electronics

MDPI

Article

Enhancing Programmability in Next-Generation Networks: An Innovative Simulation Approach

Jesús Calle-Cancho *, Cristian Cruz-Carrasco, David Cortés-Polo, Jesús Galeano-Brajones and Javier Carmona-Murillo

Department of Computing and Telematics Engineering, University of Extremadura, 10003 Cáceres, Spain; ccruzcar@alumnos.unex.es (C.C.-C.); dcorpol@unex.es (D.C.-P.); jgaleanobra@unex.es (J.G.-B.); jcarmur@unex.es (J.C.-M.)

* Correspondence: jesuscalle@unex.es

Abstract: With the advent of next-generation networks, it is crucial to persist in the research and development of key enabling technologies such as software-defined networking (SDN). This involves assessing prospective network deployments, mechanisms, or ideas; an undertaking performed by both network operators and academia to assess the advantages and limitations of the developed proposals related to programmable networks. In this context, simulators are envisioned as essential tools for replicating experiments, offering the required realism, adaptability, and scalability within a controlled environment. However, current solutions have limitations related to the SDN capabilities and indicators that allow for optimizing network performance, which is crucial for Beyond 5G (B5G) and 6G. To overcome this challenge, we propose SDNSimPy, a Python-based simulation framework built on a discrete event simulator. The proposed simulator features a modular architecture with various functional abstractions related to programmable networks, which have been partitioned into distinct modules to streamline its development and facilitate future extensions. Moreover, SDNSimPy has undergone a verification phase to check its implementation. Results obtained from the simulator reveal a significant distinction in the operation modes (proactive and reactive) with respect to end-to-end delay. This parameter is crucial in Beyond 5G (B5G) services and can impact the quality of service (QoS) of network communications.

Keywords: next-generation networks; simulation; SDN

Citation: Calle-Cancho, J.; Cruz-Carrasco, C.; Cortés-Polo, D.; Galeano-Brajones, J.; Carmona-Murillo, J. Enhancing Programmability in Next-Generation Networks: An Innovative Simulation Approach. *Electronics* **2024**, *13*, 532. https://doi.org/10.3390/electronics13030532

Academic Editors: Stefano Rinaldi and Martin Reisslein

Received: 11 December 2023
Revised: 19 January 2024
Accepted: 26 January 2024
Published: 29 January 2024

1. Introduction

Due to the continuous development of mobile communication networks in recent years, achieving efficient network management has become one of the greatest challenges in next-generation network environments. Furthermore, advances in mobile technologies and the emergence of a multitude of services provided by operators have stimulated the development of novel mechanisms and architectures to manage networks efficiently. These are designed to accommodate the novel mobile technologies, services, and applications that are currently emerging, changing the original network architecture. Moreover, all these changes are having a significant impact on the economic strategies implemented by network operators. When operators introduce new services, expand their network infrastructure, and/or optimize their resources, they must take into account the cost of these actions. The increasing user requirements for efficiency and availability are associated with a cost increment of all these efforts. Therefore, planning and design decisions for networks must consider cost estimates with the utmost precision possible. In this context, the ability of network operators to test the proposed mechanisms requires a systematic methodology to realistically replicate experiments. Evaluating new network ideas involves a high degree of experimentation before adopting a protocol or implementing a system in a production environment. The experimentation process typically involves testing a system

in a laboratory environment to verify that the idea reliably meets initial requirements or to convince network operators that the emerging network mechanism offers significant advantages [1].

Hence, it is crucial to employ a suitable platform for validating research proposals, enabling the execution of extensive experiments at a lower expense. Nonetheless, engaging in computer network research poses challenges owing to the diverse and scattered nature of network components. Depending on the experiment's complexity, researchers may encounter difficulties accessing resources for high-performance experiments, testing distributed algorithms, and programming network component behavior. To overcome these challenges, the research community has actively worked on developing network experimentation platforms. These platforms aim to assist experimenters in meeting their research needs and fulfilling the associated requirements [2].

Moreover, with the emergence of software-defined network paradigms, it is essential to comprehend all the underlying concepts to conduct innovative and efficient research from a network perspective [3]. In this context, efficient management of resources and network flows has emerged as a critical aspect of network performance optimization. To overcome these challenges, SDN simulators can be used to improve the management of network resources and flows by providing a platform for evaluating and optimizing network performance [4]. These simulators allow for the analysis of various aspects of network resource management, such as network topology, traffic control, routing, and data aggregation [5]. They also enable the comparison of different resource provisioning methods and the evaluation of their impact on network performance. Additionally, SDN simulators can be used to study the impact of flow management strategies on network throughput and quality of service [6]. By simulating different scenarios and evaluating the results, researchers can identify the most effective approaches for managing flows and improving network performance. These simulations can also help in the measurement of traffic flow, improving packet loss rate, and enhancing bandwidth utilization.

Therefore, this article introduces a framework for modeling and simulating SDN networks with a high level of detail on network flow control, coupled with an enhanced visualization layer capable of providing information about events occurring in the SDN network in a straightforward manner. It displays the behavior of SDN technology, abstracting users from its complexity. The contributions of this paper are as follows:

- We propose a review of the main network simulators, discussing the challenges and limitations in relation to SDN technology.
- The behavior of the SDN network is modeled, analyzing the overall operation of the SDN technology considered in the proposed simulator.
- A modular SDN simulator is implemented with various functionalities, including a visualization layer, simulation parameter configuration, network topology editing, and the loading of real network traffic.
- A comprehensive analysis of all simulator components is carried out, validating each one.

The rest of this paper is organized as follows: Section 2 provides background information on the software-defined networking paradigm. Section 3 provides an overview of the SDN simulation landscape. Following that, Section 4 elaborates on the proposed simulator architecture and its main components. Subsequently, Section 5 conducts a simulator verification, demonstrating its primary functionalities and outputs. Lastly, Section 6 concludes the paper.

2. Background

Software-defined networking is recognized as a crucial architecture for handling traditional IP networks known for their complexity and challenging management. The SDN architecture is characterized by being directly programmable, agile, centrally managed, and built on open standards [7]. SDN has been developed to facilitate the programmability of connectivity services provided by 5G beyond (B5G) and 6G networks, enabling the dynamic direction and management of network traffic flows to achieve the maximum

possible benefits [8]. Thus, SDN enables intelligent and flexible programmable B5G/6G networks, allowing for fine-grained orchestration and control of applications/services. It creates a virtualized control plane that can make intelligent management decisions for network functions, bridging the gap between service provisioning and QoE (quality of experience) management in new-generation mobile networks [9]. SDN can provide context-aware QoE management, ensuring network integrity, reliability, and reduced latency for delay-sensitive multimedia applications [10].

Figure 1 shows the different planes and elements that constitute an SDN network architecture. In these networks, the network control and forwarding functions (the control and data planes) are decoupled, enabling direct programmability of network control and abstracting the underlying infrastructure for network applications and services (the application plane).

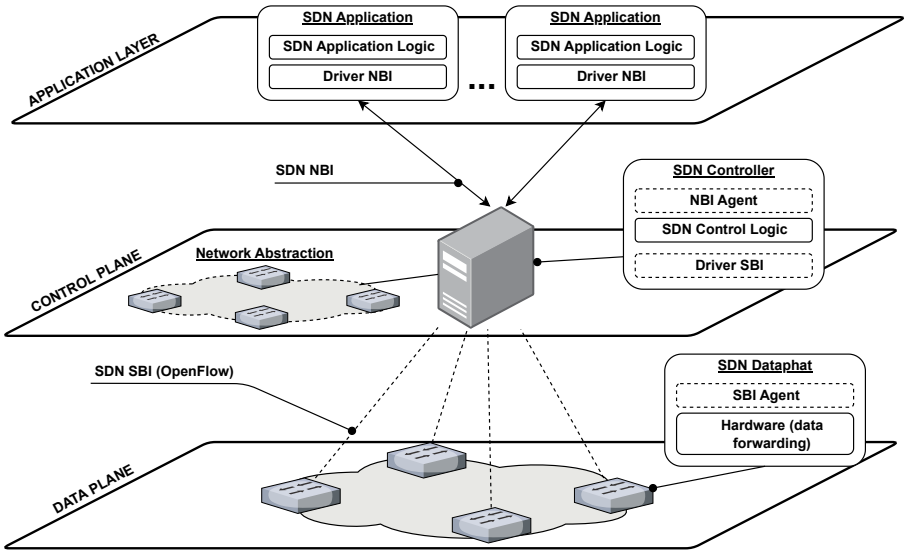


Figure 1. SDN functional architecture.

