P4Docker: Enabling Efficient P4 Switch Testbeds with Docker Integration

Dener Silva[1], Alexandre Heideker[1],Lucas Trombeta[1], Bruna Carvalho[1], João Henrique Kleinschmidt[1], Carlos Kamienski[1]

[1] Universidade Federal do ABC (UFABC)
Santo André – SP – Brazil

{dener.silva, alexandre.heideker, lucas.trombeta, bruna.carvalho
joao.kleinschmidt, carlos.kamienski}@ufabc.edu.br

Abstract. *Software Defined Networking (SDN) represents a paradigm shift in network architecture, offering unprecedented control and flexibility by decoupling the control and data planes. A critical component of this evolution is the emergence of P4, a programming language designed for the data plane, enabling precise control over packet processing. Despite its potential, the use of P4 is hampered by its complexity and the steep learning curve associated with its programming model. Addressing this challenge, this paper introduces P4P4Docker, a Docker-based P4 switch container named P4Docker, aimed at simplifying the design, management, and debugging of P4 switch test beds. P4Docker also offers an intuitive GUI platform to engage with P4 programming, reducing entry barriers and fostering innovation in network design and experimentation. By providing an intuitive interface, P4Docker enables a more accessible and efficient engagement with P4 programming, offering comprehensive access to logs and system metrics. Ultimately, P4Docker represents a significant step toward making sophisticated network programming more accessible, potentially accelerating the adoption and innovation of SDN and data plane programmability.*

1. Introduction

Software Defined Networking (SDN) is a transformative technology that redefines traditional networking architecture by decoupling the control plane (responsible for routing decisions) from the data plane (responsible for data forwarding) [Liatifis et al. 2023]. This separation introduces agility and flexibility, allowing network administrators to manage and optimize network resources dynamically. However, the transition to SDN introduces challenges, particularly in data-plane programming, where the need for customized packet processing and forwarding behavior is crucial.

Programming Protocol-Independent Packet Processors (P4) [Bosshart et al. 2014] have emerged as a significant development in this context, offering a powerful and flexible means to define how network devices process packets. The development of programmable data planes has significantly influenced network experimentation, where P4 [Bosshart et al. 2014] enables researchers and practitioners to design, test, and implement novel network protocols with unprecedented flexibility. This optimization allowed with a

programmable data plane can be used to create new protocols or even optimize traffic for specific applications.

Despite its potential, adopting P4 and similar technologies poses a steep learning curve and requires a shift in skill sets for network engineers and developers. However, the broader adoption of the P4 technology encounters several challenges. These include the necessity for specialized skills and training, interoperability hurdles, limited hardware support constraints, security concerns, and the complexity of custom packet processing [Kfoury et al. 2021, Heideker et al. 2023].

To address these challenges, we present a Docker-based P4 container named P4Docker[1] that includes a BMv2 (Behavioral Model Version 2) switch [Consortium 2024], an open-source tool designed to be a valuable resource for examining network and protocol behaviors in a controlled setting with P4 programmability. The P4Docker also implements a GUI (Graphical User Interface) to simplify interactions with advanced networking experiments, making it easier to work with the test environment, improve debugging, and offer detailed access to logs and system metrics. This tool aims to make P4-based network experimentation more accessible while maintaining high functionality.

The interface offers a straightforward and user-friendly approach to designing network topologies for P4 test beds through a graph-based framework. Following the design phase, the interface independently produces a script that facilitates the creation of necessary containers and connections and the adjustment of essential settings, thereby simplifying the establishment of the test bed and guaranteeing uniformity across the deployment setting. Additionally, the interface includes functionalities to dismantle the established environment, enabling the reset of test beds, and incorporates a utility to assist in debugging the compilation of P4 code.

This paper is structured as follows: Section 2 introduces the motivation and related works; Section 3 explores the architecture and functionalities of the tool; Section 4 explains the test bed details; Section 5 discuss and concludes this paper.

2. Background and Related Work

Data-plane programming diverges from traditional static network configurations by offering the ability to dynamically respond to changing network conditions and traffic behaviors. It precisely controls how packets are managed and routed, with decisions based on detailed criteria like header fields or protocol types [Kfoury et al. 2021]. In the academic domain, efforts to advance data plane programmability are rising. However, the practical application is often confined to simulations due to the prohibitive costs associated with P4-compatible network devices [Kfoury et al. 2021, Goswami et al. 2023].

