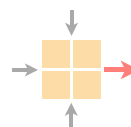




Eidgenössische Technische Hochschule Zürich  
Swiss Federal Institute of Technology Zurich



Networked Systems  
ETH Zürich — seit 2015

# Improving current P4 prototyping tools

Semester Thesis

Author: Jurij Nota

Tutor: Edgar Costa Molero

Supervisor: Prof. Dr. Laurent Vanbever

February 2021 to May 2021

## **Abstract**

In recent years there has been a paradigm shift in the networking community due to the rise of Software-defined networking devices. In this context P4, a domain-specific data plane programming language, is becoming a very hot topic. Unfortunately, P4 programmable switches are still expensive and operate them might still be somehow cumbersome. To overcome this, several software models of these devices have been designed and now can be used for P4 testing and prototyping. In this thesis, we aim at improving currently available software P4 tools to make them up-to-date and ready for future developments in this field.

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Motivation . . . . .	1
1.2	Task and goals . . . . .	2
1.3	Overview . . . . .	3
<b>2</b>	<b>Background and Related Work</b>	<b>4</b>
2.1	Mininet . . . . .	4
2.2	Unix domain socket . . . . .	5
2.3	NetworkX . . . . .	6
2.4	P4 Workflow . . . . .	6
2.5	P4-Utills . . . . .	7
2.5.1	Related Work . . . . .	7
2.5.2	Structure . . . . .	8
<b>3</b>	<b>Improving <i>P4-Utills</i></b>	<b>11</b>
3.1	Backend . . . . .	12
3.1.1	Migration to Python 3 . . . . .	12
3.1.2	Improved modularity . . . . .	12
3.1.3	TaskServer and TaskClient . . . . .	13
3.1.4	P4Mininet and FRRouter . . . . .	14
3.2	Frontend . . . . .	15
3.2.1	NetworkAPI and AppRunner . . . . .	15
3.2.2	NetworkGraph . . . . .	16
3.2.3	P4CLI . . . . .	16
3.3	P4Runtime . . . . .	17
3.4	P4 Learning . . . . .	17
<b>4</b>	<b>Using the new <i>P4-Utills</i></b>	<b>19</b>
4.1	Frontend changes . . . . .	19
4.2	New features in action . . . . .	21
<b>5</b>	<b>Conclusion and Future Work</b>	<b>25</b>
	<b>References</b>	<b>26</b>
<b>A</b>	<b>FRRouting example configuration files</b>	<b>I</b>

# Chapter 1

## Introduction

Recently, the rise of new programmable technologies in the field of network engineering has caused a major change in the way systems are deployed and configured. Indeed, thanks to the so-called Software-defined networking, operators can program both control and data plane of devices, customizing the network behaviour even more. Among others, one emerging topic in the field is P4, a domain-specific programming language that specifies how data plane devices process packets [25]. The key factor that makes P4 a very useful and interesting tool is that it has been designed to be target-independent (i.e. it can be used with a wide range of both hardware-based and software-based architecture) and protocol-independent (i.e. targets are not bound to any specific network protocol).

To prototype and test P4 data plane applications without the need for actual devices as well as to easily get acquainted with the new technology, P4 software switches [2] have been developed by the community. Moreover, thanks to existing network virtualization engines such as *Mininet* [6], more complex and realistic interactions between multiple P4 and non-P4 software devices can be analysed.

*Mininet* implements an efficient way of virtualizing nodes (hosts and switches) in a network by exploiting *Linux kernel Namespaces* [24]. Indeed, network namespaces create an isolated network stack for the process within it, allowing the virtualization of different nodes. These nodes can be then linked together with *Virtual Ethernet Devices* [26] that simulate the physical wires among machines.

Existing tools such as *P4-Utills* [10] (currently used in the course "Advanced Topics In Communication Networks" together with the exercises contained in *P4 Learning* [9]) and *p4app* [11], that allow for easy P4 prototyping using the *Mininet* network virtualization framework, are becoming outdated and do not support the most recent features deployed by the P4 community. Therefore, to provide a better framework for exercises and prototyping and enhance the readiness with which new updates can be integrated, both *P4-Utills* and *P4 Learning* have to be renovated. This thesis will give to the reader an overview of the changes that were required to reach the goal and how the usability of the platform was affected by them.

### 1.1 Motivation

There are several reasons why an update for *P4-Utills* and *P4 Learning* is needed. First of all, *P4-Utills*, its ancestor *p4app* and *P4 Learning* are written in Python 2. Unfortunately, Python 2.7 [15]), the last release of Python 2, reached its end of life on January 1 2020 so no more official support will be provided in the future. In addition, the P4 language consortium has recently defined

the *P4Runtime* API [23], a control plane specification for controlling the data plane elements of a device defined or described by a P4 program. This new communication API, which aims at being the standard for P4 devices so that even control plane programs can be target-independent, is currently not supported by *P4-Utils*, which only provides a *Thrift* API to control the devices data plane. Moreover, *P4-Utils* extracts all the information needed to create a virtual network from a JSON configuration file. This may represent a limitation for the user since there are no fully programmatic ways (i.e. using only executable scripts) to define the topology and create a virtual network. Finally, the current structure of the platform can be improved, exploiting modularity and external libraries, to make future changes easier. New functionalities can be added to allow more complex simulations: new helper features can assist the user in testing and debugging, whereas new nodes can extend the capabilities of the application. Since *P4-Utils* and *P4 Learning* are designed to work together, updates in the application structure entail changes in the exercises repository that must be done.

## 1.2 Task and goals

To address the issues mentioned above, we present a new version of *P4-Utils* (and *P4 Learning*) that aims at the following goals.

- Migrate *P4-Utils* and the exercises in *P4 Learning* to Python 3. This will ensure faster integration of further updates in the future.
- Enable *P4Runtime* on *P4-Utils* so that P4 capable switches can be controlled through both *Thrift* API and *P4Runtime* API.
- Create *P4Runtime* versions of the most important exercises present in *P4 Learning* so that they can also provide use cases of this new feature.
- Integrate *P4-Utils* with *Mininet* in a way that Python scripts can be used to define network topologies. The legacy method involving JSON configuration files should be kept to ensure backward compatibility.
- Set up a task scheduler to easily execute commands and scripts in the nodes at defined times. This can be operated at start time with a configuration file or using the *Mininet* client afterwards. In particular, this is beneficial since a task can also be used to generate traffic among the nodes of the network.
- Add **FRRrouter** node to the application so that users can create and test more complex networks and make use of all the functionalities provided by the *FRRouting* Internet protocol suite [4]. This change is particularly important because no router node was present in *P4-Utils* before.

The *Mininet* API was a key enabler for such changes. Indeed, *Mininet* provides abstractions for virtual links, nodes and interfaces that allow the user to create complex networks with ease. All these main building blocks are based on tools provided natively by the Linux kernel, making the implementation very efficient. *P4-Utils* takes advantage of these features and, by integrating specific code aimed at compiling and executing P4 code, allows deploying P4 programmable switches in the virtual network. The external code, which is crucial for the application, is provided by the P4 community.

### 1.3 Overview

First of all, an important remark must be made. Throughout this thesis, several diagrams describing software are used. These are designed to provide an overview of the modular structure and interdependency relations among the various parts of the program, but they are not meant to be complete and detailed. Indeed, for the sake of clarity, only the parts and relations most relevant for the discussion are displayed. For this reason, the diagrams of this kind are indicated as *UML-like*: they do not include all the information dictated by UML but are still compliant with the UML arrow standard.

In chapter 2 we give an overview of the technologies employed and the structure of *P4-Utils* as it was before the update. This is meant to shed light on the platform functioning to make the reader more informed about it. In addition to this, a reference to previous related work is provided. In chapter 3 we analyse in detail the most important changes that were implemented in the structure of *P4-Utils* to overcome the issues presented in section 1.1. In chapter 4, we provide usage examples of the updated version of the application. Finally, in chapter 5, we summarize the work done and list further improvements that can be carried out in future works.

## Chapter 2

# Background and Related Work

To make sense of which changes were necessary and why, we take a closer look at the previous works and underlying technologies. The following sections will cover all the main building blocks of *P4-Utils* as they were before the update and, finally, will give an overview of the application as it was before the update.

### 2.1 Mininet

*Mininet* is a framework designed for the creation of a realistic virtual network on a single machine [6]. This is made possible by the following fundamental Linux kernel features.

- *Linux kernel Namespaces* wrap a global system resource in an abstraction that makes it appear to the processes within the namespace that they have their own isolated instance of the global resource [24]. In particular, network namespaces provide the processes within with an isolated network stack that can be modified without affecting the whole system. This is the main tool used by *Mininet* to manage different networks and virtual nodes on a single machine.
- *Virtual Ethernet Devices* (**veth**) act as tunnels between network namespaces to create a bridge to a physical network device in another namespace [26]. This technology allows *Mininet* to create links between different nodes, so that connectivity is established.
- *Traffic Control* (**tc**) is a set of Linux kernel tools aimed at controlling how traffic is sent by an interface. It is used to provide the virtual links with characteristics (e.g. latency, data loss rate and maximum bandwidth) meant to emulate the behaviour of an actual wire [19].

*Mininet* is very useful for testing new tools and making network prototypes thanks to its handy Python API that can be extended to fit the user's needs: the configuration of the aforementioned kernel features, which can be tedious and cumbersome, is done automatically following the diagram presented in figure 2.1.

The user defines a network topology using **Topo**, a class that exposes intuitive methods to add nodes and links among them. The user can also specify configuration parameters for the network elements to override *Mininet* default assumptions. These parameters include IP and MAC addresses for links and default gateways for hosts, as well as the implementation classes for main *Mininet* building blocks, namely **Node** and **Link**.

**Node** is the abstraction of a shell process running in a network namespaces and emulating a node in the network. Several kinds of nodes may be present (e.g. **Host**, **Switch**), but they are all specialized subclasses of **Node**.

**Link**, on the other hand, represents a pair of connected *Virtual Ethernet Devices*, possibly placed in two different namespaces. This simulates a physical link between two nodes of the network. A **Link** relies on two instances of **Intf** (i.e. the abstract representation of an interface) to configure IP and MAC addresses and establish connectivity. Moreover, a link can be equipped with **TCIntf**, a special kind of interface that allows configuring *Traffic Control* settings (e.g. latency, loss and bandwidth) meant to provide a better emulation of real-world devices. In addition, **TCLink** is a simple **Link** subclass which uses two instances of **TCIntf** as interfaces.

