UNIVERSITÄT DES SAARLANDES

Faculty of Mathematics and Computer Science
Department of Computer Science

End-to-End Network Protocol Evaluation with Modular Simulation

Master's Thesis

written by

Marvin Meiers

December 5, 2024

1st Reviewer
Dr. Antoine Kaufmann

2nd Reviewer
Dr. Laurent Bindschaedler

Erklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Statement

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis.

Einverständniserklärung

Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

Declaration of Consent

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Saarbrücken, ________________ ________________________
(Datum/Date) (Unterschrift/Signature)

Abstract

Computer networks form the backbone of modern computing, whether in data centers or for the communication of end devices with servers for the multitude of Internet-based applications. In data centers in particular, ever more powerful servers and the trend towards distributed applications that communicate a significant amount with each other have greatly increased the demands on the network. In order to meet these requirements, network technologies are constantly evolving and there is a trend towards more diverse network systems with specialized components and network protocols. In the course of further research and development of these systems, there is a need to be able to test and evaluate them. Physical testbeds are often neither available nor feasible, so researchers turn to network simulations, which however fail to capture the complex intricacies of today's network systems. End-to-end simulations are able to adequately model all relevant components of the network, but they do not achieve the required scale with practical computational resources and simulation performance.

This thesis introduces a framework that addresses the drawbacks of full-system simulations. It builds on previous work on modular full-system simulation and provides easy-to-use elements that allow the simulation to be scaled further. First, mixed-fidelity simulations allow choosing the level of simulation detail for different components. This allows less relevant components to be simulated with less detail, thereby reducing computational resources. Next, the framework allows a large network topology to be split into multiple partitions and simulated in separate, synchronized network simulator instances. This prevents the network simulator from becoming a bottleneck and slowing down the entire simulation. Finally, the framework provides abstractions that allow complex large scale network simulations to be created without having to configure the individual simulators directly.

The evaluation shows that the framework offers a practical solution for large scale full-system simulations with feasible computational resources and simulation performance. It demonstrates that mixed-fidelity simulations and decomposing bottleneck network simulators are easy to implement with our framework and reduce both simulation time and needed resources, while still providing sufficient accuracy.

Acknowledgements

First, I would like to thank my advisor Antoine Kaufmann for his support and advice throughout my work on this thesis.

I also want to thank all members of the OS group for the interesting discussions we had and all the welcome distraction from work. Especially, I would like to thank Hejing and Jonas, with whom I worked closely on SimBricks and had many fruitful discussions.

I thank all my fellow students who helped me during my studies at Saarland University, especially Jakob and Luca. I really enjoyed countless lunches and the time we spent together outside of university as well as working together on many exercise sheets and projects.

Finally, I thank my family for their support. My parents Robert and Astrid as well as my brother Dominic have been a constant source of support and encouragement throughout my life.

Contents

Chapter 1

Introduction

1.1 Rapid Development of Networks

Computer networks are the backbone for today's computer applications. They are essential for any communication between different computers which is crucial for the vast majority of modern applications. With the increasing amount of data that modern applications process, the demand for efficiently moving huge amounts of data is rising quickly. This requires fast development and improvement of the networking technology including both hardware and software to keep up with the demand.

Over the last 60 years, there have been many further developments in hardware for computer networks. On the one hand, manufacturing processes for hardware components have constantly evolved, making computers and network components ever more powerful and able to process and exchange larger amounts of data faster. In order to maximize performance, increasingly specialized network components have been developed, such as SmartNICs [7, 8, 14] or programmable network switches [3, 18]. In addition, transmission technologies have continued to improve and new technologies such as fiber optics have been developed.

In addition to the hardware, the software has also evolved in order to make efficient use of the improved network resources and the increasingly capable and complex hardware. This includes improvements to established network protocols such as TCP [1, 4] or the development of new specialized network protocols like Homa [20].

These developments are particularly visible in data centers, where there is a great need for high-performance networks for fast communication between servers. Due to the special requirements in a data center for high throughput and low latencies, there are many developments and innovations for this special use case that aim to maximize performance.

1.2 Evaluating Networks Is Difficult

While we develop new hardware or software for networks, we have to test and evaluate them at some point. However, this is not an easy task, as networks are complex systems made up of many types of components, including computers, routers, and network switches. For each of these components there is additionally a great variety in terms of hardware and software available, such as various types of network switches that can be deployed in the network. In addition, networks can consist of thousands or even millions of individual components that are connected to each other.