Initiatives like the P4Pi project [Laki et al. 2021] aims to mitigate these constraints by transforming a Raspberry Pi 4 into a P4 switch, offering an affordable option for hands-on experimentation. While this is an excellent introductory tool, its applicability in complex network environments still needs to be improved.

Tools like Mininet [Mininet 2023] offer a cost-effective alternative for testing and validating P4 applications, enabling the simulation of complex network environments

[1] https://hub.docker.com/repository/docker/dnredson/p4d

with multiple P4 switches and hosts on a single machine [Chen et al. 2023]. This virtual setup supports OpenFlow and P4 protocols using BMV2 (Behavioral Model Version 2) switches [Consortium 2024], facilitating detailed studies of network behaviors and protocol dynamics in a controlled virtual environment. However, Mininet fails to offer complete control over the switch and its behavior and does not provide environment isolation running both hosts and switches in the same host.

Existing simulation tools support P4 and have facilitated networking research. However, programming the data plane with P4 demands a more hands-on approach with the network switch, particularly for dynamic control and immediate packet processing. Unlike conventional simulation settings, P4 necessitates a test bed capable of instantaneously adapting to network conditions and user-specified policy alterations. This degree of interaction is crucial for researchers and engineers aiming for detailed control over packet routing and network dynamics, expanding the possibilities of network simulation achievements.

3. Architecture and Functions

3.1. Architecture

The P4Docker operates with a straightforward frontend-backend architecture, illustrated in Fig. 1. The frontend, developed in JavaScript, offers a user-friendly GUI interface for creating network topologies, utilizing Cytoscape[2] for graphical representation. The backend generates scripts for deploying and managing these topologies and interfaces with the P4 Compiler in a Docker container to compile and store files.

Additionally, a "Share" volume, established during P4Docker installation, links all containers, facilitating file access across the environment. Docker volumes provide persistent data storage, allowing data to be kept across container restarts and updates and sharing content between different containers. The P4 Compiler container uses this volume to store the compiled P4 codes and share content with switch containers.

Even though the tool does not actively execute or administer Docker containers and namespaces, the P4Docker facilitates users in the construction of network topologies utilizing P4 switches and generating a bash script to instantiate the requisite environment. Moreover, the interface allows users to produce an auxiliary script to dismantle the established environment. The functionality to save and load topologies further enhances usability, enabling the exportation of topologies to different machines, thereby fostering an efficient and adaptable workflow within network design processes.

3.2. Main Functions

The P4Docker dashboard, illustrated on Fig. 2, allows the following functions:

- Add node: allows the addition of a node to the edge graph topology that can be any Docker container or a P4Docker switch;
- Add Port to Node: add a port to the current selected node;
- Add Edge: creates a connection between two nodes (a host and a switch), allowing setting of networking configuration (delay, bandwidth, MAC address, IP address);
- Remove Node: remove the selected node;

[2]https://cytoscape.org/

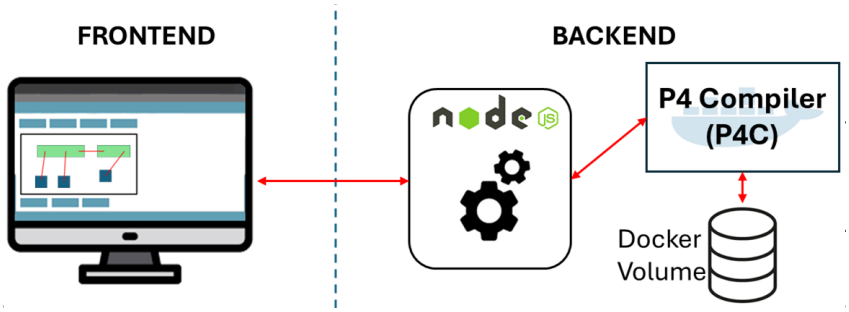


Figure 1. P4Docker Architecture

- Edit Selected Node: edit the information of the current selected node;
- Load Topology: allows loading a file with a topology created previously;
- Save Topology: allows saving the current topology in a JSON file;
- Generate Code: generate the script to create the designed topology;
- Generate Cleaner Code: generates the script to remove the designed topology.

