin was introduced, digital representations of physical products were still new and immature, mainly because most data gathering was still manual and computational capacity wasn't comparable to that available at present day. Because of this, what is considered to be the white paper of Digital Twin was published only in 2014 by Dr. Michael Grieves, who already introduced the idea at the beginning of 2000s [10].

In the last decade, also thanks to the increasing popularity of the Internet of Things, the use of digital twins exploded: nowadays there are billions of interconnected sensors that allow for data gathering and eventually for building digital twins of billions of systems.

1.2.2 Applications and challenges

The concept of digital twin has applications in various fields, from automotive to aerospace, from economy to information technology [8].

In most cases, the term digital twin refers to a network of digital twins: every component of a complex system has its own digital twin, that is interconnected with the others and contributes to the creation of the twin of the whole system.

Implementing a Digital Twin comes with a considerable number of challenges to overcome. Apart from application-specific issues, general problems come from data handling, sensor distribution and

computational load [8].

Especially when dealing with complex systems, a huge amount of data is produced: the link between physical and digital object should ensure enough throughput in both directions. When dealing with critical systems, also confidentiality of the data may be important, so that some security infrastructure should be involved [8].

Another important requirement consists in using reliable predictive models. In order to do so, sensors must be distributed so to gather relevant data. However the more data is collected, and therefore processed, the more computational power is needed: information should be filtered in order to get just what is needed, without polluting it with useless information.

Last but not least, also scalability should be taken into account: since a digital twin will ideally accompany the physical system for its entire life cycle, it is reasonable to imagine that the physical twin may evolve with time.

1.2.3 Digital twins in the telecommunication field

The use of Digital Twins in the telecommunication field isn't yet mature like that in other industrial sectors (e.g. aerospace), but it's gaining popularity. Applications are mainly focused on optimization and routing problems of mobile networks.

The basic idea is that making modifications in production networks is too dangerous, so most applications of the digital twin paradigm involve the use of deep or reinforcement learning to determine some threshold values for the system. New configurations based on the computation made on the Digital Twins are given a score, or reward, and the decision on the production network is eventually taken by maximizing such a reward. This quantity may be a computed value, or a raw predicted value (e.g. throughput, latency, etc.) [11].

1.3 Power usage of network devices

In general, the costs of operating a network infrastructure in time is larger than the cost of creating it. Surveys and studies, also published by telecommunication companies, show that in multi-year projections the operational expenditures (OpEx) may amount to as much as nine times the capital expenditures (CapEx). Studies also show that 15% to 50% of the OpEx of running a network is due to the cost of energy (Figure 1.3 by Huawei).

The overall picture becomes even more dramatic when considering mobile networks. Statistics show that most of the deployed infrastructure is over-dimensioned and under-utilized in regular traffic conditions. Figure 1.4 shows that one third of the energy consumption is attributable to idle network resources, and in general, only 15% of all the required power is involved in the transmission of data [12].

Despite the rising interest in greening the Internet, a rather small amount of data and researches on the power consumption of network devices is available. The existing literature aims at providing methods to establish the power consumption of routers and switches, while discussing ways of implementing energy saving policies [13].

Experimental data collected from various devices show that, in current networking devices, half of the energy is required by the base system functions alone, while the remaining half strongly depends on the number of active line card interfaces, even if idle. In fact, the data show that the traffic load does not affect too much the quantity of required energy, and suggest that a possible solution to the energy saving problem might consist in temporarily switching off some interfaces or all nodes in the network. This is however unpractical both from an infrastructural and an economical point of view. First, switching off entire links in large networks may eliminate the redundancy needed to provide reliability to the service. Second, the investment is made useless by limiting or excluding the usage of the expensive transmission links [14].

Data taken from a commercial Ethernet switch from one of the major manufacturers show that a single 1000baseT interface counts for 1.8W of the overall energy consumed by the device (Table 1.1)[14]. Studies show that up to 4W may be saved by lowering the bit-rate of interfaces from 1 Gbps to 100 Mbps. For this reason, one of the solutions proposed in early 2000s by the Ethernet Alliance and IEEE 802.3 Energy Efficient Ethernet Study Group is that of Adaptive Link Rate (ALR) [15].

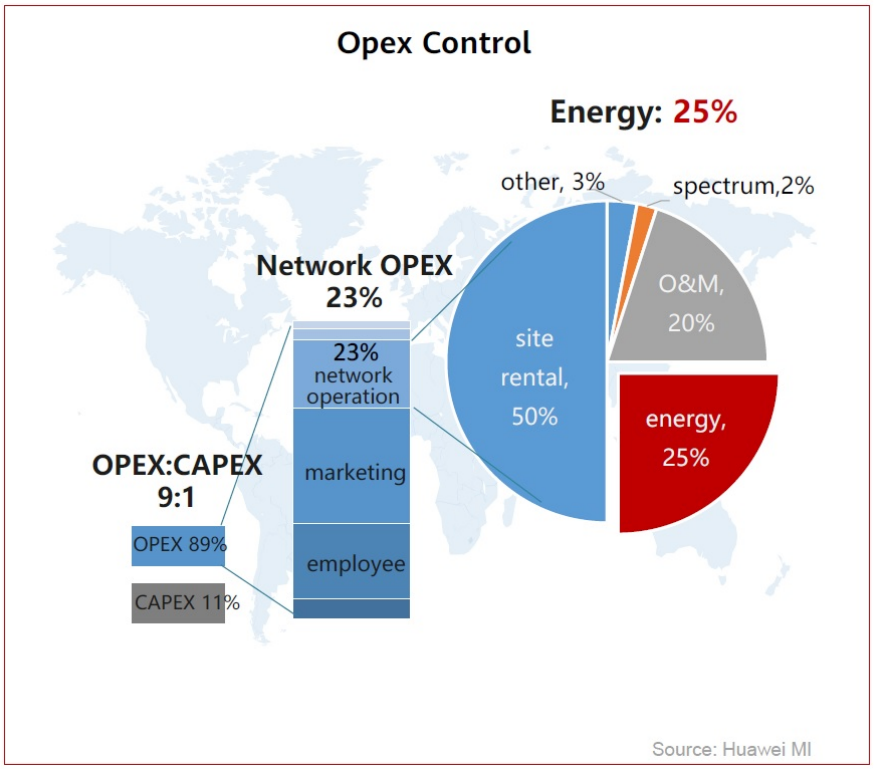


Figure 1.3: Distribution of OpEx and CapEx as estimated by Huawei

[thick,font=] [font=]**Energy consumption distribution in mobile networks;**
[thick,font=] [radius=2.5,rotate=90,text=legend]15/Energy for transmitting data, 36/Redundant and idle devices, 25/Temperature control lighting UPS etc., 10/Power loss due to transmission lines, 14/Fans and power supplies

Figure 1.4: Energy consumption distribution in mobile networks [12]

Active interfaces	10Mbps	100Mbps	1Gbps
0	69.1W	69.1W	69.1W
2	70.2W	70.1W	72.9W
4	71.1W	70.0W	76.7W
6	71.6W	71.1W	80.2W
8	71.9W	71.9W	83.7W

Table 1.1: Power required by a commercial Ethernet switch depending on active interfaces at their working speed.

Changing link rates comes with a number of problems, the most obvious being development of a control policy to determine when to make changes, also considering the unpredictability of network traffic. Moreover, ALR is implemented at layer 2, so it is necessary that all involved network nodes are aware of it, to allow its application on the network. To account for this, ALR must be advertised on the link via a handshake mechanism. One last thing to consider is that MAC and PHY layers must synchronize every time the link rates are changed, thus increasing the time needed for ALR to function.

Experimental studies show that it is possible to achieve significant energy saving by using ALR, despite some performance degradation in terms of packet delay and buffer overflows [16].

This research is focused on an improvement of this technique exploited via SDN, which relieves the hardware of being technology-aware and allows the implementation of the control policies on a whole network scope rather than just the single link, like in ALR.

2 Development of the Digital Twin

This chapter introduces the technical background needed for the development of the project. Then the focus is moved on how the digital twin and its core functionalities for energy consumption optimization have been implemented.

2.1 Technical background

This section illustrates the tools at the core of the project presented in this research. All the following content is derived by the official documentations of Mininet [17], Ryu [18] and OpenFlow [19].

2.1.1 Mininet

Since it was not possible to run this research on a real network, it has been chosen to use the opensource network emulator Mininet [17], which allows for the virtualization of various network architectures featuring switches, hosts and routers, on a single machine.

Mininet hosts run Linux-based OS, so they can run some programs and bash scripts, while switches support OpenFlow, allowing Software Defined Networking.

On top of the advantages coming from the ability to emulate complex networks, Mininet includes a CLI for debugging and running tests, and also provides a Python API for network creation and experimentation.

Last but not least, since Mininet networks run the real Linux kernel and network stack, the code developed for an OpenFlow controller and tested on the emulator can be easily ported to a real system for real-world testing and deploying.

2.1.2 SDN Controller: Ryu

The opensource networking framework Ryu [18] has been chosen as the controller for this project. It provides software components with largely documented APIs for the development of network management applications. The component approach also allows developers to easily modify or implement their own applications from scratch, in order to ensure the network can meet their requirements.

Some Ryu applications provide REST APIs for retrieving stats from the network, as well as modifying its behavior.

The Ryu Controller supports various protocols, included OpenFlow, which is the most established SDN communication standard.

2.1.3 OpenFlow

As already mentioned in Chapter 1, OpenFlow is a protocol that allows the interaction between the controller and the switches.

Switches in a software defined network are managed by a SDN controller through the exchange of messages that use the OpenFlow protocol. These messages are used to send all the information regarding a switch, from its MAC address to the status of its interfaces, as well as to allow the controller to modify the device behavior, by managing traffic flows, queues, and so on.

Some of the main components of an OpenFlow Switch are the flow tables, one or more, and they are sequentially numbered. Each flow entry in the tables is composed by [19, Section 5.2]:

- match fields: the fields of the packets' header are checked against these. They can be data from layers 2 (MAC addresses), 3 (IP addresses), and 4 (ports) of the ISO/OSI stack;
- priority: determines the precedence of an entry;
- counters: they are updated according to packet matches;
- timeouts: the time, either maximum or idle, after which an entry is removed from the table;

- cookies: data used by the controller to manage the table, but they do not affect the handling of the packets;
- instructions: actions that are performed based on the matched fields, e.g. packet forwarding/-dropping, enqueueing of packets in buffers which allows for quality of service mechanisms, etc.

When a switch receives a packet, its header fields are checked against the flow table of the switch itself. If a match is found, an action is performed accordingly, otherwise the packet is forwarded to the controller, which will use the information contained in the header to create new rules.

In the case of multiple flow tables in the same switch, matches may be concatenated, meaning that a match in the first table may lead to a lookup and subsequent match in the second table, and so forth.

In the case no match is found, the behavior is determined by the table-miss flow entry: if it does not exist, packets are dropped by default. It is not unusual that the table-miss entry is used to send packets to the controller.

The SDN Controller, according to the instructions coming from the SDN application, may add, modify, or remove flows in the tables.

The extract below shows the last two entries of a flow table. The first one states that a packet which was received on port 2 of the switch, coming from source MAC address `dl_src` and heading to destination MAC address `dl_dst`, must be forwarded to port 1. The latter is the table-miss entry used to send all uncategorized packets (notice the empty match section) to the controller for further processing. The field `duration_sec` shows the amount of time the entry has been in the table, while the timeout fields, which in this case are undefined (default value 0), can be set to allow the expiry of the entries.

```
...
{
    "priority": 1,
    "cookie": 0,
    "idle_timeout": 0,
    "hard_timeout": 0,
    "byte_count": 140,
    "duration_sec": 167,
    "duration_nsec": 421000000,
    "packet_count": 2,
    "length": 104,
    "flags": 0,
    "actions": [
        "OUTPUT:1"
    ],
    "match": {
        "in_port": 2,
        "dl_src": "2a:df:20:79:64:cc",
        "dl_dst": "76:1d:03:dc:c4:91"
    },
    "table_id": 0
},
{
    "priority": 0,
    "cookie": 0,
    "idle_timeout": 0,
    "hard_timeout": 0,
    "byte_count": 12972,
    "duration_sec": 177,
    "duration_nsec": 990000000,
```

```
        "packet_count": 184,
        "length": 80,
        "flags": 0,
        "actions": [
            "OUTPUT:CONTROLLER"
        ],
        "match": {},
        "table_id": 0
    }
```

Quality of Service via OpenFlow queues Quality of Service in general is a set of technologies and parameters which are used in order to meet certain requirements in the allocation of network resources.

OpenFlow switches provide Quality of Service support through queueing mechanism. One or more queues can be attached to a port. When a flow entry is mapped on a specific queue, it is treated according to the queue's configuration [19, Section A.2.2].

Queues are mainly used to guarantee minimum and maximum bandwidth for traffic exiting the port and they apply both traffic shaping and policing: packets can be either dropped or buffered to prevent exceeding the bandwidth limit.

Unlike flow tables, that can be managed by a SDN controller, but are otherwise managed by the OpenFlow switch, queues cannot be handled autonomously by the switch, since OpenFlow is only able to query and provide statics from the device. If it is necessary to modify the behavior of the switch queues, a SDN controller and applications, are mandatory.

2.2 Environment setup

Both Mininet and Ryu have various dependencies to be installed. For this reason, it is most convenient to use a ready-made virtual machine. For this project, the choice fell to ComNetsEmu [20], a testbed and network emulator that allows for SDN emulation, research and development.

Mininet and Ryu both execute inside the virtual machine, while the SDN applications are located both in the VM and in the host system. Those in the VM interact with Ryu via its Python API, while the others use the REST API (Figure 2.1).

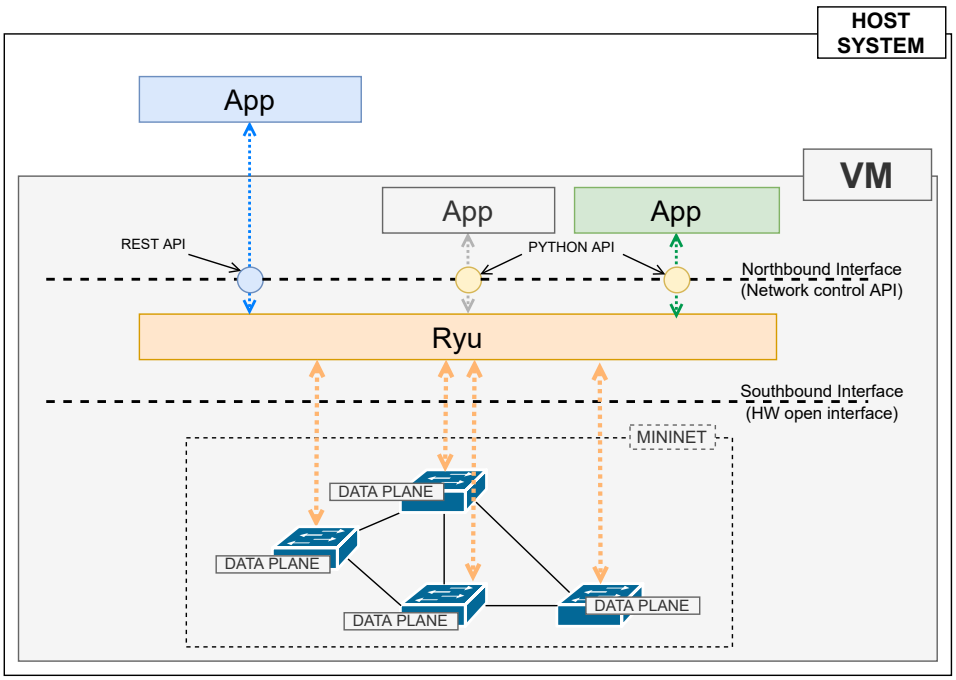


Figure 2.1: Project environment

2.2.1 Building and running a network with Mininet

The Mininet Python API [21] allows for a quick setup of any desired virtual network architecture.

