Politecnico di Torino

A Web-Based Platform for Experimenting with Virtual Networks Adaptations via Reinforcement Learning over the GENI Testbed

Master’s Degree in Computer Engineering

Supervisor(s):
Prof. Guido Marchetto, Supervisor
Prof. Flavio Esposito, Co-Supervisor
Dr. Alessio Sacco, Co-Supervisor

Candidate:
Enrico Alberti

Politecnico di Torino
October, 2022

I would like to dedicate this thesis to my loving parents

Abstract

Network emulators and simulation environments traditionally support computer networking and distributed system research. The continued use of multiple approaches highlights both the value and inadequacy of each approach. To this end, several large-scale virtual network testbeds such as GENI have emerged, allowing testing of a networked system in controlled yet realistic environments. Nevertheless, setting up those experiments first, and integrate machine learning models later to these deployments is a challenge.

In this thesis, we propose the design and implementation of a web and command line tool that integrate Reinforcement Learning (RL) with a virtual network experiment using resources acquired within the GENI testbed. After some configuration setup, users draw the network topology of their experiment and then reserve the GENI resources with a button push. A reinforcement learning algorithm is then launched to learn and steer traffic dynamically based on emulated traffic network conditions. The back end includes a Software-Defined Network (SDN) controller that reprograms the routing table of Open Virtual Switches after reading the output of the RL model.

While in this thesis the system focuses on the deployment of experiments for virtual network adaptation, the platform can be easily extended to other network management mechanisms and machine learning algorithms.

Contents

Chapter 1

Introduction

The Internet grew and is constantly growing fast, and as its size increases, more difficulties appear. As a consequence, it is impossible for a single person or a team of people manually configure and handle what is happening in real time in the 'big' network. Moreover, with increasingly demanding requirements for more and more flexibility, machine learning and autonomous networks, which constantly monitor and adapt themselves, are born. In this chapter, we will intensely discuss this and more motivations that led us to develop the project, we will highlight our contribution and the results we obtained.

1.1 Motivation

In the early days of computer networks, given the small network size, it was elementary to manage the network. As time passed, the number of nodes present increased exponentially, making management increasingly complex. One recent trend of addressing this problem was the inclusion of automation, leading to the so-called "self-driving networks". Many are the benefits of autonomous networks. Just think about the flexibility of a system that automatically detects the changes in the environment and evolves, adapting to these changes, the increase in performance, and the energy saved. While the use of Reinforcement Learning (and thousands of hours spent by our computer engineers) can lead to automated networks, the entire design and experimentation process is still tedious. In this thesis, we aim to simplify the testing with reinforcement learning algorithms even for people without any background in the network field, and easily run these algorithms in production networks. Our focus is on providing the enabling technologies to project and deploy these models, leaving room for machine learning and artificial intelligence specialist to play with specific parameters of the model, e.g., number of steps and number of neurons.

Another essential motivation that pushed us into this project is the need to experiment not anymore in an emulated environment but instead in a real one. Mininet[1] is a valuable tool that allows experimenting with networks, but it still remains an emulator, and results may be misleading. Mystique [1], the base of the generalized Reinforcement Learning-based Virtual Network Adaptations (RLVNA) algorithm we decided to implement in this project, was tested in Mininet. We decided that moving to a real environment was better for improving the algorithm and for having more concrete data, and the GENI testbed (Global Environment for Network Innovations)[2] is one of the best.

Finally, we wanted to have something simple but as dynamic as possible. Having a static network is easy to create and handle, but once an experiment is done, it requires changing many lines of code for the development of the following experiment, even if it is needed to change just one parameter. Designing and developing a dynamic code allows you to change settings faster with more flexibility and does not require starting from scratch between experiments. Of course, realizing a dynamic system bring with it many difficulties.

1.2 Thesis contribution

Behind the motivations explained before, we realized a system able to implement a reinforcement learning algorithm that provides self-driving network, as dynamic as possible over a real traffic generator testbed.

The idea was to develop a generalized Reinforcement Learning-based Virtual Network Adaptations (RLVNA) algorithm. This reinforcement learning algorithm is composed of two central cores. The first core is the presence of a Ryu controller for querying the OpenVSwitches to collect each switch's data. The second core is

[1] http://mininet.org/
[2] https://www.geni.net/

the machine learning model which, once received the measurements from the Ryu controller, evaluates if it is better to activate or deactivate new support switched and new paths in the network. The algorithm's purpose is clearly to scale up and down, reducing the congestion in the network when it occurs while simultaneously avoiding energy consumption when not needed. This development task was challenging, especially when dealing with the configuration of the flows of each switch at startup or in the path change in case a new support switch is activated.

We encountered several problems also when dealing with the GENI testbed. The first problem itself is the quality of available nodes. In GENI, it is possible to choose an aggregate all over the United States and reserve resources. Many aggregates had limitations on the number or type of machines to reserve, and often reserving even ten machines required more than 20 minutes (as long as the underlying GENI infrastructure can support it). Because of the low performance of the architecture itself, we tried to deal with GENI as much as possible in the backend, avoiding the end users spending their time on such useless questions. For example, once the desired resources are ready and available, an operation that could require much time, the system allows multiple experiments on the same topology, changing just the parameters of the machine learning algorithm, such as the support switches or the penalty in the model. In this way, the final users could maximize their productivity by focusing on the research of the best parameters of an algorithm.

Chapter 2

Background and related work

This chapter will discuss the main core techniques used to develop this project. In particular, we will mainly explain what Software Defined Networks (SDN) are and how they are used, what Reinforcement Learning (RL) is used for, and how to deal with the GENI testbed.

2.1 Software Defined Network

Nowadays, networks are continuously becoming more significant and more complex to manage. In particular, networks are challenging to configure, handle in case of failure, or adapt in case of load changes mainly because networks are vertically integrated, which means that the control and data planes are integrated into a unique component. Several efforts have been made to make computer networks more programmable. This is one of the reasons that led to the birth of Software Defined Networks (SDN).

2.1.1 Technical overview

Software-Defined Networking (SDN) is a protocol that decouples the data flow plane from the control flow plane and allows the management of many devices and services through APIs and high-level languages. One of the first implementations of SDN is Open Flow independent also from proprietary software. In particular, in SDN, the controller is the one with the logic and the one which decides the flows from source to destination; the switches instead are a -high-speed- dumb device with no knowledge but with the only objective to forward packets. Data plane switches use match (IP, mac) / action (drop, forward, edit) as routing rules. A need has arisen to use a Context-based control plane where some matches are defined, and a default rule is left towards the controller in case of particular packets. The controller (which is centralized at least logically) could implement several features such as routing, NAT, and Fire-walling. It can be reactive (no initial rule) or proactive (initial rules already defined). The main uses are in the data-center and virtual machine environments. These and more deep concepts about SND can be found in [2, 3].

Moreover, to summarize, among the advantages of SDN we remember:

- the global view of the controller which allows the best decisions
- the possibility to define rules using the "Big Switch" abstraction
- the possibility to have simpler and faster data switch
- the possibility to use different virtualized network operating systems, also known as "slicing"
- last but not least the increase of scalability (limited due to the CAP theorem).

2.1.2 Architectural overview

From the architectural point of view, we can see the implementation in Figure 2.1.

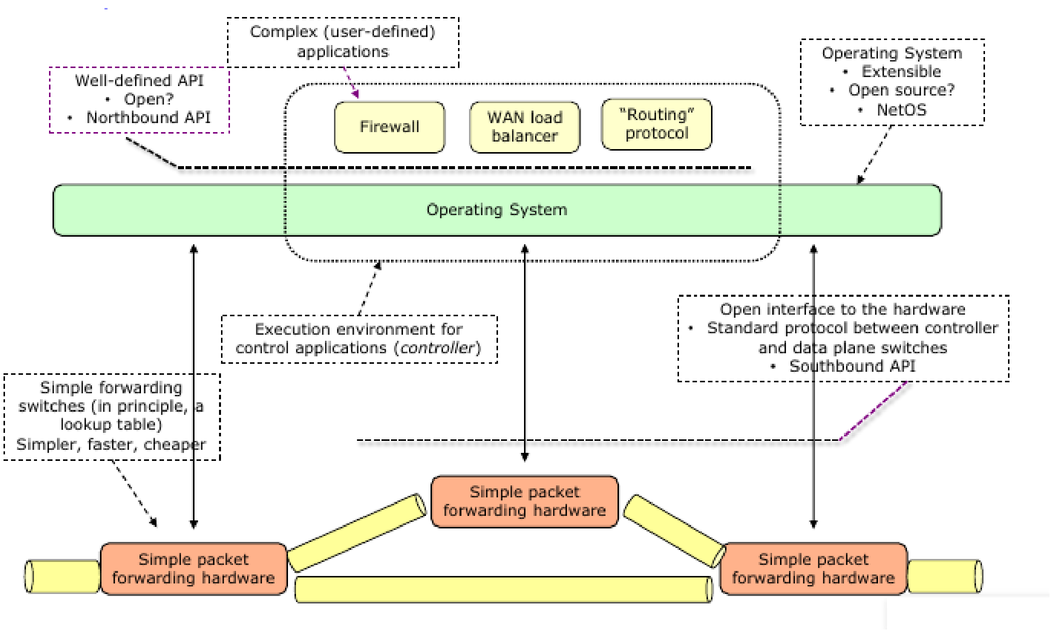

Complex (user-defined) applications
Well-defined API
• Open?
• Northbound API
Firewall
WAN load balancer
"Routing" protocol
Operating System
• Extensible
• Open source?
• NetOS
Operating System
Execution environment for control applications (*controller*)
Open interface to the hardware
• Standard protocol between controller and data plane switches
• Southbound API
Simple forwarding switches (in principle, a lookup table)
Simpler, faster, cheaper
Simple packet forwarding hardware
Simple packet forwarding hardware
Simple packet forwarding hardware

Fig. 2.1 SDN Architecture. Adapted by [4]

To recap, here we highlight the main point of the architecture:

1. **Network Applications**: We remember the main categories such as traffic engineering, mobility, wireless, measurement and monitoring, security and dependability, and data center networking.
2. **Programming Languages**: The use of high-level language brings many advantages such as modularity, simplicity, and portability.
3. **Language-based Virtualization**: Main features are modularity, abstraction, and slicing.
4. **Northbound Interface**: These are the APIs between controllers and users.
5. **Network Operating System**: It is the brain of the architecture. It provides abstractions, services, and APIs to developers hiding low-level details.
6. **Network Hypervisor**: Allow to share hardware between different virtual machines and brings advantages from virtualization like scalability, flexibility, slicing, migration, and tenant isolation.
7. **Southbound Interface**: It connects the switches to the controller. OpenFlow is one of the most used protocols.

8. **Network Infrastructure**: It is composed of switches and routers that, given a matching, apply an action.

2.2 Reinforcement Learning

Reinforcement Learning (RL) is a part of Machine Learning (ML) with increasing use[5, 6]. Nowadays, it is used a lot in the field of video games but also research because it aims to find the best solution by experimenting with different decisions. A formal definition is provided in [6]: *"Reinforcement learning is the problem faced by an agent that learns behavior through trial-and-error interactions with a dynamic environment."*.

This section will give a brief overview of Reinforcement Learning without going into further detail. In the end, to underline the importance of the dictionary, we will analyze the central key concept providing their definitions.

2.2.1 Overview

The reinforcement learning idea is elementary and probably re-adapted by everyday life. An agent interacts with an environment and receives a positive reward if the action is correct and a negative reward if the action is wrong. We can retrieve this simple idea from ordinary life. Let us think about our children. They are the agents in a completely new environment. If they break something (negative action), the parents give them some punishment (negative reward); instead, if they say hello (positive action), the parents give them some positive encouragement (positive reward). After many and many tries, the children understand not to break objects and to say hello to people even if the environment changes and it is no longer their home.

We can review these simple concepts in Figure 2.2.

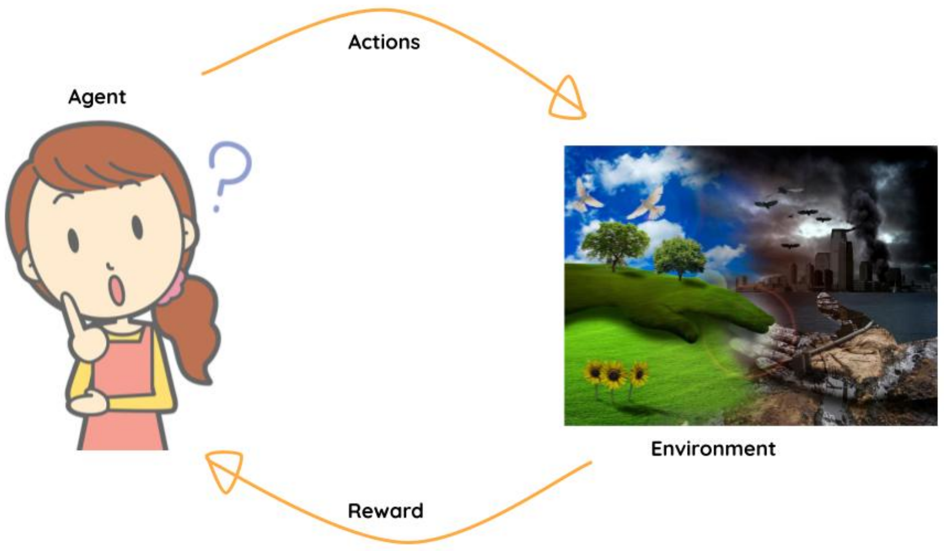


Fig. 2.2 Reinforcement Learning schema

As we explained at the beginning, many are the applications of RL. Since the objective of the project is not to deeply analyze these techniques, we will not investigate more and leave room for more specific jobs. For completeness, we will provide just some examples as reported in [7]. They discuss various applications of RL, including games, robotics, Industry 4.0, smart grid, intelligent transportation systems, computer systems, natural language processing, computer vision, business management, finance, healthcare, and education.

2.2.2 Glossary

To better fix the central concept of Reinforcement Learning, we report the terminology used here.

- **Agent**. The agent is the operator in charge of executing some action in the environment and receiving feedback. He has a clear objective in mind.
- **Environment**. The environment is something outside the control of the agent. It is the world which the agent interacts with. It receives actions as input, which modify the internal state. It returns the state as output together with rewards.

- **Action**. As explained before, it is the possible combination of movement the agent decides to take for passing in a different state while searching for the best reward.
- **State**. It is a snapshot of the environment. The relevant information is stored inside the state and changes depending on the action it receives.
- **Reward function**. It is the feedback we receive from the environment. In general, the rewards are numbers. The higher the number, the better the reward.

2.3 Self-driving networks

As networks continue to evolve and become larger and more complex, there is a growing need for automation and self-management of networks, possibly without human intervention. In this section, we will focus on an "autonomous" or "self-driving" network, which are networks that constantly monitor themselves, and depending on the changes in the environment, they adapt consequently. In [8] the author explains the reason behind the autonomous network. He claims that since the network is so complex, we should use a different approach. Instead of learning and configuring from a closed analysis, which sometimes requires the network operator, we should take the decision automatically based on predictions of machine-learning-based models. A further example of empowering self-driving networks could be found in [9]. Only for completeness, we say there exist many ways for exploiting self-driven networks. The use of intent-based networking , as explained by [10] is a natural example. In this case, the idea is to interact with the network operator with a simple chat interface using natural language, extract intent and then translate it into SDN rules.

2.3.1 Mystique

Traditional threshold-based self-management is not always able to model very complex networks well. For this reason, it was decided to use a "self-driving" solution called Mystique[1] that, using Multi-Agent Reinforcement Learning (MARL), auto-scales adapting to the network load at any given time, learning from the link the current load and, establishing the minimum number of resources needed as shown in Figure 2.3.

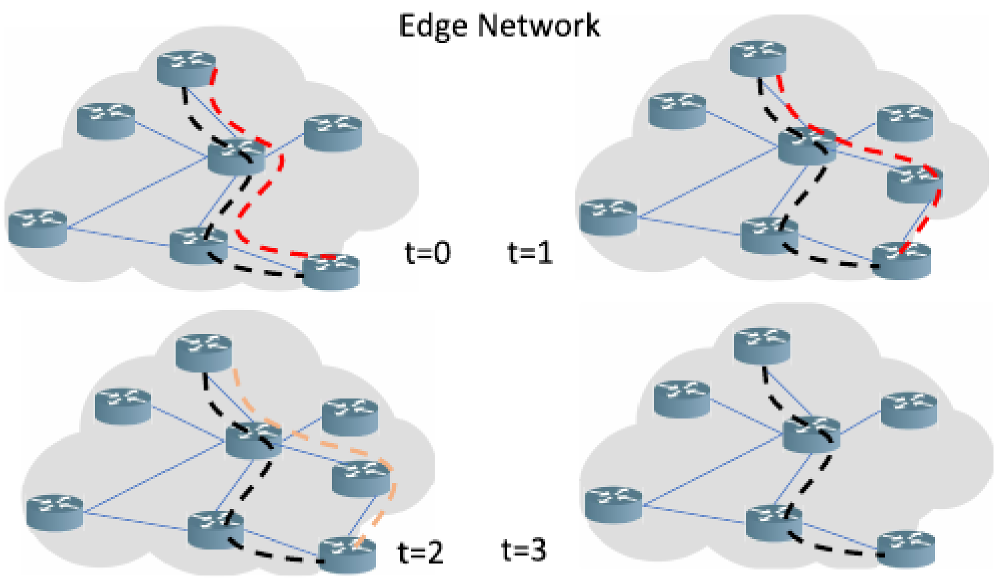


Fig. 2.3 Mystique example scaling up and down. Adapted by [1]

Among the benefits of auto-scaling, we remember the reduced cost of unused (and switched-off) resources, the ability to properly manage temporary peaks and the flexibility and ease of managing large networks. Moreover, behind Mystique advantages, we cannot forget to mention the maximization of the users' Quality of Experience (QoE), the management of possible failures and the reduced cost of energy for the switched-off unused switches as reported in [1, 11].

From the architectural point of view, we can see the implementation in Figure 2.4.