The overall behavior of a network is a combination of many different individual components' behavior and the interactions between them. The interconnected nature of a network means that we cannot look at individual components in isolation because they naturally influence each other through their interactions. Therefore, properly evaluating and testing a network requires us to capture the behavior and interactions of many pieces in order to capture the behavior of the whole network. Because of this complexity, we cannot easily predict how a new network switch or a new transport protocol will affect the performance of the network. This means that in order to evaluate a network, we must look at the entire network as a whole, even if we only change one part of the network.

1.3 Current Approaches Are Insufficient

In section 1.2 we have already established that testing and evaluating a network is a difficult task. Unfortunately, the solutions available to us today are not sufficient to adequately solve the problem. They often lack either flexibility, precision or a deep insight into the system.

Physical testbeds, for instance, are often infeasible [32] because of high costs and a long time investment to set them up, if there is not already a fitting network system in place that could be used for this purpose. Moreover, working with a physical system also means that we might have to swap physical components and configure actual computers, network switches and other hardware, which makes changing and adapting such a system a complicated process. This is especially true when we need to do this at a large scale with thousands of components. Another consequence of a physical testbed is that we need to have all physical components available, which rules out this approach in early-stage development of hardware when there is no physical prototype available yet. Lastly, although a physical testbed allows us to observe the overall behavior of the network, it is often hard to obtain in-depth insight into the system in order to also understand and explain the behavior.

There are also approaches that model the behavior of a network, like for example network simulators. However, they only focus on some parts of the overall system and use abstractions to deal with the complexity of the system. Consequently, they do not manage to model all intrinsic details of the network. This makes network simulators insufficient to model all involved components with great detail, preventing true end-to-end evaluations of the network. Instead, we have to model and evaluate different pieces of the complete system separately, basically in isolation, and then combine the results afterwards. Because of the tight integration of the components, we might miss important aspects of the interplay so that they are not reflected in the pieced together results. Additionally, network simulators often do not scale for large network systems [32], leading to very long simulation times.

Full-system simulation combines network simulators with dedicated detailed component simulators in order to provide end-to-end evaluations of the network. However, when scaling to large network systems full-system simulation needs many simulator instances requiring infeasible amounts of computation resources.

Current evaluation approaches therefore struggle to capture the behavior of diverse large scale network systems, especially involving highly specialized hardware, like programmable switches or SmartNICs. Large scale network simulations are needed, for example, for testing and evaluating network protocols, since they naturally involve thousands of hosts. Physical testbeds require the specialized hardware to be physically available and installed. To scale them, we need to purchase and install more components, which requires a lot of money, space and time. Simulators lack detailed models for the specialized hardware and therefore fail to capture the end-to-end behavior of the network and do not scale for large networks. Full-system simulation provides end-to-end evaluations but requires large amounts of resources for large scale simulations. Lastly, the current solutions are difficult to implement, as with physical testbeds, or they have a steep learning curve, like it is the case for many simulators.

1.4 Contribution

This thesis addresses the shortcomings of full-system simulation and for that proposes a framework to build, configure, and run large scale full-system network simulations with feasible resource requirements. To this end, the framework offers various features to scale the simulation efficiently and reduce the required computation resources. On the one hand, components in the simulated system can be simulated with different levels of detail by choosing which simulator to use to model the component. These mixed-fidelity simulations enable more resource efficient simulations by moving non-critical components into less detailed simulators which also require less computation resources. On the other hand, the large network topology can be split into multiple parallel, synchronously simulated partitions, preventing the network simulator from becoming a bottleneck and slowing the simulation drastically down. For that, the framework provides an abstraction for the simulated network which is integrated into the full-system simulation framework SimBricks [16]. The abstraction offers a practical approach to implement and configure mixed-fidelity simulations and parallelize bottleneck network simulators without requiring manual configuration of each underlying simulator.

1.5 Prior Publications

Parts of the results of this thesis have been submitted for publication and have been included in a corresponding preprint [17]. The publication includes the network simulator abstraction to configure mixed-fidelity network simulations and to decompose bottleneck network simulators. It uses the framework that is included in SimBricks and ns-3 for the configuration and orchestration of large scale network simulations. Finally, the evaluation of the publication makes use of the framework for experiments using network simulations.

Chapter 2

Background

2.1 Network Evaluation

There exist many ways to evaluate and test networks. We can mainly distinguish between two major approaches: physical testbeds and simulation. Each of them comes with its own set of advantages and disadvantages. In the following, we will take a closer look at the different approaches.