For this project different topologies are used, all composed of five switches, eight hosts and a controller (Figure 2.2). The switches are defined as Open vSwitches, an open-source implementation of a virtual multilayer switch, which operates at layer 2 of the OSI stack and provides some functionalities from higher layers. Open vSwitch was created with the intent of providing network automation through programmatic extensions, while supporting standard management interfaces and protocols. Open vSwitch can work both on virtualized environments and hardware distributed architectures [22].

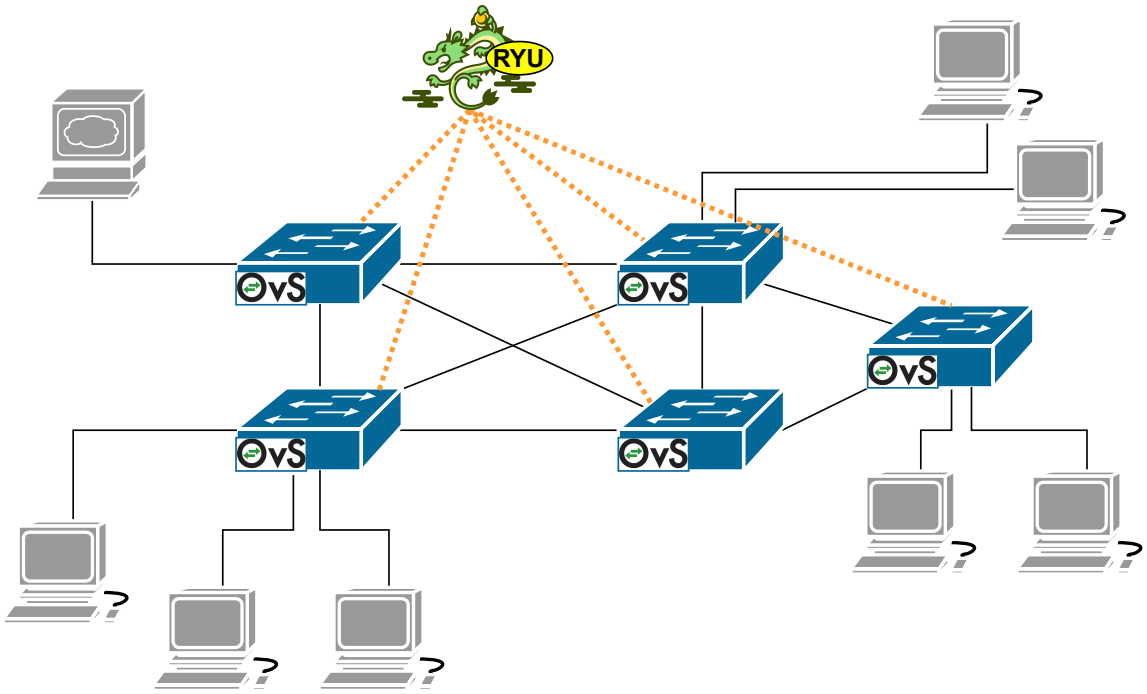


Figure 2.2: One of the topologies used in the project

Via the Python API, switches are instantiated as nodes of class `OVSSwitch`, while 1.3 is the chosen OpenFlow version. All nodes are assigned a name and a `dpid` for easier usage later in the project. When adding hosts, also their IP address is defined. (Script 2.1).

Listing 2.1: Instantiating switches and hosts

```
1  s1 = net.addSwitch('s1', dpid="1", protocols="OpenFlow13", cls=OVSSwitch)
2  s2 = net.addSwitch('s2', dpid="2", protocols="OpenFlow13", cls=OVSSwitch)
3  s3 = net.addSwitch('s3', dpid="3", protocols="OpenFlow13", cls=OVSSwitch)
4  s4 = net.addSwitch('s4', dpid="4", protocols="OpenFlow13", cls=OVSSwitch)
5  s5 = net.addSwitch('s5', dpid="5", protocols="OpenFlow13", cls=OVSSwitch)
6
7  server = net.addHost( 'srv', ip='10.0.0.100', dpid="100")
8  h1 = net.addHost( 'h1', ip='10.0.0.1', dpid="111")
9  h2 = net.addHost( 'h2', ip='10.0.0.2', dpid="112")
10 h3 = net.addHost( 'h3', ip='10.0.0.3', dpid="113")
11 h4 = net.addHost( 'h4', ip='10.0.0.4', dpid="114")
12 h5 = net.addHost( 'h5', ip='10.0.0.5', dpid="115")
13 h6 = net.addHost( 'h6', ip='10.0.0.6', dpid="116")
14 h7 = net.addHost( 'h7', ip='10.0.0.7', dpid="117")
```

Mininet offers various options for controllers, both built-in or remote: in this case, a remote one, which is Ryu, is instantiated in the topology building script (Script 2.2).

Listing 2.2: Definition of the Ryu controller

```
1 c0 = RemoteController("c0", ip="127.0.0.1", port=None)
2 net.addController( 'c0' )
```

After defining the network nodes, links between them must be added. For the project, basic connections are used, even if the API offers the class `TCLink`, which provides the setting of traffic control parameters such as delay, bandwidth, etc. Also switch interfaces may be specified (e.g. line 1 of Script 2.3)

Listing 2.3: Connecting switches and hosts

```
1  net.addLink(s1, s2, port1=1, port2=1)
2  net.addLink(s1, s3)
3  net.addLink(s2, s4)
4  net.addLink(s2, s5)
5  net.addLink(s3, s4)
6  net.addLink(s1, s4)
7  net.addLink(s5, s4)
8  net.addLink(s2, s3)
9  net.addLink(server, s1)
10 net.addLink(h1, s3)
11 net.addLink(h2, s3)
12 net.addLink(h3, s3)
13 net.addLink(h4, s4)
14 net.addLink(h5, s4)
15 net.addLink(h6, s5)
16 net.addLink(h7, s5)
```

As a last step, the switches must connect to the controller (Script 2.4).

Listing 2.4: Connecting switches to the controller. Network start up.

```
1 net.build() # Build the topoology
2 c0.start()  # Start the remote controller
3 s1.start( [c0] )    # Start the switch and connect to the controller
4 s2.start( [c0] )
5 s3.start( [c0] )
6 s4.start( [c0] )
7 s5.start( [c0] )
8 net.start() # Start all other nodes in the network
```

At this point, the network is already fully functioning and it is possible to run commands in the nodes CLI.

Traffic in the network will be emulated via iPerf, so it is necessary to prepare all hosts to listen to that type of connections. The code shown in Script 2.5 is equivalent to running the command in the shell of the specified host and it sets up a iPerf server ready to receive connections.

Listing 2.5: Setting up iPerf servers

```
1 info(server.cmd("iperf -s -i1 &"))
2 info(h1.cmd("iperf -s -i1 &"))
3 info(h2.cmd("iperf -s -i1 &"))
4 info(h3.cmd("iperf -s -i1 &"))
5 info(h4.cmd("iperf -s -i1 &"))
6 info(h5.cmd("iperf -s -i1 &"))
7 info(h6.cmd("iperf -s -i1 &"))
8 info(h7.cmd("iperf -s -i1 &"))
```

After all the set up has been done, the interactive Mininet shell is provided to the user and the real simulation can start.

The full code (Script A.1) is available in Appendix A.

2.2.2 Traffic generation

As anticipated in the previous section, traffic will be emulated via iPerf tool.

iPerf is a tool for measurement and tuning of network performance. It can generate data streams, either TCP or UDP, while the typical output contains the amount of data exchanged and the throughput.

Among the parameters available to tune the connection, this project will use:

- `-i` for setting the interval in seconds between connection statistics reports;
- `-c` for specifying the IP address of the destination host;

- `-n` for indicating the amount of data to be sent.

As an example, typing:

```
h6 iperf -c 10.0.0.2 -n 1500M -i2
```

in the Mininet CLI, means that host `h6` will send about 1.5GB of data to the host having IP address 10.0.0.2, and reports about the performance will be given every two seconds.

2.2.3 Mininet limitations

Recalling the objective of this project, which is optimizing the energy consumption of a given network by:

a) keeping a live model of it,
b) identifying the most convenient links to be used,
c) disabling switches interfaces of unused links,
d) adapting the bit rate of active ports depending on traffic volume,

it is important to notice that the solution proposed in this project is strongly influenced by a limitation that comes with the use of Mininet. The correct way to address point d) would be by acting directly on the interface: this would be possible on a physical network running OpenFlow switches. Unfortunately, in Mininet the physical interfaces are emulated by virtual Ethernet interfaces, and the speed of this kind of interfaces is hardcoded in the Linux kernel to be 10 Gbps [23]. Since it would be impractical to modify the Linux kernel, the solution proposed in the following pages uses traffic control and quality of service mechanisms to emulate the actual change of bit rate of the interfaces.

2.2.4 Using Ryu to manage the network

Ryu source code [24] comes with some applications, e.g. `simple_switch_13.py`, that provide fundamental functions to have the switches correctly process traffic flows, while others provide REST API endpoints that allow for an easier implementation of software to interact with the underlying network.

In order to exploit the quality of service mechanisms, as mentioned in Section 2.2.3, the following applications must be started on the controller when initializing the network:

- `rest_qos.py` provides a REST API for the managing of queues in OpenFlow switches;
- `qos_simple_switch_13.py` is a slightly modified version[1] of `simple_switch_13.py` that provides the logic for the management of flow tables;
- `rest_conf_switch.py` provides a set of REST API for switch configuration;
- `gui_topology.py` provides a set of REST API for gathering information and statistics about switches usage, as well as an interactive graphical representation of the network.

All the Ryu applications get started when the network is built, via a bash script (Script 2.6) invoked at line 34 of the example Script A.1. Ryu produces some logs that, in the case shown in the script below, get redirected to a file hosted in a shared folder between the virtual machine and the host system.

Listing 2.6: Start Ryu applications

```
1 #!/bin/bash
2 # Start ryu controller application
3 printf '*** Starting RYU Controller on localhost ***\n'
4 ryu-manager ryu.app.rest_qos qos_simple_switch_13.py ryu.app.rest_conf_switch ryu.app.
      gui_topology.gui_topology --observe-links &> /home/shared/ryu-logs.log &
5 sleep 10
```

[1] Official Ryu guide for using QoS functions `https://osrg.github.io/ryu-book/en/html/rest_qos.html`

2.3 Digital Twin for energy optimization

In order to monitor the energy consumption of a network and deciding which actions to take, it is necessary to model the network in such a way that could keep track of the configuration changes as they happen.

The concept of Digital Twin, introduced in Chapter 1, comes in handy: by keeping a model of the network, based just on the features of interest, it is possible to instantly perform updates according to the network needs, with a fair confidence margin since the computations have been made on a model that is fed with live data.

2.3.1 Digital Twin of the network

The digital twin of the network keeps a model of it, based just off the needed features. For this reason, since the goal is maximizing the energy saving by acting on the interfaces of the devices, the network is represented by a graph.

The nodes of the graph are the switches, while the edges, which represent the links (Figure 2.3b), are associated to a series of dictionaries keeping information such as: physical ports constituting the link, amount of data exchanged, working speed and status of the interfaces (e.g. idle, in use, saturated).

The graph is then enriched with information about links connecting switches and hosts: there is no need to keep track of the latter devices by adding them to the list of nodes, but it is important to know which interfaces of the switches are connected to them.

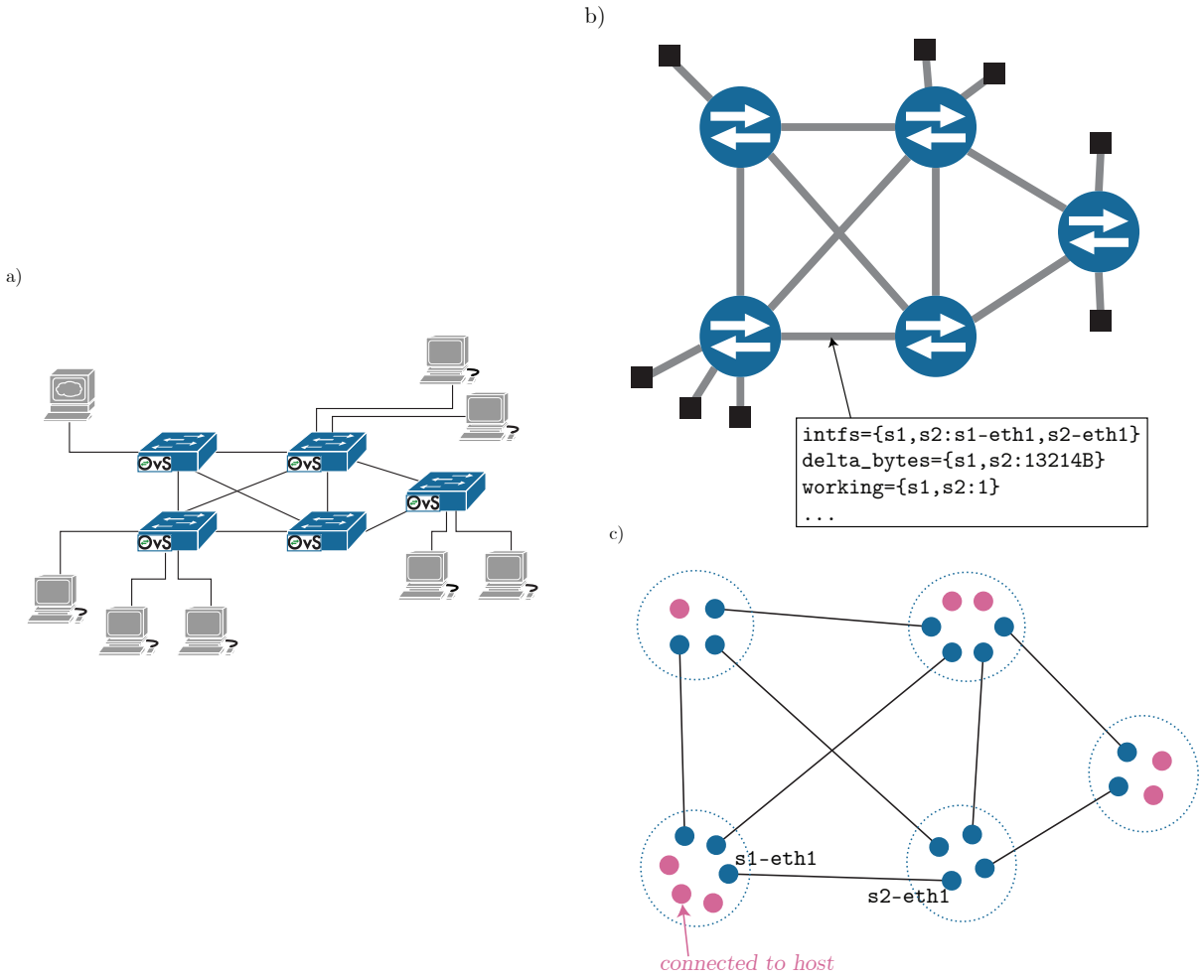


Figure 2.3: a) is the real network; b) is the summarized model; c) is the interface-centric model. Both b) and c) represent the same information in the end.