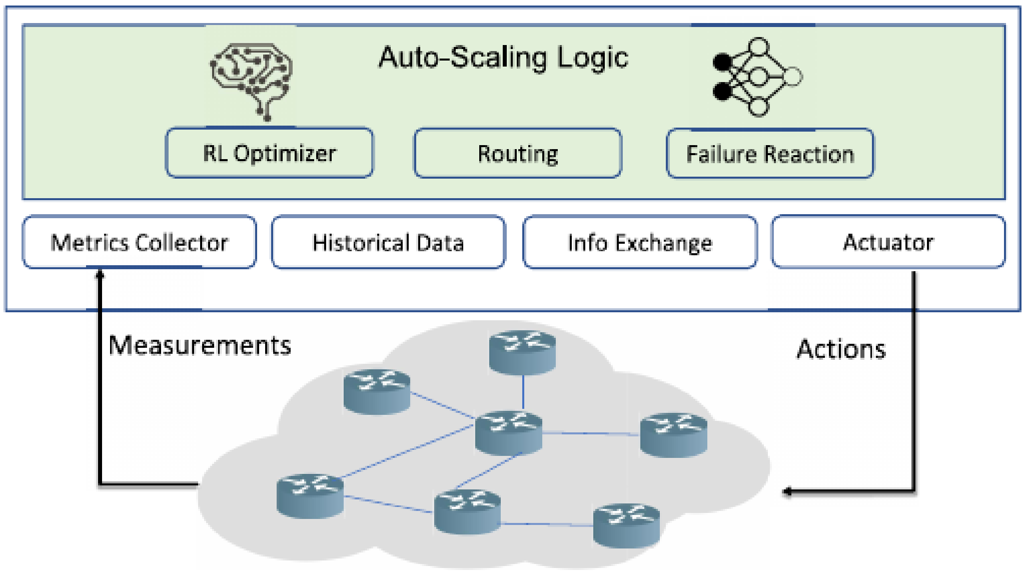


Fig. 2.4 Mystique Architecture. Adapted by [1]

As we can see in Figure 2.4, the main features of the RL algorithm are the following:

- **Reinforcement learning (RL) module**: It interacts with other processes for collecting information, selects the best action, and notifies it.
- **Routing**: Each agent dynamically creates and destroys switches and links to handle failures or changes in traffic.
- **Failure reaction**: Since the RL model mainly focuses on action to take, the Failure reaction module mainly desire to manage the failure detection and reaction.
- **RL Optimizer**: Aimed at finding subsets of networks that satisfy current traffic conditions and avoid wasting resources.

By summarizing the whole process, the switches collect the measurements and report them periodically to the network controller, where the *metric collector* component resides. The agents, taking into account *historical data*, output the best decision for the network. When the decision is made, the *actuator* receives the output of the active components and performs the appropriate commands. Moreover, the actuator also pushes the new routes into the network.

In addition, Mystique can leverage multiple controllers, each of which is responsible for a sub-network to improve system scalability and resiliency. The agents interact to obtain information about the global topology, as the routing decisions should consider a global view.

2.4 Testbed and tools

A testbed is a research platform for conducting replicable experiments. We can think of them as a giant distributed network spread across a country, and they offer the possibility to reserve nodes for the researcher's purpose. Among the most used testbed, we remember GENI (Global Environment for Network Innovations), FABRIC, Chameleon, CloudLab, and Mininet.

2.4.1 Geni

In the mid-2000s, experimentation on large and distributed networks was a problem to be addressed. To this end, the Global Environment for Networking Innovations (GENI[1]) was born[12, 13]. It is a platform that allows the creation of a virtualized, and therefore flexible and customizable, network-based on geographically distributed physical nodes [14]. The physical nodes are all over the United States as depicted in figure2.5 and this confirms the power of architectures like this. Different entities' resources are connected using 100G Internet2 layer2 service. Although not all nodes are often very powerful, GENI is particularly used to perform experiments on vast networks (e.g. traffic analysis, new routing algorithms, new layer 2 and 3 protocols), experiments for not-IP layer 3 protocols, and programmability in the network core (programmable switches) because it allows the installation of custom software or even custom operating systems on these compute resources. GENI is not perfect, but besides all limitations in our analysis, it is the best up to now, and because of these reasons we choose it for the implementation of our project.

[1]https://www.geni.net/

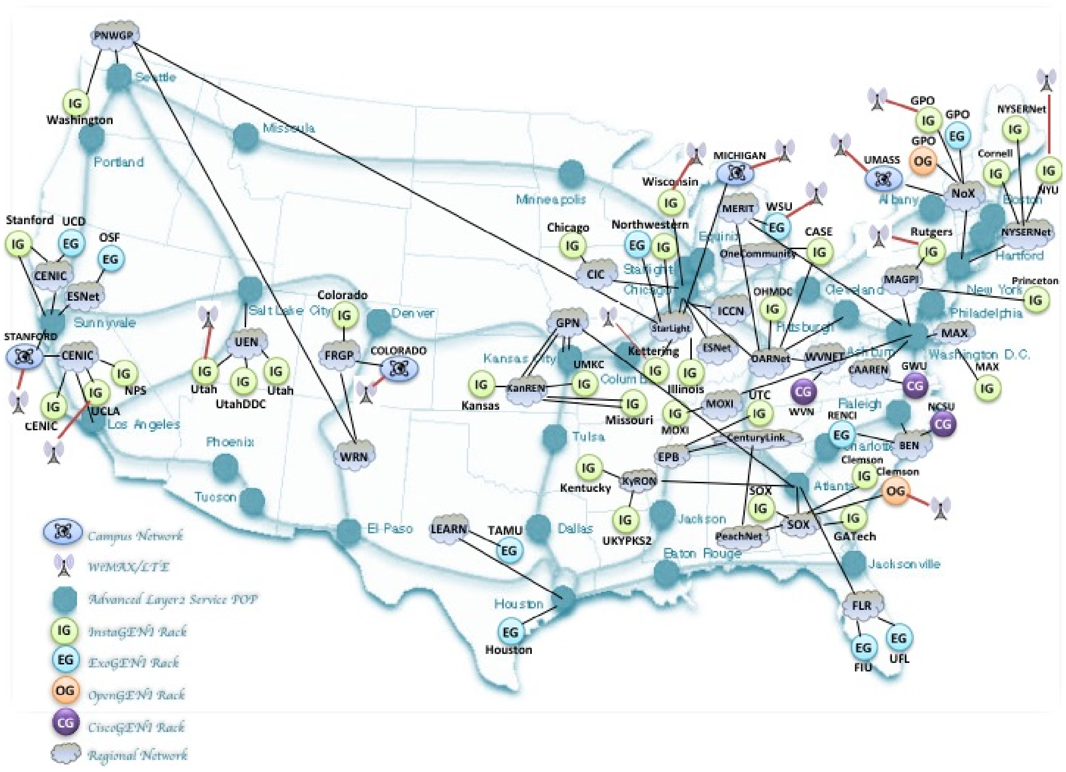


Fig. 2.5 GENI Map. Adapted by [15]

The GENI application fields are not limited only to research and education. For completeness, we report here some of the possible applications. It is used in Elastic Optical Networking [16], wireless and 4G [17, 18], services that address advanced manufacturing needs [19], scientific computing experiments [20], multi aggregate communication [21], virtual network migration [22], and electric power management with SDN [23].

The main concepts in GENI are the following:

- **Project**: It is a way to organize and group people and experiments. It is possible to configure roles and rights for resource access.
- **Slice**: It is a portion of the network where it can run experiments that remain isolated from each other.
- **Aggregates**: They provide resources to the slice. A Slice could obtain resources from multiple aggregates (University, Clouds).

- **The GENI AM API and GENI RSpecs**: AM API is a way to list/request/ status/delete resources from aggregates using standard API. RSpec's are XML documents used for describing resources you are asking (request RSpecs) Alternatively, you received (manifest RSpecs) or the lists of resources in an aggregate (advertisements RSpecs).

For a deeper discussion and a user tutorial, we would like to suggest [24].

2.4.2 Fabric

FABRIC[2] is a vast and powerful infrastructure used for at-scale experiments on networking, cybersecurity, mixed environment (Wireless, IoT, HPC), machine learning, and artificial intelligence field. It is not used in Computer Science only, but also in Weather and Climate Prediction, Physics, Space, Astronomy, and cosmology. Moreover, thanks to FABRIC Across Borders (FAB) extension the nodes are more distributed in the world (USA, Asia, Europe, and South America) allowing better experimentation.

Let us highlight some key concepts on FABRIC.

- It is a multi-user utility for concurrent experiments of different sizes.
- It is designed for small-life production workloads.
- It is highly distributed and highly extensible.
- It is not an isolated testbed. It Allows experiment slices to connect to a variety of networks and resources.
- It is a place for experimenting on new Internet architectures and algorithms, not a fast new pipe for data delivery.

2.4.3 Chameleon

Chameleon[3] is a compelling large-scale platform used nowadays for many purposes, like operating system development and virtualization, SDN, power and resource

[2]https://fabric-testbed.net/
[3]https://www.chameleoncloud.org/

management, and artificial intelligence. Chameleon gives users complete control of the switches and software stack, increasing isolation and customization. Moreover, a small amount of resources is configured as a virtualized KVM cloud to balance the need for finer-grained resource sharing. Chameleon adapted OpenStack to provide its services, an already well-defined standard known by millions of users.

2.4.4 CloudLab

CloudLab[4] is an infrastructure mainly used for computer science experiments. It provides fast access to bare metal. There are thousands of machines across the US and in addition, CloudLab inter-operates with existing testbeds like GENI increasing the available resources. It provides strong isolation supporting hundreds of simultaneous slices each possibly, with different environments needs. It supports known standards such as OpenStack, Hadoop, and Kubernetes.

2.4.5 Mininet

Very different is Mininet[5] from the testbed explained before because it is just an emulator. However, even if it is an emulator is an excellent first approach tool to use in a personal computer for developing and testing an architecture [25]. We will not spend too much time in Mininet, because it is out of our scope but since it is one of the best emulators, we will report here just the official description. *" Mininet as said earlier is an emulator however it creates a realistic virtual network, running real kernel, switches and application code, on a single machine (VM, cloud, or native), in seconds, with a single command. Because you can easily interact with your network using the Mininet CLI (and API), customize it, share it with others, or deploy it on real hardware, Mininet is useful for development, teaching, and research. Mininet is also a great way to develop, share, and experiment with Software-Defined Networking (SDN) systems using OpenFlow and P4. Mininet is actively developed and supported and is released under a permissive BSD Open Source license."* [26]

[4]https://www.cloudlab.us/
[5]http://mininet.org/

Chapter 3

System design and implementation

In this project, we tried to develop a user-friendly platform for independent network experiments using the GENI testbed. The system is transparent and could be used even by users without any background in the network field. In particular, the final user does not have to worry about how nodes are reserved, which IP addresses to assign, or which aggregate puts the experiment. The users can only draw the topology they want to experiment with and ask the system to make the resource reservation. Then they could focus on customizing the settings and evaluate the best combination.

Behind the hook, an experiment uses many essential concepts such as Software Defined Network (SDN), Self-Driving networks, and Reinforcement Learning algorithms. In particular, since we used SDN, the idea of each experiment is that it presents a controller node that implements the logic of the system and many OpenVSwitches nodes that implements the network itself. Finally, there are the host machines that implement end-user communication. As said at the very beginning, we developed the project in GENI testbed, which allows the collection and generation of real traffic present on the Internet links very different from an emulated environment as Mininet could be. We developed this platform using a reinforcement learning algorithm, whose main objective is to scale up and down based on the traffic in the network. In particular, given a network, the idea is to have some spare switches used as support. In the beginning, the support switches are off. Depending on the amount of traffic exchanged in the other switches present in the network, the machine learning model decides if it is better to activate the support switches to reduce the congestion and later deactivate them if the traffic is low.

In this chapter, we will discuss the three cores of this platform which we briefly explained in this introduction: The Frontend, the Backend, and the development of a generalized Reinforcement Learning-based Virtual Network Adaptations (RLVNA) algorithm. You can see a summary architecture in Figure 3.1

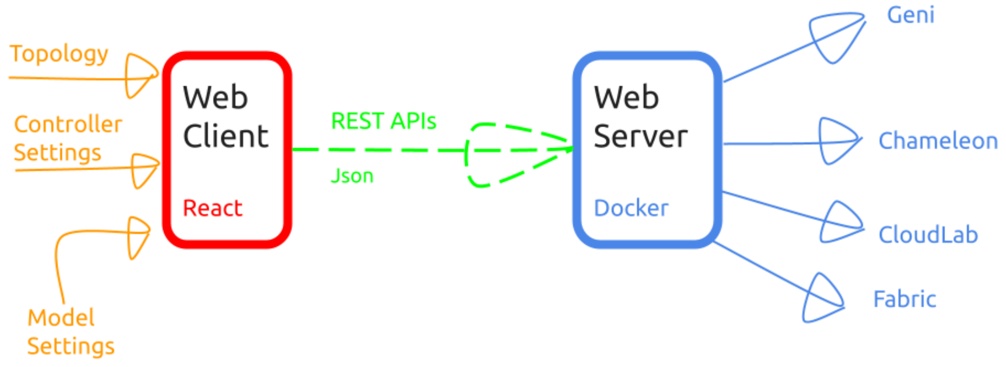


Fig. 3.1 NGI platform

It is important to underline that the schema in Figure 3.1 is as general as possible, but as we said many times, the project in this first version implements only experimenting with GENI testbed and with the RLVNA algorithm we developed.

3.1 Frontend

The following section will discuss the Frontend design and programming implementation. The frontend offers basic functionality, such as the creation of custom topology and the possibility to run Machine Learning algorithm for network management inside these custom topologies. We did not spend too much time on the care of the graphical implementation because we prefer to focus on the variety and correctness of the offered functionality. Furthermore, another reason for not spending too much time on the graphical implementation is that in the future, we would like to change the actual implementation allowing us to experiment with different machine learning algorithms. As we wrote many times, for the moment, we implemented the project to run with one single reinforcement learning algorithm. It means that we created the webpage with a well-defined and strict structure but what we want to achieve in the future is to remove these limitations and use this platform to help ML developer to test their algorithms in their custom topologies.

In the Frontend design subsection, we will report some screenshots of the web page and explain the possible actions and execution order for using the platform correctly. In the frontend programming implementation, we will focus on the technical details and the technologies we used to implement this reactive web page.

3.1.1 Frontend Design and GUI

The Frontend is the interactive point for the end user. It is as intuitive as possible, but we inserted some information boxes to explain the settings even so for completeness. The Frontend aim is to allow the user to ask for the desired resources and customize the reinforcement learning algorithm's settings.

For a better understanding of what we are referring we report now in Figure 3.2 the first version of the web page as it presents at the very beginning.

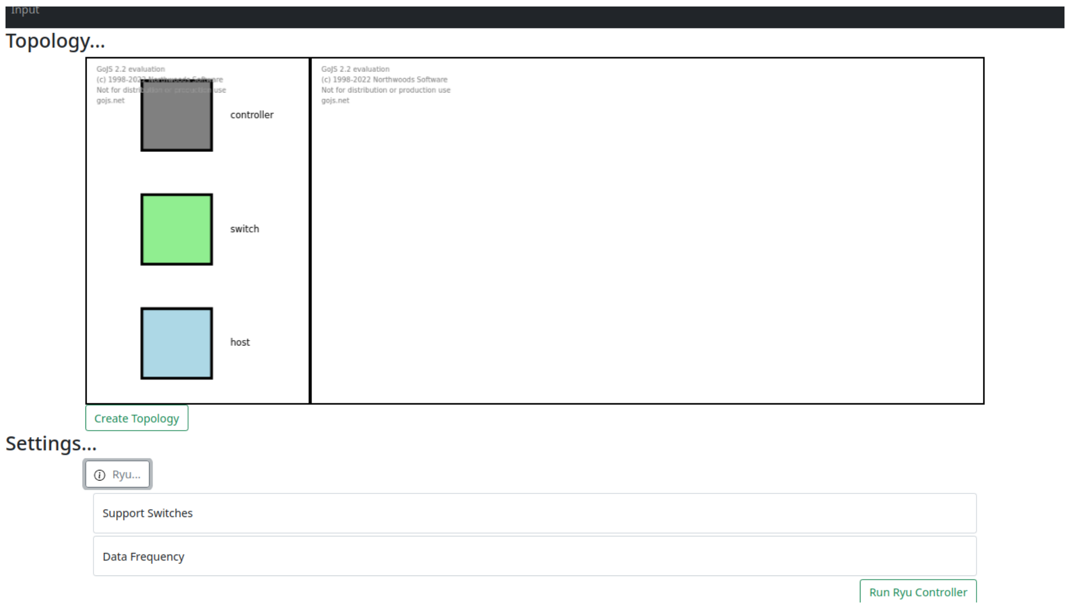


Fig. 3.2 Example of frontend

In particular, from Figure 3.2 we can see the two main sections that compose this web page:

1. **Topology definition**
2. **Machine Learning settings customization**

In the first half of the page, there is the topology definition that is composed of a big box where the users can draw the topology they want to test. The second half is for the reinforcement learning algorithm settings. As we said, the settings often refer to the RLVNA algorithm we implemented, so they adapt to the parameters it expects to receive. The first and second parts of the page present many buttons used to interact with the backend, but we will discuss this later.

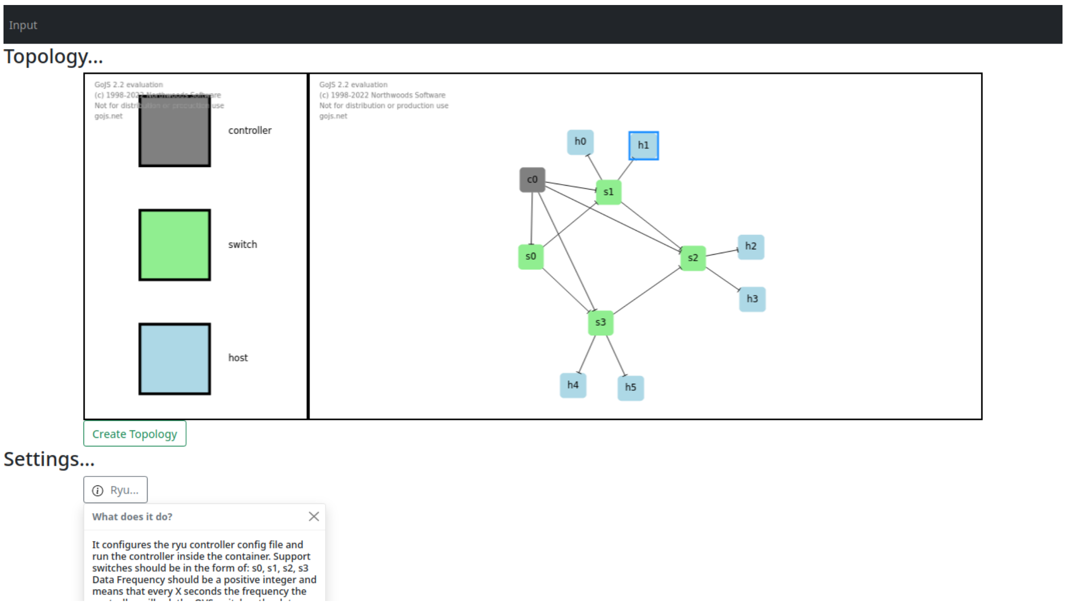
We report here the steps the user should execute to run an experiment:

1. Draw the custom topology
2. Ask for resource reservation and wait for a positive response
3. Configure and run the Ryu controller
4. Configure the ML model

5. Get the ssh command to log inside the nodes

We defined very generic steps, especially for number five, because now we will focus step by step on each point and with the help of some screenshots, we will better discuss what we are referring to.