2.1.1 Physical Testbed

A physical testbed is a real network system with actual physical components that runs the complete software stacks. The advantage of a physical testbed is that it can provide realistic results and it runs in real time allowing fast evaluations. The usefulness of the results however depends on the fact whether the testbed is representative of the system that will actually be used later. For example, when we use a small testbed it can be difficult to extrapolate the results to a large scale system, because we might not observe certain behaviors of the larger system in the testbed. Therefore, we ideally need a testbed that also corresponds to the later system, which is also a disadvantage of physical testbeds, since a large scale physical system is often neither available nor feasible [32]. Therefore, researchers and developers usually resort to smaller testbeds, possibly limiting the usefulness of the results. Further, physical testbeds do not give us unlimited insight into the system and limits what we can observe from the system. We are normally able to observe high-level behavior of the system like throughput or latency, but it might not be possible to observe complex behavior like caching with sufficient detail. Lastly, a physical testbed is not portable, since we require the physical system to use it. Both moving and rebuilding or cloning the system is complicated and infeasible in the common case.

2.1.2 Simulation

Researchers and developers typically resort to simulation, in order to overcome some drawbacks of physical testbeds. Simulation aims to model the behavior of the network system in software. The advantage of simulation compared to physical testbeds is, that we do not need to have the actual physical system available, and it even gives us the possibility to model a component, that does not physically exist yet. Another advantage of simulation is its flexibility in terms of configuring the system. We can simply change configuration parameters, add more components, and choose between different components in software. This is easy to implement compared to a physical testbed where we have to carry out the changes in a physical system. It also allows us to scale the simulated system from just a few components up to thousands of hosts. Since the simulation is done in software, we are able to execute it basically on any computer, which makes it portable and enables researchers and developers to reproduce and compare results. Furthermore, simulation can provide in-depth insight into the system depending on the level of detail of the simulation. With simulation, we can capture any of the behavior that is modelled by the simulator, for example by recording changes of variables or functions calls, without affecting the behavior of the simulated system. But on the other hand, long simulation times are a typical disadvantage. Depending on the simulator and how detailed the simulation is, it can take many hours or even days to simulate just a few seconds of a large network. The larger the simulated system, the longer the simulation typically takes. In addition, due to the abstractions of the network simulators, it is usually not possible to run unmodified real-world applications in the simulation to generate authentic traffic. Finally, the results of a simulation might not reflect the behavior of a physical system. This depends on how well the simulator is able to model the real system. In order to ensure that the results are representative, we need to validate them against a physical testbed.

There already exist many network simulators that offer different levels of detail, simulation complexity and simulation times, such as ns-3 [21], OMNeT++ [12], and htsim [11]. Network simulators typically use discrete event simulation to model how packets traverse the network, which is a good fit for packet-switched networks.

2.1.3 Mix of Simulation and Physical System

Network simulators have to properly model the system with sufficient detail in order to deliver representative results. This might require a lot of implementation effort and lead to a complex simulator and long simulation times. This is especially true if we want to use real-world applications or an already existing implementation of a new network protocol, which we first have to adapt and integrate into the simulator. To solve this problem, we can integrate physical components into the simulation and execute an application or network protocol as we would run it in a real system. We then bind this to the simulation by intercepting the physical system at some point where we establish a bidirectional communication. The advantage with this approach is that we can run complex software or even integrate special hardware without having to model them in the simulation. But while this approach covers the functional behavior of the physical parts, we cannot accurately reason about the performance of these parts anymore.

For example, the network simulator Dummynet [25] combines execution on a physical system for applications and protocol stacks with the simulation of certain network features like queues or bandwidth limitations. There also exists the framework Direct Code Execution (DCE) [27], which integrates into the network simulator ns-3 [21]. DCE provides a way to run almost unmodified applications and network protocols under Linux in a ns-3 simulation by executing them almost natively on the machine.

2.2 SimBricks

SimBricks [16] is a flexible simulation framework that enables end-to-end evaluations of computer systems. For that, SimBricks combines already existing simulators for different components of a computer system, for example processors, memory, and networks, into one overall simulation. To this end, SimBricks implements adapters for each simulator which allows them to efficiently exchange data using shared memory queues and synchronize with each other for accurate timing. Each instance of a simulator is executed in its own process and communicates with other simulator processes through the SimBricks adapters.