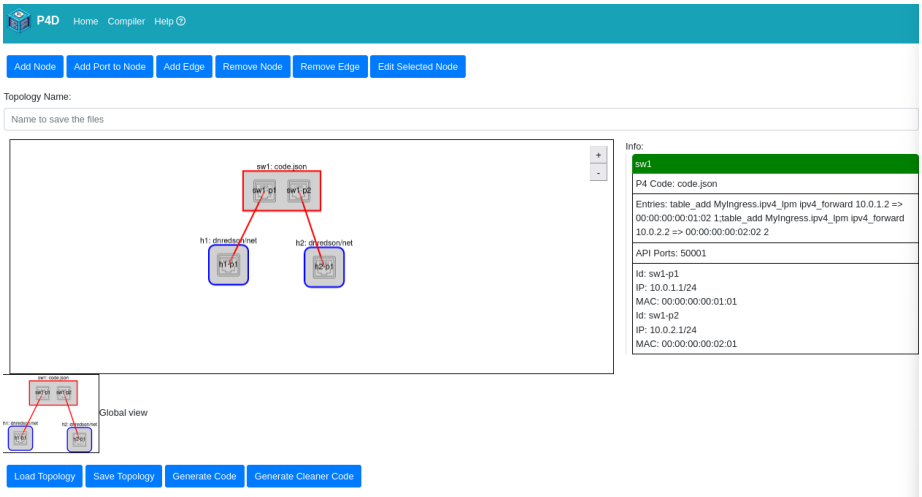


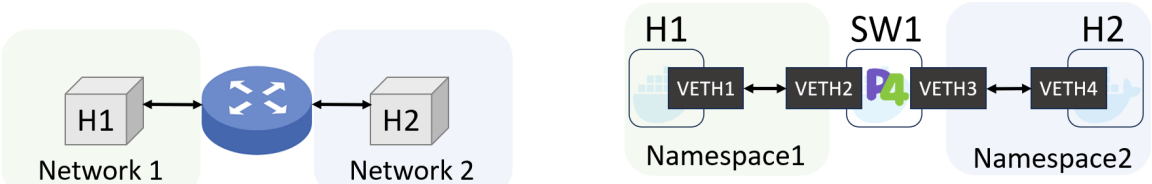
Figure 2. P4Docker Dashboard - Designing a simple topology

3.3. Topology architecture

P4Docker integrates Linux namespaces with Docker containers to simulate a detailed network environment where two physical hosts interact via a conventional switch. The use of Linux namespaces, which partition kernel resources, enables the creation of distinct network spaces on a single host, essential for emulating varied network components in SDN [Jain 2020]. This setup ensures each simulated host operates independently within its network environment, reflecting the isolation seen in real-world network infrastructures, thus providing a realistic and isolated testing scenario.

Each container functions as a separate entity on the Docker Engine, which manages its lifecycle and operations. This orchestration ensures isolation and individual interaction with the host OS. This isolation is crucial for emulating distinct network components and their interactions. Furthermore, integrating namespaces with containers creates a simulation environment where communications between hosts mimic those in a physical network, using virtual Ethernet pairs. This method offers an isolated and realistic network path.

The scenario illustrated in Figure 3a showcases a fundamental network configuration involving two hosts, designated as H1 and H2, linked through a straightforward switch. These hosts operate on two separate networks: Network 1 and Network 2. This setup demonstrates a common network structure where each host is situated in a distinct network segment, and the switch is responsible for data exchange between them.



(a) Basic Network Setup

(b) Emulated Scenario created by P4Docker

Figure 3. Basic Topology Representation

Figure 3b illustrates how the script generated by the P4Docker adapts this scenario using a container-based approach, employing the P4 language to define the network switch operations (SW1). Within this scenario, Network 1 and Network 2 are replicated as Namespace1 and Namespace2, respectively, creating distinct network contexts within a single hardware system. Virtual Ethernet (veth) pairs serve as a tunneling mechanism and function like virtual network cables, establishing connections between various network namespaces. P4Docker also allows the configuration of detailed characteristics of each connection, such as delay and bandwidth.

The script generated by P4Docker executes a sequence of operations to establish the test bed environment utilizing Docker containers and veth pairs, outlined as follows:

- Docker Containers Creation: The script initiates by instantiating isolated containers. Each container is configured without a predefined network and is endowed with elevated privileges, enabling system-level operations;
- Veth Pairs Creation: The script generates virtual Ethernet (veth) interface pairs. These interfaces act as virtual cables, establishing connections between two network points;
- Process Identification: The script retrieves each container process identifier (PIDs), facilitating direct manipulation of their network interfaces;
- Interfaces Association: The veth interfaces are associated with their respective containers, linking them virtually as per the intended topology;
- Network Configuration in Containers: The script assigns IP and MAC addresses to the interfaces within the containers and activates these interfaces, preparing them for network communication;
- Promiscuous Mode Configuration: The interfaces are set to promiscuous mode, allowing them to capture all network traffic, aiding certain network operations;
- Routes Configuration: Static routes are defined within the containers, delineating the packet forwarding pathways in the simulated network. By default, the default gateway of a container is the switch to which its first port is connected;
- Virtual Switch Initialization: the BMv2 switch is activated in the respective containers, creating a log file that can be used for real-time debugging and analysis of the switch behavior, allowing control of each package that passes through each switch;

- Forwarding Rules Addition: The script concludes by incorporating forwarding rules into the virtual switch, delineating the packet routing mechanism among the containers, thereby completing the network configuration.