**Mininet**, i.e. the network abstraction, then reads the configured **Topo** object and starts building the virtual network accordingly. During this process **Mininet** creates several instances of **Node** (one for each node in the network) and **Link** (one for each pair of connected nodes) following the implementation of these classes.

Finally, the **CLI** comes into play as a command-line client designed to easily execute **ping** and **iperf** on multiple hosts without having to log into each of them and knowing their IPs. It is informed by **Mininet** about all the network data and provides also useful commands to print out the topology of the network.

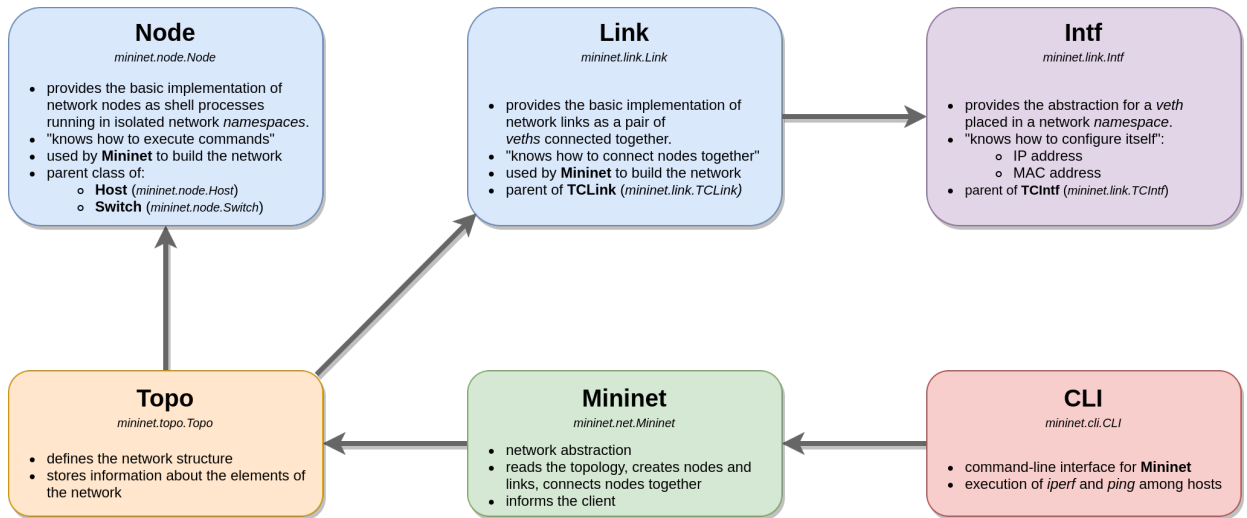


Figure 2.1: UML-like diagram of *Mininet*

## 2.2 Unix domain socket

The *Unix domain socket* is used to communicate between processes on the same machine efficiently [21]. Indeed, instead of being bound to an IP address and a port number, this socket is identified by a path in the filesystem which is its address. This fact is extremely beneficial because it provides a fast way to connect processes running on the same machine, even if they are in different network namespaces: the path of the socket (i.e. its address) does not change among network namespaces and, therefore, can be used with ease. Similarly to their network counterpart, *Unix domain sockets* can be stream-oriented (similar to TCP) or datagram-oriented (similar to UDP).



## 2.3 NetworkX

*NetworkX* is a Python library designed for the creation, manipulation, and study of the structure, dynamics, and functions of complex networks [8]. This module and its tools allow efficient computation of graph algorithms, which are important in networking. In particular, shortest paths algorithms are used to compute the best routing paths between two nodes and configure the network devices accordingly. This is the very reason why *NetworkX* is integrated into *P4-Utils*: some exercises contained in *P4 Learning* requires the user to create a script that controls the P4 switches in the network and, to do so, topology information is often required. There are two main classes provided by *NetworkX* which are of our interest:

- `networkx.classes.Graph` is the base class for an undirected simple graph, namely an undirected graph with no multiple parallel edges among nodes [5].
- `networkx.classes.MultiGraph` is the base class for an undirected multigraph, namely an undirected graph with possibly multiple parallel edges among nodes [7].

We can notice that both classes represent undirected graphs, that is graphs whose edges are always bidirectional. We chose them because the networks that *Mininet* emulate employ *Virtual Ethernet Devices* as connections and these, like their real-world counterparts, are bidirectional.

## 2.4 P4 Workflow

The workflow needed to execute P4 programs on targets is shown in figure 2.2. The process presented, which applies in particular to actual programmable hardware targets, is also useful to explain how P4 code is handled with software emulating devices like those provided by the *Behavioral Model* repository [2] that are used by *P4-Utils* because the fundamental steps needed to execute a P4 program are the same for both software and hardware targets.

The workflow starts with the P4 program written by the user: it contains all the instructions necessary to classify, forward, route and modify the packets flowing through the device. This code needs to be compiled to obtain the binary file (e.g. `target_demo.bin`) that the target uses to configure itself. The compiler deals with this task by taking into account the type of device, its available resources and by specifying how the hardware pipeline has to be set up. For the software targets implementing the *Behavioral Model*, this step is done thanks to *p4c* [12], the reference compiler provided by the P4 community, by passing a specific argument to it. The compiled file is then used to configure the target. In the case of software switches, the file is parsed by the virtual device executable which uses it to set up its internal pipeline.

What we explained, however, only involves the data plane so we now need to cover the operations needed to control the switch once it is active and configured with the compiler output: indeed forwarding rules and tables have to be changeable whenever it is needed. Hardware switches usually have implemented a server that can interact with the data plane and is accessible through its API from the control plane program. Usually, different manufacturers implement different servers with different APIs making it difficult to port control plane programs from one device to the other. This is in contrast with data plane programs which are written in P4 and, therefore, aimed at target-independence. The same applies to the *Behavioral Model* software switches which have their built-in *Thrift* server.

In recent years, to overcome this diversity of protocols and to make also control plane programs target-independent, *P4Runtime* [23] has been developed. Ideally, in the future, all manufacturers

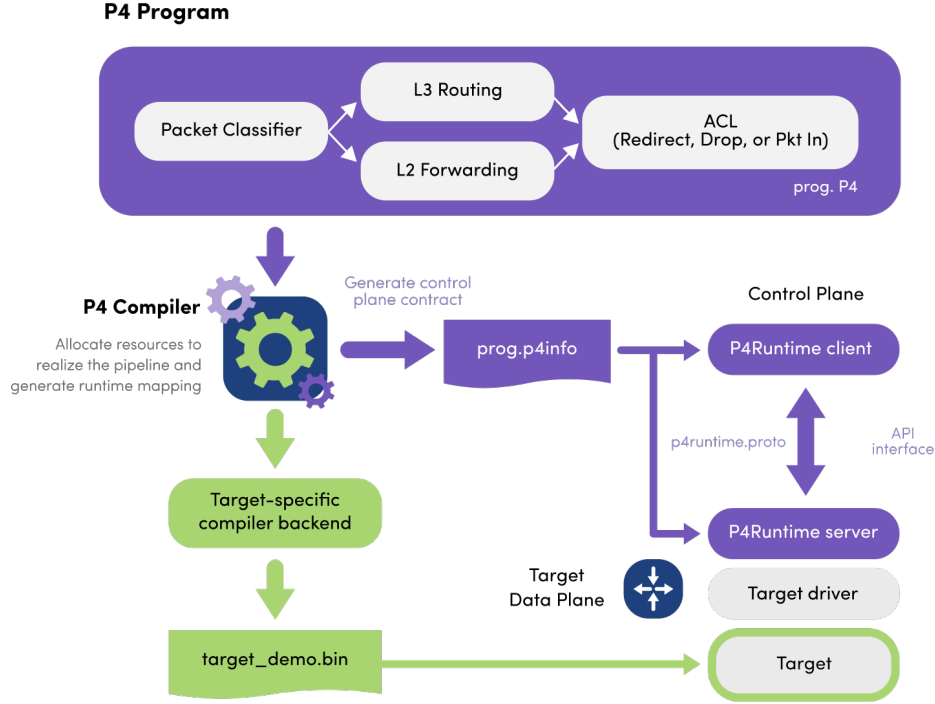


Figure 2.2: P4 workflow [25]

should adopt it as a standard for control plane programming. In response to this, also the *Behavioral Model* repository has been updated to include new switches that implement a built-in *P4Runtime* server.

In figure 2.2 it is shown that both the *P4Runtime* server in the data plane and *P4Runtime* client in the control plane need to be configured with a specific informative file (e.g. **prog.p4info**) generated by the compiler together with the binary configuration file. For what concerns the aforementioned software switches, *p4c* can output such file by enabling a specific option. This file is then pushed into the *P4Runtime* server of the switches through a gRPC connection and used by the control plane client to access and modify data plane entries.

## 2.5 P4-Utils

Hereafter we provide the reader with some background information about *P4-Utils*, as it was before the update, to better frame this semester thesis in the context of existing P4 prototyping platforms.

### 2.5.1 Related Work

The application *p4app* [11] is the ancestor of *P4-Utils* [10]: the former was created by the P4 community to provide a testing and prototyping platform based on P4 language, whereas the latter is an adaptation made by the ETH Networked Systems Group to simplify the application use and have a tool for P4 teaching. Indeed, during the course "Advanced Topics in Communication Network", held in fall 2021 at ETH, *P4-Utils* was extensively used together with the exercises contained in the repository *P4 Learning*.

Even if *P4-Utils* was created for educational purpose, it is currently used also outside the ETH

student community, as shown by the interactions in the GitHub repository. The same applies also for *P4 Learning* [9]. This proves that the community of people interested in learning and prototyping with P4 appreciates these tools. However, since three years have passed since the creation of the platform and only minor updates to maintain the code have been performed since then, deeper changes were needed.

## 2.5.2 Structure

The main components of *P4-Utills* and their mutual relations are shown in figure 2.3. The application is a wrapper around a *Mininet* core that allows to easily define, create and configure a network making use of some ancillary modules.