SimBricks' design allows users to run full-system simulations that simulate all components of a system and in turn allow capturing the complete picture which we will further discuss in subsection 2.2.1. In subsection 2.2.2 we describe the modular approach of SimBricks where different simulators can be flexibly combined and interchanged. Lastly, we touch upon the scalable nature of SimBricks in subsection 2.2.3 that allows creating large simulations.

2.2.1 Full-System Simulation

There are many simulators that are capable of simulating various computer system components. But in practice there is not one single simulator that is capable of simulating all system components at once. Therefore, SimBricks combines multiple simulators into one overall simulation, where each simulator is responsible to model one component of the system. The result-

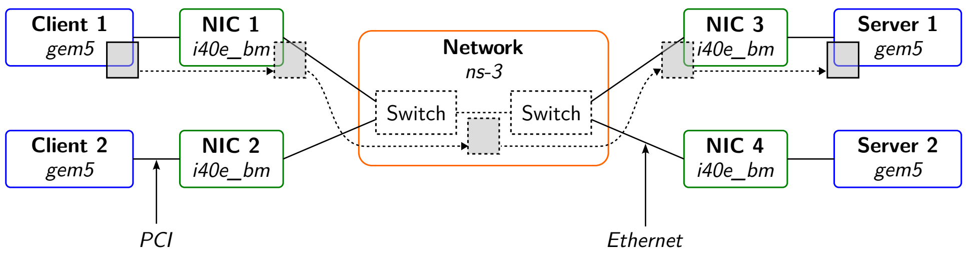


Figure 2.1: Overview of a full-system simulation with four hosts and their NICs connected to a network. It shows a packet travelling from Client 1 to Server 1. Based on Figure 2 in [16].

ing full-system simulation enables the user to test and evaluate the system end-to-end, instead of testing and evaluating each component in isolation.

Figure 2.1 shows how SimBricks connects simulators at natural boundaries, like Ethernet for network connections or PCI for connections between devices and hosts, into a full-system simulation. This works well in practice, since most simulators model system components up to the point of a well established common interface where the component would then connect to another component of the system. The system shown in Figure 2.1 consists of four host simulators that are each connected via PCI through SimBricks to a NIC simulator, which in turn connects to a network simulator via Ethernet. This setup allows capturing the behavior of the full system from complex host behaviors like caching to routing and queuing of network packets in the network. It also captures the interactions between the individual components through the SimBricks channels. For example, a packet that is generated by an application on Client 1 travels through the client's NIC and the network to the server's NIC and finally arrives at Server 1, involving five simulator instances working together.

2.2.2 Modular Simulation

Since SimBricks implements communication between simulators leveraging common protocols like Ethernet and PCI, simulators can be flexibly connected with each other in a modular fashion. For two simulators to be connectable they must both have a SimBricks adapter that implements the same protocol. This means, that after we have implemented an adapter for a simulator using some protocol, we can connect this simulator to any other simulator implementing an adapter using the same protocol. In particular, this eliminates the need to implement special adapters for each pair of simulators, keeping the implementation effort low. Consequently, simulators can be connected via SimBricks in the same way as their corresponding system components in a real computer system.

Furthermore, for most of the system components there exist not only one

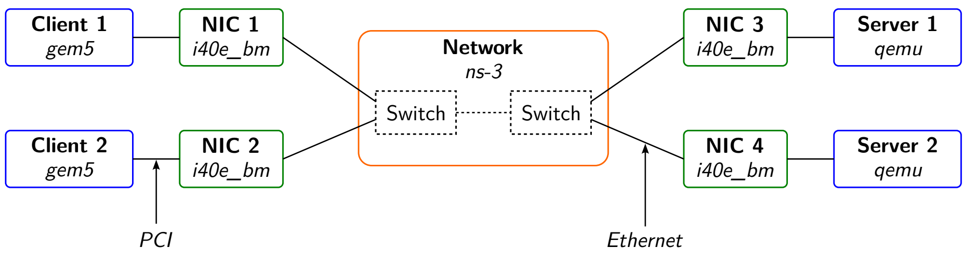


Figure 2.2: A full-system simulation that has the same structure as the system in Figure 2.1, but uses qemu instead of gem5 to simulate the servers.

single simulator that is capable of simulating it. Instead, various simulators have emerged over time, which offer different trade-offs in terms of accuracy, resource requirements and scalability. SimBricks' modular approach gives us the additional ability to easily choose between different simulators for one system component and switch between them with little effort.