Actually, this is a summarized view of what the digital twin aims at modeling: all the reasoning that will be explained in the following sections is applied to every single interface, so the most correct

and complex way of keeping a model of the network would be by building a graph whose nodes are the interfaces (Figure 2.3c). The graph should then undergo a clustering process to group the interfaces belonging to the same switch, and every node should be marked with a number of properties needed to examine the behavior of the network and take decisions accordingly. Anyway, this would add too much computational complexity compared to the possibility of dealing with pairs of interfaces, hence the choice to simplify the model to the graph previously described.

2.3.2 Routing aspects

When there is more than one layer 2 path in a network, a switch loop occurs. Such occurrences may cause unpredictable routing behavior or, worse, a network failure. It is a common practice, especially in complex architectures, to have redundant links to ensure availability: in these cases, the Spanning Tree Protocol (STP) is used to break loops in the network by blocking some paths.

During the set up of the digital twin, that represents the real network through a graph, a spanning tree of the network is built in order to determine which links to use.

Each node N in the graph is given a score

$$score_N = deg(N) \cdot 2 + hosts(N)$$

where the degree of the node is the number of connections towards other nodes, while $hosts(N)$ is the number of connected hosts, that were previously stated to not be considered in the node count.

The switch with the highest score is elected as the root of the spanning tree, that is then built following a breadth-first search visit order of the nodes.

After the spanning tree is built, if a switch becomes a leaf with no hosts directly connected to it, it is marked as a candidate to be turned-off since no traffic would and should ever be routed through it. In practice, completely shutting off a device would be unpractical because, in the case the traffic should suddenly be routed through it, it would be necessary to wait for its boot up time, that may last several minutes depending on the product. A feasible option would be that of putting the device in hibernation mode, if made possible by the vendor.

All of the interfaces involved in links that have not been added to the spanning tree may be disabled as well.

2.3.3 Data gathering

Ryu provides a considerable amount of information about the network that needs to be filtered in order to get just the needed data, so to keep the computations as lightweight as possible. And besides that, since data is gathered via REST calls that take up a significant amount of time, it is also necessary to minimize them, so that the maximum amount of information is gathered with a minimum amount of requests.

HTTP requests are performed by the main application script via cURL. One of the peculiarities of cURL is that it provides the use of multi handles that allow to perform multiple parallel transfers, all done in the same thread[2].

Apart from a limited number of requests necessary to build the digital twin and initialize the data structures connected to it, the actual periodic data gathering is achieved with a maximum number of N calls, where N is the number of switches, plus a multi-request that is used to eventually modify the network behavior.

In details, the periodic calls are N GET requests to `http://localhost:8080/stats/port/<dpid>` which return statistics about ports usage for each switch; these data are processed so to have a periodic report of the amount of traffic traversing the interface.

All REST handles are provided by the Ryu applications described in Section 2.2.4, while `localhost:8080` is where the endpoint is located based on the virtual machine network configuration.

2.3.4 Optimization algorithm

After feeding real time data to the digital twin and updating the status of its links, the new information given by it is used to decide if the bit rate of the interfaces in the real network has to be changed.

[2]Multi-request as explained in official cURL documentation: https://everything.curl.dev/libcurl/drive/multi

A port can be either:

- *Disabled*: no traffic is routed through it and its bit rate is zero;
- *Idle*: the port is active, but no traffic is passing through it, so its bit rate is minimum (10 Mbps);
- *Working*: traffic has been detected in the last interval considered;
- *Saturated*: the amount of traffic is above a threshold value.

A *saturated* port is effectively a *working* port, with the difference being that in the latter case no action will be performed on the real network, while in the first one the bit rate of the port will be increased.

Depending on the chosen threshold value, the system may be more conservative towards energy saving, rather than prone to providing better network performance. The threshold value is calculated as follows:

$$thr = sensitivity \cdot port\_speed$$

where *sensitivity* is a percentage of the theoretical amount of bytes that is exchanged in a time unit at a 1 Mbps, and *port_speed* is the current bit rate of the considered port expressed in Mbps. For instance, if the time unit is one second, sensitivity will be a fraction of 125000 B.

It follows that a higher value of *sensitivity* results in a tendency to maintain the bit rates stable, even in presence of irregular and bursty traffic. On the other hand, a lower threshold yields more frequent bit rate changes, that might have some advantages with stable traffic conditions, but could probably cause some energy waste with highly irregular traffic.

An additional parameter to be set is the amount of time units a port must spend in *saturated* state before its bit rate is increased to the next step available.

Since the bit rate update has effect on the next time unit, and is based on a sampling from the traffic in the current unit, it may be inconvenient to promptly react to what could be considered an outlier value.

Figure 2.4 shows the difference between delaying, or not, the update of the bit rate: in case a), where a port needs to be saturated for more than a single time unit, the burst is processed without the need of a higher bandwidth, while in case b) the port is immediately given more bit rate, which results in a higher, and useless, threshold for the next time units. Since the threshold depends on the ports speed, a higher value comes with higher energy usage. This is the reason behind the usage of the first approach.

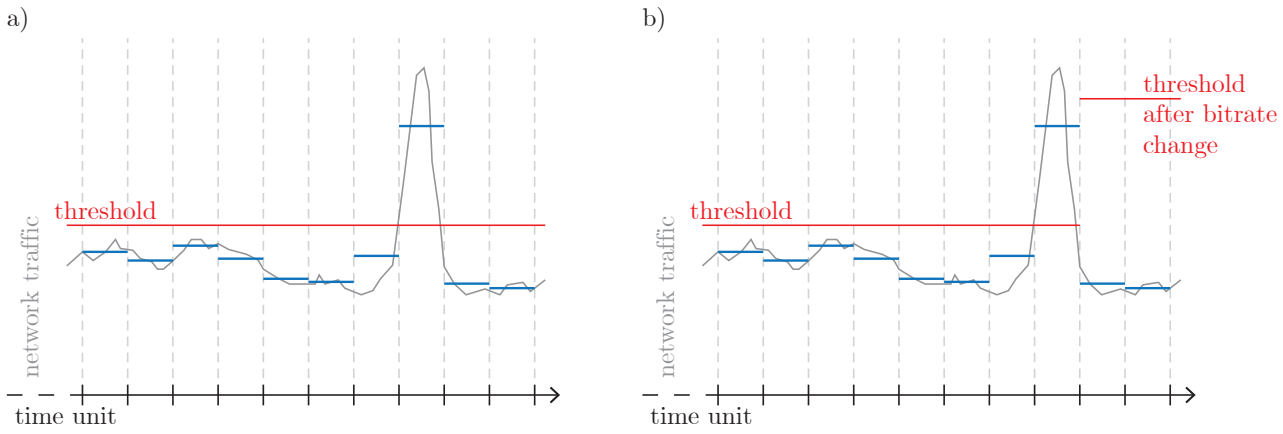


Figure 2.4: Blue lines represent the sampling of the traffic in the time unit. In a) the bit rate update is delayed, while in b) it happens as soon as the threshold is exceeded.

On the opposite, it is sufficient that a port is detected as *idle* for just one time unit to have its bit rate decreased to the minimum possible. This is possible because, without additional shaping mechanisms, a traffic flow will always try to saturate the bandwidth: a sudden drop in the port usage can be considered as the end of transmissions for the flow that was occupying the majority of the available bandwidth.

2.3.5 Code structure

Figure 2.5 shows the structure of the code that was developed in order to realize what has been described in the previous sections.

There are two main aspects to notice: the first one being the way the digital twin is created and then maintained, and the latter being the main loop which implements the energy optimization.

The software for energy optimization can be divided in two parts:

1. a preparatory phase where the digital twin is created and routing problems are addressed; in particular, the network topology is transformed into a graph, the spanning tree is defined, unnecessary links are deactivated and information about the active ports bit rate is retrieved;
2. the main loop where the actual computations are performed in order to run the digital twin and modify the network behavior: during each cycle, statistics about the interfaces are retrieved in order to update the status of the ports and change their bit rate, if needed.

The digital twin is composed of four data structures: the links memorized as a dictionary of interface pairs, the list of active interfaces selected by the spanning tree algorithm, a dictionary where the bit rate of each port is saved and another dictionary where the status of each interface is kept. Among these four data structures, the first three are populated during the preparatory phase of the energy optimization software.

The dictionaries regarding the speed and the status of the interfaces are updated at each cycle of the main loop. In particular, the ports status is retrieved from the actual network, while the current bit rate is obtained from the computations performed in the "compute bit rate change" state.

As a last feature, during "compute energy" state, live data about energy consumption of each switch in the network is logged, and such information can be used to plot and report statistics in a given time interval, e.g. at the end of a simulation run.

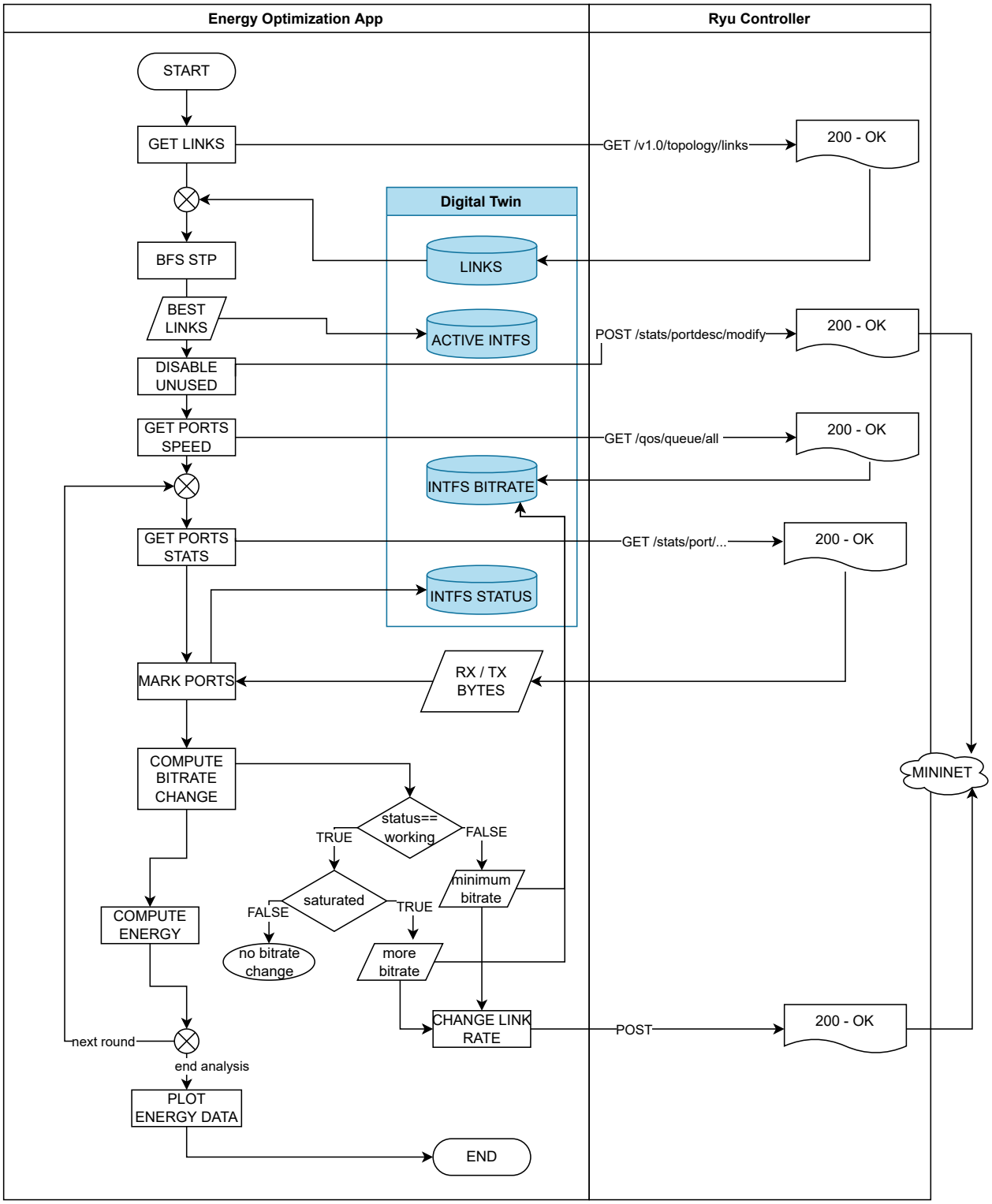


Figure 2.5: Complete scheme of the developed SDN Application for the implementation and execution of the digital twin for energy optimization.

3 Testing the Digital Twin

This chapter presents an efficient strategy for the use of a digital twin with the goal of optimizing the power consumption of a network architecture. The first sections will delve deeper in the concepts expressed in the previous chapter, providing a pseudocode for the implementation and running of the digital twin. Subsequently, some evaluation scenarios will be presented, while the main results will be eventually commented and evaluated.

3.1 Preparation for the tests

Section 2.3 introduced the reasoning behind the implementation of the SDN application, which is needed to build and run the digital twin of a network. This section will provide technical details regarding the code developed for this research.

3.1.1 Pseudocode

The software was developed in Python: its already comprehensive standard library, enriched by a huge amount of libraries developed by the community, and its legibility made that language the most suitable choice for the project presented in this dissertation.

In the initial phase of the software, some global variables and data structures are defined, along with the setup of switches to allow the use of quality of service mechanisms, as mentioned in Section 2.2.3. The data structures are divided in those composing the digital twin of the network, already discussed in Section 2.3.5, and some that store temporary data needed for correctly processing the gathered information. Finally, there are global variables that are used to set some working conditions of the optimization algorithm, as presented below:

- `BASE_POWER` is the value in Watts, representing the minimum power needed by a powered-up switch; this is mainly needed to plot the results;
- `INITIAL_SPEED` is the value in Mbps of the minimum bit rate of an active port;
- `SENSITIVITY` is the threshold value needed to change the port status, as explained in Section 2.3.4, and represents the number of bytes exchanged per time unit at a fixed bit rate of 1 Mbps;
- `ADAPTIVE_BITRATE` is a boolean variable which allows, or denies, the change of the ports' bit rate based on their status;
- `DISABLE_UNUSED` is a boolean variable that determines if unused ports may be shut down; it also applies to switches that are not involved in the routing of traffic, as anticipated in Section 2.3.2;
- `MAX_10G` is a boolean variable that allows the bit rate to go as high as 10 Gbps;
- `ANALYSIS_DURATION` is the number of time units, i.e. the number of cycles of the optimization algorithm to be run before the end of the simulation; this is strictly related to the following analysis and evaluation.

At the end of the setup phase, the spanning tree of the network is build as explained in Section 2.3.2.

Finally, the energy optimization can be performed. The pseudocode in Algorithm 1 gives an insight of the main loop. The data at lines 17 and 18 is needed just for analysis purposes and to plot the results: it is reported just to be thorough.