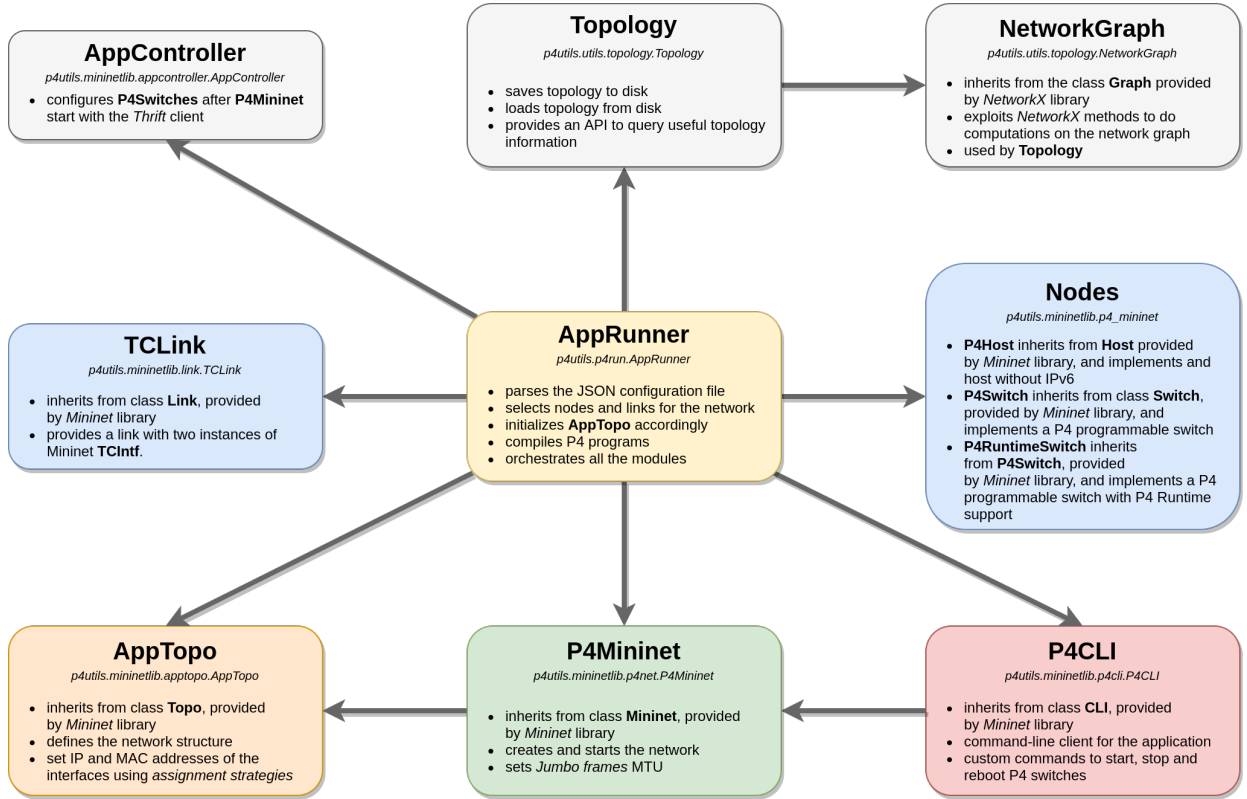


Figure 2.3: UML-like diagram of *P4-Utills* before the update

The operations start with **AppRunner** gathering all the information needed to set up the application. In particular, **AppRunner** gathers the network information including nodes, edges and their properties from a JSON file provided by the user. Afterwards, it uses *p4c* to compile the necessary P4 programs for the switches and prepares the network initialization by importing custom nodes and links which are subclasses of those provided by *Mininet*. *P4-Utills* makes use of three types of customized nodes:

- **P4Host** is a subclass of `mininet.node.Host`. It is a host with IPv6 disabled.
- **P4Switch** is a subclass of `mininet.node.Switch`. It is an extension that allows the integration in *Mininet* of the software switch `simple_switch` [16] provided by the *Behavioral Model*

repository. The class accepts as an argument the binary file generated by the compiler to configure the switch. This switch does not support *P4Runtime* and, therefore, it can be only accessed using its *Thrift* server. Apart from the main code involved in the creation of the network, *P4-Utills* has a *Thrift* Python API which can be imported in control plane scripts to configure the switches. This API is a modified version of the *Thrift* Python client developed by the P4 community [20].

- **P4RuntimeSwitch** is a subclass of **P4Switch**. It is provided by the *Behavioral Model* repository and integrates the software switch `simple_switch_grpc` [17]. The only difference between **P4Switch** and **P4RuntimeSwitch** is that the latter has a *P4Runtime* server in addition to the *Thrift* one. Unfortunately, before the update, P4-Utills did not have any *P4Runtime* API, so control plane scripts could not use it.

To emulate the behaviour of actual wires, *P4-Utills* uses its customized link class **TCLink**. This class, which inherits from `mininet.link.Link` has the only purpose of deploying `tc` capable interfaces that can be configured to limit bandwidth, enforce specific delay and data loss.

Once that **AppRunner** has gathered all the necessary information from the JSON configuration file and has retrieved the main *Mininet* blocks needed to instantiate the network (i.e. links and node classes), it is ready to pass them to **AppTopo** for further process. **AppTopo** is a subclass of `mininet.topo.Topo` and it is aimed at organizing the topology data and performing the assignment of both IP and MAC addresses for the interfaces of the nodes. In particular, **AppTopo** has four different working modes:

- The *l2* assignment strategy automatically places all the hosts inside the same subnetwork so that no L3 routing is needed on the P4 switches side. This is the default strategy.
- The *l3* assignment strategy automatically places each host in a different subnetwork so that L3 forwarding is needed at the P4 switch level.
- The *mixed* assignment strategy automatically places all the hosts connected to the same P4 switch in the same subnetwork, which differs from one P4 switch to the other. So hosts connected to the same P4 switch do not need L3 routing to communicate whereas hosts connected to different P4 switches do.
- The *manual* assignment strategy allows the user to fully specify all the hosts and switches MAC and IP addresses. This permits the creation of custom subnetwork that do not follow the previous strategies.

These assignment strategies, however, must be handled with care since they rely on some assumptions in the topology that users must respect: multiple links among nodes must not be present and each host must be connected to exactly one switch. This means that no multihomed hosts nor links between hosts must exist in the topology: only single host-switch and switch-switch links are allowed.

Once that **AppTopo** has completed the assignment of IP and MAC addresses, **AppRunner**, which has the task of orchestrating all the modules, passes it to **P4Mininet** that builds the virtual network accordingly. Indeed, **P4Mininet** is a child of `mininet.net.Mininet` and, therefore, performs the same tasks. The only difference between parent and child classes is that the latter adds additional MTU configurations to allow sending *Jumbo frames*, i.e. Ethernet frames which exceed the standard maximum payload size of 1500 bytes. This is very handy when it comes to adding more headers to

the packets (e.g. for MPLS deploying in the network) because the user does not have to pay too much attention to their size.

After **P4Mininet** has started the network, **AppRunner** populates the data planes of each P4 switch using the information contained in commands files specified by the user. To do this, it relies on **AppController**, a class that automatizes the process and sends the user-defined command to each P4 switch using the *Thrift* client of the *Behavioral Model* repository. This commands file can be seen as a static control plane program since its rules are not changed dynamically and result in a static data plane configuration. After this step, **AppRunner** gathers information about links and nodes (e.g. IP and MAC addresses) and saves it to the disk using an instance of **Topology**, a class explicitly designed to manage topology data and provide an API to query them. **Topology**, in turn, makes use of **NetworkGraph**, a subclass of the undirected simple graph class **networkx.Graph**, to compute shortest paths in the network. **Topology** is very important since it is used to load topology data, that was saved after the network started, whenever the user has to write a control plane program needing network information to populate P4 switches data planes correctly and establish connectivity among hosts.

Finally, passing **P4Mininet** and the user-provided network configuration data to it, **AppRunner** can start **P4CLI**, the command-line client for *P4-Utils*. It is a subclass of **mininet.cli.CLI** and, in addition to the commands provided by its parent, implements specific instructions for the P4 switches. In particular, **P4CLI** explicitly and easily allows to start, stop and reboot P4 switch implemented using **P4Switch**, allowing to specify also a new P4 source and commands file (i.e. static control plane program) for its configuration.

## Chapter 3

# Improving *P4-Utils*

In this chapter, we show in detail the most important changes made to *P4-Utils* is provided. These are meant to provide a solution to the issues presented in section 1.1 and aligned with the goals shown in section 1.2. To treat these changes more consistently, the improvements are grouped into different categories depending on the nature of the modification.