In Figure 3.3 it is possible to understand the Topology settings better, particularly steps one and two.



It configures the ryu controller config file and run the controller inside the container. Support switches should be in the form of: s0, s1, s2, s3 Data Frequency should be a positive integer and means that every X seconds the frequency the

Fig. 3.3 Example of frontend with custom topology

The big box in the first half of the page represents the topology section. We can see that it is composed of two parts. On the left, it is possible to see a palette that shows the three possible kinds of nodes it is possible to reserve:

- Controller
- Switch
- Node

There are three possible kinds of nodes because we have in mind to experiment with Software Defined Network. The controller is the brain of all algorithms, and it contains all the logic to implement. The switches represent the physical mean of communication the hosts use to exchange messages. Since we are talking about SDN, we are referring to OVS, which means that at the beginning, the switches do not have any flows to allow hosts to communicate. Because of this, another controller task is to take care of and properly configure them. Hosts are the simplest kind of node because they are interconnected raw machines. We leave the freedom to the end users to play as much as they want with the hosts because they are highly customizable. In particular we give possible use cases examples by reporting the script present in *NGI-support*[1] public repository.

In the right part of the box instead, there is initially an empty box, as you can remember from Figure 3.2 and later, while the user draws through drag and drops from the palette, the topology schema is populated. It is possible to see an example of topology drawn by a user in Figure 3.3. In that particular example, there is one controller node where the docker with the RLVNA algorithm will be inserted and configurated, four OVS nodes -one of which (s0) will be the supportive [2] one-, and six hosts available for traffic generation.

To complete the description of the first half of the web page, we want to highlight the importance of the Create Topology button. The drawn topology is parsed and converted as JSON by clicking on it. Later, the serialized topology is sent through a REST API to the backend that handles the request and provides a positive or negative response. It is essential to underline that this operation could require many minutes, and to avoid mistakes, we decided to disable the button between the request and the response. The response for the moment is provided by a simple string flag inside a simple modal component, which is more or less a popup. In the future, we were thinking about providing to the user both the RSpec and the Manifest files generated from GENI when a new topology is requested. These files are very useful, especially the manifest because it contains all the descriptions of the nodes, their mac addresses, the IP addresses, and the physical location that GENI used to create the target topology.

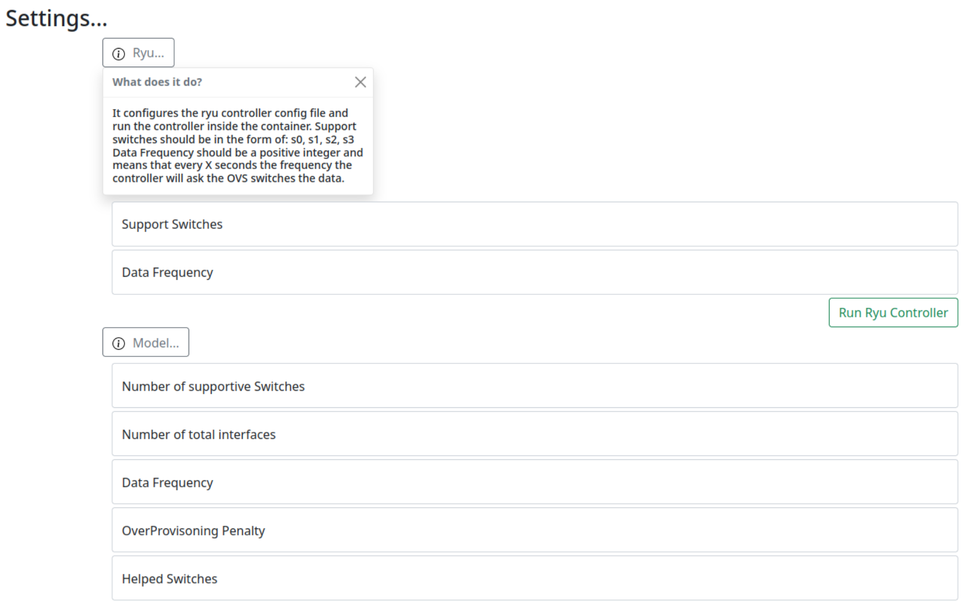
We will move our focus now to the second half of the web page, divided into three other subparts:

[1] https://github.com/Enrico-git/NGI-support

[2] As a marginal note, we remember that the concept of 'support switch' is purely linked to the RLVNA algorithm, as explained in the background section. We are forced to create (and eventually disable) it from the beginning because GENI does not allow to modify by creating and destroying nodes at runtime.

- Ryu controller configuration
- ML model configuration
- SSH nodes

To better visualize what we are talking about, it is possible to see the first two subparts in Figure 3.4. This figure is also perfect for describing RL algorithm's customization and the previous use case of the steps 3 and 4.


Settings...

Ryu...

What does it do?

It configures the ryu controller config file and run the controller inside the container. Support switches should be in the form of: s0, s1, s2, s3 Data Frequency should be a positive integer and means that every X seconds the frequency the controller will ask the OVS switches the data.

Support Switches

Data Frequency

Run Ryu Controller

Model...

Number of supportive Switches

Number of total interfaces

Data Frequency

OverProvisoning Penalty

Helped Switches

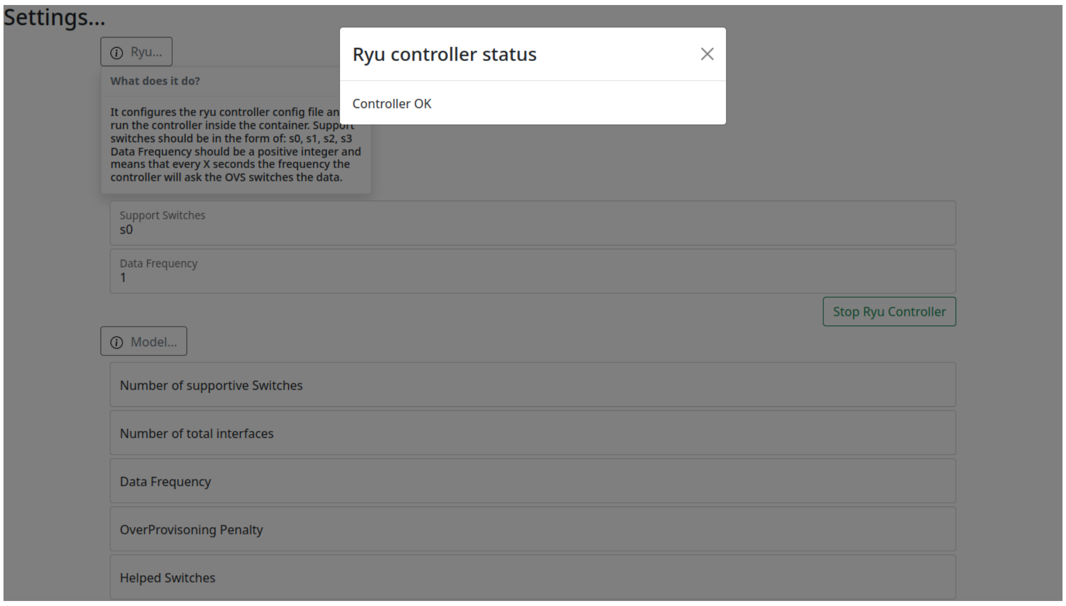
Fig. 3.4 Example of frontend with focus on the settings

These customizations are strictly linked to reinforcement learning algorithm parameters. This means by default that if in the future we want to use this platform with different machine learning algorithms probably, the settings to configure will be different.

From Figure 3.4 it is possible to see that there is an information box before each subpart (Ryu, Model, SSH). This information box is closed at the beginning and opens by clicking on it. It explains the expected parameters and the reason for the subsection itself. For example, the Ryu controller configuration needs the list of support switches and the frequency with which the Ryu controller should ask the switches the statistics. The model is more customizable than the Ryu controller. It needs the number of supportive switches, the data frequency, which means how often the model should ask the Ryu controller for the information, the overprovisioning penalty, which is a value that discourages the use of the support switches in the machine learning model, and the list of helped switches.

To improve the clarity of the configuration we explained right now, we report a numerical example of the topology depicted in Figure 3.3. For the Ryu controller, we insert ’s0’ as a supported switch and 1 second as data frequency. It means that every 1 second, the Ryu controller will ask the switches s0, s1, s2, and s3 for the statistics sent and received in each port. For the ML model, the number of support switches is 1, the data frequency could be 1, the penalty could be 300 (empirical), and the helped switch is s2. It means that every 1 second, the ML model will ask the Ryu controller all the switches statistics. It will evaluate a reward thanks to the penalty and the value in the helped switch, and it will send to the Ryu controller the action it decides to be the best.

We explained in detail the configuration, but before the Ryu controller and ML model run, the user has to click on the respective button of each subsection and wait for the response. As we said before, even in this case, the response is a simple string inside a modal. Figure 3.5 shows an example of the response.


Settings...

Ryu...

What does it do?

It configures the ryu controller config file an run the controller inside the container. Support switches should be in the form of: s0, s1, s2, s3 Data Frequency should be a positive integer and means that every X seconds the frequency the controller will ask the OVS switches the data.

Ryu controller status

Controller OK

Support Switches: s0

Data Frequency: 1

Stop Ryu Controller

Model...

Number of supportive Switches

Number of total interfaces

Data Frequency

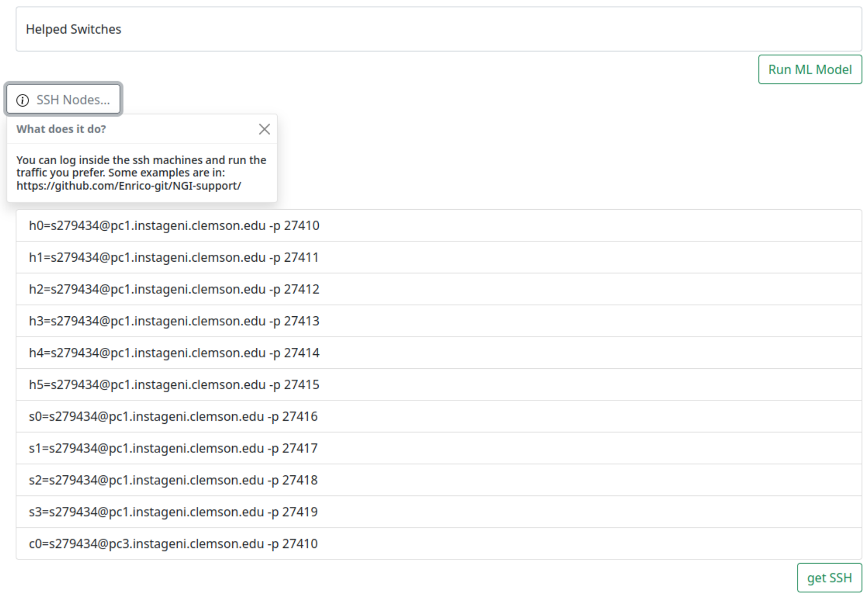
OverProvisoning Penalty

Helped Switches

Fig. 3.5 Example of frontend response

What happened in Figure 3.5 is that the user configured the Ryu controller and clicked on the ’Run Ryu Controller’ button. This click generates a necessary configuration file inside the controller node (where the Ryu controller operates) and insert the flows inside all the switches created using the standard shortest path first algorithm. After the backend handled these two aspects, it provided 'Controller OK' as a response. Please also note a further detail in Figure 3.5. After the user successfully runs the Ryu controller, the button changes the name to 'Stop Ryu controller'. It means that by clicking again on the button the Ryu controller will stop. A similar logic is provided for the model settings.

In conclusion, we also want to say a few words about the last step, number 5, regarding the SSH nodes depicted in Figure 3.6.



You can log inside the ssh machines and run the traffic you prefer. Some examples are in: https://github.com/Enrico-git/NGI-support/

```
h0=s279434@pc1.instageni.clemson.edu -p 27410
h1=s279434@pc1.instageni.clemson.edu -p 27411
h2=s279434@pc1.instageni.clemson.edu -p 27412
h3=s279434@pc1.instageni.clemson.edu -p 27413
h4=s279434@pc1.instageni.clemson.edu -p 27414
h5=s279434@pc1.instageni.clemson.edu -p 27415
s0=s279434@pc1.instageni.clemson.edu -p 27416
s1=s279434@pc1.instageni.clemson.edu -p 27417
s2=s279434@pc1.instageni.clemson.edu -p 27418
s3=s279434@pc1.instageni.clemson.edu -p 27419
c0=s279434@pc3.instageni.clemson.edu -p 27410
```

Fig. 3.6 Example of SSH command

The webpage realized until now is not enough because the experiments to run are endless. To overcome this limitation, we decided to provide the ssh command to all nodes and to provide a reference to the script already created in git. We provided all possible nodes, the controller, the switches, and the hosts.

In our experiment, we used the switches with *tcpdump* to test the correctness of the path the hosts use for communication. For the hosts, we used *iperf3* for generating traffic and have a measurement of the throughput and *netperf* for measuring the latency. The controller node helps interact with the docker that contains the RL algorithm code. In particular, it is helpful for interacting in case the Ryu controller has some error or to enable the ML model. The ML model to be activated need to run the training file first and the test file later. We leave the researcher the duty of running them properly. In our data collection, for example, we collected the results in two ways: with the model (RLVNA on) and without (RLVNA off / baseline).

3.1.2 Tecnologies and Programming implementation

Building a web page is a highly time-consuming and labor-intensive activity. It takes much attention to detail and a tremendous amount of concentration. Fortunately, to the rescue of developers come new, increasingly comprehensive, and efficient tools such as the React[3] javascript library. Just think that giants such as Facebook, PayPal, Tesla, and Netflix use it to give an idea of how powerful this library is. There are several advantages of React. First, declarativity. The programmer only has to worry about declaring what he or she wants to change, and it is the framework's job to repurpose the page as best. Another advantage is the reuse of components. It may seem abstract at first glance, but React requires you to study and define a page architecture, unlike pure programming. Once you have defined a component, which can be a box with specific characteristics, you can reuse it by changing only the content and leaving the structure unchanged. Finally, it is essential to point out that the learning curve is not overly steep because it is still a mixture of javascript and HTML, two languages that are now popular worldwide. Because of these reasons, we decided to implement the graphical user interface (GUI) in React.

It is also important to give credit to the React Bootstrap[4] library, which gives a huge help to the graphical component. This robust component provides already predefined components such as buttons, forms, and models reducing the implementation time.

The Frontend includes also another vital library named GoJS [5] used for the topology definition. As the authors report on their webpage:

"GoJS is a JavaScript library that lets you easily create interactive diagrams in modern web browsers. GoJS supports graphical templates and data-binding of graphical object properties to model data. You only need to save and restore the model, consisting of simple JavaScript objects holding whatever properties your app

[3]https://reactjs.org/
[4]https://react-bootstrap.github.io/
[5]https://gojs.net/latest/index.html

needs. Many predefined tools and commands implement the standard behaviors that most diagrams need. Customization of appearance and behavior is mostly a matter of setting properties."[27]

Before starting to describe the actual code, we would like to acknowledge Joseph Gutierrez[6] for helping us in finding the GoJS library. This has been a not easy job because it seems to be challenging to find a library that offers the possibility to draw at run time customizable objects and link each other. For example, before GoJs library, we tried many other libraries, but those did not provide all the requested features. React diagrams, React dnd tree, beautiful JS, and d3 trees do not allow to create the link between switches but just between parent and child that could be represented as Controller and Switches or Switch and Hosts. We also consider the option for being independent of other libraries and realize it with React Card[7] component, but the visualization was ugly enough. Because of these reasons, in this first version, we decided to utilize GoJS, leaving space for future development.

You can find the Frontend core implementation in appendixA. We describe here the two most important concepts used while developing it. React by standard defines the concepts of 'state' and 'props'. The idea is to define the Topology object in the root component and to pass from component to component by state and props. Even the methods of the object are passed as property and later called from the smaller component to the root, which owns the real implementation. Regarding the Topology object, we prefer to use an approach based on object-oriented programming. We defined a Class container to store the status of the topology drawn by the user. The object should contain the following:

- array of controllers
- array of switches
- array of links
- number of controller, switches, hosts
- methods for adding controller, switch, link

It is important to underline that each controller should have a reference to all switches it handles. The same is true for the switches; each switch should have a reference of

[6]joseph.gutierrez@slu.edu
[7]https://getbootstrap.com/docs/4.0/components/card/

all hosts it manages. For the link, it is enough to have a map source destination. This object is serialized, inserted in the POST request body, and sent to the backend.

As you can imagine, from Figure 3.3 two are the bigger components we defined:

- Topology container
- Settings container

The Topology container is responsible for the GoJs box that the user will use for drawing the custom topology. The Settings container instead incorporates all the machine learning parameters the end user wants to experiment with.