Figure 2.2 showcases how the modular approach lets us choose and combine different simulators for the simulated components. While we are simulating the clients with the host simulator gem5, we pick the host simulator qemu for simulating the servers. Note that the structure of the simulators is still the same compared to Figure 2.1, with the only difference that we selected other simulators for some components.

2.2.3 Scalability

The design of SimBricks makes it easy to scale simulations. By running each simulator instance in its own process, SimBricks naturally parallelizes the full-system simulation. In order to scale a simulation, we simply add more components to the simulated system. Each of the added components is then simulated by an additional simulator instance. This however also increases the number of physical CPU cores that are need to execute the simulation.

Although SimBricks enables us to efficiently scale simulations by adding more processes, the resources of a computer are limited. In order to scale the simulation beyond the resource limits of a single machine, SimBricks distributes simulator processes across multiple machines. For the communication of simulator processes on different machines SimBricks uses proxies to transparently forward communication over the network.

Chapter 3

Design

3.1 Design Goals

The goal of the presented framework is to enable practical full-system network simulation at a large scale. To address the challenges of large scale full-system network simulations, we have the following design goals:

- **Scalable:** simulate large scale networks with thousands of components within manageable requirements for compute resources.
- **Efficient:** run the simulation efficiently without high overheads and avoid individual simulators becoming a bottleneck and slowing down the simulation.
- **Flexible:** choose flexibly which simulator to use for which component and easily switch between different simulators for a component.
- **Easy to use:** offer a simple interface to compose the full-system simulation and allow for easy changes between simulators.

3.2 Design Overview

3.2.1 Approach

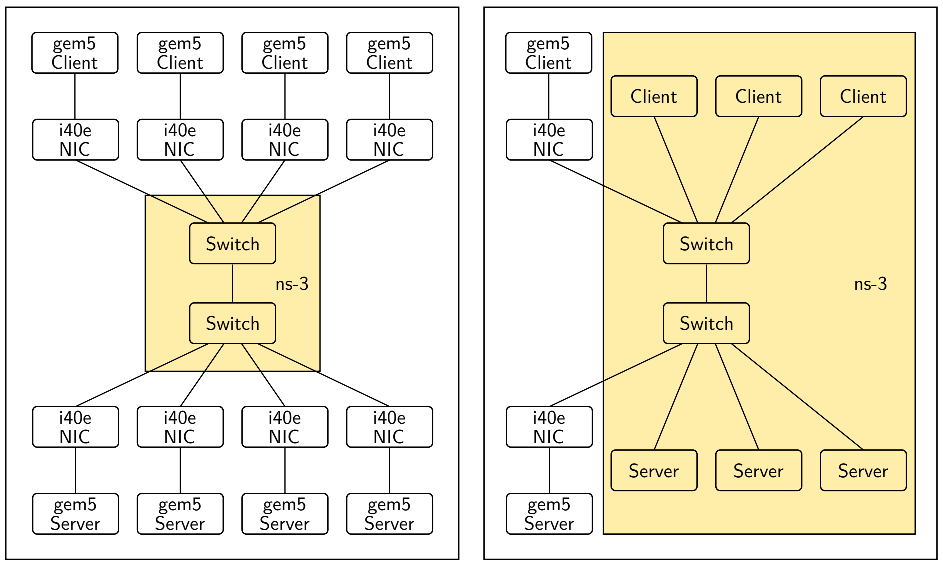
SimBricks [16] allows us to compose and run full-system network simulations, by simulating all components of the network with detailed component simulators and combining them into one overall simulation. However, when scaling such a full-system simulation to a large network with thousands of components, this requires many simulator instances leading to a high demand for compute resources. Due to the design of the SimBricks adapters which rely on polling, we need one CPU core for each simulator process, which means that we need hundreds or thousands of CPU cores for large scale simulations.

In the following, we will discuss three approaches that help us to overcome the limitations of large scale simulations. First, our framework enables mixed-fidelity simulations, where we use detailed but resource intensive simulators for one part of the system and less detailed but more resource efficient simulators in other parts. Secondly, we make more efficient use of the available resources by parallelizing bottleneck simulators that would otherwise slow down the entire simulation. Lastly, the framework provides programming abstractions to assemble and configure these large scale full-system simulations, making them practical to use.