3.4. Compiling

In addition to generating network topologies, P4Docker facilitates the debugging of P4 code through an integrated compilation tool embedded within the dashboard, as depicted in Fig.4. This compiler feature provides diagnostic feedback from the compilation process, enabling users to pinpoint and address errors within the P4 code. Furthermore, the interface allows users to download the compiled code, thereby allowing the replication of the current scenario for further examination and validation.

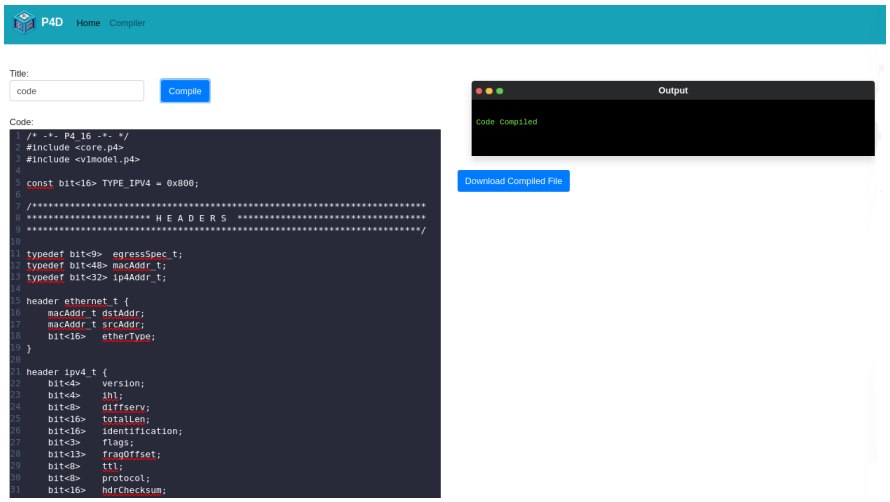


Figure 4. P4Docker-GUI Dashboard - Compiling P4 Code

3.5. Debugging

In addition to its compatibility with many network analysis tools such as Wireshark[3] and iPerf[4], a unique feature of the P4 switch, P4Docker, employed by this tool, is its capability for log analysis. Unlike most simulation tools currently supporting P4, which do not allow real-time log analysis, P4Docker enables comprehensive logging. Figure 5 demonstrates two terminals: the right terminal displays a host container executing a ping command to another container, with the traffic transiting through a P4Docker switch; the left terminal exhibits the corresponding log for each package. This feature empowers users to scrutinize each packet traversing the switch, enhancing their understanding of the impact of the P4 code on each packet processing. Such an analytical facility significantly aids in comprehending P4 data plane programming and switch behavior, contributing to a more profound learning experience in network programming.

4. Demo Description

The demonstration will be conducted using a virtual machine on which the P4Docker will be installed. The presentation will delve into its capabilities for creating network

[3]https://www.wireshark.org/
[4]https://iperf.fr/

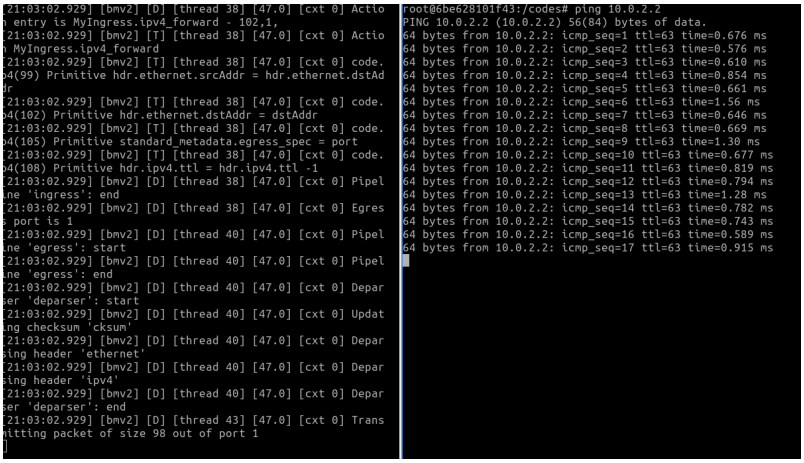


Figure 5. Debugging P4Docker switch

topologies using the dashboard, which includes adding nodes to the graph, establishing connections, and exporting the creation code.