3.2 Backend

The backend is where all the magic happens. As we said previously, we wanted to realize a platform mainly for researchers with no networking background but a strong background in Machine Learning. We wanted to test solutions they designed in an emulated yet realistic environment. The backend has to interact with GENI to choose the aggregate, select the slice, define the kind of node to reserve, and their networking configuration, such as the IP addresses. GENI checks the availability, and if the requested resources are available, the backend configures each node that, as we said before, could be a controller, switch, or host. Be aware that many improvements could be realized in the backend. We are conscious that we realized a first basic version, which for example, does not allow the platform to be ready with multiuser or to customize the possible aggregate. Both limitations come from GENI. It has a strict policy for access, and regarding the aggregate, we tried to customize it at the very beginning but without success. In particular, our initial idea was to choose the aggregate randomly but with enough nodes available for the request. The problem is that even if an aggregate has enough resources, it does not mean that the machines are good enough. We did not understand why GENI fails to reserve even a few machines, and later we discovered that the aggregate you choose could make a big difference. We empirically defined Clemson and Illinois aggregate as better than the others. Once we discovered the best aggregate, we tried to scale up without success. We did experiments with ten and twenty nodes, but it was not possible to try experimenting with hundreds of nodes. We postponed the scaling problem and focused on having the first correct and working version, at least for a few nodes.

Furthermore, to complete the analysis, we started at the beginning of this section when the backend receives the user's request which could consist in creating a custom topology, configuring the Ryu controller, configuring the ML Model, or getting the ssh command for logging inside the node, the backend executes the operation of the node reservation explained before. Finally, after the interaction with GENI, it returns the proper status as a string that explains to the end user if the operation was a success or a failure.

We will explain in the following subsections how we designed and structured the backend and a few details on the implementations intending to describe the choices we make and the possible improvement to realize.

3.2.1 Backend Design

Before starting with the description, it is important to fix properly in mind which are the limited features the backend offers. In particular, you can imagine them by having a look at the endpoints the backend offers:

- *http://127.0.0.1:8000/api/topology* for resource reservation
- *http://127.0.0.1:8000/api/ryu* for Ryu configuration
- *http://127.0.0.1:8000/api/model* for Model configuration
- *http://127.0.0.1:8000/api/traffic* for Hosts configuration
- *http://127.0.0.1:8000/api/download* for measurement statistics.

Before starting with the description of each endpoint, we would like to spend a few more words about the GENI account and preliminaries because all the APIs somehow interact with GENI. For operating in GENI, it is essential to have an account. We registered our account in the 'National Center of Supercomputing Application' of the University of Illinois at Urbana-Champaign (USA). Once the account is approved by the center, and the dual-factor authentication is defined, it is possible to log in and become a project member. Moreover, from there, it is possible to create the slices with the desired resources or get access to the certificates needed for any request. In our case, we downloaded the SSL GENI certificate, and we need to generate an ssh key pair and upload the public key to the GENI system. This operation is fundamental because when GENI creates the resources, it is possible to access them only if they contain the public key and the proper private key is matched. Furthermore, from the GENI web portal, we created the slice that the backend will use but left the honor to reserve the resources for the backend calls.

We can finally begin our description of the features offered by the backend. We will start with the most complex which is the */api/topology*. We defined the name of the slice and the aggregate statically. In general, we choose the Illinois aggregate because we saw it is the most performant aggregate. To help the reader, we first define the topology steps, and then we will discuss each point separately. The topology should execute the following actions:

1. Delete the previous allocation in the slice
2. Parse the frontend request
3. Create the Rspec request
4. Store the manifest response for future use
5. Configure the machines
6. Store ssh command for future use

The first thing to do for creating the topology is to deserialize and parse the body of the POST request sent from the frontend. This operation gives an idea of the number of nodes to reserve. Then, since a slice could contain only one uneditable experiment, we realized a function that checks if there is a previous experiment, and eventually, it deletes all previously allocated resources. Once we have the proper space in the slice, we start creating the RSpec. The RSpec is a particular request accepted as input by GENI, which contains a description of what the end user would obtain. While Parsing the frontend request for each node, we inserted the information in the RSpec. For the controllers and the hosts, we choose ubuntu 20 machines; for the switches, we choose a simple ubuntu 18 with OVS already preinstalled. Moreover, for the links between the hosts and the switches, we defined an IP address as 192.168.x.y to simplify the successive traffic generation. Once the

RSpec request is complete, we ask GENI to reserve these resources, and in case of success, it returns a manifest. In such cases, we store this manifest on the disk because it contains the request we ask for and the implementation details that GENI assumed, such as the MAC address for each interface. The manifest also includes the node and the ssh credentials for getting access. When the resources are reserved, it takes a while once they are ready and available for access. We inserted a delay function to understand when all the nodes are ready. A future improvement could be not to wait for all resources before going further, but once a node is ready, execute the subsequent actions on it. When a node is ready, it must be configurated for our scope. For example, the controller at startup is an empty machine, and we download a bash script from a public repository to start the configuration. We install the docker engine, download the RLVNA docker image, and finally create a container with the downloaded image. Unfortunately, the docker image is around 4GB, which requires some time. A further improvement could be to reduce the docker size. The RLVNA size is not too large, but we needed a conda virtual environment to resolve some ML library dependency, which takes 2GB. For the switches, the logic is almost the same. The difference is that in this case, we do not need docker; instead, we need to create the OVS switches, create the respective interfaces and map the machine to forward the packets from the natural interface to the OVS interface and vice versa. Furthermore, the OVS switches need to receive and set the controller IP to communicate with, as we explained in the background section about the SDN section. We choose to use the controller IP defined by default by GENI, which is in the form of 172.17.x.y. We also tried to define a custom IP address in the form of 192.168.x.y, but then we realized this was a useless complication because it required creating new links between each switch and the controller and probably the proper ARP definition. The last step of the 'create topology' is to parse the manifest generated by GENI as output and to write on disk the ssh command of all nodes, which will be used in the future.

Another more simple feature is the one offered by */api/ryu*. This API is designed for configuring the Ryu controller. The Ryu controller is inside the docker, and the docker is inside the controller node. The first thing to achieve is to get the ssh command for access to the controller node. Once inside the controller node, it is possible to access the docker, even in detached mode, and to execute the proper command for the requested configuration. In particular, inside the docker, we inserted a simple and general bash script to generate a config.ini file needed for the Ryu controller to start. The script receives as input the end user's settings defined in the frontend for the Ryu controller section and generate the file correctly. Then the backend executes the proper command for running the Ryu controller and gets ready to receive statistics from the OVS switches.

A similar feature is defined through */api/model*. This API is designed for configuring the ML model. As Ryu, the ML model is inside the docker, and the docker is inside the controller node. Even in this case, the first thing to achieve is to get the ssh command to access the controller node. Once inside the controller node, it is possible to access the docker, even in detached mode, and to execute the proper command for the requested configuration. In particular, inside the docker, we inserted a simple and general bash script to generate a config2.ini file needed for the ML model to start. The script receives as input the end user's settings defined in the frontend for the ML model section and generate the file correctly. After the configuration file, differently from the Ryu controller, the ML model is not run because we left space for the researchers to properly execute it with the traffic they want and any other settings.

Continuing the description of the features, a handy feature but still a work in progress is provided by */api/traffic*. The original idea was to configure all the hosts with the proper dependency, such as iperf3 and netperf, and later download and install a python3 script from a public repository to start the traffic generation. We changed this feature frequently because the traffic needs to be constantly observed and monitored. This feature is no more reachable from the frontend, and the only way to use it is by utilizing an ad hoc curl request.

In conclusion, the last feature is provided by the */api/download* API. There are two possibilities for utilizing this feature: by curl request or frontend request. The frontend can access the SSH nodes by reading the file generated in 'create topology', instead through curl, it is possible even to download the traffic generated by the host script.

After this accurate description of the backend design, we remand to the backend implementation for further details about the tools and libraries used and to the last section of the chapter for RLVNA algorithm implementation.

3.2.2 Backend Implementation

We developed the web platform in Python programming language using Django[8] tool. As the developers declare: *"With Django, you can take web applications from concept to launch in a matter of hours. Django takes care of much of the hassle of web development, so you can focus on writing your app without needing to reinvent the wheel. It is free and open source"* [28].

Between the advantages of Django, we remember that:

- It is fast and with a small learning curve.
- It is flexible and scalable.
- It has many extras such as user authentication, content administration, site maps, RSS feeds.
- It solves security problems, such as SQL injection, cross-site scripting, cross-site request forgery and clickjacking.

Another essential library we used for interacting with GENI testbed is geni-lib[9]. As the authors report: *"geni-lib is a Python library for interacting with the NSF GENI Federation, or any federation that uses components of the GENI Software Architecture. Common uses include orchestrating repeatable experiments and writing small tools for inspecting the resources available in a given federation. There are also a number of administrative API handlers available for interacting with software commonly used in experiments - particularly those exposing services to other experimenters"*[29].

Between the most important API we called, we remember:

- IGCompute.createsliver(context, sname, rspec)
- IGCompute.deletesliver(context, sname)
- IGCompute.sliverstatus(context, sname)
- geni.rspec.pg.Execute(shell, command)

[8] https://www.djangoproject.com/
[9] https://geni-lib.readthedocs.io/en/latest/

- geni.rspec.pg.IPv4Address(address, netmask)
- geni.rspec.pg.Install(url, path)
- geni.rspec.pg.StitchedLink(name)
- geni.rspec.pg.XenVM(name, component_id, exclusive)

It is also important to underline that geni-lib is designed for python2, and we need to change some source code to adapt it to python3 in our project.

In conclusion, we want to express the last note, which refers to future works. As you can see, the code has been developed as generally as possible. This means that in the future, topologies with more than one controller could be generated with minor improvements. Moreover, 'geni-lib' calls could be replaced with other testbed library calls.

3.3 Reinforcement Learning-based Virtual Network Adaptations algorithm

In the Background chapter, we spent many words describing the Reinforcement Learning concepts and Mystique algorithm from a technical point of view. In this section, we will focus on the implementing part and the related work done for the development of a generalized Reinforcement Learning Virtual Network Adaptation (RLVNA) algorithm.

Before starting with the RLVNA algorithm's explanation, we must understand which part of the architecture we are referring to (see Figure 3.1). We are in between the backend and the GENI system. In particular, as we said before, the RLVNA algorithm is dockerized from the backend and inserted inside a compact and isolated container in the controller GENI node.

Now we want to spend a few more words about Docker[10] and the content of the container we realized. It is a potent tool for emulating an environment. In particular, as the authors report:

[10]https://www.docker.com/

Docker is an open platform for developing, shipping, and running applications. Docker enables you to separate your applications from your infrastructure so you can deliver software quickly. With Docker, you can manage your infrastructure in the same ways you manage your applications. By taking advantage of Docker's methodologies for shipping, testing, and deploying code quickly, you can significantly reduce the delay between writing code and running it in production [30].

Furthermore, we initially used instaGENI custom image[11] instead of Docker for creating the custom image for the controller, but later we chose Docker for achieving platform independency and portability.

However, Docker realizes just empty boxes; what is essential is what we insert inside these boxes. Inside the container[12], we included a lot of other tools useful for minors functionalities such as Omni, Conda, and bash scripts.

Omni[13] is a powerful library used to interact with GENI by command line. We used this tool only for retrieving the GENI topology from the Ryu controller. Because of this reason, Omni dependency is easily removable. However, Ryu offers an API for topology discovery[14], but we were not able to let it work with GENI testbed nodes. By the way, by changing the SDN Controller, it should be possible to avoid the use of Omni.

Concerning Conda virtual environment[15], the authors report: *Conda is an open-source package management system and environment management system that runs on Windows, macOS, Linux and z/OS. Conda quickly installs, runs and updates packages and their dependencies. Conda easily creates, saves, loads and switches between environments on your local computer. It was created for Python programs, but it can package and distribute software for any language. Conda as a package manager helps you find and install packages. If you need a package that requires a different version of Python, you do not need to switch to a different environment manager, because conda is also an environment manager. With just a few commands, you can set up a totally separate environment to run that different version of Python, while continuing to run your usual version of Python in your normal environment* [31]. We installed Conda because the GENI nodes are not so powerful, and they do

[11]https://groups.geni.net/geni/wiki/HowTo/ManageCustomImagesInstaGENI
[12]https://hub.docker.com/repository/docker/enrico2docker/ubuntu-mystique
[13]https://groups.geni.net/geni/wiki/HowToUseOmni
[14]https://ryu.readthedocs.io/en/latest/gui.html
[15]https://docs.conda.io/en/latest/

not have a standard architecture, so it helps in installing all ML model requirements (Keras, Tensorflow), but this led to a docker image size increment.

We quickly saw the bash scripts when we talked about the controller reservation and container configuration. We wanted to realize the RLVNA algorithm as general and as customizable as possible from the frontend. It means that the algorithm's behaviour adapts depending on some configuration files. Our idea was to insert inside the docker image some predefined scripts that depending on the input value they receive, generate the proper configuration file used both for the Ryu controller and the ML model.

To summarize, we report here the steps we follow. Initially, we realized a Dockerfile that uses ubuntu 20 as the first layer, and then we installed python3.8, all Ryu controller's requirements, and all requirements for Omni. Be careful that in this Dockerfile we did not install anything for the ML model part because we have enough problems with node compatibility. We decided to create the image from an existing container with the proper conda virtual environment already in.

Finally, after this tools overview, we can start the description of the generalized Reinforcement Learning-based Virtual Network Adaptations (RLVNA) algorithm and mainly the Ryu controller generalization, where we spend most of our time. By the way, we do not want to give a full description of the whole Ryu controller because it would be too long. We will limit ourselves to remind you that we are working on an SDN architecture. All switches will send their statistics (which consist of the packets in and out of each interface) to the controller with a particular frequency , and the controller will implement the logic. Before this can happen, each switch needs a configuration. We realized a dynamic algorithm that can configure different topologies and now want to discuss it more deeply.

The first thing the Ryu controller needs is the GENI topology. As we said a few lines before, Ryu offers some methods for getting the topology, but we could not let it work with GENI, and we used the 'omni list resources' call to access the slice information. It is like obtaining the manifest we talk during the backend description. We parsed this information and generated the proper data structure containing the controllers, the switches, the hosts, and the links. Until now, just the GENI topology is parsed and available inside the Ryu controller. Moreover, once we have this base, we can build upon it what we want. The next step is obtaining the shortest path between one host and the others. Unfortunately, GENI does not allow updating the topology in real time for scaling up and down. Hence, our idea is to have a more extensive network with some free switches disabled at the beginning (support). Whenever the load becomes intensive, they are enabled to reduce the allover workload. Because of this, our idea was to create two data structures, one that contains the shortest path considering the support switch and another data structure that contains the shortest path without considering these switches. Be careful in the future because this strategy could create problems if half of the support switches are enabled, and half are disabled.

This was the basic configuration done at startup in the Ryu controller. The configuration to execute in each switch when they register to the controller was much more complex. In particular, the mapping between the name of the switches and their interface port number. For this mapping, the Ryu controller gives us a big help. From the omni topology, we stored for each interface the MAC address. Instead, the Ryu controller in the 'datapath' provides the MAC addresses and port number for each interface. We realized then a 'simple' check, and this led us to the possibility of inserting in each switch simple flows. In particular, thanks to the shortest path and this mapping, we insert flows in the form of: "If you match this *IP destination*, forward to this *port number*". We want to underline a small optimization we realized. Before inserting the flows for each host, we inserted a function that clears all previous flows in the target switch such that if we want to reset the topology paths, it is enough to stop and re-run the Ryu controller. A further improvement we led to the following developers is the use of sophisticated algorithms such as Equal Cost Multipath Protocol (ECMP). For the moment, in case multiple shortest paths with equal cost are detected, only one is utilized. ECMP could improve the total performance of the network by utilizing all paths.

Finally, we are moving our focus to analyze the communication between the Ryu controller and the ML model. it means to focus on what happens when a supportive switch turns on or off. To recap, the end user decides the frequency at which the Ryu controller sends all collected switches' statistics to the ML model. In the beginning, the ML model needs the first phase of exploration, named training. In training, it takes random actions and understands which action is better than the other. When the training ends, it starts the accurate algorithm that now knows how to operate. It evaluates a reward function and, depending on the value, decides the action to send to the controller. We defined two possible actions: support switch ON or support switch OFF. Of course, this process is repeated for all the supportive switches. It means that the action that the ML sends to the controller is an array with the combination of ON/OFF of all the support switches existing in the topology. When the Ryu controller receives the action, it executes the following operations.

If the action is to enable a ’support switch’, the first step is to find the neighbors of the target switch. The neighbor switches are the ones that need to change their flows. Moreover, we searched the hosts with the shortest path through the target switch. Half of these hosts were selected to use the new path, and half kept the old path without the support switch. Overall, this improves the network’s performance as we will discuss in the result chapter. Viceversa, if the action is to disable a ’support switch’, this improves the performance too because it means that the target switches were useless and their energy could be saved -La luce costa cara. In that case, the first step is always to find the neighbors of the target switch and their hosts. Half of the hosts that were previously enabled will be forced to use a path without the target switch, the other half instead has no modification.

Chapter 4

Evaluation

The work done until now is something costly in which we put much effort. However, it is crucial to demonstrate the goodness of the project with statistical data and results. Because of these reasons, we tested the platform and collected different results. We did not measure the goodness of the frontend with performance evaluation, but it is intuitive enough to understand how easy it was to create and customize experiments.

Before going further, it is essential to highlight that the GENI testbed is powerful enough because there is an excellent variety of aggregate spread all over the US, and it allows to reserve resources with high flexibility and customization, but it has so many limitations. So tricky has been the collection of the statistics and the resource reservation before that. Even in the best aggregate, such as Illinois or Clemson, the node reservation often failed. Consider also that sometimes even for simple topologies with four switches, passes days before a working topology is configured correctly. This caused a delay in data collection and a limitation in the number of configurations we set up. In particular, we mainly created two topologies. One with about ten nodes and one with about twenty nodes. Once we could reserve these resources, we mainly changed the parameters in the algorithm or the kind of traffic to send in the network.

This chapter will discuss the kind of traffic we realized and the problems we encountered. Moreover, we will evaluate the topologies settings we tested and the result we obtained. Moreover, after explaining the scripts we used for traffic generation, we will discuss some statistics we collected, such as the round trip time (RTT), the throughput, and the cumulative distribution function (CDF).

4.1 Setting

We want to post our attention on two particular ones of all the possible experiments it is possible to generate. The code we used for traffic generation is almost identical in the following two topologies. We will explain once now, and we will discuss later the tiny changes we adopted between the two.