The dictionary `ports_speed` (Algorithm 1, line 9), keeping track of the ports' bit rates, is initialized only at this point of the execution of the SDN application for convenience, as opposed to the other data structures composing the digital twin, that are rather instantiated during the setup phase.

Since the function `get_ports_speed()` performs N REST requests, where N is the number of active switches, and it processes data for each active port, the best moment to invoke it follows the evaluation of the global variable `DISABLE_UNUSED`.

Algorithm 1 Energy optimization algorithm - Main loop

```
1: if DISABLE_UNUSED then
2:     switches ← switches − switched_off
3:     ports ← coming from the digital twin graph's spanning tree
4: else
5:     switches ← all switches considered
6:     ports ← coming from the digital twin graph edges
7: end if
8: used_ports ← {'port' : 0, ...}                    ▷ initialize dictionary to all 0's
9: ports_speed ← get_ports_speed()                   ▷ REST GET call to Ryu API
10: time_unit ← 0
11: energy_data ← []
12: while time_unit ≤ ANALYSIS_DURATION do
13:     working, saturated ← get_working_ports(switches, ports, ports_speed)
14:     if ADAPTIVE_BITRATE then
15:         ports_speed ← change_link_rate(working, saturated, ports, ports_speed)
16:     end if
17:     time_unit ← time_unit + 1
18: end while
```

At lines 13 and 15 (Algorithm 1) the two functions at the core of the project are called. The pseudocode for them, recalling the concepts presented in Sections 2.3.3 and 2.3.4, follows.

Algorithm 2 get_working_ports(switches,ports,ports_speed)

```
Require: prev_port_bytes = {'port' : bytes}          ▷ support global data structure
1: for switch_id ∈ switches do
2:     port_stats ← cURL(GET, .../stats/port/switch_id)
3:     for stats ∈ port_stats do
4:         port ← stats[port_id]
5:         port_bytes ← stats[rx_bytes] + stats[tx_bytes]
6:         delta ← port_bytes − prev_port_bytes[port]
7:         prev_port_bytes[port] ← port_bytes
8:         if delta > SENSITIVITY · INITIAL_SPEED then
9:             port.status ← working
10:            working ← working + port
11:            if delta > SENSITIVITY · ports_speed(port) then
12:                port.status ← saturated
13:                saturated ← saturated + port
14:            end if
15:        end if
16:    end for
17: end for
18: return working, saturated
```

The function `get_working_ports` (Algorithm 2) takes as input the list of active switches containing their IDs, the list of active ports, and their actual bit rate, while outputting two lists of ports in *working* and *saturated* status, respectively.

At line 0 of Algorithm 2 one of the support global variables previously anticipated is presented: `prev_port_bytes` is a dictionary, with the names of the ports as keys, containing the amount of bytes exchanged by each port up to the last request made. The stored values are used at line 6 to compute the amount of data exchanged since the last cycle, which is then compared to the threshold value used to determine the status of the port (lines 8 and 11).

For a port to be *working*, it is sufficient that a little amount of data is passing through it, and that

is the reason why `SENSITIVITY` is multiplied by the base speed of the interface. Instead, a *saturated* port must be using a fair amount of its available bandwidth, as explained in Section 2.3.4, so the threshold value is multiplied by the actual working bit rate.

Algorithm 3 shows the function used to change the bit rate of the interfaces, according to their status. It takes as input the lists produced by `get_working_ports`, plus the lists of ports and their bit rates (`ports_speed`), and returns the updated version of the dictionary `ports_speed`.

As anticipated in Section 2.3.4, ports need to be in *saturated* state for a fixed hard-coded amount of time units before their bit rate is updated, in order to avoid unconvenient changes. The global dictionary `time_in_use`, having ports as keys, keeps track of the time units during which each port has been in *working* status, since entering it.

The values proposed in lines 7 to 9 have been determined according to on observations, and allow for a fair compromise in the reactivity of the system depending on the variations in traffic volume.

Lastly, the update of the bit rate of the interfaces is performed via a POST request, with the port name and its new speed in the payload (line 15).

Algorithm 3 change_link_rate(working,saturated,ports,ports_speed)

Require: $time\_in\_use = \{'port' : usage\_counter\}$ ▷ support global data structure
Ensure: $INITIAL\_SPEED = 10$
1: **for** $port \in working$ **do**
2: $time\_in\_use[port] \leftarrow time\_in\_use[port] + 1$
3: **end for**
4: **for** $port \in time\_in\_use.keys()$ **do** ▷ iterate on all ports available, checking status
5: $bitrate \leftarrow INITIAL\_SPEED$
6: **if** $port \in saturated$ **then**
7: **if** $time\_in\_use[port] > 1$ **then** $bitrate \leftarrow 100$
8: **if** $time\_in\_use[port] > 3$ **then** $bitrate \leftarrow 1000$
9: **if** $time\_in\_use[port] > 6$ **and** MAX_10G **then** $bitrate \leftarrow 10000$
10: **else if** $port \notin working$ **then**
11: $time\_in\_use[port] \leftarrow 0$
12: $bitrate \leftarrow INITIAL\_SPEED$
13: **end if**
14: **if** $ports\_speed[port] \neq bitrate$ **then** ▷ need to change port bit rate
15: $cURL(POST, .../qos/queue/switch\_id, data = \{port, bitrate\})$ ▷ switch id available from port
16: $ports\_speed[port] \leftarrow bitrate$
17: **end if**
18: **end for**
19: **return** $ports\_speed$

The complete code for the functions presented in this section can be found in Appendix A.

3.1.2 Tests scenarios

In order to get comparable results, the scenarios in which tests took place remained constant with respect to network topology and generated traffic.

The chosen network topology is composed of five switches and eight hosts, as depicted in Figure 3.1, while the traffic generation pattern is showed in Table 3.1.

The execution of STP, at the conditions explained in Section 2.3.2, always results in the same active topology to be used, which is depicted in Figure 3.2. Notice that no link is marked as active (blue edges) towards switch 2: this is due to the fact that no traffic is actually routed through it from the moment at which switch 4 becomes the star center of the network.

The effectiveness of the usage of a digital twin in a software defined network for energy consumption optimization has been evaluated in a number of more relevant cases, also comparing the proposed solution to a traditional approach, where no optimization is performed. This is made possible by

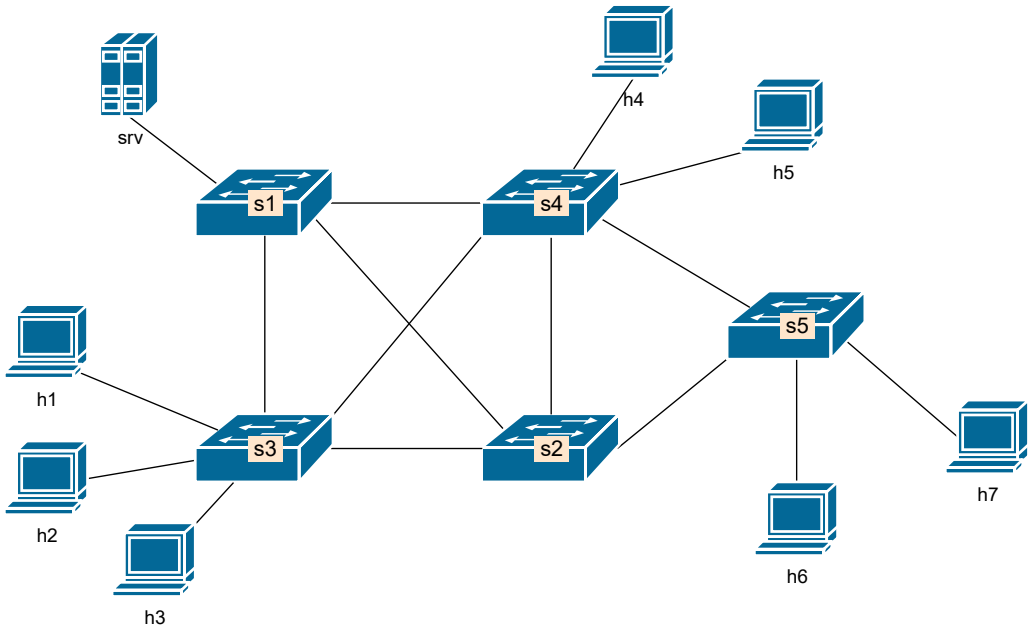


Figure 3.1: Topology used for the majority of the tests.

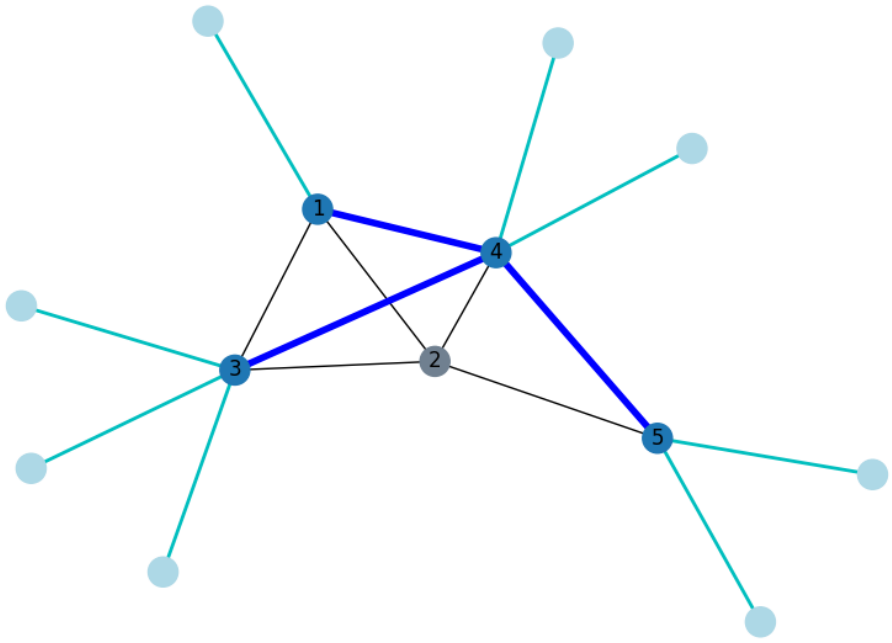


Figure 3.2: Active topology after the execution of STP. The links represented with black lines may be disabled, depending on the simulation parameters.

the presence of the previously described global variables that are used to enable or disable the main features of the SDN application.

For instance, it is possible to emulate the behavior of a traditional network by setting the base working speed of the interfaces, and by disabling both the bit rate adaptivity and the switching-off of unused devices. In that case, the digital twin is still used, but just for gathering network usage statistics to be compared against complete simulation runs. This can be noticed in the pseudocode in Algorithm 1, line 13, where the update of the ports status is performed without any previous checks, as opposite to the change of bit rate which may be disabled by setting `ADAPTIVE_BITRATE` to `False` (lines 14 and 15).

Time	Source	Destination	Data exchanged
2	h1	srv	10 MB
3	srv	h1	1500 MB
5	h2	h4	1000 MB
6	h7	h3	800 MB
12	srv	h1	1500 MB
14	srv	h6	1500 MB
15	h1	h3	1300 MB
16	h3	h1	250 MB
17	srv	h6	2500 MB
27	h2	h7	100 MB
28	h6	h2	850 MB
30	srv	h1	1500 MB
31	srv	h6	1500 MB
32	h1	h3	1300 MB
33	h3	h1	250 MB
34	srv	h5	2500 MB

Table 3.1: Traffic pattern used for the simulation runs.

3.2 Simulation runs and results

Table 3.2 contains the list of performed and analyzed simulations, with the specifications of the global parameters. The tests aim at providing a comparison between the traditional approach and the optimization performed via the digital twin.

Runs 1 and 2 will emulate the network behavior without energy consumption control mechanisms. In run 1 all interfaces will work at 100 Mbps, while in run 2 the bit rate will be 1 Gbps. Simulations 3 and 4 will implement all optimization mechanisms, with the only difference between the two being the possibility to use a bit rate of 10 Gbps in run 4.

Among the optimization mechanisms, there is the possibility to switch off unused ports and, if possible, switches. Figure 3.2 marks with blue edges the links, and therefore the ports, that must be kept active, while all the others may be disabled, with the only exception of those connecting switches to the hosts.

The value of `BASE_POWER` was set to 20W by averaging the data from product specifications related to office-sized network switches, that are usually equipped with interfaces ranging from 100 Mbps to 1 Gbps, even though some models also support 10 Gbps. When dealing with industrial or data-center-sized switches, the base operating power may exceed 100 W.

	`BASE POWER`	`INITIAL SPEED`	`SENSITIVITY`	`ADAPTIVE BITRATE`	`DISABLE UNUSED`	`MAX_10G`	Cycles
1	20 W	100 Mbps	67500	False	False	False	60
2	20 W	1000 Mbps	67500	False	False	False	60
3	20 W	10 Mbps	67500	True	True	True	60
4	20 W	10 Mbps	67500	True	True	False	60

Table 3.2: List of performed simulations, with their parameters.

3.2.1 Energy consumption analysis

As already explained in Section 1.3, the power required by interfaces is mostly influenced by the bit rate they are working at, rather than the traffic load [14].

It follows that all measurements related to energy will be based on the link rates, as per the values in Table 3.3 [14]. The values are further supported by data contained in various data sheets of switches from well-established brands [25].

Every simulation lasts for 60 cycles, which corresponds to about three minutes, due to the computational power of the testing environment, which influences the time needed to perform and process all the REST calls.

Bit rate	Power per interface
10 Mbps	0.1 W
100 Mbps	0.2 W
1000 Mbps	0.5 W
10000 Mbps	5.0 W

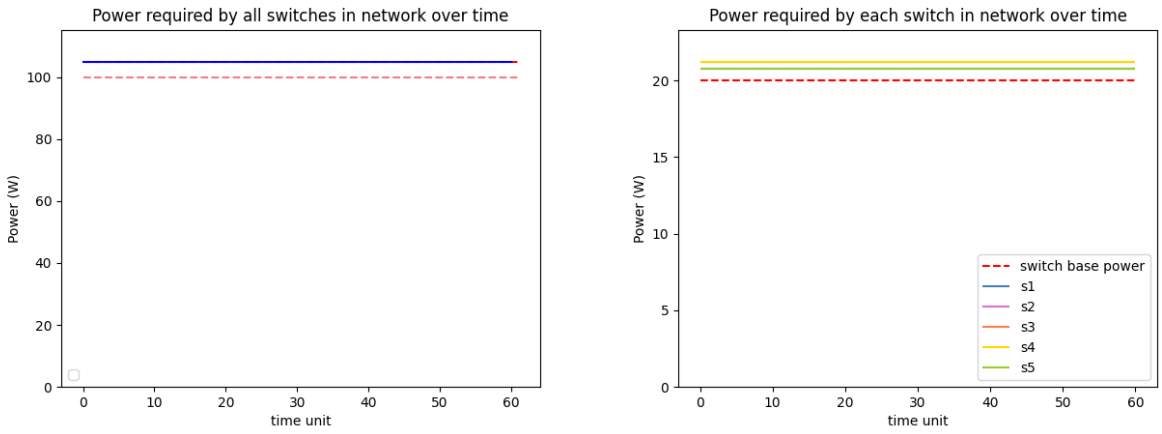
Table 3.3: Power required by each port at a given bit rate.

3.2.2 Run 1: traditional approach, 100 Mbps fixed bit rate

The first simulation run reproduces the behavior of a network where all ports work at a fixed bit rate of 100 Mbps: none of them is disabled after the execution of the spanning tree algorithm, but they are simply left unused.