Compiling P4 code using the compiler interface will also be addressed, illustrating its functionality and the ability to debug compiler output.

Subsequently, the script generated by the P4Docker will be executed, establishing the desired network topology using containers and veth-pairs.

Finally, the connectivity between hosts will be assessed by connecting to the terminal of one host and performing a connectivity test with a second host using communication through the P4 switch. Moreover, the method for accessing the P4 switch log to analyze its operations and understand its behavior will be demonstrated.

Source Code Repository: https://github.com/dnredson/P4Docker

Instruction video: https://youtu.be/P$_2jDUeSOI0$

Manual and Documentation: https://dnredsons-organization.gitbook.io/p4docker/

5. Conclusion

In this paper, we introduced P4Docker, a Docker-based P4 container with a Graphic User Interface, to mitigate the complexities associated with P4 programming and test bed configuration. Our tool significantly streamlines designing, deploying, and managing network topologies, enabling novice and experienced users to engage more effectively with P4-based network experimentation. Through an intuitive, graph-based interface, P4Docker automates the generation of necessary scripts for container management and network configuration, enhancing the accessibility and efficiency of network research and experimentation.

The utility of P4Docker was demonstrated through its ability to simplify complex network topology designs, facilitate rapid deployment and teardown of test beds, and support in-depth debugging and analysis of P4 code. These capabilities lower the entry barrier for individuals embarking on P4 programming and enhance productivity for seasoned practitioners.

Looking forward, we envision continuous enhancements to P4Docker, incorpora-

ting feedback from the community to introduce new features, improve user experience, and extend support for emerging network programming paradigms. Future work will aim to explore and compare the performance and architecture of P4Docker with other simulation tools like Mininet, expanding the scope and versatility of P4Docker in various network research and development scenarios. Future work may also explore more profound integration with other network simulation tools, SDN controllers, and components, expanding the scope and versatility of P4Docker in various network research and development scenarios.

Acknowledgements

This work was supported by the São Paulo Research Foundation (FAPESP), Brazil, under Grant 20/05152-7.

References

Bosshart, P., Daly, D., Gibb, G., Izzard, M., McKeown, N., Rexford, J., Schlesinger, C., Talayco, D., Vahdat, A., Varghese, G., and Walker, D. (2014). P4: programming protocol-independent packet processors. *SIGCOMM Comput. Commun. Rev.*, 44(3):87–95.

Chen, G., Hu, Z., and Jin, D. (2023). Enhancing fidelity of p4-based network emulation with a lightweight virtual time system. In *Proceedings of the 2023 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, SIGSIM-PADS '23, page 34–43, New York, NY, USA. Association for Computing Machinery.

Consortium, P. L. (2024). BMV2: Behavioral Model version 2 (P4 Runtime environment). Git repository.

Goswami, B., Kulkarni, M., and Paulose, J. (2023). A survey on p4 challenges in software defined networks: P4 programming. *IEEE Access*, 11:54373–54387.

Heideker, A., Silva, D., Kleinschmidt, J., and Kamienski, C. (2023). Otimização de tráfego iot-lorawan usando programação de plano de dados em p4. In *Anais do XLI Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos*, pages 239–252, Porto Alegre, RS, Brasil. SBC.

Jain, S. M. (2020). *Namespaces*, pages 31–43. Apress, Berkeley, CA.

Kfoury, E. F., Crichigno, J., and Bou-Harb, E. (2021). An exhaustive survey on p4 programmable data plane switches: Taxonomy, applications, challenges, and future trends. *IEEE Access*, 9:87094–87155.

Laki, S., Stoyanov, R., Kis, D., Soulé, R., Vörös, P., and Zilberman, N. (2021). P4pi: P4 on raspberry pi for networking education. volume 51, pages 17–21.

Liatifis, A., Sarigiannidis, P., Argyriou, V., and Lagkas, T. (2023). Advancing sdn from openflow to p4: A survey. *ACM Comput. Surv.*, 55(9).

Mininet (2023). Mininet: An instant virtual network on your laptop (or other pc). [Online; accessed 14-January-2024].