Before starting the traffic generation, we want to say a few words about the tools we used:

- iperf3
- netperf

We report here a brief description of iperf3 we found on the web. *"Iperf is a tool for network performance measurement and tuning. It is a cross-platform tool that can produce standardized performance measurements for any network. Iperf has client and server functionality, and can create data streams to measure the throughput between the two ends in one or both directions. Typical iperf output contains a time-stamped report of the amount of data transferred and the throughput measured. The data streams can be either Transmission Control Protocol (TCP) or User Datagram Protocol (UDP)"* [32]. Besides that, we used iperf3 only for testing TCP data streams and left space for future experiments. Be aware that the docker, controller, and OVS switches are configured mainly for TCP traffic. It means that if in the future we want to test UDP traffic, they need to be adapted and tested. Overall, we discover the limitation of the iperf3 tool, especially in the RTT measurement, thanks to a fascinating reading about traffic generator tools in [33, 34]. In particular, after many hours of testing, we realized that the best RTT measurements were obtained by the netperf tool.

About netperf we report from the same authors: *"Netperf is a software application that provides network bandwidth testing between two hosts on a network. It supports Unix domain sockets, TCP, SCTP, DLPI and UDP via BSD Sockets. Netperf provides a number of predefined tests e.g. to measure bulk (unidirectional) data transfer or request response performance"*[35]. Adapting netperf in our code was simple because it uses the same logic as iperf. Moreover, we chose this tool because it already provides a test for the RTT, which was perfect for our scope.

Let us start now by describing the scripts we realized for the traffic generator. As we said earlier, the differences between the first and the second topologies are slight. We will explain the global logic now, and if needed, we will add further details later. Behind all the attempts, we can affirm that two are the basic versions: The dynamic one and the static one. Both versions receive as input two parameters. The number of iperf3 iterations to execute and the list of all IP hosts available in the topology. A minor optimization could be to avoid passing the list of IP addresses and to pass just the number of hosts. We saw that the request could be done using the IP destination and the hostname. The first thing operation is to define the number of clients and the number of servers. In the dynamic one, we defined the number of servers as the total number of hosts divided by three, and then we computed the clients by difference. In the static version, we manually defined the number of clients and the number of servers. The servers run the proper iperf3 or netperf command with the server setting instead, the clients run the same command but with the client settings. Our initial idea was to open in the server machine one port for each client, and then the client randomly performs a request. We realize this was too much for GENI node because when we passed from 6 hosts to 16 hosts, the node could not support one thread per port. We then defined a limitation of 3 ports per server, which was enough for our experiments. When we started writing the code for traffic generation, our idea was to create a congested network and to measure the metrics only in a single flow. In particular, the first host in the topology ’h0’ is in charge of collecting and storing the throughput if it uses an iperf3 request or the RTT in case it uses a netperf request. For simulating the congestion, we verified first that the link capacity is 100Mbps, and then all the clients perform the defined number of iterations for different bandwidths. In particular, they start with 10 Mbps and arrive to send in total 100Mbps each. For Example, for our experiments, we chose as iteration number 10. It means that we executed ten times the request for each bandwidth (10, 20, ..., 100). Concerning the iperf time, we tried many values, but we found that the best duration is around 2 minutes.

4.1.1 Four switches experiment

This is one of the first experiments we executed. We choose to reserve the following resources in Clemson aggregate:

- **1 Controller**
- **4 OVS switches**
- **6 Host**

As we explained, the controller is in charge of executing the ML model and the Ryu controller. The switches represent the network and the mean of communication. The hosts are divided into clients and servers and generate traffic for testing the algorithm.

From the architectural point of view, we can see the implementation in Figure 4.1.

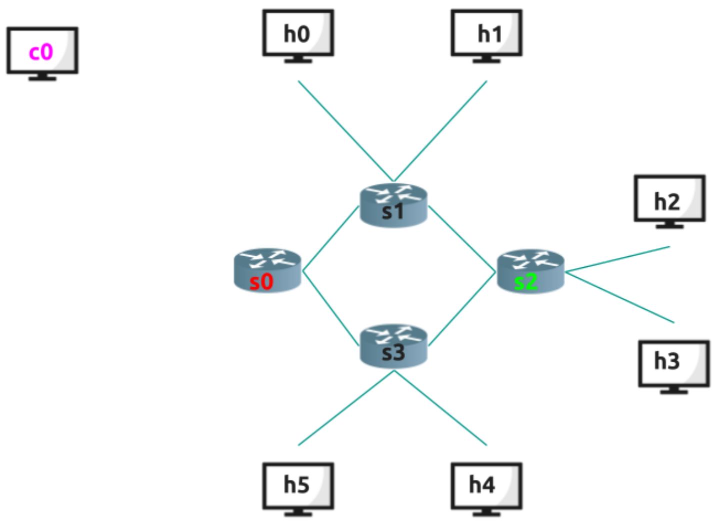


Fig. 4.1 Experiment with 4 switches

In this topology, we used ’s0’ as the support switch and ’s2’ as the helped switches. It means that the Ryu controller collects the packets in all interfaces of switch s2. The reinforcement learning model receives the statistics and, depending on their value and the over-provisioning penalty, decides whether to activate the path with s0 or not. If the action is to activate ’s0’ and their links, they are activated only for half of the hosts. For clarity, we report here the example we used for our test. However, the controller receives the statistics from all the switches, even if the links are not drawn in the picture.

We selected h0-h1-h2-h3 as clients and h4-h5 as servers in our static traffic generator. In the beginning, when 's0' is off, all the clients use the path (s1-s2-s3) for sending traffic. After a threshold of around 30Mbps, the RLVNA algorithm activates the support switch 's0', and half of the target host with the shortest path with the new path is selected. In our case, the hosts h2 and h3 continue to use their old path s2-s3 because it is the shortest. The hosts h0 and h1 instead are the targets, and only half is selected. It means that only h0, in this case, will use the path s1-s0-s3; instead, the hosts h1 will continue to use the path s1-s2-s3.

As you can imagine, even now, this led to a significant improvement in performance, but we prefer to discuss it later in the next section.

4.1.2 Eight experiment switches

In this second experiment, we reserved in the Illinois aggregate the following nodes:

- **1 Controller**
- **8 OVS switches**
- **10 Host**

The idea is the same as before, what changes is just the amount of machines. From a theoretical point of view it remains the same, but in this case, we saw the power of the generalization we realized because without modification, we were able to test this completely different topology.

From the architectural point of view, we can see the implementation in Figure 4.2.

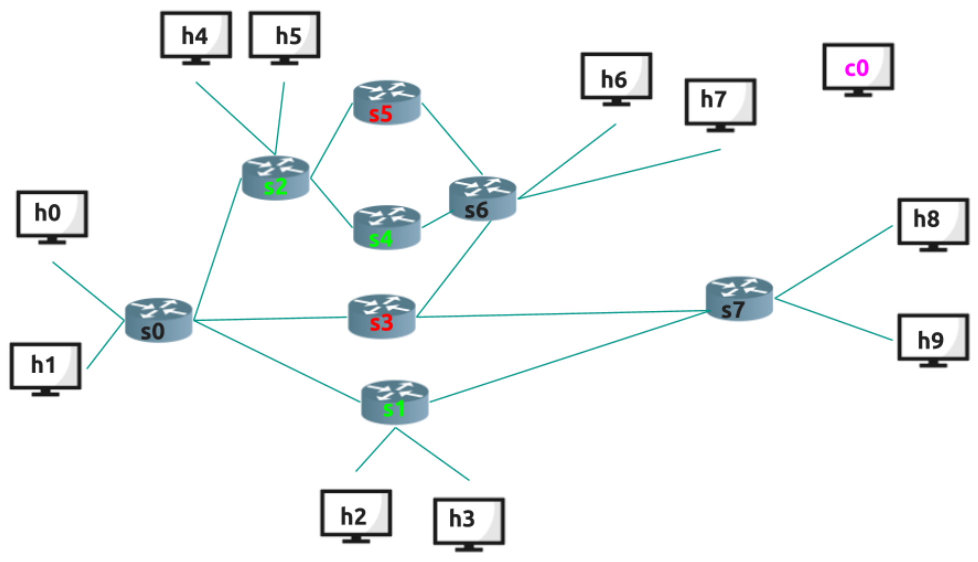


Fig. 4.2 Experiment with 8 switches

In this topology, we use h0-h1-h2-h3-h4-h5 as clients and h6-h7-h8-h9 as servers. In particular, the clients from h1 to h5 generate traffic to the servers from h7 to h9 for simulating a realistic environment. The client h0 generates and collects the traffic through the server h6. Even in this case, the amount of traffic sent is incremental. It starts at 10 Mbps per client and reaches 100Mbps per client, which is the link limit in GENI , and the host h0 per each bandwidth collects the RTT with netperf requests and the throughput with iperf3 requests. It is obvious, but as we said before, the controller receives the statistics from all the switches, even if no links are drawn in the picture.

In our example, we used ’s3’ and ’s5’ as the support switches. The helped switches of ’s3’ are ’s1’ and ’s2’. The helped switch of ’s5’ is ’s4’. It means that the Ryu controller collects the packets in all interfaces of the helped switches. Depending on the reinforcement learning model, it decides if to activate the path with the support switches or not. Even in this case, eventually, the path is activated only for half hosts.

For completeness, we report here the example we used for our test. In this case, the ML model could take the following actions:

- s3 OFF, s5 OFF
- s3 OFF, s5 ON

- s3 ON, s5 OFF
- s3 ON, s5 ON

For brevity, we will discuss just the first and the last case. The intermediate cases are particular and require an unbalanced kind of traffic that was out of our test. As we said before, the interesting traffic is in 'h0' the others are just for traffic congestion.

The default case is when 's3' and 's5' are OFF. In this case, for 'h0' the shortest and only possible path is following s0-s2-s4-s6-h6. By the way even the hosts 'h1', 'h4, and 'h5' follows the path s2-s4-s6 with 'h7' as destination. As you can imagine, the ML model will soon detect the need to activate the support switch. Let us move to the case with 's3' and 's5' ON. In this case, the host 'h0' will use the path s0-s3-s6-h6, which is completely free. Instead, as an example, 'h1' will continue to use the previous path with s2. We will discuss the performance in the next section, but as you can imagine, this greatly improves them.

4.2 Result

In this section, we will report the limited statistics we collected for the experiments with four switches topology (Figure 4.1) and with 8 switches topology (Figure 4.2). The statistics we are referring to are:

- Round Trip Time (RTT)
- Throughput
- Cumulative Distribution Function

All the experiments are reproducible because we inserted a static seed for the randomly generated values. It is not essential to provide here the requirement of the personal computer we used for the backend because the backend is used only for the topology definition and customization. After that, we are using the GENI machines. All geni machines use the default value except for the controller node. Since we installed the docker engine in the controller, we followed the docker's developer suggestion to have as a minimum requisite 2 CPU and 4GB of RAM.

It is also vital to underline that for each experiment, we realized two comparisons, as you will see in the legend of each plot:

- With the Reinforcement Learning Virtual Network Adaptation (RLVNA)
- Without any improvement, Baseline

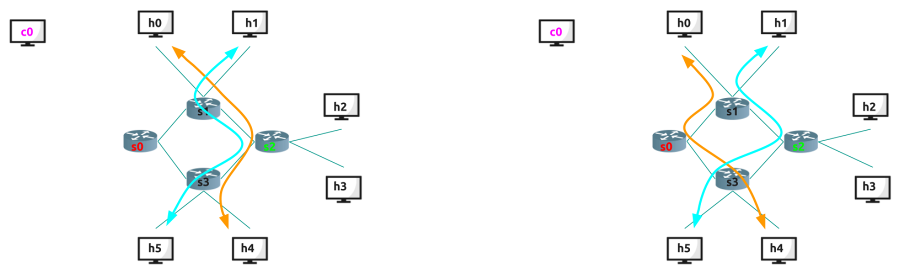
We realized the first basic version without using any RL algorithm. In this case, the hosts used the shortest path configured by the Ryu controller. The second version requires the RL model, which generally activates the path with the supportive switches.

As you will see in the plots, we realized that three are the most relevant bandwidths. The first is around 30Mbps when the links start to saturate and the ML model activates the new paths. The second is around 50Mbps, which highlights the benefits when using the RL algorithm. The third and last is around 80Mbps when in general, there is the maximum difference in value between the two comparisons. These are the reason why we depicted only these three values in the CDF.

Overall, for the RTT and the throughput we draw the confidence interval, which helps to define if one curve is better then another statistically.

4.2.1 Four experiment switches

Before starting the analysis of the results, we want to report here the paths that the hosts 'h0' follows with and without the algorithm.



(a) (b)

Fig. 4.3 Experiment Topologies. RLVNA deactivated in (a), RLVNA activated in (b)

Figure 4.3 depicts the paths used in the communication with h0-h4 and h1-h5 without the RLVNA algorithm. It is important to underline that we did not draw the other hosts' paths, but as you can remember from the setting description, they share almost the same path as h1. On the left of Figure 4.3 (a) we depicted the path while the RL algorithm is activated. On the left of Figure 4.3 (b) we depicted the path when the RL algorithm is deactivated.

We can finally start by describing the results obtained in these two cases. In Figure 4.4, it is possible to analyze the RTT for the four switches' topology.

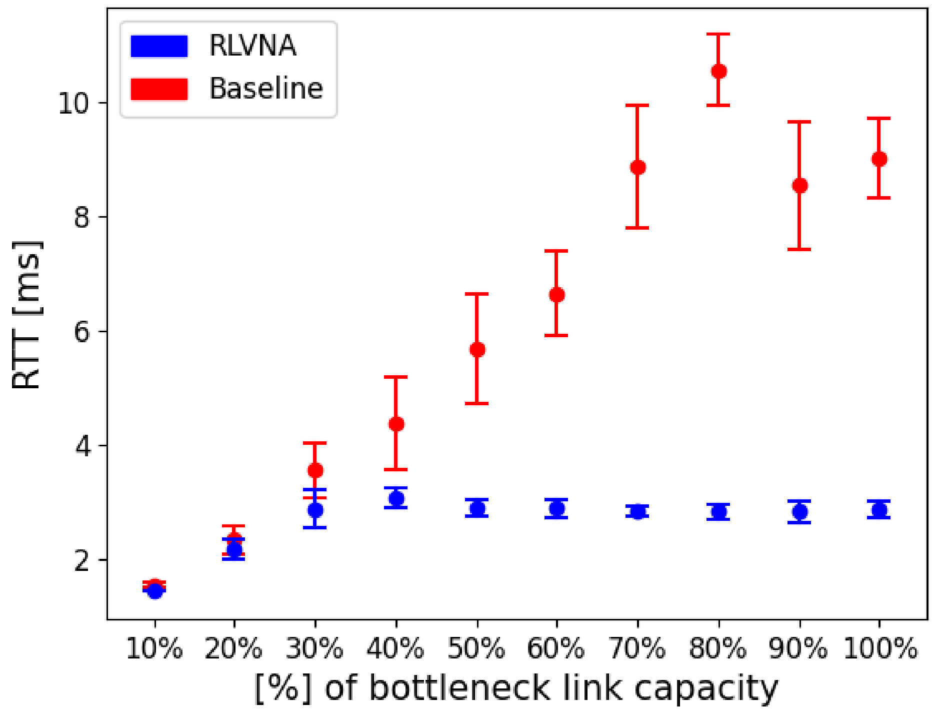


Fig. 4.4 RTT with 4 switches

As we can see from Figure 4.4 on the Y-axis, we have the Round Trip Time (RTT) measured in milliseconds. The RTT is when a packet goes forth and back from the source to the destination. On the X-axis, we have the bottleneck link capacity. It is important to remember that the GENI link saturation is at 100Mbps. The bottleneck link capacity is how much all clients send in the network. For example, 50 percent means that all clients are sending 50Mbps.

The case with RLVNA activated is almost constant because from 10 percent to 100 percent, the values are more or less 3ms. This is entirely different in the baseline case because the values always rise from 2ms to 12ms. This means that when the load is high and RLVNA algorithm is off, the packets require more time to reach the destination and return.

When the bottleneck link capacity is less than 30 percent, the values of the RTT with and without RLVNA algorithm are almost the same. The values tend to change when the load increases more. For example, around 50 percent, we have that with the RLVNA algorithm the RTT is 3ms, instead without is double around 6ms. Please also consider how the values are distributed across the confidence interval for this particular bandwidth. Without RLVNA algorithm, the values are between 5ms and 7ms. With RLVNA algorithm instead, the values are almost always the same at 3ms. The most significant difference is around 80 percent because in baseline, we can see values from 10ms to 12 ms; with RLVNA algorithm, it is around 2.5 ms.

In Figure 4.5, it is possible to analyze the throughput for the four switches' topology.

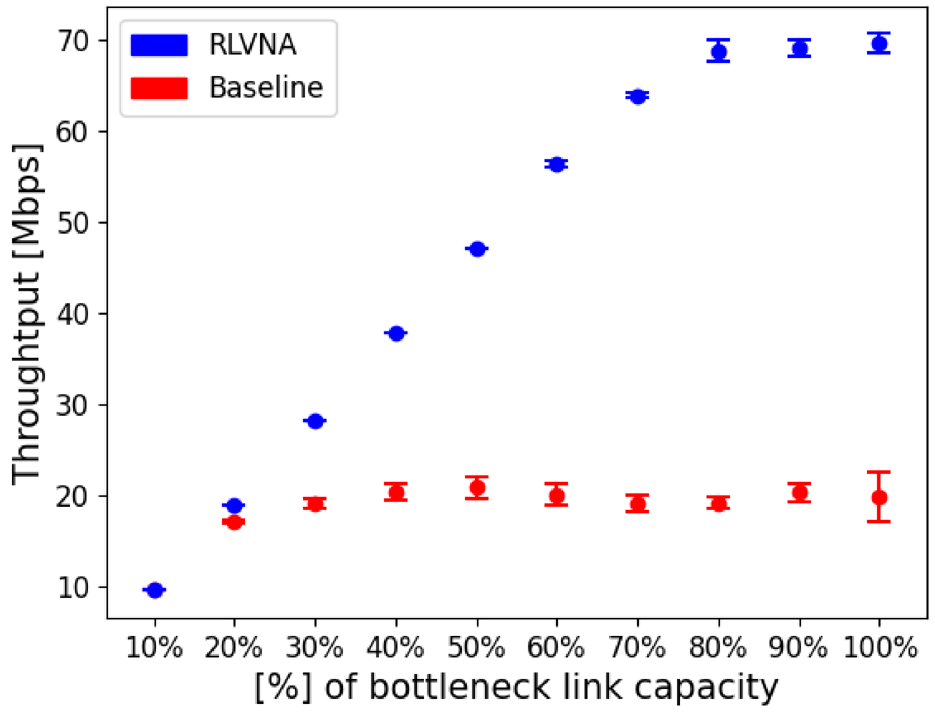


Fig. 4.5 Throughput with 4 switches

As we can see from Figure 4.5 on the Y-axis, we have the throughput measured in megabits per second. The throughput is the number of bits sent in a time interval. On the X-axis, we always have the bottleneck link capacity. We remember again that the GENI link saturation is at 100Mbps. The bottleneck link capacity is how much all clients send in the network. For example, 50 percent means that all clients are sending 50Mbps.