The SDN controller is a crucial component of the SDN architecture as it provides a global view of the network and connects applications to network resources. It is responsible for implementing flow actions based on application policies and managing traffic in the network. The general architecture of an SDN controller consists of multiple modules that need to be understood for simulator implementation [11,12].

- Link discovery. It sends queries about external ports using PacketOut messages. The responses to these request packets return as PacketIn messages, enabling the controller to construct the network topology.
- Topology manager. It is the management module responsible for maintaining the topology itself.
- Decision-making. The topology maintained by the topology manager allows this module to find optimal paths between network nodes. The paths are constructed in such a way that quality of service (QoS) policies and security policies can be applied during the path creation.
- Storage manager. It is a dedicated statistics and queue manager designed to collect information about performance and manage the different queues of incoming and outgoing packets.
- Flow manager. This module interacts directly with the entries and flow tables of the data plane, using the SBI (southbound interface) for this purpose.

OpenFlow switches (OFSs) are devices located in the forwarding plane responsible for carrying out traffic forwarding functions across the network. They have one or multiple OpenFlow channels through which they communicate with the controller(s). Through

these channels, a controller can manage the respective switch by adding, removing, and updating flow rules in its flow table(s) using the OpenFlow protocol [13]. This flow insertion can occur reactively, in response to the arrival of specific packets from a switch, or proactively, either before receiving traffic from a switch or after receiving certain packets from another switch.

As can be seen in Figure 2, an OFS switch has a DataPath that identifies it, ports (input and output), and a group table that allows OpenFlow to include additional forwarding methods. The group table contains group entries, whose main components are: group identifier, group type (to determine group semantics), counters (updated when packets are processed by a group), and action buckets (an ordered list of action buckets, where each action bucket contains a set of actions to execute and associated parameters). Additionally, it features a meter table that contains meter entries, enabling OpenFlow to implement basic QoS operations such as speed limitations, which can be combined with per-port queues to implement complex QoS policies.

The flow tables of an OpenFlow switch are sequentially numbered, starting at 0. The pipeline processing always begins at the first flow table. Therefore, when a packet arrives at the switch, it undergoes comparison with the match fields of the flow entries in table 0 (starting with the most prioritized ones). If the packet matches any flow entry, the set of instructions included in that flow entry is executed. These instructions may direct the packet to another higher-order flow table, where the same processing will be repeated [14].

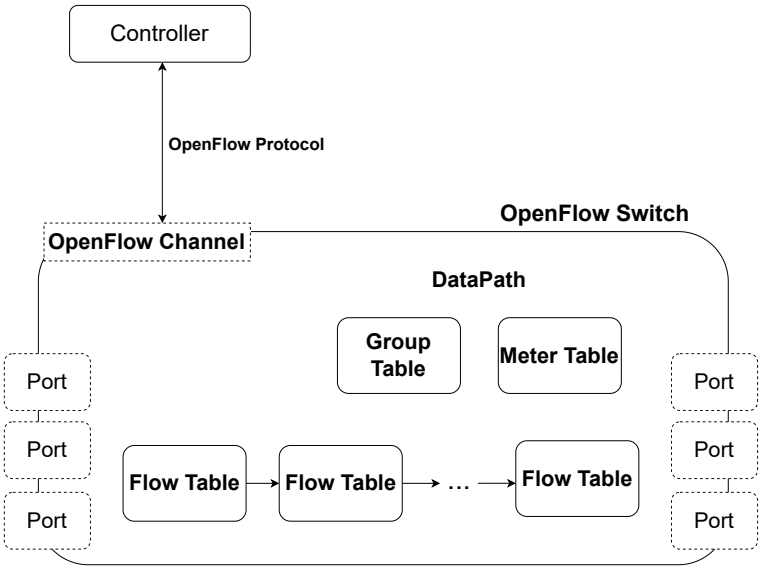


Figure 2. OpenFlow switch.

The flow tables mentioned above are the most important data structures of an OpenFlow switch. In OpenFlow switches, all incoming packets are processed by the OpenFlow pipeline. The protocol's pipeline defines how packets interact with the flow tables. An OpenFlow switch must have at least one flow table, but it can have multiple. The packet processing in the OpenFlow switch pipeline is shown in Figure 3.

Another important aspect of the SDN network is the connection between switches and the controller. To establish this connection, a logical connection is first established using TLS or TCP. Once this connection is made, the version negotiation process begins, where both ends exchange a Hello message. Upon receiving this message on each end, each one determines the negotiated version of OpenFlow. Once the version negotiation phase is completed, the feature discovery phase begins. The purpose of this phase is for the controller to learn about the capabilities of the switch. To achieve this, the controller sends a FeatureRequest message requesting the switch's capabilities, and the switch responds with a FeatureReply, providing information about its features. From this point on, the main communication phase between the controller and the switch begins, where the controller

instructs the switch (using OpenFlow messages as seen before) on how the traffic passing through the switch should flow.

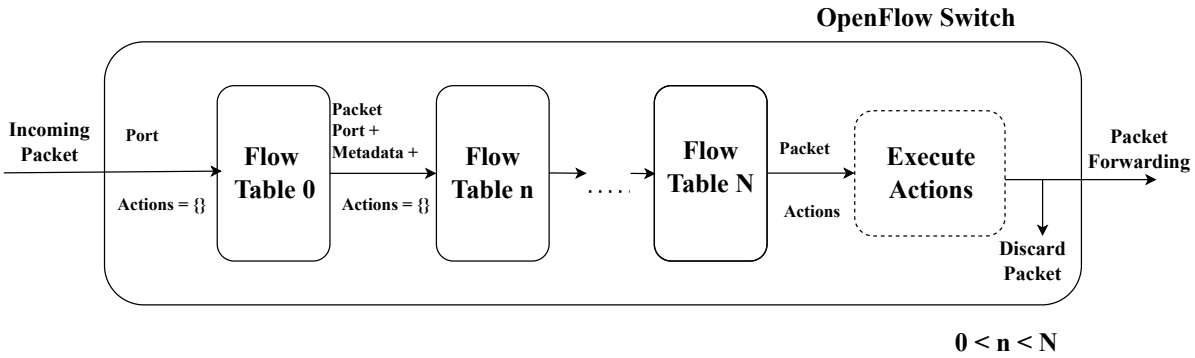


Figure 3. Packet processing on the OFS pipeline.

3. Network Simulator Landscape

Network simulators enable the evaluation of network behavior, as well as protocols and technologies involved when subjected to different scenarios and workloads. The results can be obtained economically and without the need for a real implementation. Through simulation-based studies, researchers can gain insights into events within scenarios featuring large-scale models, customized applications, and dynamic environments. Simulators enable the recreation of intricate network topologies, traffic patterns, and various network conditions, offering comprehensive evaluations without the need for expensive equipment, extended setup times, or disruptions to real network traffic. Additionally, network simulators provide insights into scalability, interoperability, and security, allowing researchers to assess a protocol's handling of high traffic volumes, diverse devices, compatibility with different systems, and resilience to security attacks.

Typically, network simulators employ discrete-event models, operating in discrete steps where events facilitate communication between simulation entities, resulting in state transitions. The alteration of these states occurs in response to events that initiate a transition and produce output results [15]. Below is a list of the most commonly used network simulators in recent times.