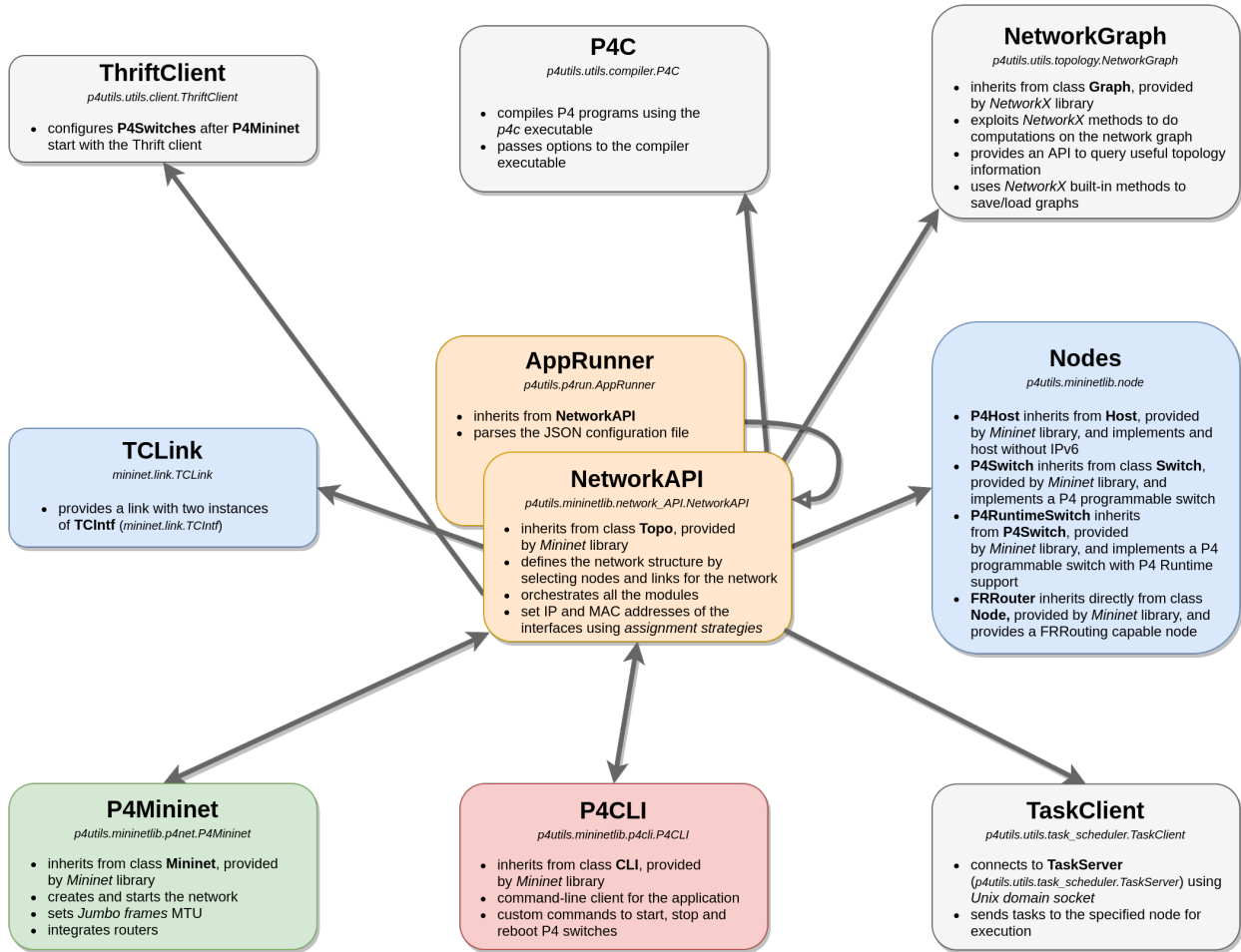


Figure 3.1: UML-like diagram of *P4-Utils* after the update

In figure 3.1, the structure and the main blocks of the updated version of *P4-Utills* are shown to give an overview of the changes.

## 3.1 Backend

Hereafter, we illustrate the changes involving parts of the application which are not directly accessible by users. In other words, the updates contained in this section regard ancillary modules that are used by *P4-Utills* to provide specific services upon request.

### 3.1.1 Migration to Python 3

As reported in section 1.2, the first step necessary for any further update was migrating *P4-Utills* to Python 3. Indeed, new libraries are natively implemented in Python 3 by now and this task is needed to make the application compatible with them. The porting has been partially automatized thanks to *2to3*, a program that reads Python 2 source code and transforms it into valid Python 3 code [1]. However, manual intervention was needed to complete the porting because of pitfalls not correctly handled by *2to3*. Indeed, some ancillary libraries slightly changed in the transition from Python 2 to Python 3 and this had to be fixed by hand.

### 3.1.2 Improved modularity

To decrease coupling among the different parts of the application and facilitate future updates, some changes in the modular structure were made and new modules were added. To start discussing them, consider the table 3.1 which is showing the way **AppRunner** and **P4CLI** used to handle P4 code compilation and static configuration of P4 switches data planes in the old version of *P4-Utills*, as already mentioned in subsection 2.5.2. Indeed, **AppRunner** and **P4CLI** have both the task of starting and configuring P4 switches: the only difference is that **AppRunner** always performs them at the network starting time, whereas **P4CLI** does it while the network is running if requested by the user.

Module	P4 source compilation	Static data plane configuration
<b>AppRunner</b>	It is handled by function <code>compile_all_p4</code> of the utility module <code>p4utils.utils.utils</code> .	It is managed by class <code>AppController</code> of the module <code>p4utils.mininetlib.appcontroller</code> .
<b>P4CLI</b>	It is managed by function <code>compile_p4_to_bmv2</code> of the utility module <code>p4utils.utils.utils</code> .	It is handled by functions <code>read_entries</code> and <code>add_entries</code> of the utility module <code>p4utils.utils.utils</code> .

Table 3.1: overlapping modules in *P4-Utills* before the update

As one may notice, even though **AppRunner** and **P4CLI** have two identical tasks, they are carried out with different methods. This results in a set of tools with overlapping purposes. To solve this, two new classes were created to create a standard way, shared across the whole *P4-Utills* platform, to perform P4 source compilation and static data plane configuration:

- **ThriftClient** is aimed at replacing **AppController**, in order to be usable also from outside **AppRunner**. Indeed, **AppController** cannot be used directly because it always attempts to configure the data planes of all the switches in the network with the user's predefined commands files: although this is useful at network starting time, it may be useless if the

network is already running and the user makes use of P4CLI to reboot and reconfigure only a single switch of the topology or provide a new commands file to it. In response to this issue, **ThriftClient** was developed as a per-switch client able to send different commands files at different times to the switch *Thrift* server.

- **P4C** is basically a Python wrapper around the compiler executable *p4c* [12]. The main benefits obtained by replacing the old compiling functions in the utility module with this new class are twofold: firstly we group the code that has the same purpose in one block widely accessible, and secondly, **P4C** can implement features aimed at reducing the number of recompilations. Indeed, as noticed by the GitHub community [3], the recompilation of P4 code, upon a switch reboot command is given in P4CLI, happens even if the P4 source has not changed. Of course, this is wasteful since compiling P4 code is quite a time-consuming operation. To overcome this, **P4C** implements a checksum mechanism that checks whether the P4 source has changed before starting the compilation of the code.

Thanks to the changes explained above, all the modules that will need access to P4 source compilation or static data plane configuration will always rely on the aforementioned classes.

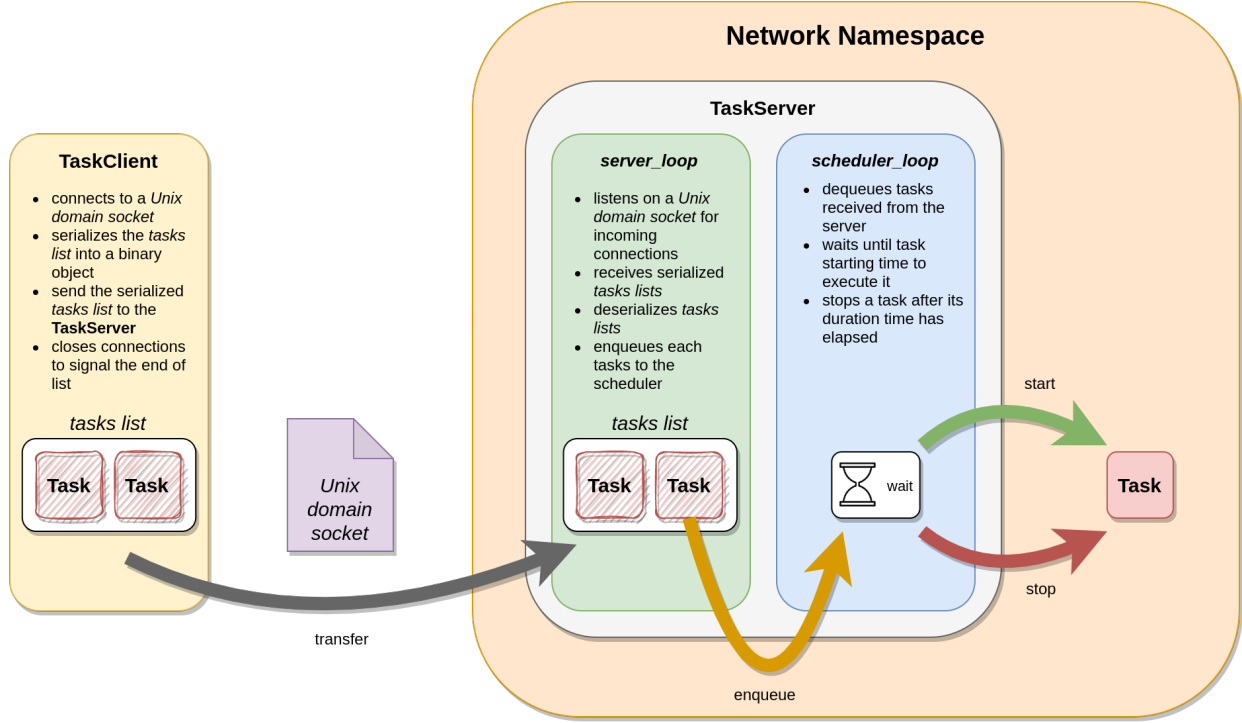
### 3.1.3 TaskServer and TaskClient

A tool often convenient for network prototyping is a traffic generator or, more generally, a task scheduler that can also generate traffic. Indeed, when dealing with multiple hosts in different network namespaces, it becomes cumbersome to run executables, commands or Python functions in each one or even some of them. As remarked in section 1.2, a solution to this is implementing a task scheduler.

The *P4-Utils* task scheduler implementation is entirely contained in the module `p4utils.utils.task_scheduler` and, as shown in the figure 3.2, it is made of two main blocks:

- **TaskServer** is a class that implements the task scheduler running in a network namespace possibly different from the one in which **TaskClient** is used. It has two main threads which run continuously:
  - The `server_loop` listens for connections on its specific *Unix domain socket*. Upon connection, it retrieves a list of tasks containing the commands, executables or Python functions to run, their starting times and, possibly, their durations. Then, it splits the list and sends each element to the `scheduler_loop`.
  - The `scheduler_loop` receives tasks from the `server_loop` and starts them in separated processes when the starting time is reached. Once that the task duration time has expired, `scheduler_loop` kills the task process. Since the duration time of the tasks can be left unspecified, endless tasks are allowed.
- **TaskClient** is a class that implements a simple client to send lists of tasks to **TaskServer** instances running in possibly different namespaces. When a list of tasks is to be forwarded to a task server, it is passed to an instance of **TaskClient** configured to use the server *Unix domain socket*. **TaskClient** establishes a connection and starts sending the tasks list. Eventually, when all the data are sent, the client closes the connection signalling the end of the tasks list to the server. If more tasks have to be sent, one connection for each new list must be created.