The case with RLVNA algorithm activated is continuously rising because from 10 percent to 80 percent, the values increase from 10Mbps to 80Mbps. This is completely different in the baseline case because the values are constant, around 20Mbps. This means that when the load is high, and RLVNA algorithm is off, the packets sent are much fewer because the links are saturated instead when RLVNA algorithm is activated, as you can remember from Figure 4.3, a new path is entirely available for h0, triplicating the performance.

In particular, when the bottleneck link capacity is less than 20 percent, the throughput values with and without RLVNA algorithm are almost the same at

20Mbps. The values tend to change when the load increases more. For example, around 50 percent we have that with RLVNA algorithm the throughput is 45Mbps, instead without is less than the half around 15Mbps. The most significant difference is around 80 percent because in the baseline case, we can see values around 20Mbps; with RLVNA algorithm instead, it is almost four times more, around 70Mbps.

Please also consider the overall value distribution across the confidence interval. In this case, the values are almost the same, probably because we are using iperf3 and not netperf as in the RTT.

We now start the description of the Cumulative Distribution Function. As we said at the beginning, we will consider the values collected in the ten iterations of the following bandwidths in the following figures:

- Throughput at 30Mbps, in Figure 4.6
- Throughput at 50Mbps, in Figure 4.7
- Throughput at 80Mbps, in Figure 4.8

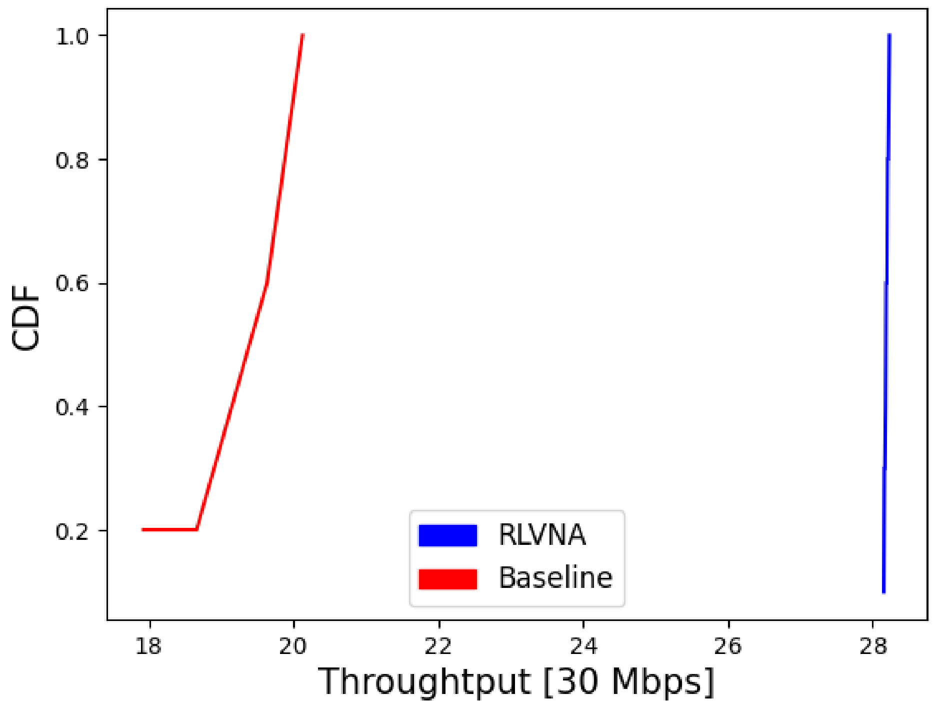


Fig. 4.6 Cumulative Distribution Functions with 4 switches 30M

In Figure 4.6, we can see how values are distributed when all the hosts send 30Mbps. Please notice the big difference in distribution with and without RLVNA algorithm. In the baseline, values are from 18Mbps to 21Mbps. With RLVNA algorithm instead, they are perfectly constant at 28Mbps. In both cases, we can see the absence of outliners because the values are uniformly distributed after the mean value of 0.5.

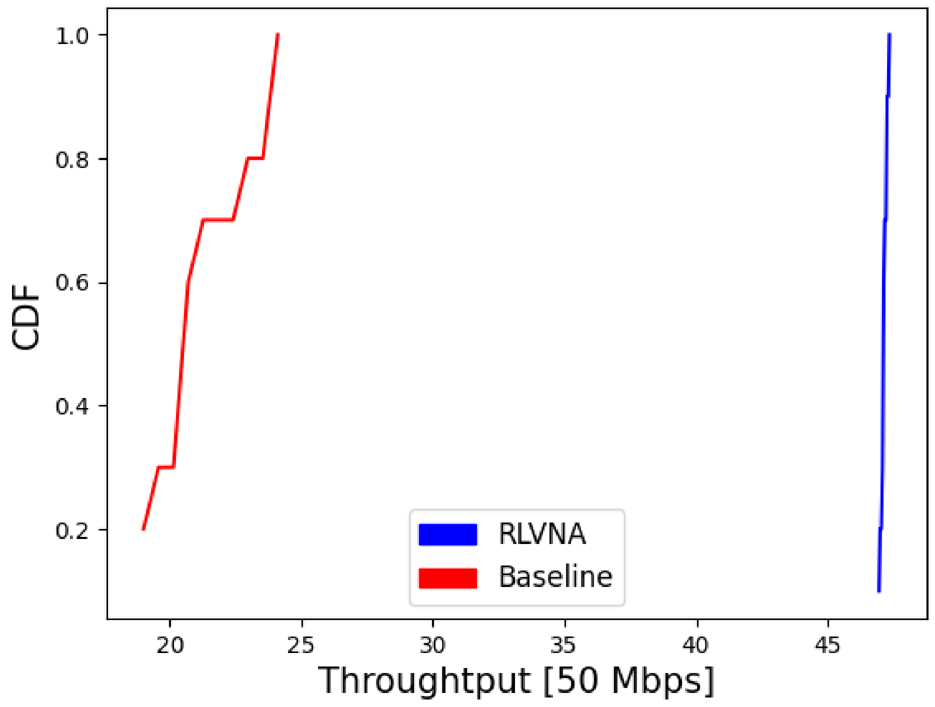


Fig. 4.7 Cumulative Distribution Functions with 4 switches 50M

In Figure 4.7, we can see how values are distributed when all the hosts send 50Mbps. Please notice the big difference in distribution with and without RLVNA algorithm. With RLVNA algorithm, it is almost perfect, with all values close to 45Mbps. In the baseline, values are spread from 20Mbps to 25Mbps. In this case, we can see the absence of outliners when RLVNA algorithm is activated, but outliners are present in the default case. We can affirm that outliners generally have a value between 0.9 and 1.0 and are in the range from 23.5Mbps to 25Mbps.

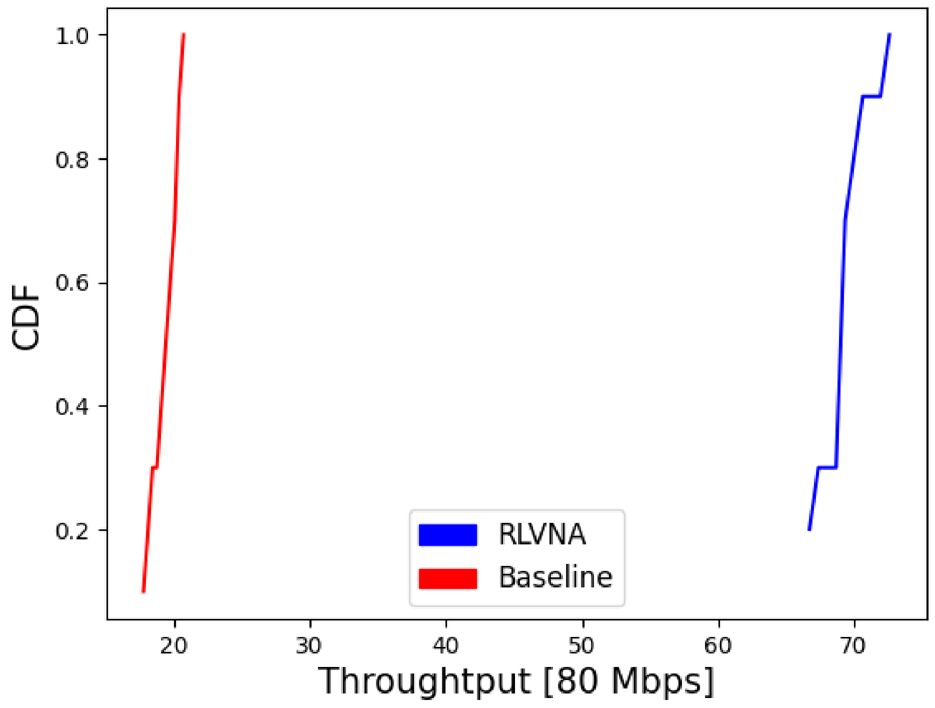
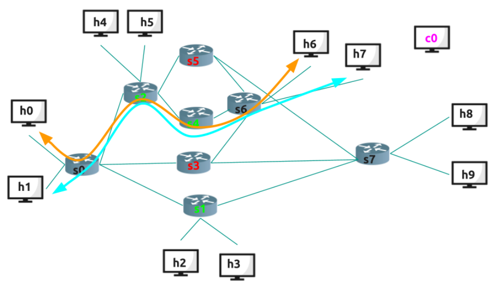


Fig. 4.8 Cumulative Distribution Functions with 4 switches 80M

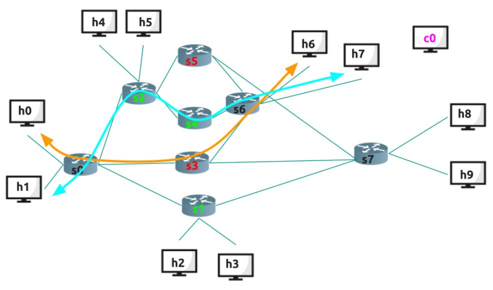
In Figure 4.8, we can see how values are distributed when all the hosts send 80Mbps. Please notice the big difference in distribution with and without RLVNA algorithm. In the baseline, the values are distributed between 20Mbps and 22Mbps. With RLVNA algorithm, values are spread from 50Mbps to 70Mbps. In this case, we can see the absence of outliners when RLVNA algorithm is disabled, but outliners are present when RLVNA algorithm is enabled. We can affirm that outliners generally have a value between 0.9 and 1.0 and are in the range from 70.5Mbps to 72.5Mbps.

4.2.2 Eight experiment switches

Before starting the analysis of the results, we want to report here the paths that the hosts ’h0’ and ’h1’ follows with and without the RL algorithm.



(a)



(b)

Fig. 4.9 Experiment Topologies. RLVNA algorithm deactivated in (a), RLVNA algorithm activated in (b)

Figure 4.9 depicts the paths used in the communication with h0-h6 and h1-h7 without the RLVNA algorithm. It is essential to underline that we did not draw the other hosts' paths, but as you can remember from the setting description, they share almost the same path as h1. On the left of Figure 4.9 (a) we depicted the path while the RL algorithm is activated. On the left of Figure 4.9 (b) we depicted the path when the RL algorithm is deactivated.

We can finally start by describing the results obtained in these two cases.

In Figure 4.10, it is possible to analyze the RTT for the eight switches' topology.

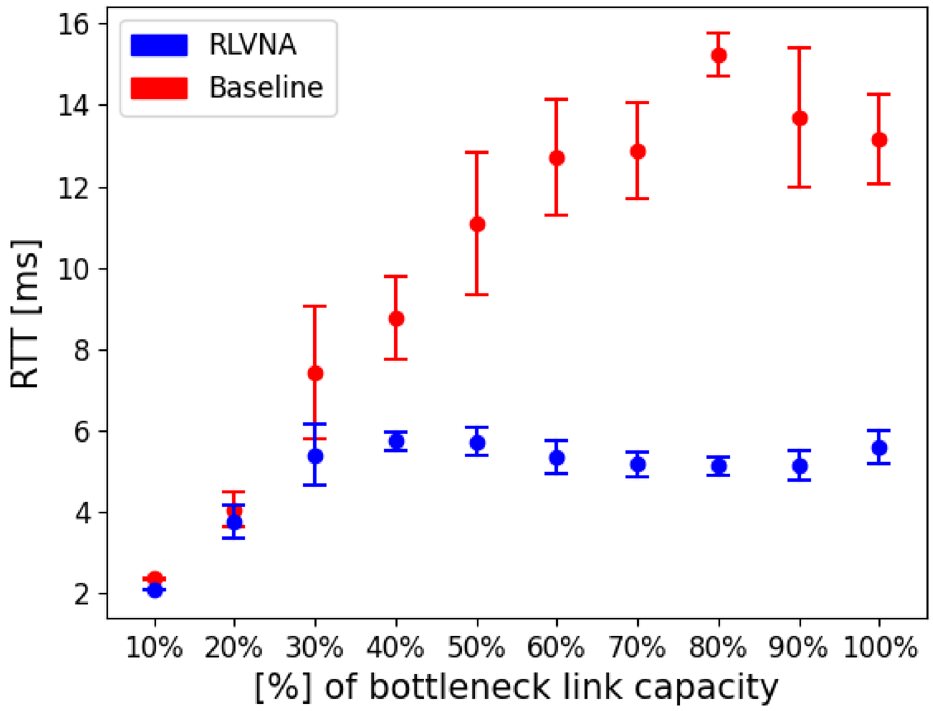


Fig. 4.10 RTT with 8 switches

As we can see from Figure 4.10 on the Y-axis, we have the Round Trip Time (RTT) measured in milliseconds. The RTT is when a packet goes forth and back from the source to the destination. On the X-axis, we have the bottleneck link capacity. It is important to remember that the GENI link saturation is at 100Mbps. The bottleneck link capacity is how much all clients send in the network. For example, 50 percent means that all clients are sending 50Mbps.

The case with RLVNA algorithm activated is almost constant because from 30 percent to 100 percent, the values are in the range from 5ms and 6 ms. This is entirely different in the default case because the values always rise from 6ms to 16ms. This means that when the load is high and RLVNA algorithm is off, the packets require more time to reach the destination and return. Viceversa, when the load is high and RLVNA algorithm is on, thanks to the new path we saw in Figure 4.9 the latency is almost the same.

In particular, we can see that when the load is low, the ML model probably decides not to use the new path. When the bottleneck link capacity is less than 30 percent, the values of the RTT with and without RLVNA algorithm are comparable with 5ms and 7ms. The values change when the load increases, and the use of the new paths is substantial. For example, around 50 percent, we have that with RLVNA algorithm the RTT is 6ms, instead without is around 9ms. Please also consider how the values are distributed across the confidence interval for this particular bandwidth. With RLVNA algorithm, the values are between 5.9ms and 6.1ms; Without instead, the values are in the range between 9ms and 13ms.

The most significant difference is around 80 percent because, without RLVNA algorithm, we can see values of 15ms which are three times more compared to the 5ms when RLVNA algorithm is activated.

In Figure 4.11, it is possible to analyze the throughput for the eight switches' topology.

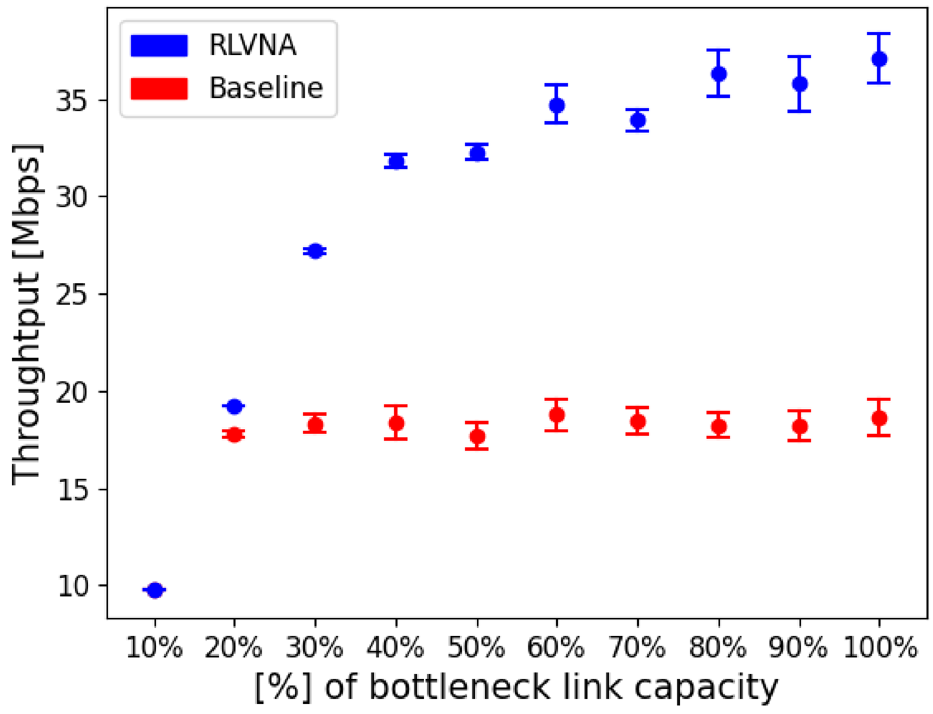


Fig. 4.11 Throughput with 8 switches

As we can see from Figure 4.11 on the Y-axis, we have the throughput measured in megabits per second. The throughput is the number of bits sent in a time interval.

