The Last WAVE: Integrating with Mininet for More Flexible and Reproducible Network Experiments

Ivalcleb L. Benigno de Souza, Paulo Ditarso Maciel Jr. and Leandro C. de Almeida

[1]Academic Unit of Informatics – Federal Institute of Paraíba (IFPB)
João Pessoa – PB – Brasil

ivalcleb.leoncio@academico.ifpb.edu.br,

{paulo.maciel,leandro.almeida}@ifpb.edu.br

Abstract. *Experimentation is essential in computer network research because it enables controlled and reproducible evaluation of applications and protocols. This paper presents the third version of WAVE (Workload Assay for Verified Experiments), evolving from the original multiple-workload generator and its later extensions with new load models and microburst support. WAVE controls the execution of real application instances over time according to mathematical workload models, currently supporting sinusoid, flashcrowd, and step patterns, as well as video-oriented workloads and microbursts. The main contribution of this version is the native integration with Mininet, which removes the previous integration barrier with simulated environments and allows workload generation and network emulation to be configured in a unified workflow. In this integrated setting, researchers can define different topologies, such as linear and tree, and inject realistic network parameters, including additional delay and packet loss. These capabilities provide a more flexible and reproducible environment for evaluating distributed applications and network behavior under diverse experimental conditions.*

1. Introduction

This paper introduces the third version of WAVE (Workload Assay for Verified Experiments), consolidating the trajectory started with the original multiple-workload generator proposed in 2023 [1] and extended in 2025 with new workload patterns and microburst support [2]. In this new stage, WAVE is integrated with Mininet [5], one of the most widely adopted network emulation environments in the computer networking community, which increases the flexibility and reproducibility of experiments by enabling multiple application-driven workloads to be executed under configurable topologies and network conditions.

WAVE currently supports three workload models, namely *sinusoid*, *flashcrowd*, and *step*, as well as microburst generation and native support for video-oriented application workloads. The platform combines mathematical load modeling with practical orchestration technologies, including a Web interface for experiment configuration and automated environment provisioning, so that researchers can reproduce experiments with real application traffic instead of relying only on synthetic packet generators. In previous

versions, however, integrating the workload generator into a simulated network environment such as Mininet required significant manual effort and ad hoc adjustments, which limited repeatability. This limitation is fully addressed in the present work.

With the Mininet-integrated version, WAVE now provides native support for defining multiple network topologies directly as part of the experiment workflow (e.g., *linear* and *tree*) and for configuring more realistic network conditions. In practice, researchers can inject additional delay and packet loss parameters into the emulated environment while controlling workload dynamics, enabling more faithful and systematic evaluations of distributed applications under diverse and reproducible scenarios. The WAVE source code is publicly available in a repository[1], which also includes a user manual[2] detailing the installation steps and usage instructions. In addition, demonstration videos of the tool can be accessed through the link[3].

The remainder of this paper is organized as follows. Section 2 describes the architecture, modules, functionalities, and workload-related capabilities of WAVE. The prerequisites for the WAVE demonstration and the intended presentations are explained in Section 3. Use cases that illustrate how WAVE can contribute to scientific research are outlined in the Appendix.

2. WAVE architecture

This section describes the modules and their technological components within the WAVE architecture. Currently, WAVE is composed of four main modules: Initialization, Web, Provisioning, and Monitoring, as illustrated in Figure 1. In addition, this section presents the integration of WAVE with Mininet, a network emulator that enables the creation of configurable topologies for the experiments.