- The objective modular network testbed (OMNeT++) [16] is an open-source discrete event simulator based on the C++ programming language, which enables the modeling of communication networks, multiprocessors, and other distributed or parallel systems. A model in OMNeT++ consists of several modules. These communicate through message passing, which can be sent via connections that extend between different modules or directly to the target module(s).
- Network simulator 3 (ns-3) [17] is an open-source discrete-event network simulator written in C++. It was developed as a replacement for the widely used ns-2 simulator in the educational and scientific sectors. ns-3 focuses on enhancing central architecture, software integration, models, and educational components for network devices and protocols. It simulates both unicast and multicast protocols and is extensively employed in research on ad hoc mobile networks.
- The optimized network engineering tool (OPNET) [18] is a comprehensive development environment designed for specifying, simulating, and evaluating communication network performance. It provides a user-friendly interface for modeling both wired and wireless communication networks, featuring a discrete-event simulator with a hierarchical structure to model network component behavior. Specialized libraries support existing protocols, allowing for modification and the creation of customizable libraries. OPNET models are compiled into executable code for debugging or execution, producing output data. The platform includes tools for experimenters to specify detailed models, identify elements of interest, run simulations, and analyze results.

- Software-defined networking for communication over real-time Ethernet (SDN4CORE) [19] is an extension of the OMNeT++ simulator. It is an open-source, event-based simulator for programmable real-time Ethernet networks. It offers programmable network controllers and devices for various Ethernet extensions and management protocols.
- CloudSimSDN [20] is designed to replicate the utilization of hosts and networks, as well as the response time of requests within SDN-enabled cloud data centers. It functions as an additional package integrated with CloudSim, making it advisable to acquire proficiency in CloudSim before delving into CloudSimSDN. Notably, CloudSimSDN facilitates the computation of power consumption for both hosts and switches. For example, it enables the assessment of network-aware VM placement policies.
- The global mobile information system simulator (GloMoSim) [21] is a wireless network simulator that uses both sequential and parallel models. It consists of library modules that simulate wireless communication within the protocol stack. GloMoSim's libraries are built upon the parallel simulation environment for complex systems (PARSEC), which is a parallel simulation language based on C.
- QualNet [22] is a network simulation tool with a commercial application, employed for modeling and analyzing communication networks. This simulator boasts a modular architecture, granting users the flexibility to tailor it by selecting various modules and configuring their parameters. Specifically designed as a discrete-event simulator, QualNet is adept at handling heterogeneous networks and distributed applications. It empowers users to test custom protocols, develop prototypes, and execute large-scale network simulations through its comprehensive simulation suite.

After analyzing the various network simulation models and the features offered by different tools developed to date, the reasons for the development of the proposed simulator in this article (SDNSimPy) are presented. It has been deemed appropriate to develop a simulation tool that provides all the advantages and features offered by the Python programming language, widely used in emulated models. The following stand out: powerful built-in data structures such as lists and dictionaries; an interpreted language that enables interactive exploration of code or data; and an extensive library that offers comprehensive functionality for data processing. SDNSimPy provides a visualization of SDN technology behavior. It can be used as a tool to aid in understanding such a broad and complex technology as SDN, enabling integration with key functionalities along with real traffic load or enhanced visualization of each process carried out in a programmable network. In the subsequent sections, a comprehensive breakdown of all developed functionalities within the executed simulator is presented, accompanied by its validation process.

4. Simulator Architecture

In order to comprehend the modular development executed within this simulator and its diverse functionalities, a comprehensive analysis of all its components is undertaken, examining each implementation detail. Initiating this exploration, the distinct constituent elements of the simulator are meticulously delineated, thereby elucidating the specific divisions within its developmental framework. Figure 4 shows the architecture of SDNSimPy and its general operation scheme.

SDNSimPy features a topology editing functionality within the simulator itself, allowing us to load existing topologies through a JavaScript object notation (JSON) file. This enables the modification of pre-existing topologies or the creation of new ones from scratch. On the other hand, SDNPySim provides the option to load or create the packets and flows that hosts will generate in the simulated network. Additionally, preferences can be configured to define the behavior of the discrete event simulator. Moreover, the simulator's core manages the simulation's operation and oversees the visualization layer, allowing a comprehensive view of its functionality. Finally, the simulation results are generated, showcasing different charts depicting the outcomes achieved during the simulation execution. It

is possible to export the network state along with all the events that occurred throughout the execution. Next, the development of each of the parts described above is detailed.

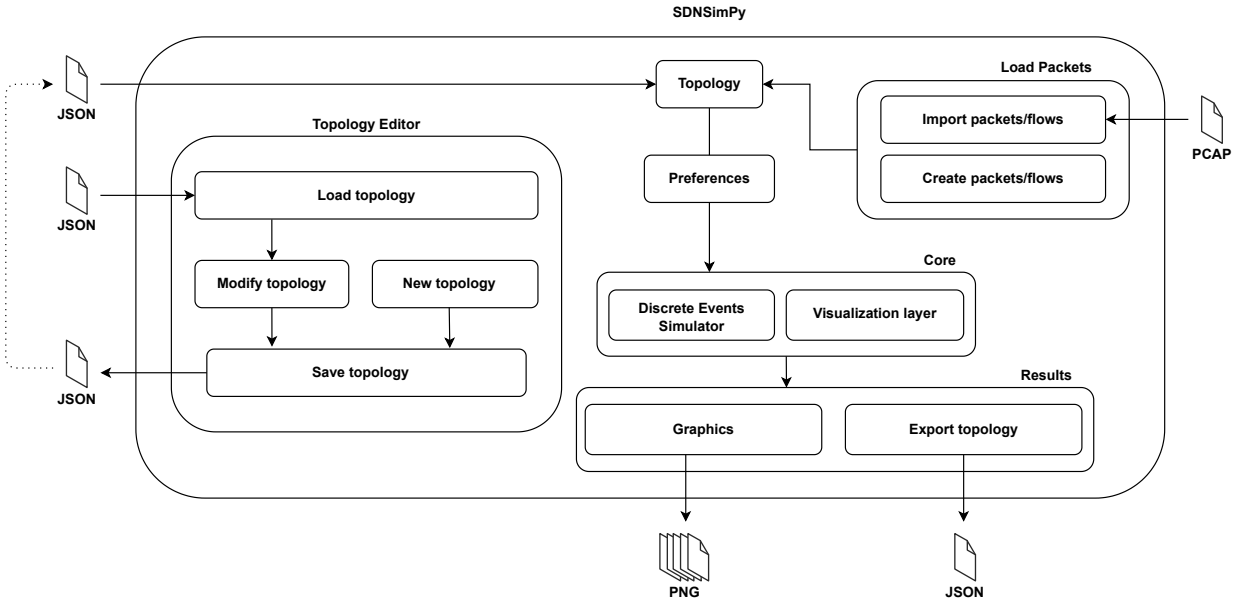


Figure 4. SDNSimPy architecture and operation schema.

4.1. Topology Editor

The functionality for creating and editing topologies has been implemented through an adaptation of the MiniEdit tool [23], taking into account the specific requirements of our simulator. This tool has been adapted in such a way that it allows the introduction of the different elements (switches, hosts, controllers, and links) that make up the network topology, as shown in Figure 5.