On the X-axis, we always have the bottleneck link capacity. We remember again that the GENI link saturation is at 100Mbps. The bottleneck link capacity is how much all clients send in the network. For example, 50 percent means that all clients are sending 50Mbps.

The case with RLVNA algorithm activated is continuously rising because from 10 percent to 80 percent, the values increase from 10Mbps to 38Mbps. This is completely different in the default case because the values are constant at around 18Mbps. This means that when the load is high and RLVNA algorithm is off, the packets sent are much fewer because the links are saturated; instead, when RLVNA algorithm is activated, as you can remember from Figure 4.9, a new path is entirely available for h0, duplicating the performance. In the previous case with the four switches, the performance was three times more, probably because the switches cannot manage vast packets.

In particular, when the bottleneck link capacity is less than 20 percent, the throughput values with and without RLVNA algorithm are almost the same at 20Mbps. The values tend to change when the load increases more. For example, around 50 percent we have that with RLVNA algorithm the throughput is 32Mbps, instead without is the half around 16Mbps. The most significant difference is around 80 percent because in the default case, we can see values around 17.5Mbps, and with RLVNA algorithm instead, it is almost double around 35Mbps.

We now start the description of the Cumulative Distribution Function. As we did before, we will consider the values collected in the ten iterations of the following bandwidths:

- Throughput at 30Mbps, in Figure 4.12
- Throughput at 50Mbps, in Figure 4.13
- Throughput at 80Mbps, in Figure 4.14

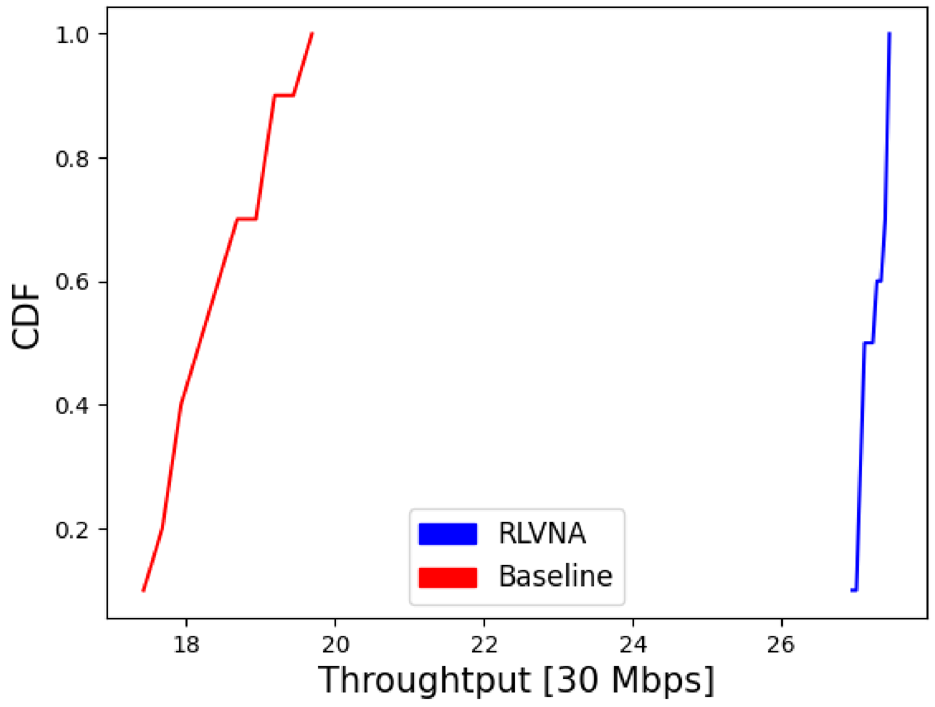


Fig. 4.12 Cumulative Distribution Functions with 8 switches 30M

In Figure 4.12, we can see how values are distributed when all the hosts send 30Mbps. Please notice the big difference in distribution with and without RLVNA algorithm. In the default, values are from 17Mbps to 21Mbps. With RLVNA algorithm instead, they are between 25.1Mbps and 25.5Mbps. We can see the presence of some outliner in the default case, between 19.7Mbps and 20Mbps.

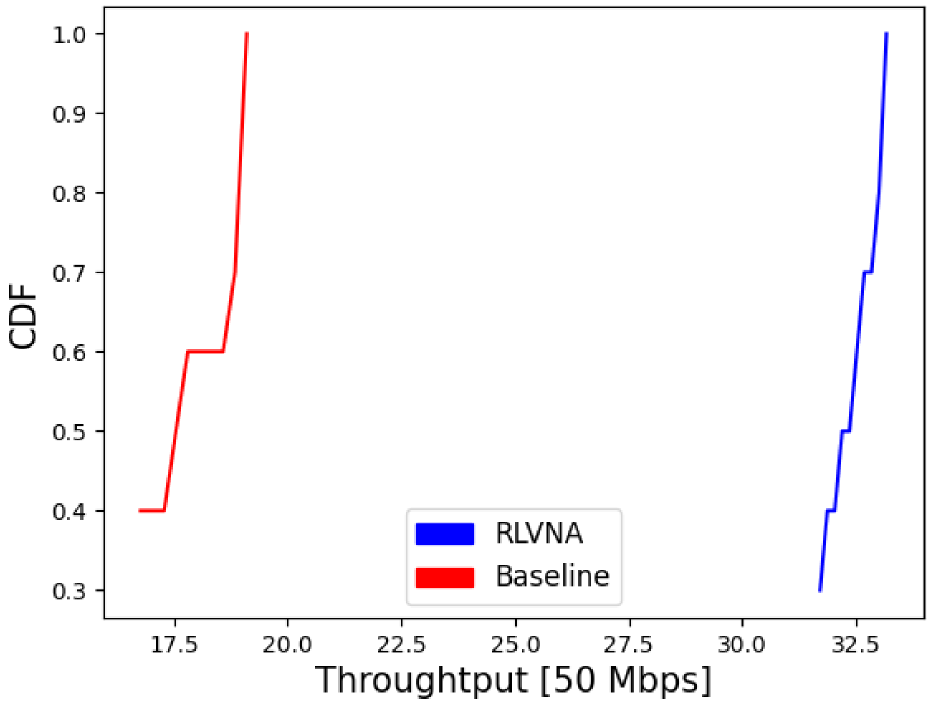


Fig. 4.13 Cumulative Distribution Functions with 8 switches 50M

In Figure 4.13, we can see how values are distributed when all the hosts send 50Mbps. Please notice the are no significant differences in distribution with and without RLVNA algorithm. With RLVNA algorithm, it is segmented between to 31.5Mbps and 33Mbps; Without values are spread from 17Mbps to 19.5Mbps. In this case, we can see outliners' absence when RLVNA algorithm is both activated and deactivated.

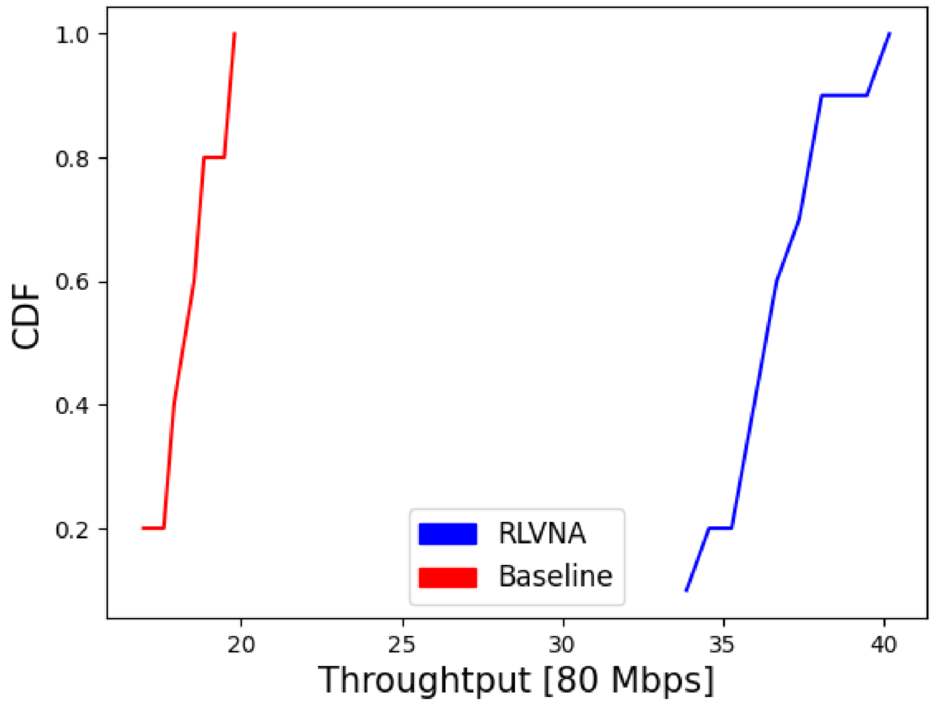


Fig. 4.14 Cumulative Distribution Functions with 8 switches 80M

In Figure 4.14, we can see how values are distributed when all the hosts send 80Mbps. Please notice the big difference in distribution with and without RLVNA algorithm. In the default case, the values are distributed between 17Mbps and 22Mbps. With RLVNA algorithm, values are spread from 34Mbps to 40Mbps. In this case, we can see the presence of outliners both when RLVNA algorithm is activated and disabled. We can affirm that outliners generally have a value between 0.9 and 1.0. In case RLVNA algorithm is OFF, it goes from 20Mbps to 20.5Mbps; when it is ON it goes in the range from 38.5Mbps to 40.5Mbps.

Chapter 5

Conclusion

In this thesis, we build a platform for experimenting with the GENI testbed with reinforcement learning. This work is innovative because GENI offers raw resources, but much effort is required to handle and customize them. With this project, we tried to implement a general platform that allows us to reserve resources and experiment with reinforcement learning algorithms. We are saying that even if we used the GENI testbed, it should not be so tricky to experiment with other architecture like Fabric (The successor of GENI) or Amazon Web Services. Moreover, changing the docker container should be easy enough, enveloping the reinforcement learning algorithm with others. However, changing the backend means changing the frontend as a consequence, but since we are talking of reinforcement learning algorithms, probably many of the settings available now remain equal. The advantage of the front end is that since we used React, the components are already defined, so what is needed to change is only the component content.

From this point, the path for future work is endless. We want here to draw the path for two exciting ideas. The first could be to scale this system in a distributed environment with many controllers, each taking care of a different subnetwork. In this case, all controllers should communicate with each other to have a unique vision of the global topology. Talking about this future work, the code developed until now partially takes care of multi-controller because we had it in mind when we designed the project, but the controller communication will require enough effort. Another interesting extension of the actual project could be the use of multi-sites, also known as a distributed federated testbed. It means to scale geographically and use multiple aggregates. For example, for an experiment, it would be beautiful to connect ten nodes in California aggregate with ten Nodes in Illinois aggregate using a compatible link over Internet2. In this case, the process is much more difficult. We already tried to implement this feature, but unfortunately, the geni-lib does not provide any comfortable mechanism for achieving this deal. By the way, we are confident enough that if not with GENI, a successor like FABRIC will provide this feature as agile as possible.

References

[1] Alessio Sacco, Matteo Flocco, Flavio Esposito, and Guido Marchetto. Supporting sustainable virtual network mutations with mystique. *IEEE Transactions on Network and Service Management*, 18(3):2714–2727, 2021.

[2] Diego Kreutz, Fernando M. V. Ramos, Paulo Esteves Veríssimo, Christian Esteve Rothenberg, Siamak Azodolmolky, and Steve Uhlig. Software-defined networking: A comprehensive survey. *Proceedings of the IEEE*, 103(1):14–76, 2015.

[3] Nick Feamster, Jennifer Rexford, and Ellen Zegura. The road to sdn: An intellectual history of programmable networks. *SIGCOMM Comput. Commun. Rev.*, 44(2):87–98, apr 2014.

[4] Fulvio Risso. Sdn architecture figure. Technical report, Politecnico di Torino, 2021.

[5] Hongzi Mao, Mohammad Alizadeh, Ishai Menache, and Srikanth Kandula. Resource management with deep reinforcement learning. In *Proceedings of the 15th ACM Workshop on Hot Topics in Networks*, HotNets '16, page 50–56, New York, NY, USA, 2016. Association for Computing Machinery.

[6] Leslie Pack Kaelbling, Michael L. Littman, and Andrew W. Moore. Reinforcement learning: A survey. *CoRR*, cs.AI/9605103, 1996.

[7] Yuxi Li. Deep reinforcement learning: An overview. *CoRR*, abs/1701.07274, 2017.

[8] Nick Feamster and Jennifer Rexford. Why (and how) networks should run themselves. *ANRW '18: Proceedings of the Applied Networking Research Workshop*, pages 20–20, 07 2018.

[9] Patrick Kalmbach, Johannes Zerwas, Péter Babarczi, Andreas Blenk, Wolfgang Kellerer, and Stefan Schmid. Empowering self-driving networks. In *Proceedings of the Afternoon Workshop on Self-Driving Networks*, SelfDN 2018, page 8–14, New York, NY, USA, 2018. Association for Computing Machinery.

[10] Arthur Selle Jacobs, Ricardo José Pfitscher, Ronaldo Alves Ferreira, and Lisandro Zambenedetti Granville. Refining network intents for self-driving networks. In *Proceedings of the Afternoon Workshop on Self-Driving Networks*, SelfDN 2018, page 15–21, New York, NY, USA, 2018. Association for Computing Machinery.

[11] Alessio Sacco, Flavio Esposito, and Guido Marchetto. A distributed reinforcement learning approach for energy and congestion-aware edge networks. In *Proceedings of the 16th International Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '20, page 546–547, New York, NY, USA, 2020. Association for Computing Machinery.

[12] Mark Berman, Chip Elliott, and Lawrence Landweber. Geni: Large-scale distributed infrastructure for networking and distributed systems research. In *2014 IEEE Fifth International Conference on Communications and Electronics (ICCE)*, pages 156–161, 2014.

[13] Larry Peterson, Tom Anderson, Dan Blumenthal, Dean Casey, David Clark, Deborah Estrin, Joe Evans, Dipankar Raychaudhuri, Mike Reiter, Jennifer Rexford, Scott Shenker, and John Wroclawski. Geni design principles, September 2006.

[14] Mark Berman, Jeffrey S. Chase, Lawrence Landweber, Akihiro Nakao, Max Ott, Dipankar Raychaudhuri, Robert Ricci, and Ivan Seskar. Geni: A federated testbed for innovative network experiments. *Computer Networks*, 61:5–23, 2014. Special issue on Future Internet Testbeds – Part I.

[15] GENI. Geni resources. https://www.geni.net/about-geni/geni-maps/, 2022.

[16] Lei Liu, Wei-Ren Peng, Ramon Casellas, Takehiro Tsuritani, Itsuro Morita, Ricardo Martínez, Raül Mu noz, Masatoshi Suzuki, and S. J. Ben Yoo. Dynamic openflow-based lightpath restoration in elastic optical networks on the geni testbed. *J. Lightwave Technol.*, 33(8):1531–1539, Apr 2015.

[17] Abhimanyu Gosain and Ivan Seskar. Geni wireless testbed: An open edge ecosystem for ubiquitous computing applications. In *2017 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*, pages 54–56, 2017.

[18] Abhimanyu Gosain, Mark Berman, Marshall Brinn, Thomas Mitchell, Chuan Li, Yuehua Wang, Hai Jin, Jing Hua, and Hongwei Zhang. Enabling campus edge computing using geni racks and mobile resources. In *2016 IEEE/ACM Symposium on Edge Computing (SEC)*, pages 41–50, 2016.

[19] Alex Berryman, Prasad Calyam, J. Cecil, George B. Adams, and Douglas Comer. Advanced manufacturing use cases and early results in geni infrastructure. In *2013 Second GENI Research and Educational Experiment Workshop*, pages 20–24, 2013.

[20] Tae Hwang. Nsf geni cloud enabled architecture for distributed scientific computing. In *2017 IEEE Aerospace Conference*, pages 1–8, 2017.

[21] Ilia Baldine, Yufeng Xin, Anirban Mandal, Chris Heermann Renci, Unc-Ch Jeff Chase, Varun Marupadi, Aydan Yumerefendi, and David Irwin. Networked cloud orchestration: A geni perspective. In *2010 IEEE Globecom Workshops*, pages 573–578, 2010.

[22] Yimeng Zhao, Samantha Lo, Ellen Zegura, Mostafa Ammar, and Niky Riga. Virtual network migration on the geni wide-area sdn-enabled infrastructure. In *2017 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, pages 265–270, 2017.

[23] Ali Sydney, David S. Ochs, Caterina Scoglio, Don Gruenbacher, and Ruth Miller. Using geni for experimental evaluation of software defined networking in smart grids. *Computer Networks*, 63:5–16, 2014. Special issue on Future Internet Testbeds - Part II.

[24] Jonathon Duerig, Robert Ricci, Leigh Stoller, Matt Strum, Gary Wong, Charles Carpenter, Zongming Fei, James Griffioen, Hussamuddin Nasir, Jeremy Reed, and Xiongqi Wu. Getting started with geni: A user tutorial. *SIGCOMM Comput. Commun. Rev.*, 42(1):72–77, jan 2012.

[25] Karamjeet Kaur, Japinder Singh, and Navtej Ghumman. Mininet as software defined networking testing platform. In *Mininet as Software Defined Networking Testing Platform*, 08 2014.

[26] Mininet. Mininet tool description. http://mininet.org/, 2022.

[27] GoJs Developers. *https://gojs.net/latest/intro/index.html*.

[28] Django Developers. *https://www.djangoproject.com/*.

[29] Django Developers. *https://geni-lib.readthedocs.io/en/latest/index.html*.

[30] Docker Developers. *https://docs.docker.com/get-started/overview/*.

[31] Conda Developers. *https://docs.conda.io/en/latest/*.

[32] Wikipedia Contributors. *https://en.wikipedia.org/wiki/Iperf*.

[33] Oluwamayowa Ade Adeleke, Nicholas Bastin, and Deniz Gurkan. Network traffic generation: A survey and methodology. *ACM Comput. Surv.*, 55(2), jan 2022.

[34] Jonas Jelten. Moongen tutorial. *Seminars FI / IITM WS 15/16*, 2016.

[35] Wikipedia Contributors. *https://en.wikipedia.org/wiki/Netperf*.