Figure 3.2: *P4-Utills* task scheduling process

### 3.1.4 P4Mininet and FRRouter

As remarked in section 1.2, one important feature to add to *P4-Utills* is the possibility of adding routers as nodes in the virtual network. Indeed, this new tool can be very useful for emulating more complex networks and analysing better how different types of nodes (hosts, P4 switches and routers) behave when communicating with each other. In addition, a whole set of routing protocols (e.g. OSPF, BGP, LDP) can be made available, adding more options for the design and the functioning of the network.

In response to this need, since *Mininet* does not provide any router node natively, the new **FRRouter** node was implemented: a node exploiting the *FRRouting* Internet protocol suite [4] to route packets among its interfaces. **FRRouter** was added to the module `p4utils.mininetlib.node`, which contains all the custom types of nodes used by *P4-Utills*. This was possible thanks to the fact that *Mininet* built-in nodes can be easily extended with customized options: indeed, the **FRRouter** class inherits directly from the class **Node** provided by *Mininet*. To integrate this new node in the structure of *P4-Utills* and its *Mininet* backbone, further small changes were applied to **P4Mininet**, the network class used by *P4-Utills*.

As a consequence, with the updated version of *P4-Utills*, the user can instantiate multiple **FRRouter** nodes in the network, specifying which routing services must be activated on the targets and their initial configuration. Moreover, an **FRRouter** node can always be accessed and reconfigured while the network is running by using VTYSH, a common shell for all *FRRouting* daemons [22].

Finally, for what concerns the other node types that were already present in *P4-Utills* (i.e. **P4Host**, **P4Switch**, **P4RuntimeSwitch**), they were affected by small updates aimed at improving their integration with the rest of the application.

## 3.2 Frontend

This section is focused on all those components of the application which are in direct interaction with the user. Indeed, all the updates reported here involve parts of *P4-Utils* that are meant to be configured or queried by the user.

### 3.2.1 NetworkAPI and AppRunner

According to section 1.2, the main update in the way *P4-Utils* interacts with the user is the introduction of the **NetworkAPI**, which allows defining the network topology as well as nodes and links properties. This API allows defining and starting the network by using a Python script instead of using the JSON configuration file. This new programmatic way of creating networks can be useful to handle large topologies. This new class is placed in the module `p4utils.mininetlib.network_API` and, besides providing an API for the network definition, it also orchestrates all the other ancillary modules of *P4-Utils*.

**NetworkAPI** handles P4 source compilation and static data plane configuration thanks to the fact that it has access to the classes **P4C** and **ThriftClient**, covered in subsection 3.1.2. In other words, the new **NetworkAPI** replaces what was managed by **AppRunner** in the old version of the application.

In addition, **NetworkAPI** can use **TaskClient** instances to communicate with **TaskServer** processes running in other namespaces to distribute tasks to each node. Indeed, the new task scheduler feature can be exploited by the user to run multiple processes in different nodes at specified times. To activate this, the user creates a file containing all the tasks information (e.g. node, command, starting time, duration). This is then parsed by **NetworkAPI** and sent to the involved servers using **TaskClient** instances.

In order to integrate better *Mininet* into *P4-Utils*, **NetworkAPI** is defined as a subclass of `mininet.topo.Topo`. In this way, a configured instance of **NetworkAPI** can directly pass itself to an instance of **P4Mininet** to start the application. This is shown in figure 3.1 where **NetworkAPI** and **P4Mininet** are connected by a bidirectional association: this means that, during operations, the instance of **NetworkAPI** contains a reference to the instance of **P4Mininet** and vice versa.

The API, in particular, can be divided into the following categories of methods:

- *External modules management* methods allow specifying non-default ancillary modules and classes (e.g. those needed for P4 source compilations, static data plane configuration, etc.) so that *P4-Utils* can use them.
- *Network management* methods collect a set of instruction that globally affect the network. For example, the user can specify the log level, activate or deactivate packets dumping to disk, set or unset static ARP tables for hosts, enable or disable the application client after the network starts.
- *Nodes management* methods provide all the necessary tools to add, remove and update the nodes and their properties such as port numbers, interface names and default gateway.
- *Links management* methods provide all the necessary tools to add, remove and update the links and their properties such as bandwidth, delay, loss, IP and MAC addresses of the interfaces.
- *Topology querying* methods give the user an easy way to iterate over links or nodes, to check the node type and check whether two nodes are connected by a link. These functions may be useful when it comes to programmatically building large topologies.

- *Assignment strategies* methods are meant to automatize the way IP and MAC addresses are assigned without any user's intervention. These assignment strategies<sup>1</sup> did not change from the previous version of *P4-Utils* so they do not support routers nor multiple parallel links.

For compatibility reasons, the **AppRunner** is kept as a legacy network definition method via JSON file. However, to make the way **AppRunner** parse data from the file and instantiate the network more consistent, the names of some fields were changed so the old JSON network configurations files have to be updated, even though these changes are very few. In the new version of the platform, **AppRunner** is now a subclass of **NetworkAPI** and has the only additional task of parsing the configuration file. Afterwards, it simply uses the methods provided by its parent to create and start the network and orchestrate the ancillary modules. As in the previous version of *P4-Utils*, **AppRunner** is placed in the module `p4utils.p4run`.

### 3.2.2 NetworkGraph

Another important improvement is the new topology querying API provided by **NetworkGraph**. According to figure 2.3, in the past version of *P4-Utils*, this API was implemented by **Topology** which, in turn, used **NetworkGraph** to carry out shortest paths computations. Now, as shown in figure 3.1, **NetworkGraph** is the only responsible for this task. Instead of having all the methods related to topology querying dispersed across different classes, they are now grouped in one.

In addition, also loading and saving topology data are now cleaner processes. Indeed, *Mininet* has a built-in method to convert the topology information contained in an instance of **Topo** into a custom graph instance that inherits from a *NetworkX* class [8]. Therefore, since **NetworkAPI** is a subclass of **Topo** and **NetworkGraph** inherits from `networkx.Graph`<sup>2</sup>, the information contained in an instance of **NetworkAPI** can be directly transferred to an instance of **NetworkGraph**. Finally, *NetworkX* has easy methods to save and load generic graphs instances, so any **NetworkGraph** object can be easily dumped to disk and retrieved later for querying.

Before, all the aforementioned procedures were handled by the internal methods of **Topology** which worked only for networks containing hosts and switches with no multiple parallel links. With the new approach, these limitations have been partially removed. Indeed, **NetworkGraph** can store new types of nodes without problems, although no methods may be available to query them. As a final remark, **NetworkGraph** still cannot handle multiple parallel links among nodes because it inherits from `networkx.Graph`, which handles simple graphs only. A new **NetworkGraph** class inheriting from `networkx.MultiGraph` would solve this issue.

### 3.2.3 P4CLI

To keep up with the newly introduced features, also **P4CLI** needed an update. If we consider figure 2.3, it can be noticed that **P4CLI** had access to **P4Mininet** while was accessed by **AppRunner**. Now, as shown in figure 3.1, **P4CLI** has access to and it is accessed by **NetworkAPI**, which is the new modules orchestrator. This is very beneficial since **P4CLI** now can indirectly use all the modules to which **NetworkAPI** has access: in this way, the client not only has access to **P4Mininet** but also to **P4C** and **ThriftClient** which, according to subsection 3.1.2, are now the only responsible for P4 source compilation and static data plane configuration.

In addition, new commands to send tasks to the nodes were introduced in the new **P4CLI**. These commands allow the user to start processes in the nodes while the network is running using the

---

<sup>1</sup>See subsection 2.5.2 for further details.

<sup>2</sup>See section 2.3 for further details.

command-line provided by the client. The implementation of this new feature is simple: P4CLI has indirect access to `TaskClient` through `NetworkAPI` and exploits it to send new tasks to the servers running in the namespaces.

### 3.3 P4Runtime

Beside updating of the main code of P4-Utills, a new controller API was written for P4 Switches supporting *P4Runtime*: `p4utils.utils.sswitch_p4runtime_API`. This *Runtime* API was developed and tested explicitly for `simple_switch_grpc` [17] targets but, thanks to the multi-target nature of P4Runtime, should be usable with little or no changes with other devices as well. This new API provides a set of the most used commands when it comes to configuring P4 switches. In particular, the features currently supported are the following:

- *Tables* methods allow the user to insert, modify and delete entries in the P4 tables of the switch. These methods can also configure resources directly attached to the entries such as *Direct Meters* and *Direct Counters*.
- *Meters* and *Direct Meters* methods allow specifying the rate at which packets are sent by the switch.
- *Counter* and *Direct Counters* methods allow to read and write the values of the counters in the switch.
- *Multicast Groups* methods permit to forward a packet simultaneously to multiple interfaces.
- *Clone Sessions* methods permit to clone a packet and send it to an interface. This happens for example when want to send packets to an external analysing device.
- *Digests* methods implement an efficient way to send small pieces of information to a controller via the gRPC connection to the *P4Runtime* server.

One important feature missing is access to P4 registers. Indeed, although the P4Runtime specification includes registers, these are not yet supported by *PI* [14], the gRPC server implemented in the software switch `simple_switch_grpc`, and can only be accessed through the *Thrift* server. This is a well known and still open issue of *PI* [18].

All the methods provided by the *P4Runtime* API are based on the classes and methods of the module `p4utils.utils.p4runtime_API`, which is a modified version of the *p4runtime-shell* application provided by the P4 community [13]. Indeed, `p4runtime_API` compared to *p4runtime-shell*, does not have any client feature and implements an API that can connect to a *P4Runtime* capable switch. To manage more than one switch, multiple instances of `p4runtime_API` can be used, each one connecting to a specific switch.

### 3.4 P4 Learning

To catch up with the aforementioned updates, some changes were made to the exercises contained in the *P4 Learning* repository. According to the list in subsection 1.2, the following tasks were done:

- All the exercises scripts were migrated to Python 3.

- According to the changes made to the **AppRunner**, covered in subsection 3.2.1, the JSON network configuration files of the exercises were updated so that they can now be executed with the legacy method.
- A *P4Runtime* version of the following exercises, chosen among the most important ones, was created to provide usage examples for the new *P4Runtime* API:
  - *Repeater*
  - *L2 Basic forwarding*
  - *L2 Flooding*
  - *L2 Learning*
  - *MPLS*
  - *RSVP*
  - *Simple routing*
- All the exercises are provided with the newly introduced network configuration script<sup>3</sup> that can be used in place of the legacy JSON file.

It is important to remark that the examples of the *P4 Learning* repository were not updated and currently may not be used with the new version of the application.