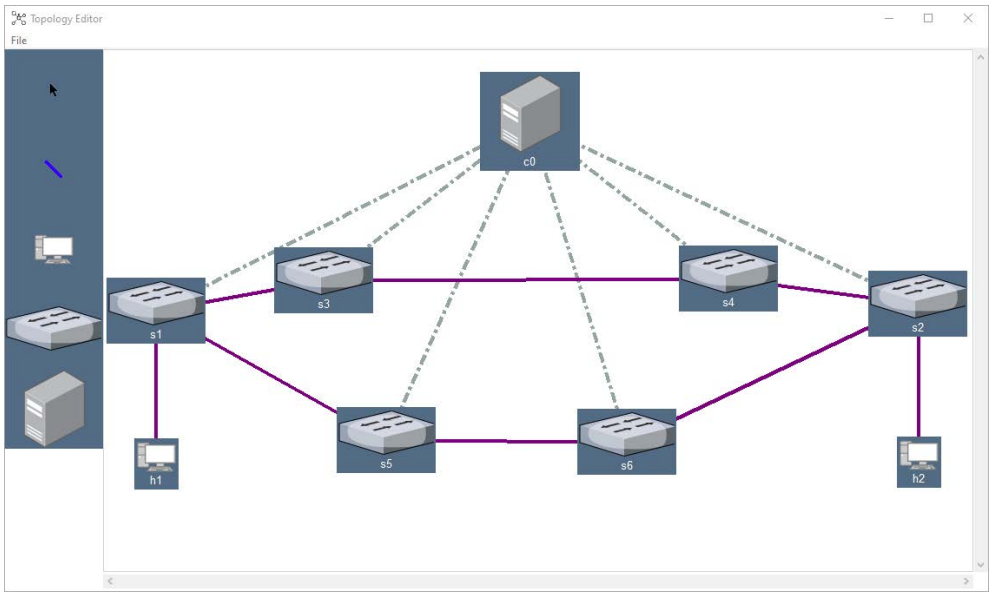


Figure 5. SDNSimPy topology editor.

In Figure 5, an example of a topology formed by several switches, hosts (connected by various links), and a controller can be seen. Additionally, each of these devices can be configured by allowing the input of the IP address, MAC, and port of each one (see Figure 6).

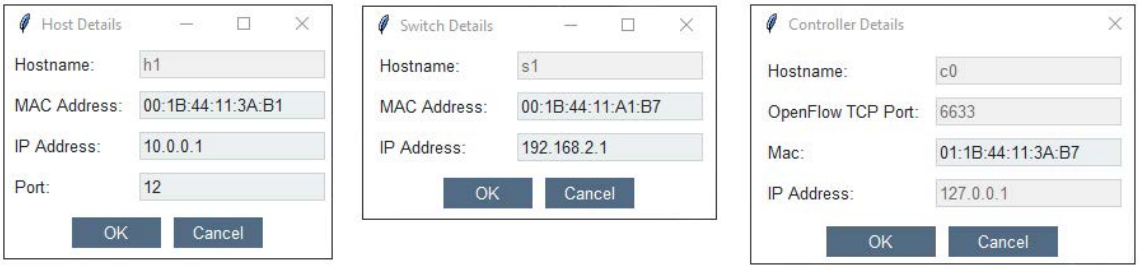


Figure 6. Configuration for each type of device: host, switch, and controller.

Simultaneously, various important characteristics can be introduced for each of the links (bandwidth, distance, and propagation speed) since, as will be seen later, these properties will significantly influence decision-making in routing packets through the network and define the time it takes to propagate from it (see Figure 7).

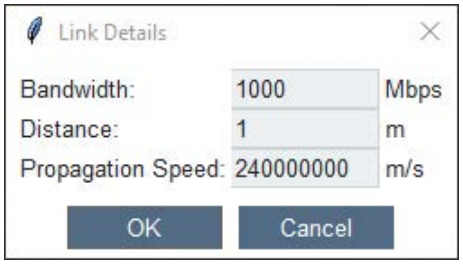


Figure 7. Link configuration.

4.2. Network Topology

To model the network topology and have a constant overview of the complete network state, the NetworkX library [24] has been used. This Python library is used for the creation, manipulation, and study of the structure, dynamics, and functions of complex networks. The simulator considers a network topology represented as an undirected graph $G = (V, E)$, where V and E denote the sets of nodes and links. Three types of nodes can be identified:

- Hosts and controllers: These two types of nodes only include attributes for link-layer and network-level addressing, along with the transport-level ports used by them.
- Switches: To these types of nodes, in addition to including the characteristics described in the two previous types, a flow table has been added as an attribute. The flow table consists of a list of flow entries that define the behavior of the switch.

Regarding the links, they have been included with attributes such as bandwidth, distance, propagation speed, and a load list to store the load throughout the simulation execution. This will be crucial for obtaining simulation results. Finally, algorithms provided by this library have been employed; specifically, Dijkstra's algorithm, which calculates the minimum paths between nodes within the topology itself. This is of vital importance for decision-making related to SDN network routing.

4.3. Packets Load

Once the topology has been loaded into the simulator, packets can be loaded onto the hosts. To do this, SDNSimPy uses the Scapy library [25]. Scapy is a Python library used for sending, analyzing, dissecting, and falsifying network packets; in essence, it is a powerful tool for creating and manipulating network packets. SDNSimPy primarily uses this tool for creating and importing packets into the hosts present in the topology, as shown in Figure 8. This way, the packets transported by the switches in the simulation will always have the data structure containing the fields of a real packet.

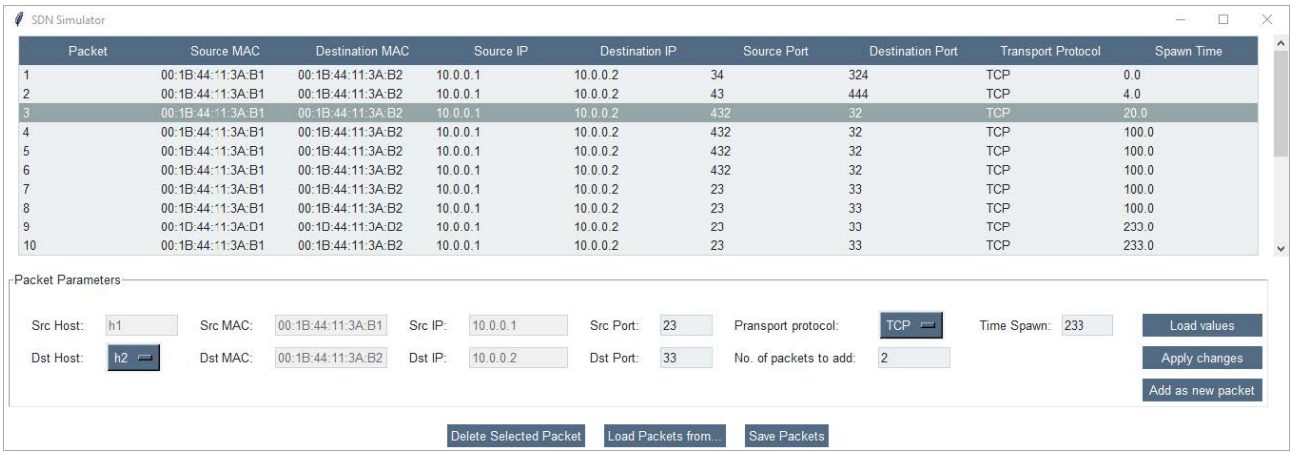


Figure 8. Packet load process on the host.

4.4. Simulation Preferences

The simulation preferences play a fundamental role in the core of the simulator. Specifically, they will define the behavior of discrete event simulation and certain aspects of the visualization layer. Figure 9 shows the simulation preferences.

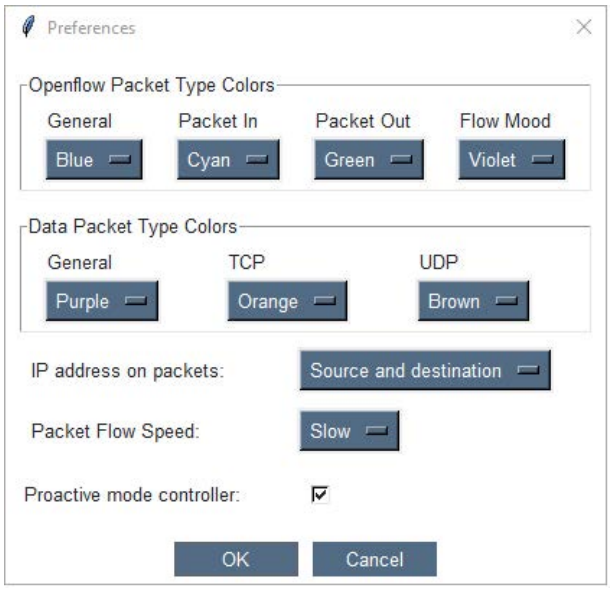


Figure 9. Simulation preferences.