The power consumption related to the bit rate is constant, since no adaptation is performed (Figure 3.4). The average power required is 104.80 W, where only 4.5% is due to the interfaces. This derives from the fact that they are working at a very low bit rate, that is reflected on the performance.

The average throughput of each switch for the duration of the simulation is just above 400 MB, and it is worth noticing that no traffic is routed through switch 2, which is unnecessarily left on (Figure 3.4a).



(a) (b)

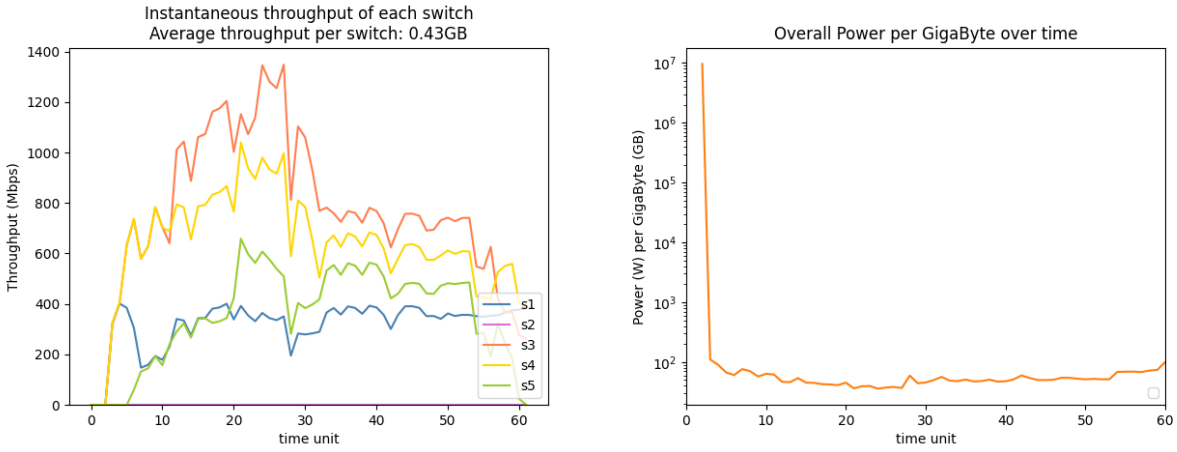
Figure 3.3: Run 2, fixed bit rate of 100 Mbps - 3.3a shows the total energy required for the entire simulation, while 3.3b shows the detail for each switch. Switches s1, s2, s5 require the same power, just like s3 and s4.

On one hand it is true that, at such a bit rate, the energy consumption is minimum, but the throughput is so low that the average power required to transmit a gigabyte of data across each switch results to be ≈ 53 W. Figure 3.4b shows the required energy per gigabyte over time, averaged on all the switches. At the beginning, the value tends to infinite because all interfaces are working to transmit just few packets related to STP.

3.2.3 Run 2: traditional approach, 1 Gbps fixed bit rate

Since the usage of devices with interfaces working at 100 Mbps is now anachronistic in the majority of networks, the same simulation of Run 1 has been repeated in Run 2, with the bit rate of each port fixed at 1 Gbps.

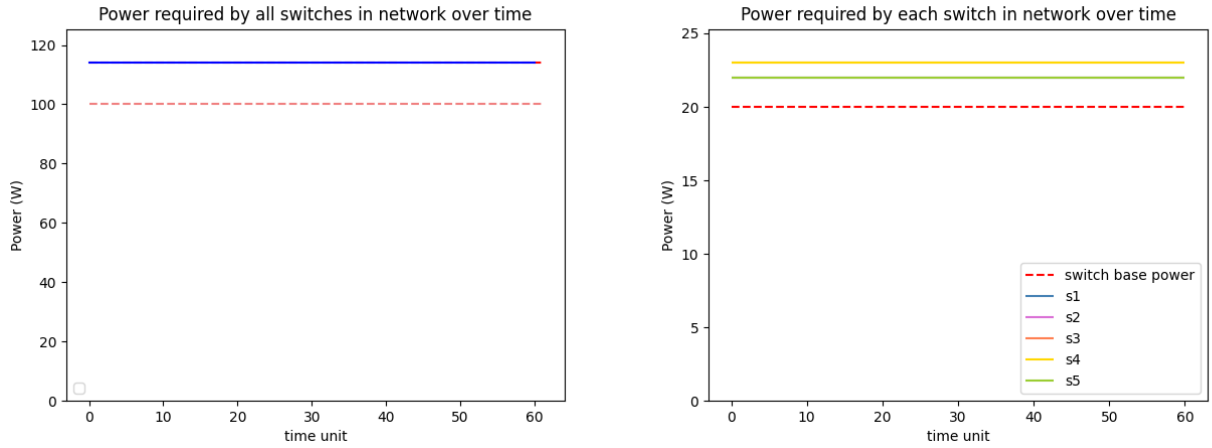
As for Run 1, this simulation requires a constant amount of energy for a greater total value of 112 W, which is due to the higher bit rate (Figure 3.5). The percentage of power required by the interfaces



(a) (b)

Figure 3.4: Run 2, fixed bit rate 100 Mbps - 3.4a shows the throughput of each switch per time unit. Notice how switch 2 never experiences traffic, as a consequence of the execution of STP. 3.4b shows the power per gigabyte over time.

now amounts to 11% of the total.



(a) (b)

Figure 3.5: Run 2, fixed bit rate of 1 Gbps - 3.5a shows the total energy required for the entire simulation, while 3.5b shows the detail for each switch. Switches s1, s2, s5 require the same power, just like s3 and s4.

A minimal increase in the energy requirements, comes with bigger benefits in terms of performance: the average throughput per switch rises from 0.4 GB to almost 1.3 GB (Figure 3.6). It is worth recalling that the throughput is computed per time unit, which corresponds to almost 3 seconds in the specific case of the presented results: this justifies the fact that the throughput is measured above 1 GB with a bit rate of 1 Gbps.

This configuration also provide a good ratio between the exchanged quantity of data and the power required to do so: in average, ≈ 23 W are required to exchange 1 GB of traffic (Figure 3.7).

3.2.4 Run 3: optimized approach, adaptive bit rate up to 10 Gbps

The third run is the first simulation where all the digital twin features for energy consumption optimization are used. In particular, the bit rate of each interface varies according to the traffic volume, as explained in Section 2.3.4, by following the increasing sequence given by 10 Mbps, 100 Mbps, 1 Gbps, 10 Gbps. All unused interfaces and, if possible, switches are turned down.

Following the execution of BFS Spanning Tree, the active links form a star topology, where switch

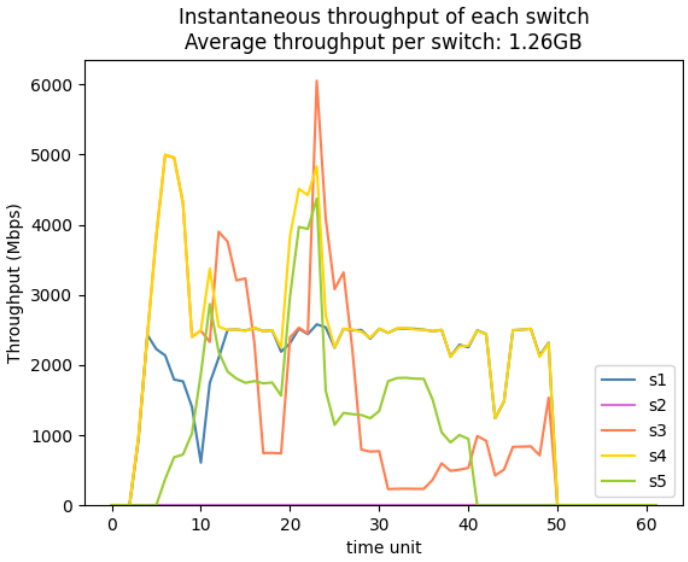


Figure 3.6: Run 2, fixed bit rate 1 Gbps - Throughput of each switch per time unit. Once again, switch 2 never experiences traffic, as a consequence of the execution of STP.

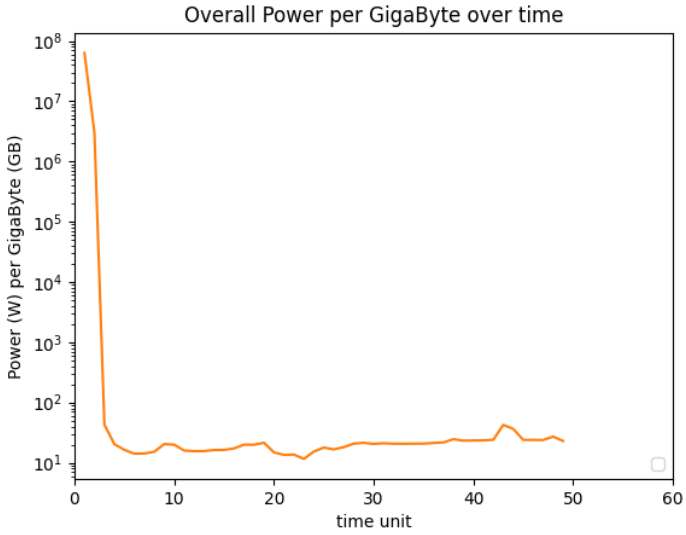


Figure 3.7: Run 2, fixed bit rate of 1 Gbps - Power per gigabyte over time, averaged on all switches.

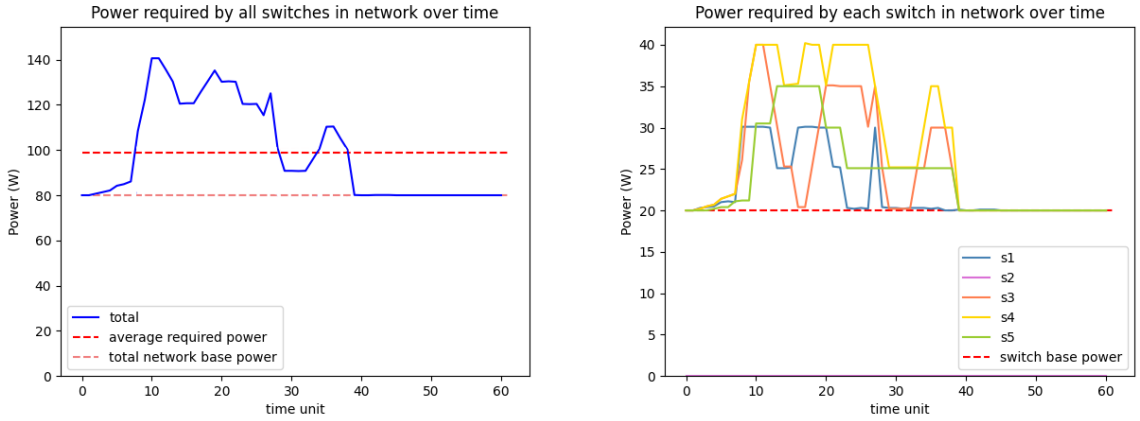
$s2$ becomes a leaf of the tree. For this reason, the device is switched off and doesn't account in the computation of the required energy, thus decreasing the base energy in Figure 3.8a to 80 W instead of 100 W, as happened in the two previous runs.

The average power required by the entire network for the duration of the simulation is just 99 W, with a third of the time spent at the minimum operating power of 80 W. Despite the peaks of energy consumption due to the interfaces working at 10 Gbps, which are well above the constant value of 112 W required in Run 2, the network reduced its energy usage of 12%.

However, it must be acknowledged that a major part of the saving is due to the fact that one switch is completely turned off, causing a 20% reduction of the base energy required by the network.

With the described running configuration, the average of the required Watts per gigabyte amounts to 21 W (Figure 3.9b), which is approximately equal to the average Watt per gigabyte of Run 2.

An interesting data is that the average throughput for each switch during the entire simulation is lower than that of Run 2: 1.2 GB instead of 1.3 GB. The explanation for this is quite trivial: by looking at Figure 3.9a, it can be noticed that for almost half of the simulation the throughput is equal to zero, since the larger bandwidth allows for some of the traffic flows to be exchanged in the extent

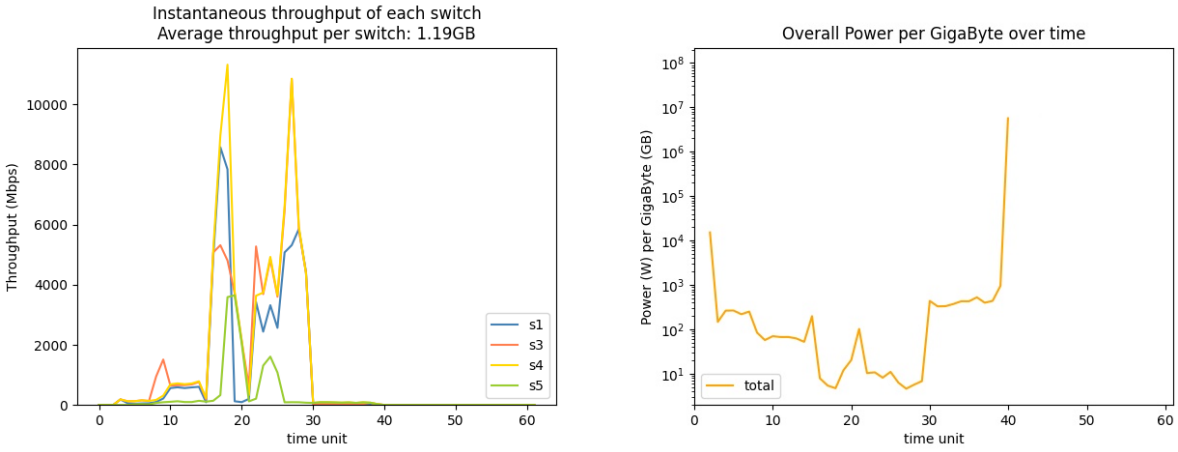


(a) (b)

Figure 3.8: Run 3, adaptive 10 Mbps to 10 Gbps - 3.8a shows the total energy required for the entire simulation, while 3.8b shows the detail for each switch.

of few time units.

If just the interval between time unit 15 and time unit 30 were considered, the average throughput per switch would be above 2 GB.



(a) (b)

Figure 3.9: Run 3, adaptive from 10 Mbps to 10 Gbps - 3.9a shows the throughput of each switch per time unit, while 3.9b shows the power per gigabyte over time.