---

<sup>3</sup>See section 3.2.1 for further details.

## Chapter 4

# Using the new *P4-Utils*

In this chapter, we provide usage examples of the new version of *P4-Utils* and comparisons with the old one.

### 4.1 Frontend changes

Let us consider the *Thrift* version exercise *L2 Learning* from the *P4 Learning* repository. In the table 4.1, we do a comparison between the old and the new JSON network configuration files to point out the most important differences:

- The new version allows skipping modules declaration so that only non-default modules have to be specified. This feature was already present in the old version of the application, even though it was not exploited in the exercises JSON configuration files. Indeed, in the old JSON file, the lines between 10 and 25 are meant to specify the external components in use.
- The old fields "compiler" and "options" were removed in the new version of *P4-Utils*. Indeed they were used to configure the compiler *p4c*. As shown in subsection 3.1.2, the newly introduced module **P4C** now carries out the task of compilation. Therefore, the new configurations for the compiler are passed through the field "compiler\_module".
- The old field "program" was replaced by "p4\_src" to simplify the instantiation of **P4Switch** and **P4RuntimeSwitch** nodes.
- All the default options concerning the nodes and the links of the topology are now put inside the "default" field of "topology": indeed we can see that line 28 of the old file is now placed in line 9 of the new configuration. This makes it easier parsing and setting the options in **AppRunner**.

In conclusion, the main structure of the configuration files has not changed a lot. As shown in the JSON snippets, the user can specify the links, hosts and switches by simply listing them and can enable or disable logging and .pcap files dumping in the same way. These operations have not changed from the old to the new version of *P4-Utils*.

<pre> 1 { 2   "program": "p4src/l2_learning_digest.p4", 3   "switch": "simple_switch", 4   "compiler": "p4c", 5   "options": "--target bmv2 --arch vmodel --std     ↪ p4-16", 6   "switch_cli": "simple_switch_CLI", 7   "cli": true, 8   "pcap_dump": true, 9   "enable_log": true, 10  "topo_module": { 11    "file_path": "", 12    "module_name": "p4utils.mininetlib.apptopo", 13    "object_name": "AppTopoStrategies" 14  }, 15  "controller_module": null, 16  "topodb_module": { 17    "file_path": "", 18    "module_name": "p4utils.utils.topology", 19    "object_name": "Topology" 20  }, 21  "mininet_module": { 22    "file_path": "", 23    "module_name": "p4utils.mininetlib.p4net", 24    "object_name": "P4Mininet" 25  }, 26  "topology": { 27    "assignment_strategy": "l2", 28    "auto_arp_tables": false, 29    "links": [{"h1", "s1"}, {"h2", "s1"}, {"h3",     ↪ "s1"}, {"h4", "s1"}], 30    "hosts": { 31      "h1": {}, 32      "h2": {}, 33      "h3": {}, 34      "h4": {} 35    }, 36    "switches": { 37      "s1": {} 38    } 39  } 40 }</pre>	<pre> 1 { 2   "p4_src": "p4src/l2_learning_digest.p4", 3   "cli": true, 4   "pcap_dump": true, 5   "enable_log": true, 6   "topology": { 7     "assignment_strategy": "l2", 8     "default": { 9       "auto_arp_tables": false 10    }, 11    "links": [{"h1", "s1"}, {"h2", "s1"}, {"h3",     ↪ "s1"}, {"h4", "s1"}], 12    "hosts": { 13      "h1": {}, 14      "h2": {}, 15      "h3": {}, 16      "h4": {} 17    }, 18    "switches": { 19      "s1": {} 20    } 21  } 22 }</pre>
---	--

Table 4.1: comparison between old (left) and new (right) configuration files

In table 4.2, we do a comparison between the two possible ways of setting up a network: the JSON configuration file and the Python configuration script, which makes use of `NetworkAPI`. Indeed, thanks to the new API, the user can simply execute the Python script to start the network. In this case, we use as an example the *P4Runtime* version of the exercise *L2 Learning*. The two snippets do the same things, so let us focus on how they define the network:

- The topology definition, done in lines 18-33 in the JSON file, is carried out with intuitive methods in lines 11-20 in the Python file.
- Global network and applications options have corresponding names among the two configuration methods so that no confusion is generated in the user.
- `P4RuntimeSwitch` nodes are non-default node so the JSON file has to import the module (as shown in lines 6-10) to use it for the switches instances. On the other hand, the Python script is much more user-friendly since provides a simple method to directly instantiate `P4RuntimeSwitch` (as shown in line 11).
- To use `P4RuntimeSwitch` instances, it is necessary to configure their gRPC server with a file generated by the compiler module `P4C`. However, this is not a default option, so it must be explicitly requested by the user by setting to `True` the parameter `p4rt` of `P4C`. The JSON configuration file performs this in lines 11-17, while the Python script does it with line 7.

<pre> 1  { 2    "p4_src": "p4src/l2_learning_digest.p4", 3    "cli": true, 4    "pcap_dump": true, 5    "enable_log": true, 6    "switch_node": 7    { 8      "module_name": "p4utils.mininetlib.node", 9      "object_name": "P4RuntimeSwitch" 10   }, 11   "compiler_module": 12   { 13     "options": 14     { 15       "p4rt": true 16     }, 17   }, 18   "topology": { 19     "assignment_strategy": "l2", 20     "default": { 21       "auto_arp_tables": false 22     }, 23     "links": [ ["h1", "s1"], ["h2", "s1"], ["h3", 24               ↪ "s1"], ["h4", "s1"] ], 25     "hosts": { 26       "h1": {}, 27       "h2": {}, 28       "h3": {}, 29       "h4": {} 30     }, 31     "switches": { 32       "s1": {} 33     } 34   } </pre>	<pre> 1  from p4utils.mininetlib.network_API import 2      ↪ NetworkAPI 3 4  net = NetworkAPI() 5 6  # Network general options 7  net.setLogLevel('info') 8  net.setCompiler(p4rt=True) 9  net.disableArpTables() 10 11 # Network definition 12 net.addP4RuntimeSwitch('s1') 13 net.setP4Source('s1', './p4src/l2_learning_digest.p4') 14 net.addHost('h1') 15 net.addHost('h2') 16 net.addHost('h3') 17 net.addHost('h4') 18 net.addLink('s1', 'h1') 19 net.addLink('s1', 'h2') 20 net.addLink('s1', 'h3') 21 net.addLink('s1', 'h4') 22 23 # Assignment strategy 24 net.l2() 25 26 # Nodes general options 27 net.enablePcapDumpAll() 28 net.enableLogAll() 29 net.enableCli() 30 31 # Start the network 32 net.startNetwork() </pre>
--	---

Table 4.2: comparison between JSON (left) and Python (right) configuration files

## 4.2 New features in action

In this section, we present a network setup that makes use of the most important features introduced with the update of *P4-Utils*. The network topology considered is shown in figure 4.1. This example has been defined and tested thanks to network configuration files exhibited in appendix A.

There topology is divided into two *Autonomous Systems*, each responsible for a prefix: 1.0.0.0/8 is assigned to AS 1, whereas 2.0.0.0/8 is bound to AS 2. AS 1 has four routers connected in such a way that R4 is a border router connected to the other AS, R3 is at the core of the network, R1 and R2 are connected to two switch S1 and S2 respectively. Each switch is, in turn, connected to two switches: H1 and H4 are connected to switch S1, H2 and H5 are linked to switch S2. On the other hand, AS 2 has only one router R5 connected to both the AS 1 and the switch S3. S3 is, in turn, connected to both H3 and H6. Each link has a maximum bandwidth capacity of 5 Mbps.

The goal of this example is to provide connectivity so that each host can reach any other one. In particular, the following technologies will be used:

- In **FRRouter** nodes, OSPF, LDP and BGP will be enabled so that connectivity among ASes and within the same AS is ensured. In particular, the usage of LDP allows having a BGP-free network core, so R3 does not need to have BGP configured. The OSPF weights of the links are the red numbers shown in figure 4.1.
- In **P4Switch** nodes, L2 learning is implemented. The switch will associate the source MAC address of each inbound packet with its ingress interface and, later on, it will use this information for forwarding.



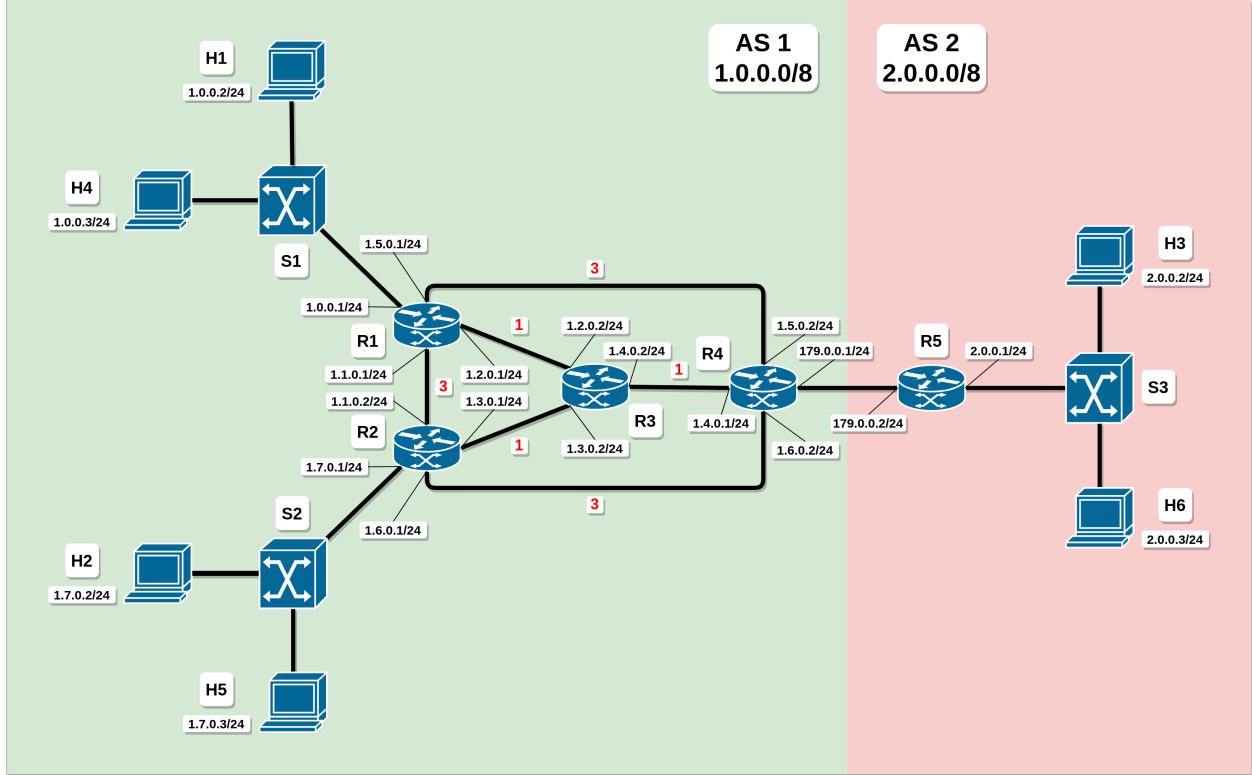


Figure 4.1: FRRouting example topology

We want to test the connectivity in the following cases.