There are different aspects that we can modify, such as the general color of OpenFlow messages and user data packets or the colors of their sub-types. Figure 10 provides an example describing different types of these messages/packets based on the parameters set in Figure 9.

As shown in Figure 10, within the data packets, two sub-types are distinguished: the one on the left uses the UDP protocol (brown), while the packet on the right uses the TCP protocol (orange). If you look at the OpenFlow messages at the top, you can see (from left to right) a PacketOut message (green) next to a FlowMod (pink), and in the top right, a PacketIn (cyan blue).

Regarding the discrete event simulator, the behavior of the controller can be defined, allowing to establish whether it should have a reactive or proactive behavior. In the case of selecting proactive operation, when the controller receives a PacketIn message from a switch, in addition to responding with its corresponding PacketOut along with a FlowMod

message, it will anticipate and introduce a flow entry into the other switches through which the packet encapsulated in the mentioned PacketIn will pass, using FlowMod messages.

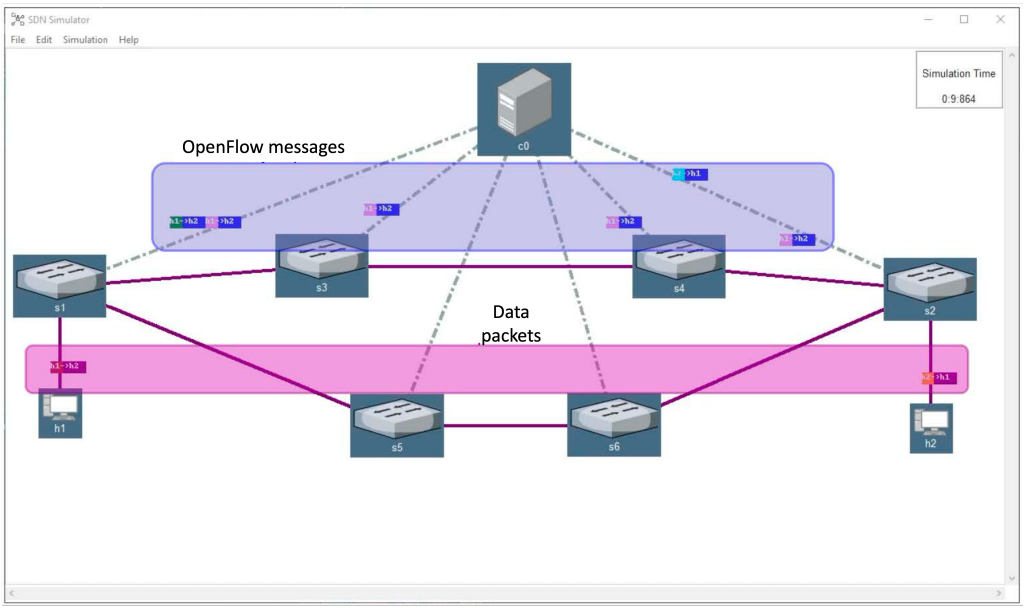


Figure 10. Information about the different types of messages/packets based on their color.

4.5. Core Simulator

The entire simulation execution is controlled by the simulator core. On one hand, we have the discrete event simulator, which will have the logic and control of the simulation. On the other hand, we have the visualization layer, which synchronously reflects all events from the discrete event simulator in the graphical interface, allowing the user to visualize everything happening throughout the simulation in real time.

To enable user interaction with the various elements of the topology (switches, hosts, controllers, and links) and view the characteristics alongside the different data structures that may undergo modifications due to the processing of events during the simulation, a multi-threaded execution is proposed as shown in Figure 11.

As can be seen in Figure 11, on one side, there is the main program that maintains the primary execution of the discrete event simulation. This, in addition to initializing the event list, launches the execution of a thread responsible for selecting the first event from the list and checking if the time at which the event should be processed is appropriate, in which case it will perform the following actions:

- Initialization of the discrete event list from the packets loaded into each host in the topology.
- Selection of the first event from the event list.
- Update of the simulation time to the timestamp of the selected event and the state variables of the simulator.
- Processing of the event.

Moreover, the thread responsible for handling the specific event will access the shared memory with the main program, allowing it to update the simulation time and state variables, and introduce new events generated by the processing of a given event.

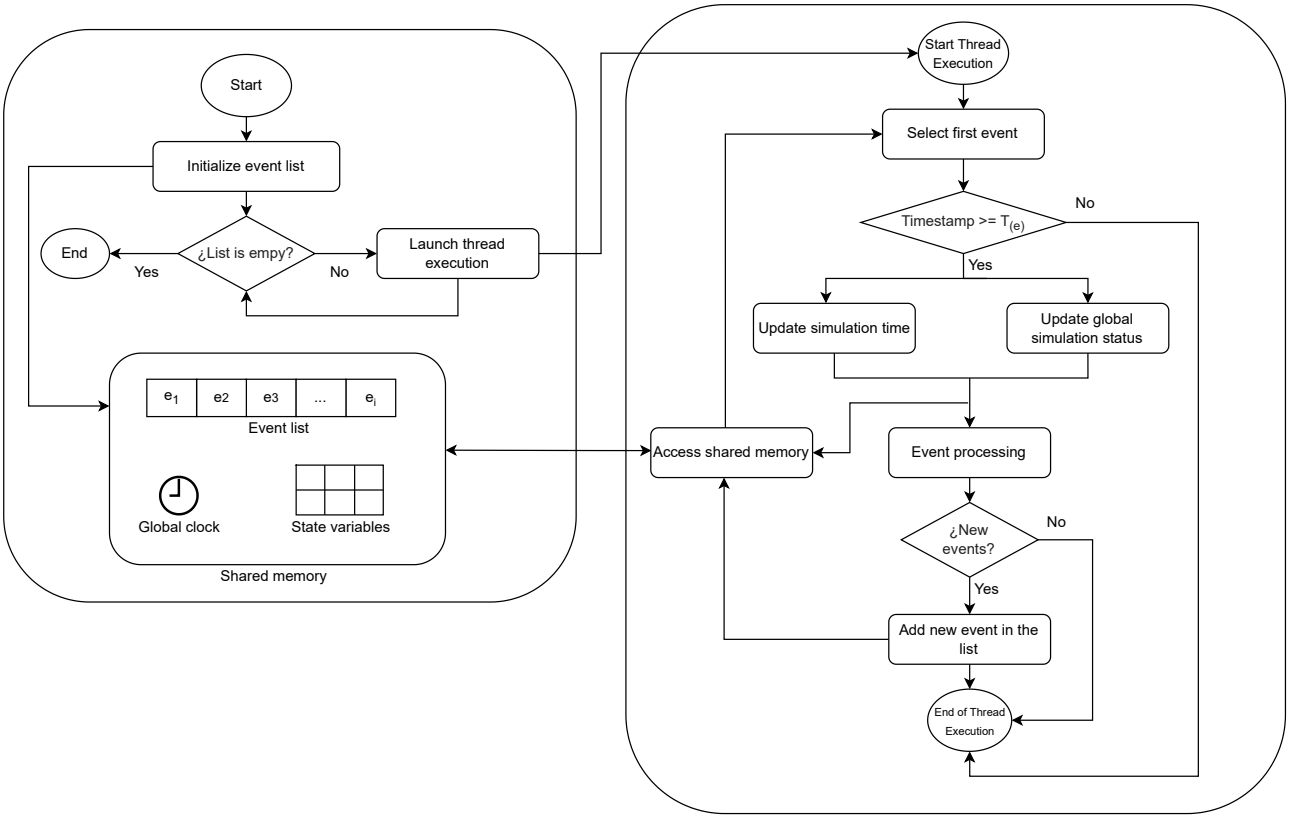


Figure 11. Discrete event simulation flowchart.