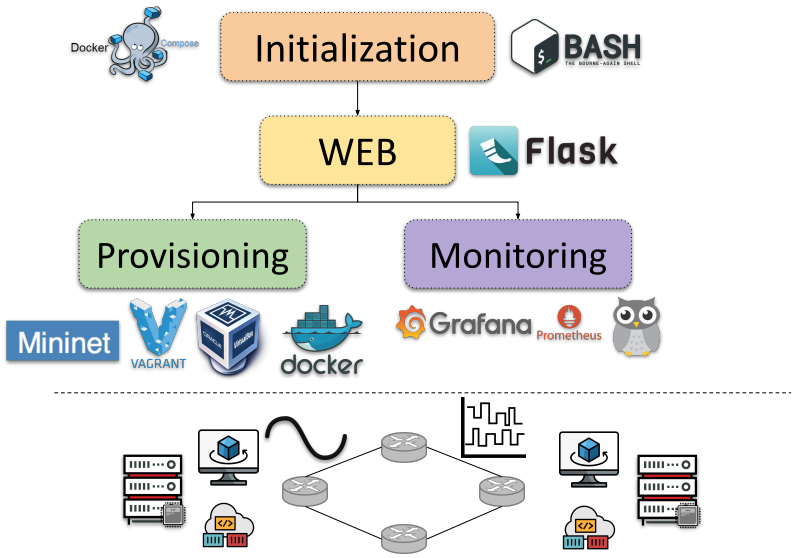


Figure 1. The WAVE architecture and its technological components.

[1]https://github.com/ifpb/last_wave
[2]https://github.com/ifpb/last_wave/blob/main/WAVE_User_Manual.pdf
[3]https://tinyurl.com/videos-WAVE

The Initialization and Web modules form the frontend; and are the ones the researcher interact with. The Provisioning and Monitoring modules form the backend, automatically executing the tasks required for provisioning and configuring the environment. These modules are triggered by the Web module based on the parameters defined by the user during the experiment configuration.

The Initialization module is responsible for configuring the environment and preparing the necessary components for the system execution. Its role is to instantiate all the other modules in the architecture. This module is executed through Docker Compose[4], which must be triggered from the command-line interface using a Bash script.

As a technological component, the Web module is developed using the Flask framework[5]. In its previous versions, WAVE Web already enabled interaction between the researcher and the system, allowing the definition of the parameters required for the execution of the experimentation environment and workload generation. In this version, the platform includes a Mininet Network Configuration card through which the researcher can select built-in topologies, such as *tree* and *linear*, and also provide custom topology definitions for experiment-specific scenarios. For each selected topology, WAVE Web collects the corresponding parameters and forwards them for automatic network provisioning, as illustrated in Figure 2. In addition, WAVE Web includes a communication API that interacts with the Provisioning and Monitoring modules. The API allows WAVE Web to transmit the parameters defined by the researcher for the configuration and execution of the experimentation environment and the workload.

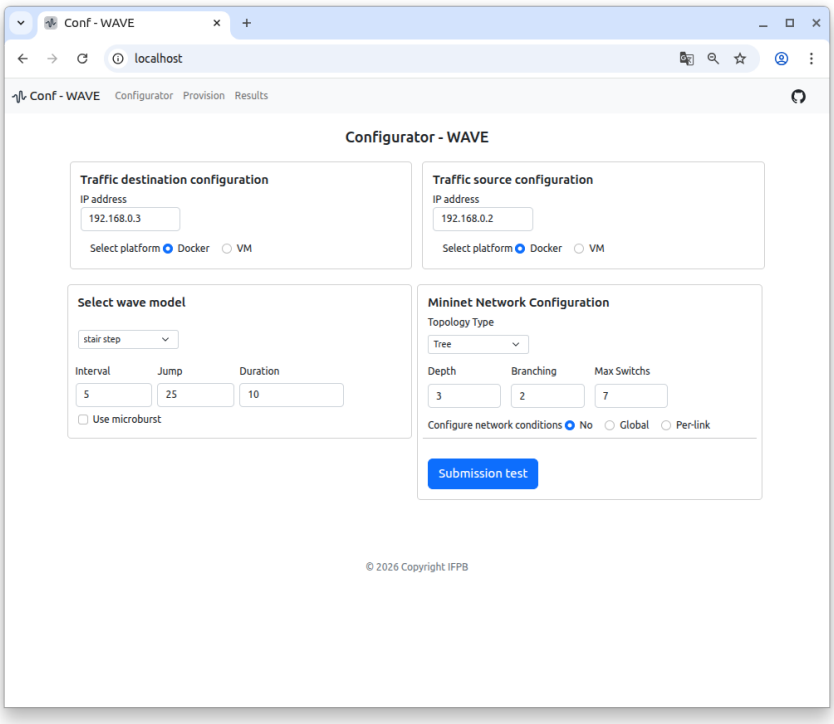


Figure 2. Web WAVE module receiving input parameters.