- To test connectivity among nodes in the same subnetwork, we send a UDP flow from H1 to H4.
- To test connectivity among nodes in different subnetworks of the same AS, we send a UDP flow from H4 to H2.
- To test connectivity among nodes in different ASes, we send a UDP flow from H2 to H3.

The task scheduler can be very useful to start the flows and to monitor the bandwidth at some node interfaces to check if the traffic makes it to the destination. Hereafter, we analyse the content of `tasks.txt`, the file that contains all the tasks of the nodes. This file is specified in the network configuration files (on line 6 for the JSON file A.1 and on line 85 for the Python script A.2) so that can be parsed and used. As an additional remark, the monitoring is done with `p4utils.utils.monitor`, a module that collects information about the receiving and transmitting rates of an interface and then dumps it to a `.csv` file. This `.csv` file can be then used to create a time graph of the interface data rates.

In the following lines, we define a UDP flow of 10 Mbps from H1 to H4 to test connectivity for hosts in the same subnetwork. The second line activates the sender, whereas the third one enables the receiver. We can see that we start the flow 15 seconds after the network and it lasts 30 seconds.

```
# Communication in the same subnet
h1 15 30 send_udp_flow --dst 1.0.0.3 --sport 5000 --dport 5051 --rate 10M
h4 15 30 recv_udp_flow --dport 5051
```

The following lines enable the monitoring on the interfaces `s1-eth1` (the interface of S1 facing H1) and `s1-eth2` (the interface of S1 facing H2). We've been monitoring for 100 seconds since the network started.

```
# Monitor bandwidth
s1 0 0 "python -m p4utils.utils.monitor -i s1-eth1 -d 100 s1-eth1.csv"
s1 0 0 "python -m p4utils.utils.monitor -i s1-eth2 -d 100 s1-eth2.csv"
```

The following lines enable the other flows and testing following the same logic reported above and enables two additional flows with related monitoring:

- a UDP flow of 10 Mbps from H4 to H2 starting 45 seconds after the network (to allow the convergence of OSPF) and lasting 30 seconds to test inter-AS connectivity,
- a UDP flow of 10 Mbps from H2 to H3 starting 60 seconds after the network (to allow the convergence of BGP) and lasting 30 seconds to test inter-AS connectivity.

```
# Communication in the same AS
h4 45 30 send_udp_flow --dst 1.7.0.2 --sport 5001 --dport 5051 --rate 10M
h2 45 30 recv_udp_flow --dport 5051
# Monitor bandwidth
s2 0 0 "python -m p4utils.utils.monitor -i s2-eth1 -d 100 s2-eth1.csv"

# Communication among ASes
h2 60 30 send_udp_flow --dst 2.0.0.2 --sport 5002 --dport 5051 --rate 10M
h3 60 30 recv_udp_flow --dport 5051
# Monitor bandwidth
s3 0 0 "python -m p4utils.utils.monitor -i s3-eth1 -d 100 s3-eth1.csv"
```

Since all the flows were configured to have a data rate of 10 Mbps and the links of the network have a maximum capacity of 5 Mbps, we expect the actual flow rate to be 5 Mbps. This would show that *P4-Utils* can correctly limit the bandwidth.

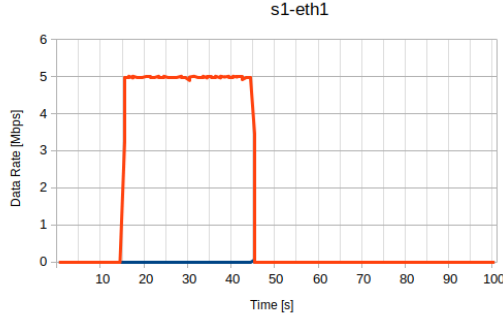
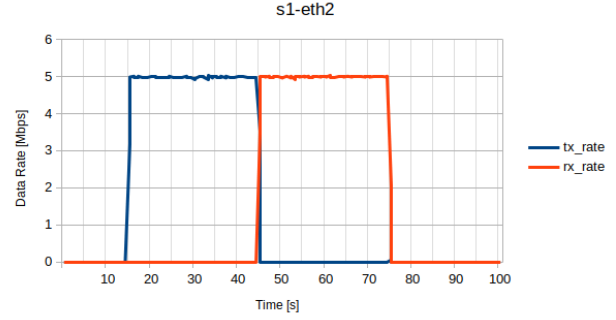
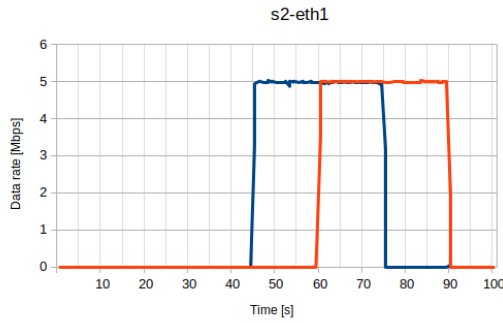
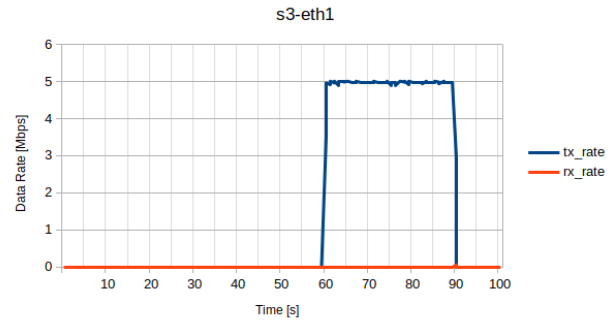
(a) Traffic on interface **s1-eth1**(b) Traffic on interface **s1-eth2**(c) Traffic on interface **s2-eth1**(d) Traffic on interface **s3-eth1**

Figure 4.2: traffic at switches S1, S2 and S3

The figures 4.2a and 4.2b show that the traffic is correctly sent from H1 to H4. Indeed 15 seconds after the network started, the traffic is originated by H1 and arrives at interface **s1-eth1** (red line in figure 4.2a) of the switch S1. It is then forwarded to H4 via **s1-eth2** (blue line in figure 4.2b). This shows that there is connectivity for hosts in the same subnetwork.

The figures 4.2b and 4.2c show that the flow originated by H4 is correctly received by H2. Indeed 45 seconds after the network started, the traffic is sent by H4 and arrives at interface **s1-eth2** (red line in figure 4.2b). After being routed across the network of AS 1, the traffic reaches the switch S2 and it is forwarded to H4 via **s2-eth1** (blue line in figure 4.2c). This proves that there is connectivity among hosts in different subnetworks of the same AS.

The figures 4.2c and 4.2d show that the traffic is correctly sent from H2 to H3. Indeed 60 seconds after the network started, the traffic sent by H2 arrives at interface **s2-eth1** (red line in figure 4.2c). After being routed from AS 1 to AS 2, the traffic reaches the switch S3 and it is forwarded to H3 via **s3-eth1** (blue line in figure 4.2d). This proves that there is connectivity among hosts in different ASes.

As a final remark, we notice that all the flows have the expected duration and none of them reaches the bandwidth of 10 Mbps. In this way, we tested the ability of P4-Utils to correctly limit the bandwidth.

## Chapter 5

# Conclusion and Future Work

In this thesis, we discussed several different updates aimed at improving *P4-Utils*. Thanks to the migration to Python 3, the application is now ready for further developments. Moreover, additional features like the **NetworkAPI**, the **TaskClient** and the **TaskServer** allows the user to manage the whole platform with ease. Finally, the newly introduced *P4Runtime* API is important to test the most recent specifications released by the P4 community and the **FRRRouter** node is now available to extend the capabilities of *P4-Utils*.

Even though in this thesis we improved and added several new features, *P4-Utils* has still room for some improvements. In particular, one could improve:

- the assignment strategies such that they support **FRRRouter** nodes and multiple parallel links between two nodes, as noted in subsection 3.2.1;
- The class `p4utils.utils.topology.NetworkGraph`, which provides a topology querying API, does not support multiple parallel links and still lacks of methods to query possible **FRRRouter** nodes informations, as remarked in subsection 3.2.2.

For what concerns the *P4-Learning* repository, there is still room for improvements:

- A *P4Runtime* version of all the exercises can be done once that the *PI* issue affecting registers has been fixed [18].
- According to section 3.4, the examples contained in *P4-Learning* are not compliant with the new version of *P4-Utils* and, therefore, need to be updated.

Beside these issue, a new update can improve the way **FRRRouter** nodes interacts with the rest of the application integrating them better. Finally, more node types can be added to make the network simulations more complex and useful.