5. SDNSimpy Verification

To verify that the network behavior aligns with the functioning of a software-defined network, the simulator has been executed along with an analysis of the results obtained by switching between the two operating modes implemented in the controller.

5.1. Proactive Controller

The topology used to conduct these tests consists of one controller, six switches, and two hosts, which are interconnected through two possible paths, as shown in Figure 5. All links in the topology have a bandwidth of 1000 Mbps, except for the link connecting switches s3 and s4, which has a bandwidth of 10,000 Mbps.

On the other hand, in Figure 12, the two packet flows created on host h1 with a destination of h2 can be observed. To facilitate the visualization of the test, the packets in each flow are generated at increasing points in time with respect to the previous packet.

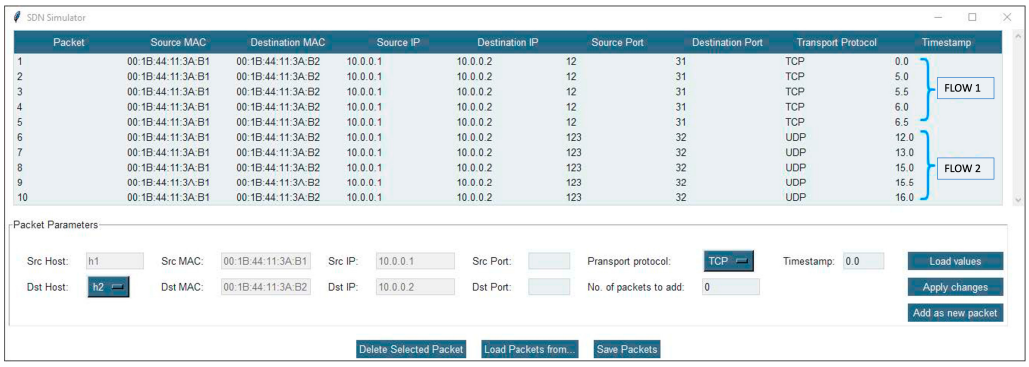
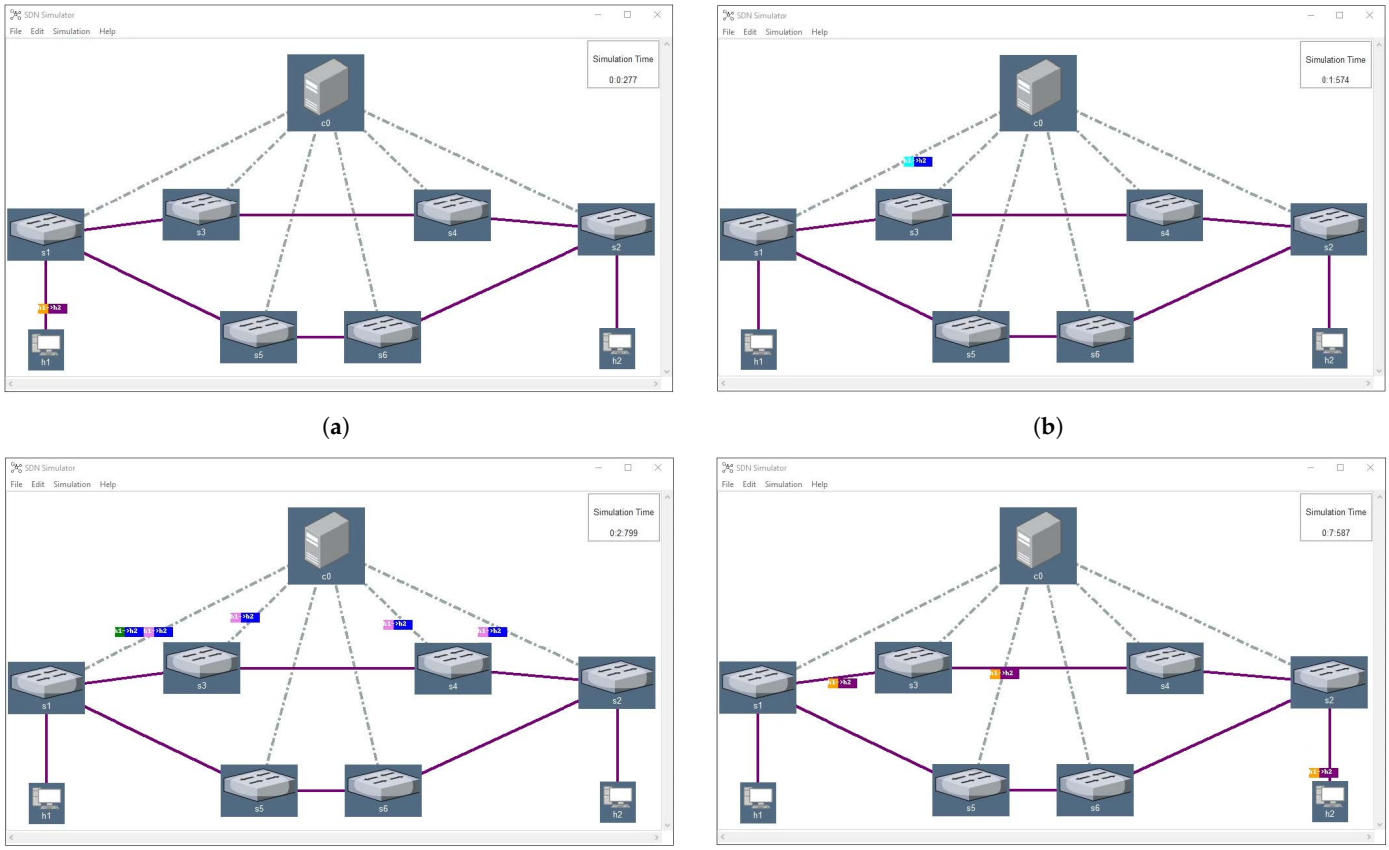


Figure 12. Packet flows created on a host.

When the simulation starts, it can be observed that upon the arrival of the first packet from Flow 1 (Figure 13a), the switch sends a PacketIn message to the controller (Figure 13b), and the controller responds with a PacketOut accompanied by a FlowMod. Furthermore, since the proactive mode of the controller is enabled, it will proactively send a FlowMod to the other switches, inserting a flow entry for all packets belonging to the same flow. This adds an entry to the flow table of those switches for subsequent packets belonging to the same flow that may pass through them (Figure 13c). Finally, in Figure 13d, it can be observed how the remaining packets match in the switches and are routed toward their destination, h2.



(**a**) (**b**)

(**c**) (**d**)

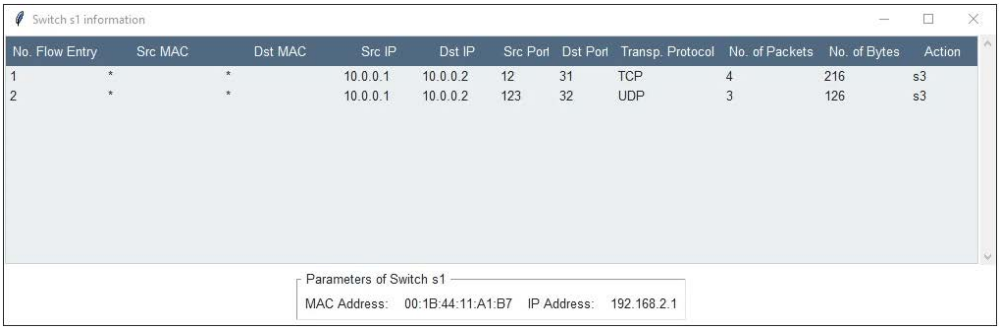
Figure 13. Verification of the proactive operating mode. (**a**) Sending the first flow packet. (**b**) PacketIn from s1 to the controller. (**c**) Controller response and add flow entries. (**d**) Flow 1 routing.