3.2.5 Run 4: optimized approach, adaptive bit rate up to 1Gbps

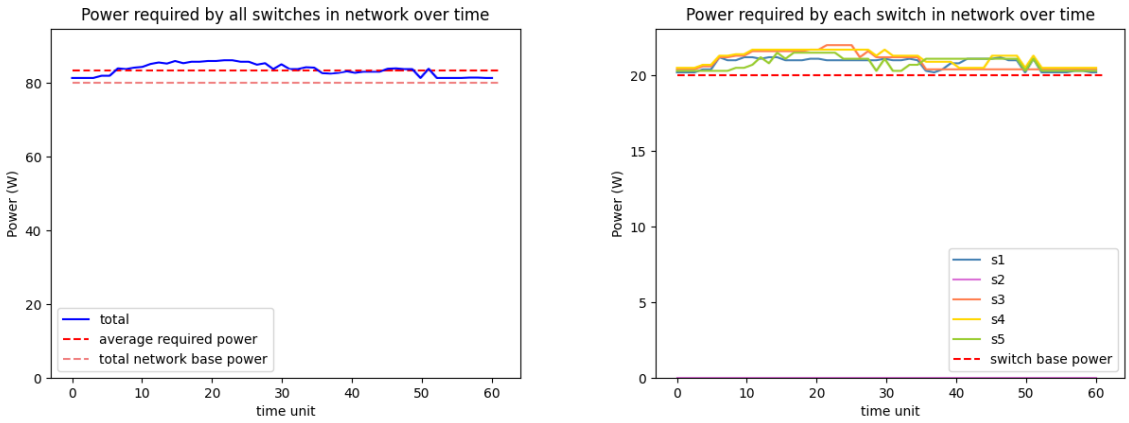
Results from Run 3 may not seem like an improvement compared to Run 2, since not only the energy saving comes with a lower grade of redundancy of the network, but also, in the end, the average power required per gigabyte exchanged is lower in the traditional run.

However it must be noticed that the comparison was not entirely fair: a link bit rate of 1 Gbps is suited for office environments, while 10 Gbps links are commonly used in the context of datacenters and, in general, in higher traffic volume scenarios.

Run 4 aims at proposing an unbiased comparison: as in Run 3, the bit rate is adaptive, but it ranges among 10 Mbps, 100 Mbps and 1 Gbps. This results in an extremely low energy consumption related to the interfaces (Figure 3.10a), which is upper-bounded by the case in which a bit rate of 1 Gbps is used on all links, and all ports are switched on. In this simulation run, switch 2 is turned off because no traffic is routed through it after the execution of the spanning tree algorithm.

This adjustment significantly influences the average energy consumption of the network, that is

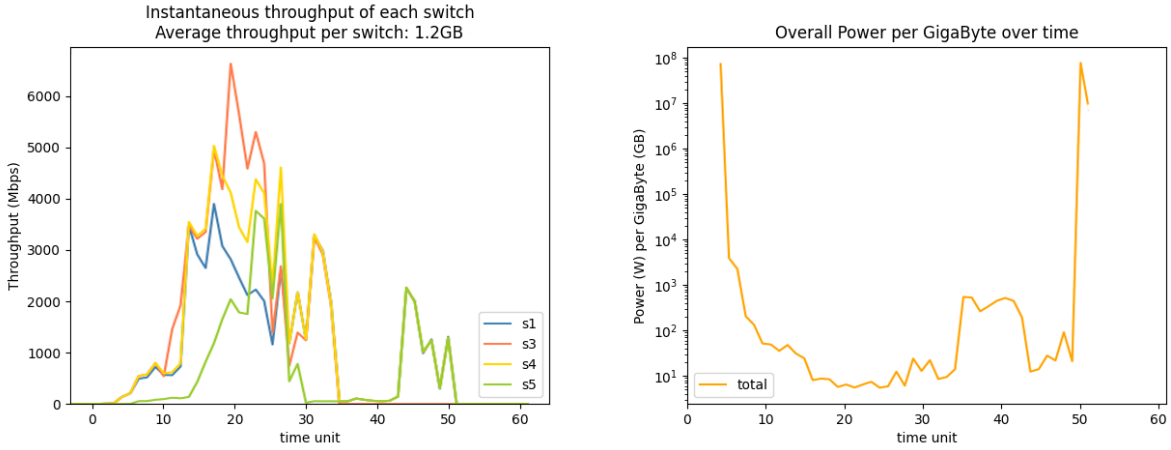
shrank to ≈ 83 W: a reduction of ≈ 26% if compared to Run 2.



(a) (b)

Figure 3.10: Run 4, adaptive from 10 Mbps to 1 Gbps - 3.10a shows the total energy required for the entire simulation, while 3.10b shows the detail for each switch.

The performance of Run 4 is comparable to that of Run 2, with just a minimum reduction in throughput (Figure 3.11a). What is most significant is the average Watt per gigabyte required during the simulation (Figure 3.11b): Run 4 amounts at ≈ 15 W/GB, which represents a reduction of 35% if compared to Run 2.



(a) (b)

Figure 3.11: Run 4, adaptive from 10 Mbps to 1 Gbps - 3.11a shows the throughput of each switch per time unit, while 3.11b shows the power per gigabyte over time.

4 Results evaluation

Chapter 3 presented the scenarios and the strategies used to test the digital twin and presented some preliminary results. This chapter will compare them, while analyzing some particular cases. Finally, some insights on possible future developments are given.

4.1 Comparing the results

The traffic pattern used in the simulations could be compared to that of an office-sized network. In such environments, it is not unusual to find switches with 100 Mbps interfaces, however the related Run 1 has been shown in Subsection 3.2.2 to provide the worst results among all the simulations: in fact, the energy saving due to the reduced bit rate is completely canceled out by the lack of throughput, causing the transmission to last longer than in the other runs. This causes the average Watt required per gigabyte transmitted to be the highest value of the four recorded.

On the other hand, Run 3 (Subsection 3.2.4) is the one providing the best performance, but at a much higher energetic cost: the average energy consumption of just 99 W comes from the fact that one switch is turned off. Were it turned on, the average power would raise to 120 W, making it the highest value among the simulations.

However it must be noticed that Run 3 uses 10 Gbps links, that are typical of small to medium datacenters or large office environments. Running the same simulation with a fixed bit rate of 10 Gbps on all interfaces, as in Run 1 and 2, would return a throughput similar to that of Run 3, but with a constant energy required, which can be computed as:

$$100W + 5.0W \cdot 28 = 240W$$

where 28 is the number of the interfaces connected in the network (four each for switches 1, 2 and 5, six each for switches 3 and 4), and 5.0 W is the energy required by an interface per time unit, as reported in Table 3.3.

Using the digital twin approach in this scenario with 10 Gbps links would therefore allow for a saving of more than a third of the required energy.

The fairer comparison is between Run 2 and Run 4, since the latter can be seen as an optimization of the first, as regards power consumption. The two simulations provide almost identical throughput, with Run 4 being marginally penalised by the time units spent at lower bit rate at the start of a new traffic flow. The optimized run however needs $\approx 26\%$ less energy than Run 2.

Whilst this result is strongly influenced by the fact that one switch is turned off after the execution of the spanning tree algorithm, it is also true that, with all switches on, a total of 20.2 W should be added to the total energy consumption in each time unit, bringing the average value to ≈ 103 W: this still represents a 9% saving in Run 4 with respect to Run 2. This is explained by the fact that the maximum instant energy consumption of Run 4 will at most be equal to that of Run 2, as shown in Figure 4.2, provided that no interface is shut down after the spanning tree execution. Moreover, in order for Run 4 to be level with Run 2, it would take that all ports are used at the maximum bit rate of 1 Gbps at the same time.

Based on observations, it is reasonable to suppose that in presence of traffic with regular volume, a network implementing mechanisms for energy consumption monitoring and optimization would achieve values of throughput and required power comparable to that of a traditional network with a fixed bit rate proportional to the amount of traffic. In other words, Run 2 and 3 would give almost identical results if the traffic volume would be sized in such a way that the threshold value needed to rise the port speed to 10 Gbps is never exceeded.

When considering just the average energy required, the configuration used in Run 4 proves to be the best. This result is still influenced by a number of variables, primarily the traffic pattern

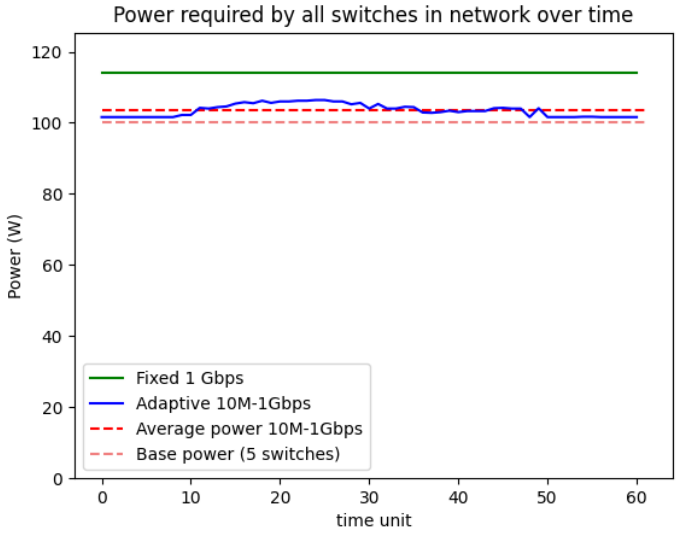


Figure 4.1: Comparison of the power required over time in Run 2 and Run 4, supposing switch 2 is on in the latter case.

characterised by bursts and the network topology, especially the one used after the execution of the spanning tree algorithm. This leads to the need for a measure that is as independent as possible from at least one of these two aspects.

Figure 4.2 shows the curves representing the energy (measured in Watt) required to exchange 1 GB of traffic over time during the four simulation runs. The measure has been chosen since it allows for a normalization of the energy consumed by the network with respect to the throughput at each time unit.

It can be noticed that at the beginning the values are really high: this is due to the disproportion between the limited number of packets exchanged during the execution of spanning tree algorithm and the bit rate of the interfaces that, in the best case of 10 Mbps, is still exaggerated if compared to the amount of traffic passing through the network.

The curves level out once traffic volume stabilizes in relation to the used bit rate. The peaks in the graph related to Run 3 (labeled as "Adaptive 10M-10Gbps") can be traced back to the moments when links working at a full speed of 10 Gbps experience a sudden decrease in traffic volume due to a flow ending. The same phenomenon is common to the curve of Run 4 to a lesser extent.

Table 4.1 summarizes the results gathered for clarity. For completeness, the last row shows the estimate value of a traditional run with links at 10 Gbps, as previously computed in this paragraph.

Run	**Bit rate range**	**Unused ports**	**Base power**	**Switches (Active)**	**Average power**	**Average W/GB**
1	Fixed 100Mbps	Powered	20 W	5 (5)	104.8 W	53 W/GB
2	Fixed 1Gbps	Powered	20 W	5 (5)	112 W	23 W/GB
3	10Mbps - 10Gbps	Shutted	20 W	5 (4)	99 W	21 W/GB
4	10Mbps - 1Gbps	Shutted	20 W	5 (4)	83 W	15 W/GB
-	Fixed 10Gbps	Powered	20 W	5 (5)	240 W (estimated)	-

Table 4.1: Summary of the discussed results.

4.2 The particular case of ring and linear topologies

The results analyzed up until now show a clear correlation between the traffic patterns and the efficiency of the digital twin configuration. In addition to that aspect, also the network topology must be taken into account: the way the spanning tree is built by the digital twin aims at generally

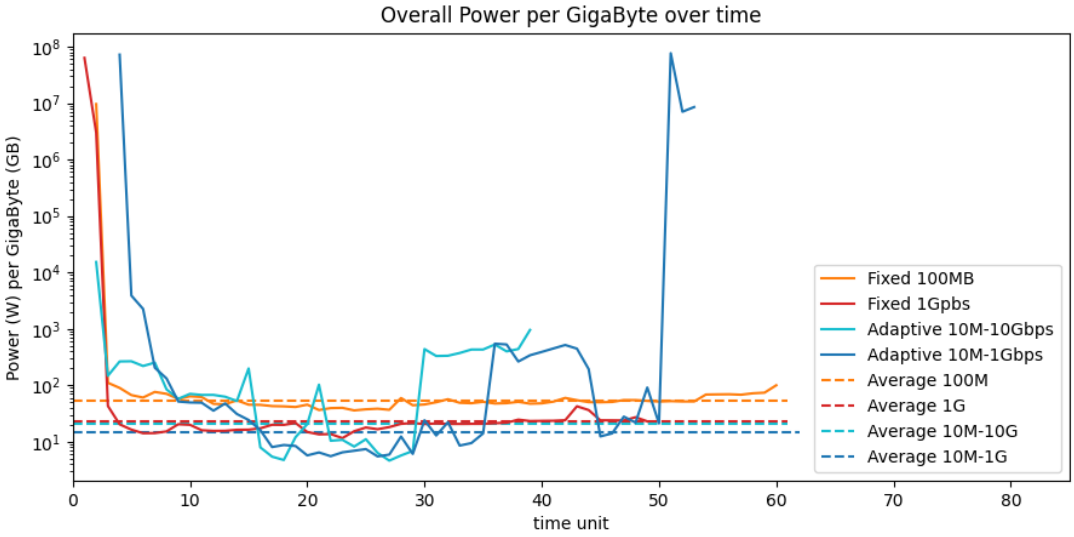


Figure 4.2: Comparison of the Watts per gigabyte required for each run.

minimizing the hops between terminal hosts generating traffic, thus leading in the majority of case to an active topology that is tree or star shaped.

With this in mind, linear and ring topologies may be seen as the limit cases. The execution of the spanning tree algorithm is ineffective in linear topologies, since there's no path to be chosen over another, while ring topologies are simplified to linear ones by the algorithm used in this project for retrieving the spanning tree.

In some extreme cases, the algorithm may lead to highly inconvenient distributions of hosts across the network, as shown in Figure 4.3. The scoring system used to determine the root of the tree described in Subsection 2.3.2 favours the node labeled with A, leading to nodes B and C being put at opposite sides of the new topology. With the highest number of hosts distributed at the two ends, there is a good chance that the majority of traffic flows will need to traverse the whole network, therefore keeping all interfaces in working and possibly saturated state.

This aspect highlights the need to develop algorithms for the marking of active links that are well suited to the network which the digital twin should work on. However, software defined networking provides all the necessary instruments to address this kind of issues, though they will not be further analyzed since they go beyond the purpose of this research.

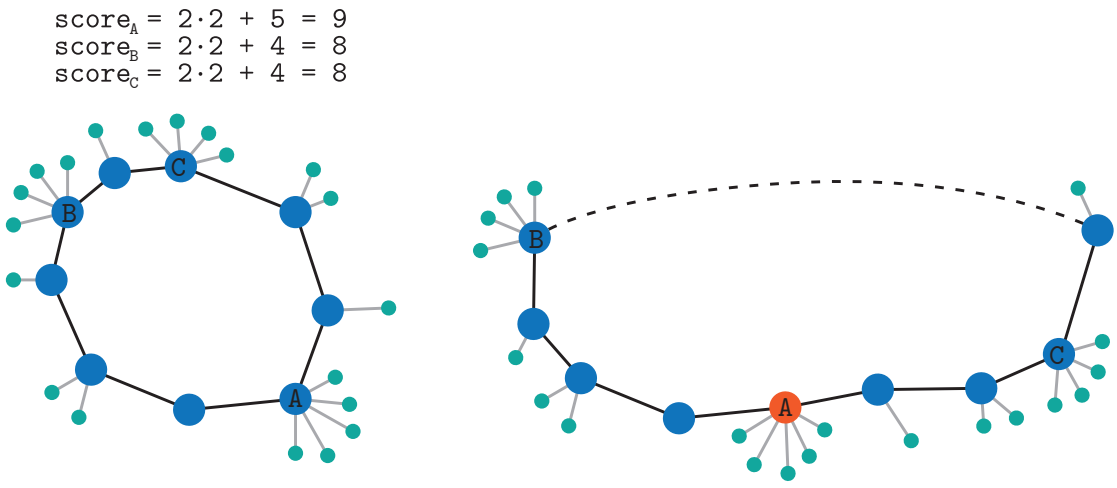


Figure 4.3: The algorithm for the spanning tree developed for this project is fair for mesh networks, but could be not tailored for ring topologies.