[4]https://docs.docker.com/compose/
[5]https://flask.palletsprojects.com/

The Provisioning module receives the parameters defined in WAVE Web to create the experimentation environment and execute instances that follow the workload model specified by the researcher. Currently, WAVE supports Vagrant[6], VirtualBox[7], and Docker[8] as technologies for provisioning the execution environment. In this new version, Mininet[9] is integrated into this provisioning flow, enabling the emulation of configurable networks together with workload execution. Client and server components can still be provisioned in containers or virtual machines, which simplifies orchestration and supports fast, scalable deployment of experimental scenarios. To clarify how this integration is implemented and how user-defined parameters are translated into network topologies, Subsection 2.1 details the role of Mininet in WAVE.

2.1. Integration with Mininet

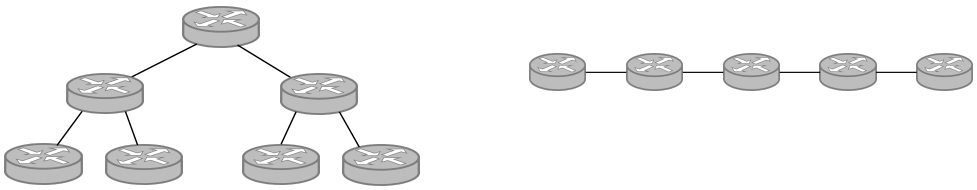
Mininet is a network emulator that creates a network of virtual hosts, switches, controllers, and links. Mininet hosts run standard Linux networking software, and its virtual switches support OpenFlow for highly flexible custom routing and Software-Defined Networking (SDN) [5].

Mininet uses process-based virtualization to run hosts and switches within a single Linux kernel. To achieve this, it employs resources such as *namespaces*, which allow the isolation of network interfaces and the network stack between processes. This enables each component (hosts, switches, etc.) in a Mininet network to have its own TCP/IP stack. Another resource used by Mininet is *veth*, which allows switches and hosts to be connected through virtual interfaces. These mechanisms enable the efficient emulation of complete networks on a single machine.

In WAVE, Mininet is used to enable the creation of configurable network topologies during experiments. To achieve this, the topologies are generated through Python scripts that use the Mininet API. This approach provides greater flexibility, allowing the dynamic configuration of network parameters, such as delay and packet loss, in addition to automating the process of creating the experimental environment. The scripts receive parameters defined by the user in the WAVE Web interface and, based on these values, automatically build the network topology.

In this version, WAVE can provision two topologies: *Tree* and *Linear*. Each of them has a specific set of parameters that defines its structure and size. The *Tree* topology receives three parameters: *depth*, which represents the depth of the tree; *branching*, which defines the number of branches at each level; and *max switches*, which limits the maximum number of switches. The *Linear* topology, on the other hand, has a much simpler structure and is defined only by the *number of switches*. Figure 3 presents examples of these topologies. A *Tree* topology with the parameters Depth: 3, Branching: 2, Max Switches: 7 can be observed in Figure 3(a), and a *Linear* topology with 5 components is shown in Figure 3(b).

[6]`https://www.vagrantup.com/`
[7]`https://www.virtualbox.org/`
[8]`https://www.docker.com/`
[9]`https://mininet.org/`