# Bibliography

- [1] 2to3 - automated python 2 to 3 code translation — python 3.9.5 documentation. <https://docs.python.org/3/library/2to3.html>. Accessed on 28-05-2021.
- [2] Behavioral model (bmv2). <https://github.com/p4lang/behavioral-model>. Accessed on 20-05-2021.
- [3] Frequent re-compiling · issue 15 · nsg-ethz/p4-utils. <https://github.com/nsg-ethz/p4-utils/issues/15>. Accessed on 28-05-2021.
- [4] Frouting. <https://frouting.org/>. Accessed on 23-05-2021.
- [5] Graph—undirected graphs with self loops — networkx 2.5 documentation. <https://networkx.org/documentation/stable/reference/classes/graph.html>. Accessed on 26-05-2021.
- [6] Mininet: An instant virtual network on your laptop (or other pc). <http://mininet.org/>. Accessed on 20-05-2021.
- [7] Multigraph—undirected graphs with self loops and parallel edges — networkx 2.5 documentation. <https://networkx.org/documentation/stable/reference/classes/multigraph.html>. Accessed on 26-05-2021.
- [8] Networkx — networkx documentation. <https://networkx.org/>. Accessed on 24-05-2021.
- [9] P4 learning. <https://github.com/nsg-ethz/p4-learning>. Accessed on 20-05-2021.
- [10] P4-utils. <https://github.com/nsg-ethz/p4-utils>. Accessed on 20-05-2021.
- [11] p4app. <https://github.com/p4lang/p4app>. Accessed on 20-05-2021.
- [12] p4c. <https://github.com/p4lang/p4c>. Accessed on 26-05-2021.
- [13] p4lang/p4runtime-shell: An interactive python shell for p4runtime. <https://github.com/p4lang/p4runtime-shell>. Accessed on 30-05-2021.
- [14] p4lang/pi: An implementation framework for a p4runtime server. <https://github.com/p4lang/PI>. Accessed on 30-05-2021.
- [15] Pep 373 – python 2.7 release schedule. <https://www.python.org/dev/peps/pep-0373/>. Accessed on 20-05-2021.
- [16] Simple switch. [https://github.com/p4lang/behavioral-model/tree/main/targets/simple\\_switch](https://github.com/p4lang/behavioral-model/tree/main/targets/simple_switch). Accessed on 26-05-2021.

- [17] Simple switch grpc. [https://github.com/p4lang/behavioral-model/tree/main/targets/simple\\_switch\\_grpc](https://github.com/p4lang/behavioral-model/tree/main/targets/simple_switch_grpc). Accessed on 26-05-2021.
- [18] Support to read and write register by pi · issue 376 · p4lang/pi. <https://github.com/p4lang/PI/issues/376>. Accessed on 30-05-2021.
- [19] tc(8) - linux manual page. <https://man7.org/linux/man-pages/man8/tc.8.html>. Accessed on 23-05-2021.
- [20] Thrift client. [https://github.com/p4lang/behavioral-model/blob/main/tools/runtime\\_CLI.py](https://github.com/p4lang/behavioral-model/blob/main/tools/runtime_CLI.py). Accessed on 26-05-2021.
- [21] unix(7) - linux manual page. <https://www.man7.org/linux/man-pages/man7/unix.7.html>. Accessed on 27-05-2021.
- [22] Vtysh — frr latest documentation. <http://docs.frrouting.org/projects/dev-guide/en/latest/vtysh.html>. Accessed on 29-05-2021.
- [23] P4runtime specification, version 1.3.0. <https://p4lang.github.io/p4runtime/spec/v1.3.0/P4Runtime-Spec.html>, Dec 2020. Accessed on 20-05-2021.
- [24] namespaces(7) - linux manual page. <https://man7.org/linux/man-pages/man7/namespaces.7.html>, Mar 2021. Accessed on 20-05-2021.
- [25] P4 – language consortium. <https://p4.org/>, 2021. Accessed on 20-05-2021.
- [26] veth(4) - linux manual page. <https://man7.org/linux/man-pages/man4/veth.4.html>, Mar 2021. Accessed on 20-05-2021.

# Appendix A

## FRRouting example configuration files

```
1  {
2      "p4_src": "l2_learning_digest.p4",
3      "cli": true,
4      "pcap_dump": true,
5      "enable_log": true,
6      "tasks_file": "tasks.txt",
7      "exec_scripts": [
8          {"cmd": "python l2_learning_controller.py s1 digest &", "reboot_run": true},
9          {"cmd": "python l2_learning_controller.py s2 digest &", "reboot_run": true},
10         {"cmd": "python l2_learning_controller.py s3 digest &", "reboot_run": true}
11     ],
12     "topology": {
13         "default": {
14             "auto_arp_tables": false,
15             "auto_gw_arp": false,
16             "bw": 5
17         },
18         "links": [
19             ["h1", "s1", {"params1": {"ip": "1.0.0.2/24"}}],
20             ["h4", "s1", {"params1": {"ip": "1.0.0.3/24"}}],
21             ["h2", "s2", {"params1": {"ip": "1.7.0.2/24"}}],
22             ["h5", "s2", {"params1": {"ip": "1.7.0.3/24"}}],
23             ["h3", "s3", {"params1": {"ip": "2.0.0.2/24"}}],
24             ["h6", "s3", {"params1": {"ip": "2.0.0.3/24"}}],
25             ["s1", "r1", {"intfName2": "port_S1"}],
26             ["s2", "r2", {"intfName2": "port_S2"}],
27             ["s3", "r5", {"intfName2": "port_S3"}],
28             ["r1", "r2", {"intfName1": "port_R2", "intfName2": "port_R1"}],
29             ["r1", "r3", {"intfName1": "port_R3", "intfName2": "port_R1"}],
30             ["r1", "r4", {"intfName1": "port_R4", "intfName2": "port_R1"}],
31             ["r2", "r3", {"intfName1": "port_R3", "intfName2": "port_R2"}],
32             ["r2", "r4", {"intfName1": "port_R4", "intfName2": "port_R2"}],
33             ["r3", "r4", {"intfName1": "port_R4", "intfName2": "port_R3"}],
34             ["r4", "r5", {"intfName1": "port_AS2", "intfName2": "port_AS1"}]
35         ],
36         "hosts": {
37             "h1": {"defaultRoute": "via 1.0.0.1"},
38             "h2": {"defaultRoute": "via 1.7.0.1"},
39             "h3": {"defaultRoute": "via 2.0.0.1"},
40             "h4": {"defaultRoute": "via 1.0.0.1"},
41             "h5": {"defaultRoute": "via 1.7.0.1"},
42             "h6": {"defaultRoute": "via 2.0.0.1"}
43         },
44         "switches": {
45             "s1": {},
46             "s2": {},
47             "s3": {}
48         },
49         "routers": {
50             "r1": {"int_conf": "./routers/r1.conf", "ldpd": "True"},
51             "r2": {"int_conf": "./routers/r2.conf", "ldpd": "True"},
52             "r3": {"int_conf": "./routers/r3.conf", "ldpd": "True"},
53             "r4": {"int_conf": "./routers/r4.conf", "ldpd": "True"},
54             "r5": {"int_conf": "./routers/r5.conf", "ldpd": "True"}
55         }
56     }
57 }
```

Listing A.1: JSON configuration file

```

1  from p4utils.mininetlib.network_API import NetworkAPI
2
3  net = NetworkAPI()
4
5  # Network general options
6  net.setLogLevel('info')
7  net.execScript('python l2_learning_controller.py s1 digest &', reboot=True)
8  net.execScript('python l2_learning_controller.py s2 digest &', reboot=True)
9  net.execScript('python l2_learning_controller.py s3 digest &', reboot=True)
10 net.enableCli()
11 net.disableArpTables()
12 net.disableGwArp()
13
14 # Network definition
15 # Switches
16 # AS 1
17 net.addP4Switch('s1')
18 net.addP4Switch('s2')
19 # AS 2
20 net.addP4Switch('s3')
21 net.setP4SourceAll('l2_learning_digest.p4')
22
23 # Hosts
24 # AS 1
25 net.addHost('h1')
26 net.setDefaultRoute('h1', "1.0.0.1")
27 net.addHost('h4')
28 net.setDefaultRoute('h4', "1.0.0.1")
29 net.addHost('h2')
30 net.setDefaultRoute('h2', "1.7.0.1")
31 net.addHost('h5')
32 net.setDefaultRoute('h5', '1.7.0.1')
33
34 # AS 2
35 net.addHost('h3')
36 net.setDefaultRoute('h3', "2.0.0.1")
37 net.addHost('h6')
38 net.setDefaultRoute('h6', '2.0.0.1')
39
40 # Routers
41 # AS 1
42 net.addRouter('r1', int_conf='./routers/r1.conf', ldpd=True)
43 net.addRouter('r2', int_conf='./routers/r2.conf', ldpd=True)
44 net.addRouter('r3', int_conf='./routers/r3.conf', ldpd=True)
45 net.addRouter('r4', int_conf='./routers/r4.conf', ldpd=True)
46
47 # AS 2
48 net.addRouter('r5', int_conf='./routers/r5.conf', ldpd=True)
49
50 # Links
51 # AS 1
52 net.addLink('h1', 's1')
53 net.setIntfIp('h1', 's1', '1.0.0.2/24')
54 net.addLink('h4', 's1')
55 net.setIntfIp('h4', 's1', '1.0.0.3/24')
56 net.addLink('s1', 'r1', intfName2='port_S1')
57
58 net.addLink('h2', 's2')
59 net.setIntfIp('h2', 's2', '1.7.0.2/24')
60 net.addLink('h5', 's2')
61 net.setIntfIp('h5', 's2', '1.7.0.3/24')
62 net.addLink('s2', 'r2', intfName2='port_S2')
63
64 net.addLink('r1', 'r2', intfName1='port_R2', intfName2='port_R1')
65 net.addLink('r1', 'r3', intfName1='port_R3', intfName2='port_R1')
66 net.addLink('r1', 'r4', intfName1='port_R4', intfName2='port_R1')
67 net.addLink('r2', 'r3', intfName1='port_R3', intfName2='port_R2')
68 net.addLink('r2', 'r4', intfName1='port_R4', intfName2='port_R2')
69 net.addLink('r3', 'r4', intfName1='port_R4', intfName2='port_R3')
70
71 # Inter-AS
72 net.addLink('r4', 'r5', intfName1='port_AS2', intfName2='port_AS1')
73
74 # AS 2
75 net.addLink('h3', 's3')
76 net.setIntfIp('h3', 's3', '2.0.0.2/24')
77 net.addLink('h6', 's3')
78 net.setIntfIp('h6', 's3', '2.0.0.3/24')
79 net.addLink('s3', 'r5', intfName2='port_S3')
80
81 # Links general options
82 net.setBwAll(5)
83
84 # Nodes general options
85 net.addTaskFile('tasks.txt')
86 net.enablePcapDumpAll()
87 net.enableLogAll()
88
89 # Start the network
90 net.startNetwork()

```

Listing A.2: Python configuration script