In order to give evidence of the expressed limitations, two additional sets of tests have been

performed involving a ring network topology. Each set consists of two runs, the first implementing no optimization mechanisms, while the second allows for variable adaptive bit rate of all active ports in the range 10 Mbps - 1 Gbps. All simulation runs have been performed under the same conditions as regards network configuration, while the traffic pattern changes between the two sets: in the first one, packets flow between hosts connected to adjacent switches, while in the latter packets must traverse all the network. A summary of the simulation parameters is provided in Table 4.2, while Table 4.3 shows the traffic pattern used in the two sets.

	Set 1		**Set 2**	
	Run 1.1	**Run 1.2**	**Run 2.1**	**Run 2.2**
Traffic flow	Between adjacent switches		Must traverse all network	
Disable unused ports	No	Yes	No	Yes
Adaptive bit rate	Fixed 1 Gbps	10 Mbps - 1 Gbps	Fixed 1 Gbps	10 Mbps - 1 Gbps
Sensitivity	67500	67500	67500	67500
Base power (switch)	20 W	20 W	20 W	20 W
Simulation cycles	30	30	30	30

Table 4.2: List of performed simulations, with their parameters.

	Set 1		**Set 2**		
Time	**Source**	**Destination**	**Source**	**Destination**	**Data exchanged**
4	h1	h5	srv	h7	200 MB
7	h4	h2	h6	srv	1500 MB
9	h5	h1	h5	srv	800 MB
13	h2	h4	srv	h6	2500 MB
17	h3	h5	srv	h5	1500 MB
20	h4	h2	h6	srv	800 MB
21	h5	h3	h5	srv	250 MB

Table 4.3: Traffic patterns generated during the two sets of simulations summarized in 4.2.

The considered topology is shown in Figure 4.4. The links selected by the spanning tree algorithm for the routing of the packets are represented by the blue edges. It's worth noticing that the ring topology is reduced to a linear one, with switch 2 not routing any traffic through it.

Energy consumption measurements from Runs 1.1 and 2.1 both returned the same results, since all interfaces are powered and working at the same bit rate for the entire duration of the simulations. In particular, the average energy required is constant and amounts to ≈ 109 W.

Similarly to what was presented in Section 4.1, both in Run 1.2 and in Run 2.2, that value represents the upper limit of the required energy. The substantial difference between the two runs is strictly depending on the traffic flows: in fact, despite the short duration of the simulations and the low number of involved devices, Run 1.2 requires 2% less energy than Run 2.2 (Figure 4.5), because, in the former, as it can be observed in Figure 4.6, only ports from two switches are used.

4.3 Future developments

The tests gave positive results as regard the use of the digital twin for modeling, monitoring and optimizing the energy consumption of the network, but also showed some limitations, mainly due to the strategies involved in the definition of the spanning tree, and to the developed optimization algorithm. These aspects have been proven to be strictly related to the nature of the traffic traversing the network and to the used topology.

This underlines once more the importance of adopting a new approach when managing complex network architectures. With SDN providing the possibility to have a global view of every aspect concerning a network, and the ability to completely program it, there is no limit to the algorithms

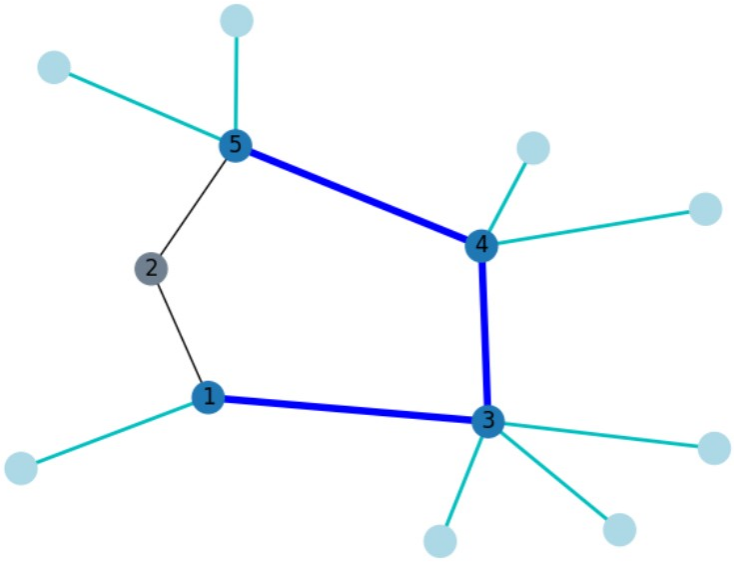
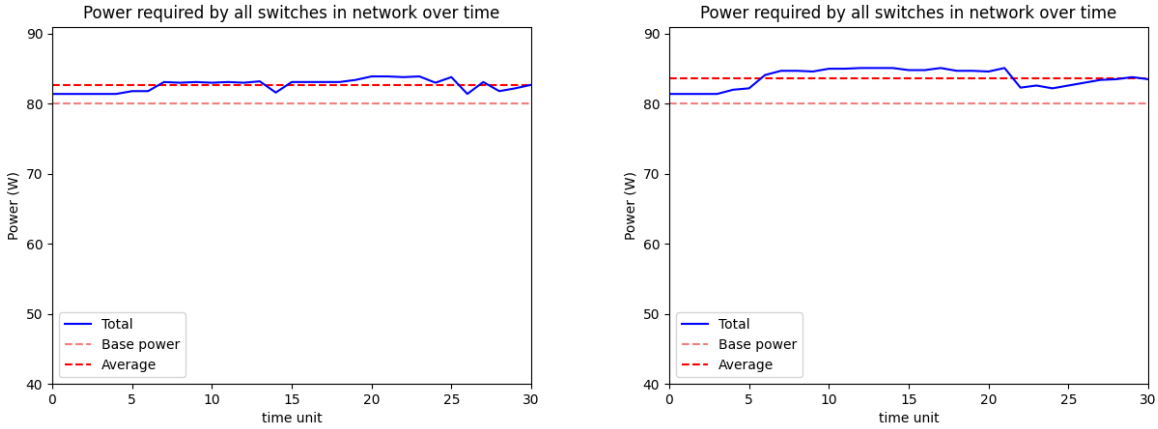


Figure 4.4: Topology used during the simulations. Blue edges represent the links selected in the spanning tree.



(a) (b)

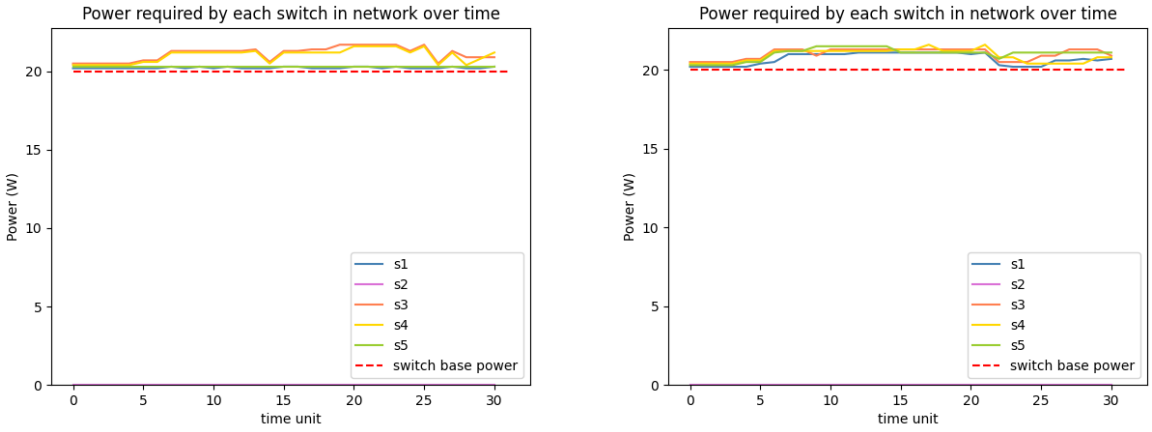
Figure 4.5: 4.5a shows the total energy required for Run 1.2 (traffic between switches near each others), while 4.5b shows that of Run 2.2. (traffic between switches at the two ends of the topology)

that can be run on the proposed digital twin to best suit the flows, the topology and the given business requirements.

What follows is an overview of some further developments that could be done starting from the presented project.

As concerns the routing, instead of defining a spanning tree based just on the topology, it would be better to use an algorithm that selects the links to use for each traffic flow in order to minimize the traversed interfaces. This could be done periodically and a priori, e.g. computing the shortest paths between hosts and populating the flow tables of the switches, or it could be part of a learning process that aims at aggregating flows on the same links in order to maximize their usage.

Further improvements can be also achieved as regards the managing of idle and working interfaces. The greedy optimization algorithm used in this project has been developed exclusively for testing purposes and, despite this fact, has proven effective under some conditions. Its main limitation is that the decision-making implemented by it is static, meaning that the used parameters are defined once and for all at the start of the execution. Machine learning mechanisms could be employed in order to dynamically compute and update the values of the sensitivity and the amount of time a port



(a) (b)

Figure 4.6: 4.6a shows the energy required by each switch for Run 1.2, while 4.6b shows that of Run 2.2.

should be in saturated state before updating its bit rate. The literature reports examples of usage of reinforcement learning algorithms executed on digital twins in various application fields.

Finally, an extreme approach to the problem of adapting the bit rate of the interfaces depending on traffic could include switching off ports that are not exchanging packets, even if they are selected as active by the routing algorithm, and turning them on as soon as some amount of dropped packets is sensed. Obviously, choosing to adopt such a solution implies accepting a compromise between energy saving and performance degradation. Moreover, in order for the entire system to be responsive enough to packet drops, the data gathering should happen at more frequent intervals, resulting in an increase of both computational load and network traffic generated by the digital twin.

Overall, the proposed future improvements highlight once more the versatility and adaptability of a digital twin implemented with the structure presented in this research. This is the most important result achieved and paves the way for interesting and useful applications in most network architectures.

Conclusions

In this research, a concrete and efficient strategy to implement a digital twin of a network for energy optimization has been analyzed and realized, yielding significant results which have been previously described.

Specifically, it has been shown that it is possible to achieve significant savings of energy consumption, even with the simplest among the discussed optimization strategy. However, the strength of this research lies in the easily-maintainable structure of the developed digital twin: this can be integrated with any algorithms of choice, thanks to its modularity and the immediate interaction with the data structures.

The fact that the digital twin can be adapted to the application scenario is crucial to the achievement of the expected results: it has been largely discussed how network topology and the nature of traffic may affect the outcome of the optimization strategy.

The greedy approach presented in Chapter 3, Algorithm 3, has proven to be effective in a generic meshed topology with irregular traffic flows. The mechanisms for adapting the bit rate of the ports, depending on the traffic volume traversing them, allow for values of energy consumption that can be lower than half the power required by a traditional networking approach (Table 4.1, Chapter 4). Even better results are achieved by shutting down unused interfaces and, if possible, entire devices. The latter aspect in a real life scenario could be unpractical since a physical switch would require some time to boot, if necessary. It could be advisable only in the case such device implements hibernation mechanisms that allow for a faster restart when needed.

Chapter 4 explores some scenarios in which the algorithms used for routing the traffic and adapting the bit rate of the ports are not the best choices. Moreover, it advances some improvements and new strategies to implement algorithms that are more appropriate to particular topologies or traffic patterns.

However, it would be imprecise to talk about compromises or limitations in the usage of the digital twin: the routing and optimization algorithms are independent entities that are executed on the digital twin and influence the underlying real network, but as they are pieces of a larger structure, they can be substituted by more suited ones. As a matter of fact, when dealing with network infrastructures, there is no bullet-proof solution, since they are designed and realized in order to meet specific requirements depending on the desiderata of the stakeholders, the available resources of the service provider, and, last but not least, the quality of service that the end user should experience.

Appendix A Code snippets

This Appendix contains some of the code developed for the implementation of the digital twin, as well as for performing the tests.

A.1 Building a network in Mininet

Script A.1 shows the Python code necessary to build a network topology in Mininet. From line 43 to 57 hosts and switches are added to the topology, while the links between them are defined in lines 61 to 77. Lines 82 to 86 are needed to start up the switches and connect them to the SDN controller, that was defined in line 39.

Listing A.1: Building a network topology in Mininet

```
1  #!/usr/bin/env python
2
3  import subprocess
4  import os
5  import logging
6
7  from mininet.net import Mininet
8  from mininet.node import RemoteController
9  from mininet.log import setLogLevel, info
10 from mininet.node import OVSSwitch
11 from mininet.cli import CLI
12 from mininet.util import quietRun
13
14 import sys
15
16 flush = sys.stdout.flush
17
18 # Create logger
19 logger = logging.getLogger(__name__)
20 stream_handler = logging.StreamHandler()
21 stream_formatter = logging.Formatter("%(asctime)s %(name)s %(levelname)s %(message)s")
22 stream_handler.setFormatter(stream_formatter)
23 logger.setLevel(logging.INFO)
24 logger.addHandler(stream_handler)
25
26 cwd = os.getcwd()
27
28 def buildTopology():
29
30     # Select TCP Reno
31     output = quietRun( 'sysctl -w net.ipv4.tcp_congestion_control=reno' )
32     #assert 'reno' in output
33
34     subprocess.run([f"{cwd}/start_ryu.sh"])
35
36     net = Mininet( controller=RemoteController, waitConnected=True, switch=OVSSwitch )
37
38     info( '*** Adding controller\n' )
39     c0 = RemoteController("c0", ip="127.0.0.1", port=None)
40     net.addController( 'c0' )
41
42     info( '*** Adding hosts\n' )
43     server = net.addHost( 'srv', ip='10.0.0.100', dpid="100")
44     h1 = net.addHost( 'h1', ip='10.0.0.1', dpid="111")
45     h2 = net.addHost( 'h2', ip='10.0.0.2', dpid="112")
46     h3 = net.addHost( 'h3', ip='10.0.0.3', dpid="113")
```

```
47      h4 = net.addHost( 'h4', ip='10.0.0.4', dpid="114")
48      h5 = net.addHost( 'h5', ip='10.0.0.5', dpid="115")
49      h6 = net.addHost( 'h6', ip='10.0.0.6', dpid="116")
50      h7 = net.addHost( 'h7', ip='10.0.0.7', dpid="117")
51
52      info( '*** Adding switch\n' )
53      s1 = net.addSwitch( 's1', dpid="1", protocols="OpenFlow13", cls=OVSSwitch)
54      s2 = net.addSwitch( 's2', dpid="2", protocols="OpenFlow13", cls=OVSSwitch)
55      s3 = net.addSwitch( 's3', dpid="3", protocols="OpenFlow13", cls=OVSSwitch)
56      s4 = net.addSwitch( 's4', dpid="4", protocols="OpenFlow13", cls=OVSSwitch)
57      s5 = net.addSwitch( 's5', dpid="5", protocols="OpenFlow13", cls=OVSSwitch)
58
59      info( '*** Creating links\n' )
60
61      net.addLink(s1, s2)
62      net.addLink(s1, s3)
63      net.addLink(s2, s4)
64      net.addLink(s2, s5)
65      net.addLink(s3, s4)
66      net.addLink(s1, s4)
67      net.addLink(s5, s4)
68      net.addLink(s2, s3)
69
70      net.addLink(server, s1)
71      net.addLink(h1, s3)
72      net.addLink(h2, s3)
73      net.addLink(h3, s3)
74      net.addLink(h4, s4)
75      net.addLink(h5, s4)
76      net.addLink(h6, s5)
77      net.addLink(h7, s5)
78
79      info( '*** Starting network\n' )
80      net.build()
81      c0.start()
82      s1.start( [c0] )
83      s2.start( [c0] )
84      s3.start( [c0] )
85      s4.start( [c0] )
86      s5.start( [c0] )
87      net.start()
88
89      # from Ryu docs
90      # set the version of OpenFlow to be used in each router to version 1.3
91      # and set to listen on port 6632 to access OVSDB
92      info(net['s1'].cmd("ovs-vsctl set Bridge s1 protocols=OpenFlow13"))
93      info(net['s1'].cmd("ovs-vsctl set-manager ptcp:6632"))
94      info(net['s2'].cmd("ovs-vsctl set Bridge s2 protocols=OpenFlow13"))
95      info(net['s2'].cmd("ovs-vsctl set-manager ptcp:6632"))
96      info(net['s3'].cmd("ovs-vsctl set Bridge s3 protocols=OpenFlow13"))
97      info(net['s3'].cmd("ovs-vsctl set-manager ptcp:6632"))
98      info(net['s4'].cmd("ovs-vsctl set Bridge s4 protocols=OpenFlow13"))
99      info(net['s4'].cmd("ovs-vsctl set-manager ptcp:6632"))
100     info(net['s5'].cmd("ovs-vsctl set Bridge s5 protocols=OpenFlow13"))
101     info(net['s5'].cmd("ovs-vsctl set-manager ptcp:6632"))
102
103     info(server.cmd("iperf -s -i1 &"))
104     info(h1.cmd("iperf -s -i1 &"))
105     info(h2.cmd("iperf -s -i1 &"))
106     info(h3.cmd("iperf -s -i1 &"))
107     info(h4.cmd("iperf -s -i1 &"))
108     info(h5.cmd("iperf -s -i1 &"))
109     info(h6.cmd("iperf -s -i1 &"))
110     info(h7.cmd("iperf -s -i1 &"))
111
112     CLI(net)
113     info( '*** Stopping network\n' )
114     net.stop()
115     subprocess.run(["mn", "-c"])
116
117
118 if __name__ == '__main__':
119     setLogLevel( 'info' )
120     buildTopology()
```