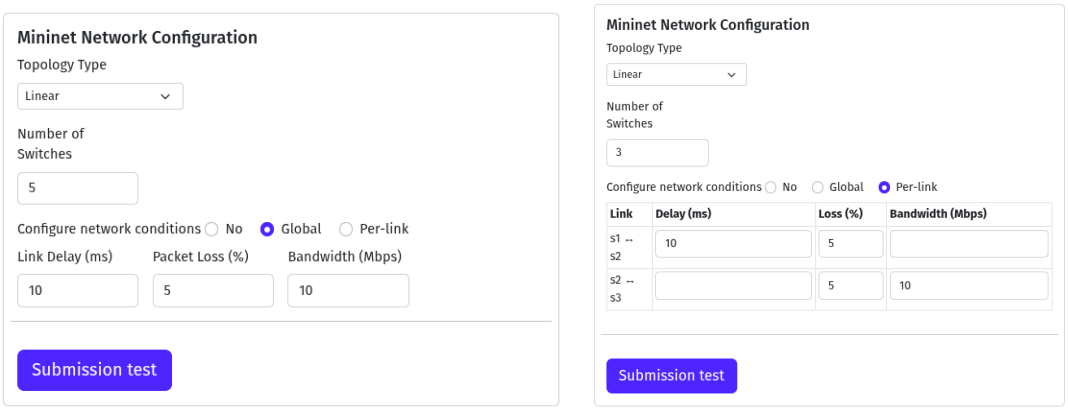


(a) Tree Topology (b) Linear Topology

Figure 3. Topologies currently supported by WAVE: (a) Tree with $depth = 3$, $branching = 2$, and $max\ switches = 7$; (b) Linear with $number\ of\ switches = 5$.

In addition to defining the network topology, WAVE also allows the configuration of link characteristics such as delay, packet loss, and bandwidth. This functionality enables the replication of various network scenarios by varying these parameters, allowing researchers to emulate different network conditions.

These configurations can be applied in two modes: Global and Per-link. In *Global* mode, the parameters specified by the user are applied uniformly to all links in the topology. The *Per-link* mode allows users to configure each link individually, enabling the creation of heterogeneous network environments in which different links exhibit distinct characteristics.



(a) Global mode. (b) Per-link mode.

Figure 4. Network configuration modes available in WAVE.

During the experiment configuration process, the parameters defined by the user are stored in a file called *config.yaml*. This file is later read by the scripts responsible for network provisioning. Depending on the selected topology, WAVE automatically executes the corresponding script: *net-tree.py* for *Tree* topologies and *net-linear.py* for *Linear* topologies. These scripts use the Mininet API to instantiate switches and create links according to the defined parameters. Below is a snippet of the *config.yaml* file.

```
- topology:
    type: "tree"
    depth: "3"
    branching: "2"
    max_switches: "7"
```

After the researcher clicks “provision” in the WAVE Web interface, the system starts the provisioning process. First, the network is instantiated in Mininet. Then, the client and server components are started. Finally, the workload models are executed and the monitoring module collects the metrics generated during the experiment.

The Monitoring module is responsible for performing measurements within the experimentation environment. By default, WAVE Web displays the number of active application instances over time, the rate of bytes received on the network interface, as well as the CPU and memory usage of the virtual components, as illustrated in Figure 5. Additional metrics can be included with small configuration adjustments. This module is implemented using Prometheus[10], Grafana[11], and cAdvisor[12].

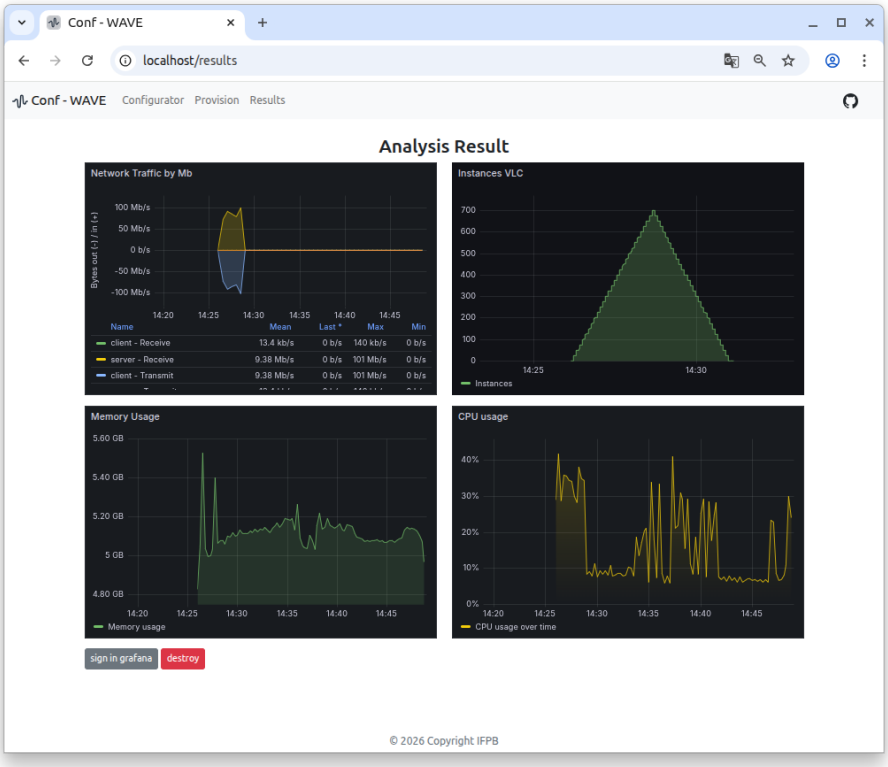


Figure 5. Monitoring module that displays network traffic, the number of video player instances, and resource usage.