Mixed-Fidelity Simulations

In order to overcome the limitations of large scale full-system network simulations, we propose to simulate some parts of the simulation with less detailed but more efficient simulators, while only simulating essential parts of the network simulation with the full level of detail, resulting in *mixed-fidelity* simulations. This can save substantial compute resources and still provide sufficient detail and insight into the simulation through the detailed simulators. The idea is that we move less critical areas of the system to less detailed simulators, so that it does not affect the accuracy of our simulation results. An example where this is useful is during the evaluation of a network protocol in a large network. In order to create a representative environment, we must generate realistic traffic in the network. This allows us to observe, for example, the effects of background traffic or congestion on the evaluated network protocol. However, we usually do not need a detailed simulation of the hosts that generate the background traffic, so that a protocol-level simulation modelled by a less detailed simulator is sufficient for those hosts.

For which components we use detailed simulators and for which less detailed simulators depends on the specific use-case and evaluation goal. This must therefore be decided on a case-by-case basis, and we might even want to take different approaches for the same basic simulation. For example, when we look at a client-server system we might want to achieve different evaluation goals. On the one hand, we can evaluate the peak throughput of the system, or on the other hand we can also evaluate the end-to-end request latency between the clients and the servers. We might not necessarily need to model the internal behavior of the client in detail when evaluating the throughput. As long as the client sends correct requests at the required rate, it makes no difference to the server whether a detailed or less detailed simulator generates those requests. However, when we evaluate the end-to-end request latency, the internal behavior of the client might have substantial influence on the measured results, since the internal processing of the client makes up one part of the overall end-to-end latency. Therefore, we probably need to simulate the clients that measure the latency with higher detail for

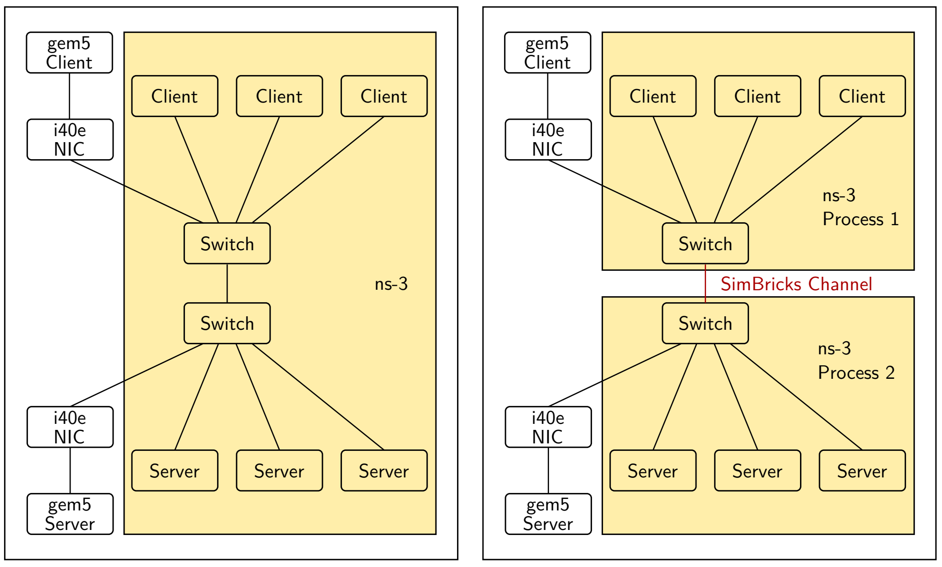


(a) End-to-End (b) Mixed-Fidelity

Figure 3.1: Two configurations for a full-system network simulation. The end-to-end configuration simulates all components with a high level of detail. The mixed-fidelity configuration moves the simulation of some hosts and their NICs into the network simulator ns-3.

realistic results.

In the previous examples, we proposed to use different levels of fidelity for the various host components in the full-system network simulation. We can achieve a high fidelity end-to-end simulation of the hosts with detailed architectural simulators, like gem5 or qemu. For a less detailed simulation at the protocol level we can, for example, use network simulators, such as ns-3 or OMNeT++. Figure 3.1 shows an end-to-end and a mixed-fidelity configuration of a network simulation. In the end-to-end simulation we use the architectural simulator gem5 for a detailed model for all the hosts. In the mixed-fidelity case we move some of the hosts to the network simulator ns-3, which models the hosts at the protocol level. Of course, the host is not the only component for that we can choose between simulators with different levels of detail. We can, for example, simulate a NIC with an accurate but resource intensive RTL-level simulation or with a faster behavioral model. Since we are focusing on large scale full-system network simulations in this thesis, we will mainly look at the trade-off of simulating a component either in a network simulator or in a detailed component simulator.