A.2 Building the graph representing the network

Listing A.2 contains the code for the construction of the network graph and the selection of the active links, that compose the spanning tree. Comments provide an explanation for the main passages. The function returns the graph representing the network together with lists of links and hosts, that will later be used to populate the data structures of the digital twin.

Listing A.2: Graph building and spanning tree definition

```
def bfs_stp():
    # ========== get switches and hosts from controller ==========
    switches = get_all_switches()
    switches.sort()
    sw_hosts, host_per_switch = get_all_hosts(switches)
    # ==============================

    # ========== add nodes (switches) to the graph ==========
    for s in switches:
        G.add_node(s)
    # ==============================

    # ========== get links from controller ==========
    b_obj = BytesIO()
    crl = pycurl.Curl()
    crl.setopt(crl.URL, 'http://localhost:8080/v1.0/topology/links')
    crl.setopt(crl.WRITEDATA, b_obj)
    crl.perform()
    crl.close()

    get_body = b_obj.getvalue()
    raw_data = json.loads(get_body.decode('utf8'))

    links = {}  # here we save which ports create the link in the form of: 's1,s2': 's1-
        eth1,s2-eth1'

    for entry in raw_data:
        s1 = entry['src']['dpid'].lstrip('0')
        p1 = entry['src']['name']
        s2 = entry['dst']['dpid'].lstrip('0')
        p2 = entry['dst']['name']
        G.add_edge(s1,s2)
        edge = f"s{s1},s{s2}"
        links[edge] = f"{p1},{p2}"
    # ==============================

    # ========== build a tree from the graphs so to break the loops ==========
    # each node/switch is given a value based on the connections to other switches
    # and number of connected hosts. The bfs tree will start from the node with the
    # highest score. This way, it is more likely that switches with no hosts connected
    # will become leaves. Those switches can be shut down since no traffic will ever
    # need to pass there.
    bfs_value = {}
    for s in switches:
        s_degree = G.degree[s]
        bfs_value[s] = s_degree * 2 + host_per_switch[s]
    bfs_origin = max(bfs_value, key=bfs_value.get)

    # "optimized spanning tree"
    T = nx.bfs_tree(G,bfs_origin)
    new_links = list(T.edges())

    # shut unwanted links
    links = shut_links(links,new_links)

    # if a leaf switch has no hosts attached, it can be turned off
    # let's find them:
    switch_off = [x for x in T.nodes() if T.out_degree(x)==0 and host_per_switch[x]==0]
    # mark as "disabled" all ports of the found switches
    links, remove_edges = switch_off_unrouted(switch_off, links)
    for e in remove_edges:
        m = re.match("s(\d+)?,s(\d+)?",e)
        e1 = (m[1],m[2])
        e2 = (m[2],m[1])
        if e1 in T.edges():
```

```
65              T.remove_edge(*e1)
66          if e2 in T.edges():
67              T.remove_edge(*e2)
68      # ================================
69      original_network = G.to_undirected()
70      # ========== populate the tree graph with hosts ==========
71      hosts = []
72      ports_to_hosts = []
73      edges_to_hosts = []
74      for port,host in sw_hosts.items():
75          m = re.match("s(\d+)?-eth(\d+)?",port)
76          s = m[1]
77          p = m[2]
78          hosts.append(host)
79          e = (s,host)
80          edges_to_hosts.append(e)
81          T.add_node(host)
82          T.add_edge(s,host)
83          original_network.add_node(host)
84          original_network.add_edge(s,host)
85          ports_to_hosts.append(port)
86
87      network = T.to_undirected()
88      # ================================
89
90      # ========== print the new topology ==========
91      edges = original_network.edges()
92      stp_edges = network.edges()
93      node_colors = []
94      for n in original_network.nodes():
95          if n in switch_off:
96              node_colors.append('slategrey')
97              continue
98          if n in hosts:
99              node_colors.append('lightblue')
100             continue
101         node_colors.append('tab:blue')
102
103     colors = []
104     weights = []
105     labels = {}
106     for n in original_network.nodes():
107         if n in hosts:
108             labels[n] = ''
109         else:
110             labels[n]=n
111     for e in edges:
112         if e in edges_to_hosts:
113             colors.append('c')
114             weights.append(2)
115             continue
116         if e in stp_edges:
117             colors.append('b')
118             weights.append(4)
119             continue
120         colors.append('k')
121         weights.append(1)
122
123     nx.draw(original_network, edgelist=edges, edge_color=colors, width=weights, node_color=
            node_colors, labels=labels, with_labels=True)
124     plt.show()
125     # ================================
126
127     return links, ports_to_hosts, switch_off, network
```

A.3 Core functions of the Digital Twin

The code in Listing A.3 is the function that retrieves data about the traffic traversing each active interface per time unit. Notice that both ingress and egress bytes must be counted (line 31). The bytes delta (line 32) is then compared to the measure of the global variable `SENSITIVITY` to determine if the port is in *working* or *saturated* status (lines 36 and 38). Lists containing the ports in a defined

status are updated accordingly and returned to the main loop to be further processed.

Listing A.3: Function that retrieves the bytes routed through the interfaces

```
def get_working_ports(active_switches, active_ports, ports_speed):
    if DEBUG_LOG:
        print("GET WORKING PORTS")

    working_ports = []
    saturated = []

    for s in active_switches:
        switch_name = f"s{s}"

        curl = pycurl.Curl()
        byte_response = BytesIO()

        url=f"http://127.0.0.1:8080/stats/port/{str(s)}"

        curl.setopt(curl.CUSTOMREQUEST, 'GET')
        curl.setopt(curl.URL, url)
        if DEBUG_LOG:
            curl.setopt(curl.VERBOSE, 1)
        curl.setopt(curl.WRITEDATA, byte_response)

        curl.perform()
        curl.close()

        get_body = byte_response.getvalue()
        json_port = json.loads(get_body.decode('utf8'))
        port_stats = json_port[f'{s}']
        for stats in port_stats:
            if stats['port_no'] != 'LOCAL':     # LOCAL: port vs controller
                port_name = f"{switch_name}-eth{stats['port_no']}"
                port_bytes = stats['rx_bytes'] + stats['tx_bytes']
                delta_bytes = port_bytes - prev_port_bytes[port_name]
                delta_port_bytes[port_name].append(delta_bytes)
                prev_port_bytes[port_name] = port_bytes
                if port_name in active_ports:
                    if delta_bytes > (SENSITIVITY * 10):
                        working_ports.append(port_name)
                        if delta_bytes > (SENSITIVITY * ports_speed[port_name]):
                            saturated.append(port_name)

    return working_ports, saturated
```

Listing A.4 shows the function that performs the bit rate adaptation of the saturated or no-more-working ports. `change_link_rate()` is fed with data structures coming from `get_working_ports()` and from the network graph. The algorithm is greedy and it can be noticed that the amount of time units that must be spent in *saturated* status before the bit rate change is hardcoded (lines 18, 20 and 22).

Listing A.4: Function for adapting the bit rate to traffic volume

```
def change_link_rate(working_ports, saturated, active_ports, ports_speed):
    if DEBUG_LOG:
        print("CHANGE LINK RATE")
    for port in working_ports:
        used_ports[port] += 1   # global - keeps track of the amount of time a port has
            been in working status

    if DEBUG_LOG:
        print("CHANGE LINK RATE - got used ports")

    urls = []   # for pycurl
    data = []   # for pycurl
    to_change = 0
    # ========== Decide if bit rate must be changed ==========
    for port in used_ports.keys():
        if port in active_ports:
            rate_mbps = INITIAL_SPEED   # global
            if port in saturated:
                if used_ports[port] > 1 and INITIAL_SPEED<100:
                    rate_mbps = 100
```

```
                if used_ports[port] > 3 and INITIAL_SPEED<1000:
                    rate_mbps = 1000
                if used_ports[port] > 6 and MAX_10G:
                    rate_mbps = 10000
            if port not in working_ports:
                used_ports[port] = 0
                rate_mbps = INITIAL_SPEED

            # ========== set up for rest multirequest ==========
            if ports_speed[port] != rate_mbps:
                if DEBUG_LOG:
                    print("CHANGE LINK RATE - changing link rate")
                crl = pycurl.Curl()
                rate = str(rate_mbps * 1000 * 1000) # expressed in bits
                dpid = re.match("s(\d+)?-eth\d+",port)[1]
                dpid = dpid.rjust(16, '0')
                url = f"http://localhost:8080/qos/queue/{dpid}"
                d = json.dumps({"port_name": port , "max_rate": "10000000000", "queues": [{
                    "max_rate": rate }]})
                urls.append(url)
                data.append(d)
                to_change += 1
                ports_speed[port] = float(rate_mbps)
    # ==============================

    # ========== PyCurl MultiRequest ==========
    # making a request for each interface that needs to change is
    # too slow and makes the time units too long.
    # This way the time units are all of about the same length

    # Pre-allocate a list of curl objects
    m = pycurl.CurlMulti()
    m.handles = []
    for i in range(to_change):
        c = pycurl.Curl()
        c.fp = None
        c.setopt(pycurl.POST, 1)
        c.setopt(pycurl.URL, urls[i])
        c.setopt(pycurl.POSTFIELDS, data[i])
        c.setopt(pycurl.WRITEFUNCTION, lambda x: None)
        m.add_handle(c)

    while 1:
        ret, num_handles = m.perform()
        if ret != pycurl.E_CALL_MULTI_PERFORM:
            break

    # Cleanup
    for c in m.handles:
        if c.fp is not None:
            c.fp.close()
            c.fp = None
        c.close()
    m.close()
    # ==============================

    return ports_speed  # used to update the global variable
```

Bibliography

[1] Steve Morgan. Humans on the Internet will triple from 2015 to 2022 and hit 6 billion. https://cybersecurityventures.com/how-many-internet-users-will-the-world-have-in-2022-and-in-2030/#home/?view_1_sort=field_4|desc&view_1_page=4, July 2019.

[2] W. Stallings. *Foundations of Modern Networking: SDN, NFV, QoE, IoT, and Cloud*. Pearson Education, 2015.

[3] ONF. Software-defined networking: The new norm for networks. Technical report, Open Networking Foundation, April 2012.

[4] Open Data Center Alliance. Open data center alliance master usage model: Software-defined networking rev. 2.0. *White Paper.* Technical report, 2014.

[5] ONF. Software-defined networking: The new norm for networks. Technical report, Open Networking Foundation, April 2012.

[6] Fabrizio Granelli. Reti Programmabili Software Defined. https://www.youtube.com/watch?v=BcdZRMHLKfA, June 2022.

[7] ONF. SDN architecture, June 2014.

[8] Yiwen Wu, Ke Zhang, and Yan Zhang. Digital twin networks: A survey. *IEEE Internet of Things Journal*, 8(18):13789–13804, 2021.

[9] Edward Glaessgen and David Stargel. The digital twin paradigm for future nasa and u.s. air force vehicles. 04 2012.

[10] Michael Grieves. Digital twin: Manufacturing excellence through virtual factory replication. 03 2015.

[11] Juan Deng, Qingbi Zheng, Guangyi Liu, Jielin Bai, Kaicong Tian, Changhao Sun, Yujie Yan, and Yitong Liu. A digital twin approach for self-optimization of mobile networks. In *2021 IEEE Wireless Communications and Networking Conference Workshops (WCNCW)*, pages 1–6, 2021.

[12] Nokia Networks. *Flatten network energy consumption, Technology Vision 2020, White Paper.* 12 2013.

[13] Arnaud Adelin, Philippe Owezarski, and Thierry Gayraud. On the Impact of Monitoring Router Energy Consumption for Greening the Internet. In *IEEE/ACM International Conference on Grid Computing (Grid 2010)*, pages p. 298–304, Bruxelles, Belgium, October 2010.

[14] Sergio Ricciardi, Davide Careglio, Ugo Fiore, Francesco Palmieri, Germán Boada, and Josep Solé-Pareta. Analyzing local strategies for energy-efficient networking. In *Lecture Notes in Computer Science*, pages 291–300, 01 2011.

[15] Mike Bennet, Ken Christensen, and Bruce Nordman. *Improving the Energy Efficiency of Ethernet: Adaptive Link Rate Proposal.* 07 2006.

[16] Baoke Zhang, Karthikeyan Sabhanatarajan, Ann Gordon-Ross, and Alan George. Real-time performance analysis of adaptive link rate. In *2008 33rd IEEE Conference on Local Computer Networks (LCN)*, pages 282–288, 2008.

[17] The mininet network emulator.

[18] Ryu sdn framework.

[19] ONF. *OpenFlow Switch Specification (Version 1.3.0 - Wire Protocol 0x04)*. Jun. 2012.

[20] Comnetsemu public repository.

[21] Mininet python api reference manual.

[22] Open vswitch.

[23] Mininet mailing list archives - [mininet-discuss] interface link_speed/bandwidth 10g harcoded.

[24] Ryu source code.

[25] Various data sheet from cisco networks.