In this section, we presented an overview of the WAVE architecture, describing its modules and technological components. Each module plays a specific role in work-

[10] https://prometheus.io/
[11] https://grafana.com/
[12] https://github.com/google/cadvisor

load generation and experiment monitoring. In addition, the technology stack aims to ensure scalability, efficiency, and ease of integration with different experimentation environments. In this way, WAVE provides a robust and flexible solution for researchers and professionals seeking to simulate and analyze various network scenarios.

3. Demonstration

For the *in-loco* demonstration, only a table, electrical power, and either a monitor or a projection screen will be required. The session is organized to show the complete workflow of WAVE in a reproducible and didactic way, including environment startup, experiment configuration, workload execution, and monitoring of runtime metrics.

The demonstration begins with a short video that introduces the platform and summarizes the architecture presented in this paper. Immediately after that, the live execution starts with the initialization of the WAVE modules and access to the Web interface. Participants will follow the full sequence of actions: selecting workload parameters, choosing a Mininet topology, applying network conditions (e.g., delay and packet loss), and triggering provisioning.

Once the experiment is running, participants will observe how workload dynamics and network parameters affect the monitored indicators in real time, such as active instances, traffic behavior, CPU usage, memory consumption, RTT, and throughput. This stage highlights the practical integration between workload orchestration and network emulation, allowing direct comparison between baseline and constrained scenarios.

Finally, the session concludes with a brief discussion of reproducibility. We will show how the same configuration can be replayed, how outputs can be compared across runs, and how the generated evidence supports experimental analysis in networking research. During the interactive portion, participants will be encouraged to modify selected parameters and inspect the impact on the observed results.

References

Almeida, L., Silva, J., Lins, R., Maciel Jr., P. D., Pasquini, R., and Verdi, F. (2023). WAVE - Um gerador de cargas múltiplas para experimentação em redes de computadores. In *Anais Estendidos do 41º Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos*, pages 9–16, Brasil. SBC.

Beuttenmuller, D. C., Valério, M. F. d. A., Silva, C. L. L. T., Da Silva, I. M., Maciel Jr., P. D., and De Almeida, L. C. (2025). A new wave: Exploring new load pattern models for experimentation in computer networks. In *Anais do Salão de Ferramentas do Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos (SBRC)*, pages 11–22, Natal/RN. Sociedade Brasileira de Computação. ISSN 2177-9384.

Hafez, N. A., Hassan, M. S., and Landolsi, T. (2023). Reinforcement learning-based rate adaptation in dynamic video streaming. *Telecommunication Systems*, 83(4):395–407.

Kim, M. and Chung, K. (2022). Reinforcement Learning-Based adaptive streaming scheme with edge computing assistance. *Sensors (Basel)*, 22(6).

Lantz, B., Heller, B., and McKeown, N. (2010). Mininet: An instant virtual network on your laptop (or other pc). In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, pages 1–6.

Lin, H., Shen, Z., Zhou, H., Liu, X., Zhang, L., Xiao, G., and Cheng, Z. (2020). Knn-q learning algorithm of bitrate adaptation for video streaming over http. In *2020 Information Communication Technologies Conference (ICTC)*, pages 302–306.

Sandvine (2023). Global Internet Phenomena. Technical report, Sandvine.

Spang, B. et al. (2023). Sammy: Smoothing video traffic to be a friendly internet neighbor. In *Proceedings of the ACM SIGCOMM 2023 Conference*, page 754–768, New York, NY, USA. Association for Computing Machinery.

Wei, X. et al. (2021). Reinforcement learning-based QoE-oriented dynamic adaptive streaming framework. *Information Sciences*, 569:786–803.