Appendix A

Frontend pseudo-code

In this apprendix we will report the most important pieces of code we implemented for the Frontend. We will start with a brief introduction of the React architecture, the object, and the components we choose, then we will discuss the GoJs library used for the topology definition. For a deeper and complete analysis of the code we suggest you to have a look to the private repository: https://github.com/Enrico-git/NGI

React

```
export default class Input extends React.Component{
        ...
                topology: new CreateTopology(),
        ...
                <Topology topology={this.state.topology}/>
                <Settings topology={this.state.topology}/>
        ...
}

class Topology extends React.Component{
        ...
        createTopology = () => {
                const req = {
                        method: "POST",
                        headers: {"Content-Type": "application/json"},
                        body: JSON.stringify({
                                topology: JSON.stringify(
                                        this.state.topology)
                        })
                };
                fetch("/api/topology", req)
                        .then((res) => res.json())
                        .then((data) => console.log(data))
        }
        ...
        <Container>
                <GoJsApp topology = {this.props.topology}  />
                <Button...>Create Topology</Button>
        </Container>
        ...
}
class Settings extends React.Component{
        ...
        <Container>
                <Ryu />
                <MLModel topology={this.props.topology}/>
                <SSH />
        </Container>
        ...
}

export default class CreateTopology{
        constructor(){
                this.ctrl = {}
                this.sw = {}
                this.links = {}
                this.num_ctrl = 0
                this.num_sw = 0
                this.num_h = 0
                this.num_link = 0
        }

        increase_h(){ ... }
        add_ctrl(){ ... }
        add_sw(){ ... }
        link_ctrl(ctrl, sw){ ... }
        link_h(sw, h){ ... }
        link_sw(src, dst){ ... }
}
```

GoJs

```
export default class GoJsApp extends React.Component{
        ...
        <ReactPalette
                initPalette={this.initPalette}
                divClassName="paletteComponent"
                nodeDataArray={[
                        { ... "controller"},
                        { ... "switch"},
                        { ... "host"}
                ]}
        />

        <ReactDiagram
                initDiagram={this.initDiagram}
                divClassName='diagram-component'
                onModelChange={this.handleModelChange}
                topology={this.props.topology}
        />
        ...

        initDiagram = () => {
                const $ = gos.GraphObject.make;
        const diagram =
                $(gos.Diagram,
                model: $(gos.GraphLinksModel,
                ...
                );

        diagram.nodeTemplate =
                $(gos.Node, 'Auto',
                $(gos.Shape, 'RoundedRectangle',
                ...
        );

        diagram.linkTemplate =
                $(gos.Link,
                $(gos.Shape, {toArrow: "Line"})
                ...
                )
        ...
        return diagram;
}

initPalette = () => {
        const $ = gos.GraphObject.make;

        var myPalette =
                $(gos.Palette,
                        ...
                );

        myPalette.nodeTemplate =
                $(gos.Node,
                "Horizontal",
                $(new gos.Binding("fill", "color")
                ),
                ...
        );

        return myPalette;
}

handleModelChange = (changes)  => {
        if (changes.modifiedLinkData != undefined){
                let from = changes.modifiedLinkData[0].from;
                let to = changes.modifiedLinkData[0].to;

                if ((from.startsWith('c'))
                        && (to.startsWith('s'))) {
                        this.props.topology.link_ctrl(from, to)
                }else if ((from.startsWith('s'))
                                && (to.startsWith('s'))){
                        this.props.topology.link_sw(from, to)
                }else if ((from.startsWith('s'))
                                && (to.startsWith('h'))){
                        this.props.topology.link_h(from, to)
                }
        }else if (changes.insertedNodeKeys != undefined){
                node = changes.insertedNodeKeys[0].
                if (node.startsWith('c')){
                        this.props.topology.add_ctrl();
                }else if (node.startsWith('s')){
                        this.props.topology.add_sw()
                } else if (node.startsWith('h')){
                        this.props.topology.increase_h()
                }
                this.setState(() => ({
                        topology: this.props.topology,
                }));
        }
```

```
          }
} ;
```

Appendix B

Backend pseudo-code

In this apprendix we will report the most important pieces of code we implemented for the Backend. We will start with a long introduction of the GENI context and REST APIs we defined, then we will discuss the scripts used for the node configuration. Finally, we will have a look to the most important part of the Reinforcement Learning-based Virtual Network Adaptation (RLVNA) algorithm generalization. For a deeper and complete analysis of the code we suggest you to have a look to the private repository: https://github.com/Enrico-git/NGI

GENI context

```
def buildContext ():
    print("buildContext: building the 'portal' by 's279434'")
    framework = FrameworkRegistry.get("portal")()
    framework.cert = ".../geni-s279434.pem"
    framework.key = ".../geni-s279434.pem"

    user = User()
    user.name = "s279434"
    user.urn = "urn:publicid:IDN+ch.geni.net+user+s279434"
    user.addKey(".../id_ed25519.pub")

    context = Context()
    context.addUser(user)
    context.cf = framework
    context.project = "network-scalability-tests"

    return context
```

Backend APIs

```
class Experiment:
def __init__(self):
    self.target_slice = 'EAtest'
    self.target_aggregate = IGAM.Illinois
    self.context = mycontext.buildContext()
    ...

def _delete_previous_aggregate(self):
    # This function clear previous instance
    # of nodes reserved in this aggregate
    # and slice. It is an internal function
    # called by create_topology()
    ...

def _wait_resource_ready(self):
    # This function waits for resource allocation.
    # This is needed before starting the node
    # customization. It is an internal function
    # called by create_topology()
    ...

def _get_ssh(self):
    # This function parses the manifest and
    # defines a list of ssh commands for
    # connecting to nodes. It is an internal
    # function called by create_topology()
    ...

def create_topology(self, data):
    # Decode the topology
    data = json.loads(data.topology)
    ...

    self._delete_previous_aggregate()
    ...

    # Creating the rspec/manifest based by front-end request
    rspec = PG.Request()
    rspec.addResource(...)
    rspec.writeXML('XX.xml')
    ...
    manifest = self.target_aggregate.createsliver(
            self.context, self.target_slice, rspec)
    manifest.writeXML("YY.xml")
    ...

    # Geni resource has been allocated but not ready yet
    self._wait_resource_ready()
    ...

    # Read the manifest and configure the nodes as follow:
    for i in range(len(nodes_geni)):
        ...
        if node_id.startswith('c') :
            ssh.connect(hostname, port, usr)

            #Get IP CTRL assigned by geni
            IP_CTRL = ssh.exec_command("hostname -I")
            ...

            # Install Docker, download image, create container
            ssh.exec_command("bash config_docker.sh")
            ...

            ssh.close()

    # Configure OVS switches
    for i in range(len(nodes_geni)):
        if node_id.startswith('s') :
            ssh.connect(hostname, port, usr)
            ssh.exec_command(cmd_config_OVS)
            ssh.close()
            ...
        elif node_id.startswith('h') :
            ssh.connect(hostname, port, usr)
            ssh.exec_command(cmd_install_iperf3_netperf)
            ssh.close()
            ...

    # create login file
    self._get_ssh()

    return 'ok'

def configure_controller(self, data):
    # This function log with ssh into the controller
    # node and generates the config.ini file used
    # for Ryu SDN controller

    for i in range(len(nodes_geni)):
        if node_id.startswith('c') :
            # parse parameters from front-end settings
            ...
            cmd_config_ryu='config_ryu.sh '+settings
            cmd_docker_ryu='sudo docker exec -d RLVNA '
                            + cmd_config_ryu
            ssh.exec_command(cmd_docker_ryu)
            ...
    return ret_msg

def _get_num_intfs(self):
    # This function count the interfaces
    # in all switches. It called by
    # configure_model()
    ...

def configure_model(self, data):
    # This function log with ssh into the controller
    # node and generates the config2.ini file used
    # for RL model

    for i in range(len(nodes_geni)):
        if node_id.startswith('c') :
            # parse parameters from front-end
            ...
            NumIntf=' '+str(self._get_num_intfs())
            cmd_config_model='bash config_model.sh'+settings
            cmd_docker_model='sudo docker exec -d RLVNA '
                                    + cmd_config_model
            ssh.exec_command(cmd_docker_model)
            ...
    return ret_msg

def configure_hosts(self, config, traffic_type):
    # This function log with ssh into each host
    # node and download the traffic generator
    # scripts

    for i in range(len(nodes_geni)):
        if node_id.startswith('h'):
            cmd_config='git clone .../NGI-support.git'
            ssh.exec_command(cmd_config)
            ...

    return ret_msg

def download_file(self, file):
    # This function logs inside the target node
    # and downloads the measurement done
    # during traffic generation.

    if file == 'traffic':
        ssh.connect(hostname, port, usr)
        with SCPClient(ssh.get_transport()) as scp:
            scp.get('iperf3_total.txt')
        ssh.close()
    elif file == 'latency':
        with SCPClient(ssh.get_transport()) as scp:
            scp.get('netperf_latency.txt')
    elif file == 'reward':
        cmd_reward='sudo docker cp RLVNA:/.txt .'
        ssh.exec_command(cmd_reward)
        ...
        with SCPClient(ssh.get_transport()) as scp:
            scp.get('cpu_300_rewards.txt')
    elif file == 'login':
        return ssh_login.txt
    ...
    return ret_msg
```

Configuration script

```
#!/bin/bash

#Read node's intefaces
```

```
INTFS=$(ip address | grep 192.168.[0-6])
NUM_INTFS=$(ls -A /sys/class/net | wc -l)

#Params from commandline
IP_CTRL=${1}
SW=${2}

#Creating OVS switch
sudo ovs-vsctl add-br $SW

# Disabling interfaces from node and Mapping the
# interfaces from node to OVS
for ((i=0; i < $NUM_INTFS; i++)); do
    INTF='eth'$i
    if [[ $INTFS =~ $INTF ]]
    then
        sudo ifconfig $INTF 0
        sudo ovs-vsctl add-port $SW $INTF
    fi
done

#setting the IP of the controller
sudo ovs-vsctl set-controller $SW tcp:$IP_CTRL:6633
sudo ovs-vsctl set-fail-mode $SW secure
```

Reinforcement Learning-based Virtual Network Adaptation Generalizzation

```
FROM ubuntu:20.04
COPY RLVNA /RLVNA
WORKDIR /RLVNA

RUN apt update && apt install python3.8 python3-pip -y \
    && pip3 install -r ryu_controller/requirements.txt
```

```
RUN DEBIAN_FRONTEND=noninteractive apt install -y swig \
        python2-dev python2.7 python-is-python2 \
        python-dateutil python-openssl libxmlsec1 \
        xmlsec1 libxmlsec1-openssl libxmlsec1-dev && \
        python2 get-pip.py && pip2 install M2Crypto && \
    tar -xf geni-tools-2.11.tar.gz
        && mv geni-tools-2.11 /usr/local/bin/ && \
    /usr/local/bin/geni-tools-2.11/src/omni-configure.py \
    -z omni.bundle

# Miniconda created with container commit
    ...

class Configuration_V1:
    def __init__(self, support_switches, slice, aggregate):
        self._get_slice_info()
        ...

        self.shortest_paths =
            self._find_all_shortest_paths(...)

        self.datapaths = {}
        self.path_to_port = {}
        ...

    def _get_slice_info(self):
        # This function generates the manifest
        # of the slice and get the GENI topology

        cmd = 'omni.py listresources "'
            +self.slice_name+'" -a "'
            +self.aggregate_name+'" -o;'
        subprocess.run(cmd , shell=True, executable='bash')

        ...
        self.num_hosts = ...
        for i in range(self.num_hosts):
            h = {}
            h[mac] = (ip, intf_name)
            ...
            self.hosts[h_id] = h

        self.num_ctrls = ...

        self.num_switches = ...
        for i in range(self.num_switches):
            intfs = []
            for j in range(len_intfs):
                intf = {}
                intf[mac] = intf_name
                intfs.append(intf)

            self.switches[s_id] = {}
            self.switches[s_id]['intfs'] = intfs
            self.switches[s_id]['flows'] = {}

        self.num_links = ...
        for i in range (self.num_links):
            self.links.append(...)

    def _find_all_shortest_paths(...):
        # For each Switch, find the shortest
        # path to ALL host combining the path
        # with the support_switches and without
        ...
        return shortest_paths

    def config_switch(self, datapath):
        # Called for each switch that registers
        # to the controller. Insert flows in OVS

        self.datapaths[switch_id] = datapath
        # mapping between path and port to forward
        ...
        self.path_to_port[path] = port_info.port_no

        #Clear previous flows
        Interfaces.remove_flows(...)

        #Insert new Flows
        self.add_default_flow(datapath)
        self.add_all_flows(datapath, s_id)

    def add_all_flows(self, datapath, s_id):
        # Insert the flows from (all) the switch to
        # all the dest. At startup do not consider
        # path with support_switches (off).

        for i in range(len(self.shortest_paths[s_id])):
            ...
            if not skip_path :
                ...
                self.switches[s_id]['flows'][dest] = out_port
                self.add_l3_flow(datapath, ip_dst, out_port)

def enable_sw(self, s_id):
    # Called when the model decides to enable a support
    # switch. It finds all neighbour of the support one.
    # Given a neighbour find all hosts that could be reached
    # through target support switch. These hosts are
    # filtered (only half are selected for the new path)

    if self.support_switches[s_id] :
        return

    target_neighbours = self._find_sw_neighbours(s_id)
    for neighbour_id, s_info in target_neighbours.items():
        #Select just the half of the available hosts
        ...
        self._shift_intf(switch_id=neighbour_id,
            curr_out_port=cur_port, new_out_port=new_port,
            h_id=h)

    self.support_switches[s_id] = True

def disable_sw(self, s_id):
    # Called when ML model decides to disable a support
    # switch. The same idea of enable switch function
    # but here we tried to rollback (like in config_switch)

    if not self.support_switches[s_id] :
        return

    target_neighbours = self._find_sw_neighbours(s_id)
    ...
    self._shift_intf(switch_id=neighbour_id,
        curr_out_port=cur_port, new_out_port=new_port,
        h_id=h)

    self.support_switches[s_id] = False

def _find_sw_neighbours(self, s_id):
    # Find all switches attached to the 'support' one
    # and the relative port. Then it finds all hosts
    # reacheable from the switch through the support.
    ...
    return neighbours
```

Appendix C

Traffic generator pseudo-code

In this apprendix we will report the most important pieces of code we implemented for traffic generation. We will discuss mainly the static traffic generation for the 8 switches topology. For a deeper and complete analysis of the code we suggest you to have a look to the public repository: https://github.com/Enrico-git/NGI-support

Traffic generator 8 switches context

```
class TrafficGenerator:
    def __init__(self):
        self.num_iperf=int(sys.argv[1])
        self.list_ip=sys.argv[2].split(',')

        self.svr_num = 4 #h6, h7, h8, h9
        self.cl_num = int(len(self.list_ip) - self.svr_num)

        # NOTE: last addresses in the list are used as iperf3 -s
        self.ip_svrs = self.list_ip[-self.svr_num:]

        #Grab IP host
        proc = subprocess.run(['hostname', '-I'], ...)
        list_addresses = proc.stdout.split(' ')
        self.my_addr = list(filter(lambda el:
            el.startswith('192.168'), list_addresses))[0]

        self.hostname='h'+str(self.list_ip.index(self.my_addr))

        self.num_load = 10 # from 10 Mbps to 100 Mbps
        self.first_port=6969
        random.seed(19951018+int(self.hostname[1:]))

    def run_server(self, port):
        #run one thread for each port
        if port == 6969: #h0 uses netperf
            subprocess.run(['sudo', 'pkill', 'netserver'], ...)

            #start process in background
            subprocess.run(['sudo', 'netserver', '-4',
                            '-p', f'{port}'], ...)
        else:
            server = iperf3.Server()
            server.bind_address=self.my_addr
            server.port=port
            print(f'iperf3 -s from {self.my_addr}, port {port}')
            while True:
                test = server.run()
                print(test)

    def generate_traffic(self):
        if self.my_addr in self.ip_svrs:
            processes = []
            if self.hostname == 'h6':
                #run num port needed for each server
                for i in range(1):
                    proc_port = self.first_port + i # 6969=h0
                    proc = Process(self.run_server, proc_port)
                    processes.append(proc)
                    proc.start()
        for i in range(1):
            processes[i].join()
    elif self.hostname == 'h7':
        for i in range(3):
            # 6973=h4, 6974=h5, h1=6975
            proc_port = 6973 + i
            proc = Process(self.run_server, proc_port)
            processes.append(proc)
            proc.start()
        for i in range(3):
            processes[i].join()
    elif self.hostname == 'h8':
            proc_port = 6971 # 6971= h2
            proc = Process(self.run_server, proc_port )
            processes.append(proc)
            proc.start()
            processes[0].join()
    elif self.hostname == 'h9':
            proc_port = 6972 # 6972= h3
            proc = Process(self.run_server, proc_port)
            processes.append(proc)
            proc.start()
            processes[0].join()
else:
    for j in range(self.num_load): # [10, .., 100] Mbps
        for i in range(self.num_iperf):
            if self.hostname == 'h0':
                server_hostname = self.ip_svrs[0]  #h6
                port = self.first_port
                        + int(self.hostname[1:])
                time=20
                proc = subprocess.run(['netperf',
                    '-H', f'{server_hostname}',
                    '-p', f'{port}', '-l', f'{time}',
                    '-t', 'TCP_RR', '--',
            '-o', 'mean_latency'], ...)

        #save measurement on file.
        filename = 'netperf_latency.txt'
        with open(filename, "a") as f:
            f.write(json.dumps(mean_rtt)+'\n')
    else:
        client = iperf3.Client()
        client.port=self.first_port
                    + int(self.hostname[1:])
        if self.hostname == 'h1':
            client.port=6975
        client.bandwidth = 10 * (j + 1) #Mbps
        client.duration = 20 # seconds

        if self.hostname == 'h4'
            or self.hostname == 'h5'
            or self.hostname == 'h1':
            client.svr = self.ip_svrs[1] #h7
        elif self.hostname == 'h2':
            client.svr = self.ip_svrs[2] #h8
        elif self.hostname == 'h3':
            client.svr = self.ip_svrs[3] #h9

        while True:
            test = client.run()
            if test.error != None:
                print(test.error)
                time.sleep(10)
                continue
        del client
        time.sleep(random.randint(1, 5))
```