To route the packets, the controller executes the Dijkstra algorithm, and since all links have the default bandwidth except for the s3-s4 link, which has a higher capacity, the path along which the switch will route the flow will be s1-s3-s4-s2.

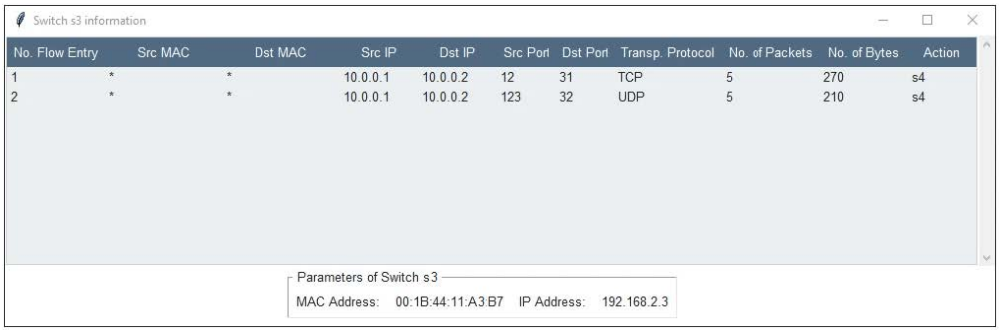
As a result of sending this initial flow, there is an entry in the flow table of switch s1 through which the four packets have passed (since the first packet encapsulated in the PacketOut skips the pipeline and exits directly through the link to s3) and have matched in that entry. In addition, the other switches will also have a flow entry, but in this case, with five packets that have matched.

In Figure 12, at time 12.0, host h1 will start sending Flow 2. However, in this case, since the second packet is sent so close in time to the first one, it will not have an entry to match with yet. Therefore, the switch will once again send a PacketIn. The remaining packets in this flow will match the entry inserted by the controller and follow the same path as in Flow 1. The flow table entries of switches s1 and s3, showing the expected results described earlier after the simulation execution, can be seen in Figure 14. Moreover, ports are defined on hosts depending on the type of flow configured. Flow 1: On h1, port 12 is

configured, and on h2, port 31 is configured. Flow 2: On h1, port 123 is configured, and on h2, port 32 is configured.



(**a**)



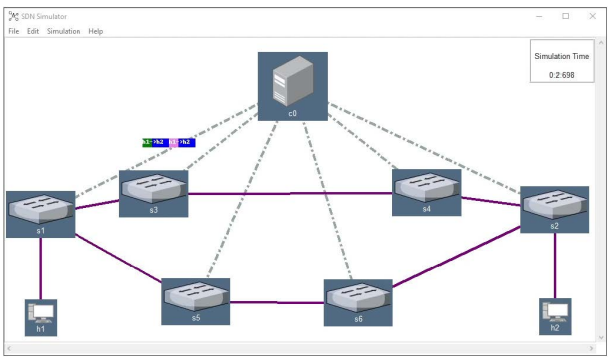
(**b**)

Figure 14. Flow tables of switches s1 and s3 after running the simulation. (**a**) Flow table of switch s1. (**b**) Flow table of switch s3.

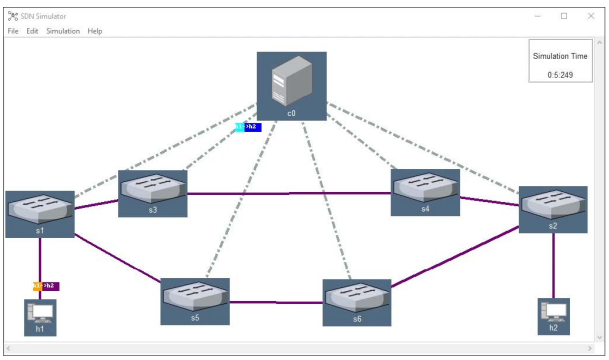
5.2. Reactive Controller

The topology used to conduct these tests consists of one reactive controller, six switches, and two hosts, which are interconnected through two possible paths, as shown in Figure 5.

As the reactive mode is being used, when the controller receives a PacketIn, it will respond with a PacketOut and a FlowMod, as in the previous section. However, in this case, it will not insert flow entries in the other switches that make up the path the flow will take in advance. Therefore, the first packet received by each switch along the flow path to its destination will result in a PacketIn from each switch that receives that packet. Each PacketIn will be responded to by the controller with PacketOut and FlowMod. In Figure 15a, the main difference between this reactive mode of operation compared to the proactive one can be observed (compared with Figure 13c).



(**a**)



(**b**)

Figure 15. Verification of reactive operating mode. (**a**) Controller response and add flow entries. (**b**) PacketIn from s3 switch to controller.

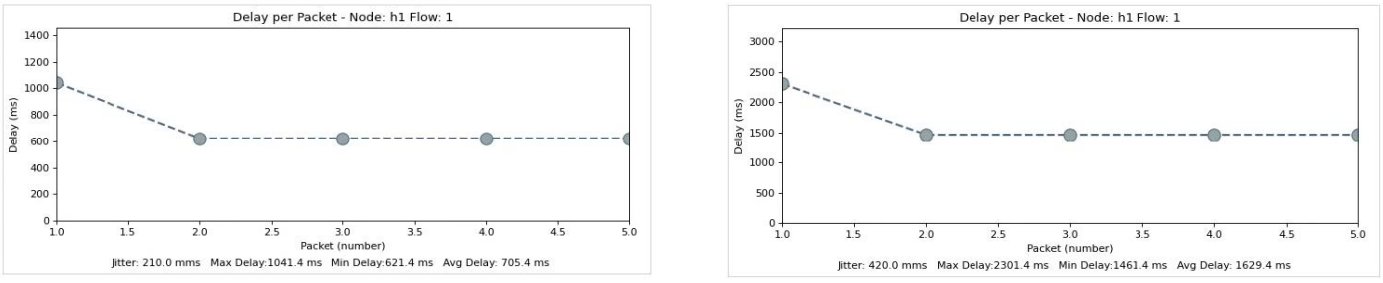
Next, it can be observed that upon the arrival of the first packet at the next switch, it queries the controller again with a PacketIn (see Figure 15b). This behavior will repeat successively until the last switch through which this flow will pass.

5.3. Graphical Results

SDNSimPy generates graphical results providing information after the simulation execution. Specifically, it provides information about the network link load and the delay for flows generated by different hosts in the network. The graphical results window after the execution of a generic simulation can be seen in Figure 16. At the top, we find information related to the delay of each of the flows generated by each host. Within the graph itself, one can observe the delay for each packet belonging to the flow, the jitter (variability of delay), and the maximum, minimum, and average delay of the respective flow. At the bottom, we have the load information for each network link at each time instance of the simulation, in bytes. In addition to the load at each time instance, it provides the maximum, minimum, and average loads for each of them.

Figure 16. Graphical results after running a simulation.

Depending on the controller's operating mode, notable differences are detected in terms of delay. In the proactive mode, the controller anticipates and inserts flow entries in all switches, involving a single communication with the controller from the first switch. In the reactive mode, for each switch through which the first packet passes, there is communication with the controller. Figure 17 shows the delay of the first flow for both executions (proactive and reactive).



(**a**) (**b**)

Figure 17. Comparison of Flow 1 in both executions. (**a**) Proactive mode operation. (**b**) Reactive mode operation.

It can be observed that in the delay of the flow in both executions, the first packet belonging to that flow has a higher value, as it is the one triggering the sending of FlowMod.

6. Conclusions

Throughout the development of this paper, we conducted a deep dive into understanding next-generation networks. These networks will be essential in providing the features required by advances in new applications, services, etc. Additionally, these networks will support the continuous increase in internet-connected users, which significantly grows every year. Due to the limitations of current networks, it is necessary to research and develop new technologies that provide these networks with the necessary properties for their implementation. Hence, we conducted a detailed study of SDN technology, providing a foundation for a flexible, scalable, agile, and programmable network.