APPENDIX: USE CASES

In the context of workload generation tools for computer networks, defining clear use cases is essential to demonstrate the applicability and relevance of the proposed solution. In this section, we briefly present two use cases of WAVE. The first use case addresses a video streaming scenario in a CDN (Content Delivery Network), while the second analyzes the impact of changing Mininet parameters on experimental behavior and performance metrics. Together, these cases illustrate how the tool supports reproducible evaluations under distinct workload and network conditions.

Video Streaming Scenario

Video-on-demand streaming accounts for approximately 60–75% of all Internet traffic [7], attracting significant attention from the scientific community [6, 9, 4, 3, 8]. WAVE can control video player instances to simulate thousands of viewers consuming content through a CDN. When many users request the same video simultaneously, edge servers face increased concurrent connections and substantial outbound traffic. This workload pattern can lead to bandwidth saturation, increased latency, and service degradation if the infrastructure is not properly provisioned.

Another key aspect of workload generation in CDN-based video streaming is dynamic adaptation to demand fluctuations. Events such as live broadcasts, product launches, or viral content can cause sudden traffic spikes, requiring real-time resource scaling. In response, CDNs employ intelligent traffic routing, dynamic resource allocation, and predictive analysis to optimize workload distribution and prevent service disruptions.

To demonstrate this new WAVE functionality, we created an experimental network environment using WAVE integrated with Mininet. In this environment, we deployed a tree topology with 7 switches, representing the communication path between a client and a server responsible for delivering video content. Figure 3(a) presents the topology configured and used in the experiment, illustrating the tree structure and the interconnection between network elements. We used the *stair-step* workload model (*interval* = 5, *jump* = 10, and *duration* = 10) to analyze how WAVE, integrated with Mininet, supports controlled and reproducible evaluations under different network conditions. Figure 6 illustrates a scenario with approximately 400 simultaneous video clients generating load in the experimental environment, a scale that can easily represent a neighborhood or even a small city consuming video streaming services.

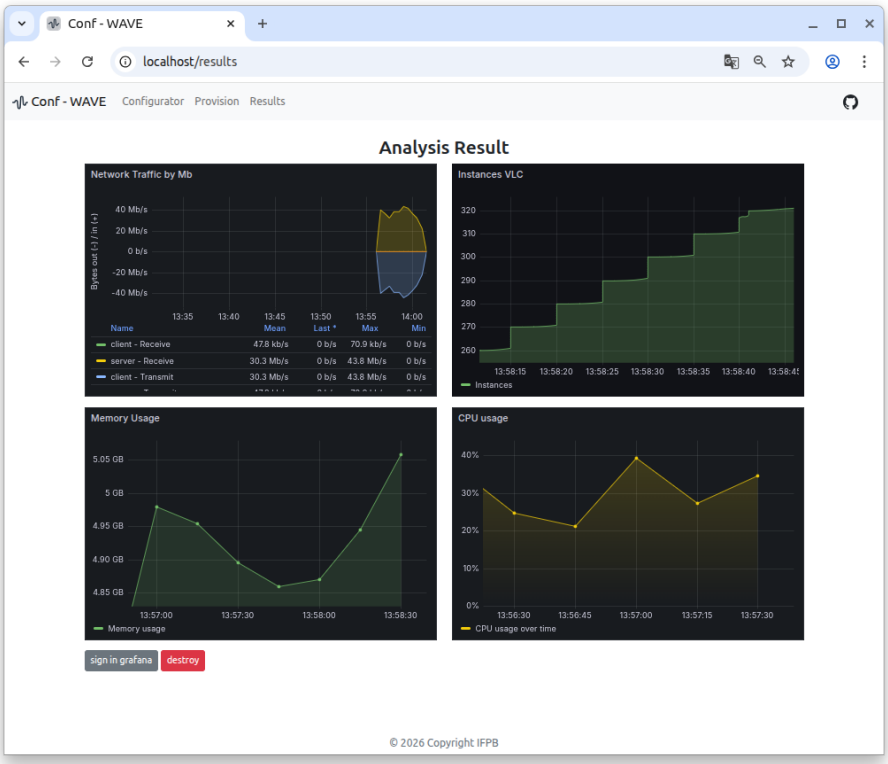


Figure 6. CDN-oriented experiment with approximately 400 simultaneous video clients generated by WAVE. Beyond controlling video instance execution, the monitoring module also provides complementary metrics, such as throughput, CPU usage, and memory consumption.

Impact of Mininet Parameters

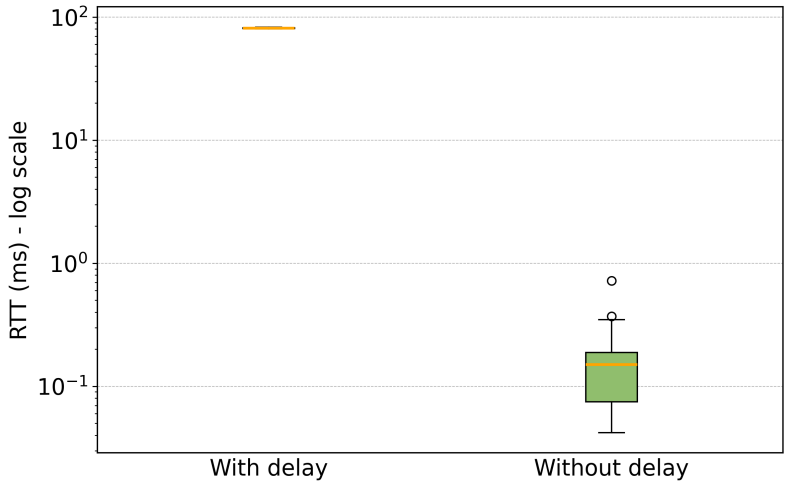
In this subsection, we conduct experiments aimed at exploring the capability of WAVE to manipulate Mininet parameters during the execution of network scenarios. For this analysis, variations in the *delay* and *packet loss* parameters were considered, as these factors directly affect latency-sensitive and reliability-sensitive applications, such as video streaming services. To compare and quantify the impact of these configurations, we measured RTT, end-to-end throughput, and packet loss rate under different network conditions.

The experiments were performed using the linear topology from Figure 3(b), composed of 5 switches. Initially, two distinct scenarios were evaluated, varying only the *delay* parameter: in the first scenario, a delay of 10 ms was configured, while in the second, a delay of 50 ms was applied. Subsequently, a second set of tests was conducted using the same topology and the same number of components, but varying the packet loss rate. In this case, two scenarios were defined: one with 1% *packet loss* and another with 15%. In all evaluated scenarios, the *stair step* load model was used, configured with the parameters *interval* = *5*, *jump* = *10*, and *duration* = *10*.

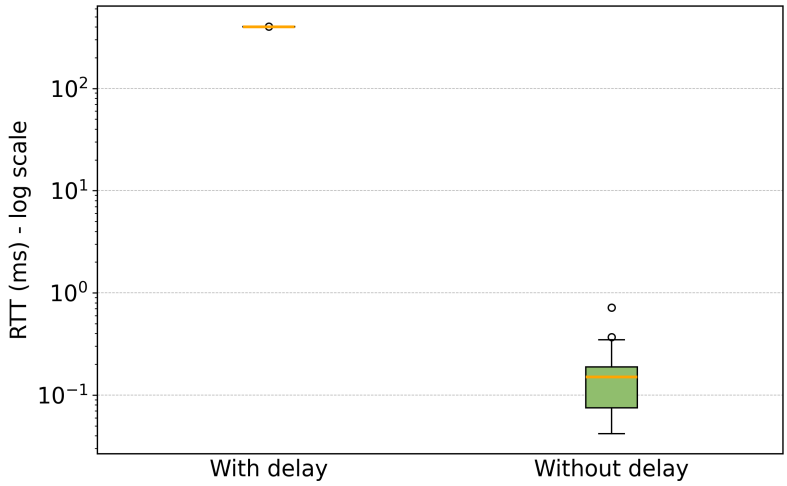
The boxplots comparing the baseline scenario with the delay configurations of 10 ms (Figure 7(a)) and 50 ms (Figure 7(b)) reveal a consistent and well-defined separation between the RTT distributions. In the absence of delay, RTT values remain strongly

concentrated at the lower end of the scale, reflecting minimal latency and low variability. As delay is introduced, the distributions exhibit a clear upward shift, with the 10 ms configuration resulting in RTT values on the order of tens of milliseconds, while the 50 ms configuration shifts the distribution even further to the order of hundreds of milliseconds. This progression demonstrates a proportional and deterministic relationship between the configured delay and the observed RTT.