In conclusion, to understand SDN networks and have a tool to test new mechanisms on these networks, we introduce SDNSimPy, a simulation framework implemented in Python and built upon a discrete event simulator. Our proposed simulator adopts a modular architecture, incorporating various functional abstractions related to programmable networks. This modular design, partitioned into distinct modules, not only enhances the development process but also facilitates future extensions, such as incorporating new routing algorithms in SDN networks, leveraging the capabilities offered by NetworkX. Additionally, SDNSimPy has undergone a thorough verification phase to ensure the accuracy of its implementation.

Author Contributions: Conceptualization, J.C.-C., C.C.-C. and J.C.-M.; Methodology, J.C.-C., C.C.-C. and D.C.-P.; Software, C.C.-C., J.C.-C. and J.C.-M. Funding acquisition, J.C.-M.; Validation, J.C.-C., C.C.-C., D.C.-P., J.G.-B. and J.C.-M.; Formal analysis, C.C.-C., J.C.-C. and J.C.-M.; Investigation, C.C.-C., J.C.-C. and J.C.-M.; Data Curation, C.C.-C., J.C.-C. and J.C.-M.; Writing—original draft preparation, J.C.-C., C.C.-C., D.C.-P., J.G.-B., J.C.-M. and D.C.-P.; Writing—review and editing, J.C.-C., C.C.-C. and D.C.-P.; Supervision, D.C.-P. and J.C.-C.; Project Administration, J.C.-M. All authors have read and agreed to the published version of the manuscript.

Funding: This research was supported in part by the Ministry of Science and Innovation under the project "TED2021-131699B-I00/MCIN/AEI/10.13039/501100011033/" and the European Union NextGenerationEU/PRTR.

Data Availability Statement: The SDNSimPy simulator is available at https://github.com/CristianCr99/SDNSimPy (accessed on 25 January 2024).

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Brandom, H. Reproducible Network Research with High-Fidelity Emulation. Ph.D. Thesis, Stanford University, Stanford, CA, USA, 2013.
2. Gomez, J.; Kfoury E.F.; Crichigno, J.; Srivastava G. A survey on network simulators, emulators, and testbeds used for research and education. *Comput. Netw.* **2023**, *237*, 110054. [CrossRef]
3. Blanco, B.; Fajardo, J.O.; Giannoulakis, I.; Kafetzakis, E.; Peng, S.; Pérez-Romero, J.; Trajkovska, I.; Khodashenas, P.S.; Goratti, L.; Paolino, M.; et al. Technology pillars in the architecture of future 5G mobile networks: NFV, MEC and SDN. *Comput. Stand. Interfaces* **2017**, *54*, 216–228. [CrossRef]
4. Tyagi, V.; Singh, S. Network resource management mechanisms in SDN enabled WSNs: A comprehensive review. *Comput. Sci. Rev.* **2023**, *49*, 100569. [CrossRef]
5. Wang, Y.C.; Hsiao, T.J. URBM: User-Rank-Based Management of Flows in Data Center Networks through SDN. In Proceedings of the 2022 4th International Conference on Computer Communication and the Internet (ICCCI), Chiba, Japan, 1–3 July 2022; pp. 142–149.
6. Myunghoon, J.; Namgi, K.; Yehoon, J.; Byoung-Dai, L. An Efficient Network Resource Management in SDN for Cloud Services. *Symmetry* **2020**, *12*, 1556.
7. Alvizu, R.; Maier, G.; Kukreja, N.; Pattavina, A.; Morro, R.; Capello, A.; Cavazzoni, C. Comprehensive survey on t-sdn: Software-defined networking for transport networks. *IEEE Commun. Surv. Tutorials* **2017**, *19*, 2232–2283. [CrossRef]
8. Barakabitze, A.A.; Walshe, R. SDN and NFV for QoE-driven multimedia services delivery: The road towards 6G and beyond networks. *Comput. Netw.* **2022**, *214*, 109133. [CrossRef]

9. Collantes, L.H.; Wibawa, A.P. SDN: A Different Approach for the Design and Implementation of Converged Networks. In Proceedings of the 2021 3rd East Indonesia Conference on Computer and Information Technology (EIConCIT), Surabaya, Indonesia, 9–11 April 2021; pp. 450–455.
10. Nisar, K.; Jimson, E.R.; Ahmad Hijazi, M.H.; Welch, I.; Hassan, R.; Mohd Aman, A.H.; Sodhro, A.H.; Pirbhulal, S.; Khan, S. A survey on the architecture, application, and security of software defined networking: Challenges and open issues. *Internet Things* **2020** *12*, 100289. [CrossRef]
11. Jellalah, A.; Abdalwart, A.; Abubaker, A.; Abdulwahed, E.; Wisam, E.; Salem, S. SDN Controllers Comparison Based on Network Topology. In Proceedings of the 2022 Workshop on Microwave Theory and Techniques in Wireless Communications (MTTW), Riga, Latvia, 5–7 October 2022; pp. 204–209.
12. Rodriguez, A.; Quiñones, J.; Iano, Y.; Barra, M.A.Q. A Comparative Evaluation of ODL and ONOS Controllers in Software-Defined Network Environments. In Proceedings of the 2022 IEEE XXIX International Conference on Electronics, Electrical Engineering and Computing (INTERCON), Lima, Peru, 11–13 August 2022; pp. 1–4.
13. McKeown, N.; Anderson, T.; Balakrishnan, H.; Parulkar, G.; Peterson, L.; Rexford, J.; Shenker, S.; Turner, J. OpenFlow: Enabling Innovation in Campus Networks. *ACM SIGCOMM Comput. Commun. Rev.* **2008**, 38, 69–74. [CrossRef]
14. Medhi, D.; Ramasamy, K. Routing and Traffic Engineering in Software Defined Networks. In *Network Routing*, 2nd ed.; Morgan Kaufmann: Burlington, MA, USA, 2018.
15. Wehrle, K.; Gunes, M.; Gross, J. *Modeling and Tools for Network Simulation*; Springer Science and Business Media: Berlin/Heidelberg, Germany, 2010.
16. Varga, A.; Hornig, R. An overview of the OMNeT++ simulation environment. In Proceedings of the 1st International Conference on Simulation Tools and Techniques for Communications, Networks and Systems and Workshops, Marseille France, 3–7 March 2008.
17. Henderson, T.; Riley, G.; Floyd, S.; Roy, S. ns3—Network Simulator. Available online: https://www.nsnam.org/ (accessed on 25 January 2024).
18. Chang, X. Network simulations with OPNET. In Proceedings of the WSC'99, 1999 Winter Simulation Conference Proceedings. 'Simulation—A Bridge to the Future' (Cat. No.99CH37038), Phoenix, AZ, USA, 5–8 December 1999.
19. Häckel, T.; Meyer, P.; Korf, F.; Schmidt, T. DN4CoRE: A Simulation Model for Software-Defined Networking for Communication over Real-Time Ethernet. In Proceedings of the 6th International OMNeT++ Community Summit 2019, Hamburg, Germany, 4–6 September 2019.
20. Son, J.; He, T.; Buyya, R. CloudSimSDN-NFV: Modeling and Simulation of Network Function Virtualization and Service Function Chaining in Edge Computing Environments. *Softw. Pract. Exp.* **2019**, *49*, 1748–1764. [CrossRef]
21. Zeng X.; Bagrodia R.; Gerla M. GloMoSim: A library for parallel simulation of large-scale wireless networks. In Proceedings of the Twelfth Workshop on Parallel and Distributed Simulation, Banff, AB, Canada, 26–29 May 1998.
22. Scalable Network Technologies, QualNet network Simulation Software. Available online: https://www.keysight.com/ (accessed on 25 January 2024).
23. MiniEdit. Available online: https://github.com/mininet/mininet/blob/master/examples/miniedit.py (accessed on 25 January 2024).
24. NetworkX. Available online: https://networkx.org/ (accessed on 25 January 2024).
25. Scapy. Available online: https://scapy.net/ (accessed on 25 January 2024).