Furthermore, the relatively narrow interquartile ranges and short whiskers across all scenarios with delay indicate low variability, suggesting that the emulated network maintains stable latency conditions over time. The absence of significant dispersion or irregular outliers reinforces the reliability of the measurements. These results confirm that the emulation environment not only applies the configured delay parameters but also preserves consistency and reproducibility, enabling a precise quantitative analysis of network behavior under controlled latency conditions.



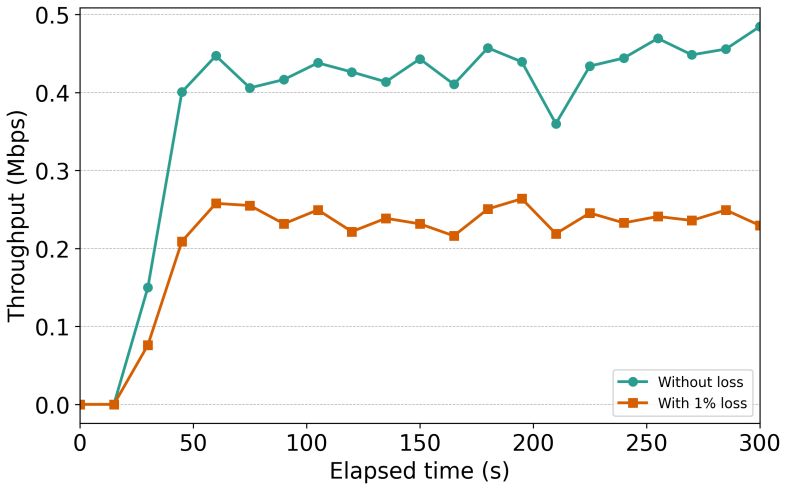
(a) 10 ms delay.



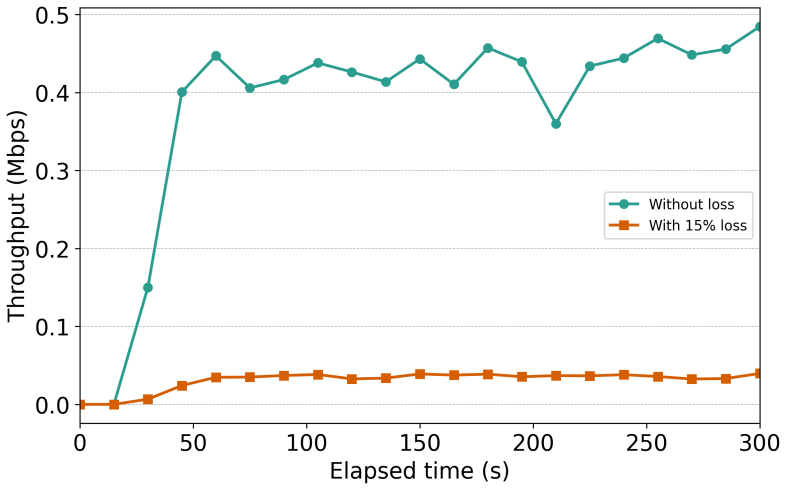
(b) 50 ms delay.

Figure 7. RTT comparison between scenarios with injected delay and without delay.

The next set of experiments evaluates the effect of packet loss on network performance. For this purpose, two scenarios with loss rates of 1% and 15% were considered. The throughput graphs present a comparison between these scenarios, considering that packet loss was configured in Global mode and accumulates across the five switches present in the topology. In the scenario with 1% loss, the observed throughput drops to approximately half the value obtained without loss, remaining stable over time with low variability (Figure 8(a)). In the scenario with 15% loss, throughput suffers significant degradation; it can be observed that throughput remained in the range of 0.01 to 0.06 (Figure 8(b)), indicating that the accumulation over multiple hops significantly increases network degradation. The low variability in both loss scenarios reinforces the consistency of the emulation environment.



(a) 1% packet loss.



(b) 15% packet loss.

Figure 8. Throughput comparison between scenarios with packet loss and without loss.