(a) One ns-3 Process (b) Two ns-3 Processes

Figure 3.2: Parallelizing an instance of the network simulator ns-3 by splitting the network topology along an Ethernet connection into two parts and connect them through a SimBricks channel.

Decomposition of Bottleneck Simulators

One consequence of synchronized simulations is that the entire simulation is only as fast as the slowest simulator. We found that this is typically a detailed simulator, such as an architectural simulator like gem5 or a RTL-level simulation. However, if we move many components into a less detailed simulator, this can become a bottleneck. For example, if we simulate hundreds or even thousands of hosts in a network simulator at the protocol-level, the amount of computation that this simulator has to carry out becomes very large, making it a bottleneck in the full-system simulation. We propose to parallelize the bottleneck simulators to distribute the computation across multiple processes. This speeds up the simulation and makes more efficient use of the available resources, since the bottleneck simulator no longer holds other simulators up that otherwise waste cycles with waiting. However, parallelizing simulators is in general a difficult problem and this is no different for network simulator, which we will mostly focus on in this thesis. Although some network simulators, such as ns-3, already offer capabilities for parallelization, they often scale poorly [32]. Additionally, we typically have to adapt the simulation accordingly in order to parallelize it, which takes some effort.

In our approach, we decompose simulators at natural boundaries, which in the case of a network simulator is Ethernet. We then simulate each part that we get after the decomposition in its own simulator process and connect the individual instances through adapters with each other, effectively parallelizing the simulator. For that, we make use of already existing SimBricks adapters in the network simulators, which let us connect and synchronize the parallel processes through SimBricks channels. Figure 3.2 shows an example of how we can parallelize an instance of the network simulator ns-3 in two processes. We split the network topology at the Ethernet connection between the two switches into two parts and subsequently replace the ns-3 internal Ethernet connection with a SimBricks channel.

Configuration and Orchestration

Running full-system network simulations is already a complicated task as it requires starting and configuring many simulator instances. When we scale up the network simulation, using both mixed-fidelity simulations and splitting bottleneck simulators, orchestrating the simulation becomes even more complex. For that, we introduce abstractions for the simulators in the orchestration framework, especially for the network simulator. The goal is to have a separation between the specification of the simulated system and its implementation. The specification defines the components in the simulation, including their simulator independent configurations, and how they are connected with each other. To do this, we provide basic building blocks from which the specification can be assembled. The implementation specifies which simulator should be used to simulate which component and how the simulator should simulate the component. This approach makes both mixed-fidelity and decomposing simulators easier to implement for the user.

If we want to convert an end-to-end simulation into a mixed-fidelity simulation, we only have to change the implementation, however the specification remains the same. In the implementation we can choose, for example, for a host component whether we want to simulate it with a detailed architectural simulator or with a network simulator at the protocol level. This changes how we simulate the system, but the structure of the system itself stays the same. This approach also allows us to easily try out different mixed-fidelity configurations by moving components between detailed and less detailed simulators in the implementation. Similarly, we can also split bottleneck network simulators into smaller, parallel simulated parts by adapting the implementation accordingly. The trivial implementation choice is to put all the components that we want to simulate in a network simulator into one network simulator instance. However, we can also partition the network topology that is defined by the specification and then choose to simulate each partition in a separate process. In the implementation we also take care of introducing the necessary SimBricks channels between the network simulator instances. Therefore, just by deciding in the implementation which component to simulate in which network simulator instance we can parallelize parts of the simulation.

Another advantage of this abstraction is that we can create large scale full-system network simulations without having to configure the underlying simulators directly, but purely within the orchestration framework. In this thesis, we present in particular a flexible abstraction of network simulators, the design of which we discuss in more detail in section 3.3. With this abstraction we can configure mixed-fidelity simulations and decompose bottleneck simulators basically transparent to the underlying simulators. This presents a very practical and easy-to-use approach for large scale network simulations to users. However, the user can still configure simulators manually at any time and integrate this into the simulation. This allows for maximum flexibility and enables users to utilize the full feature set of simulators.

3.2.2 Overview of Framework Architecture

We integrate our framework tightly with SimBricks, in order to make use of SimBricks adapters and channels for connecting simulators with each other and to make use of the orchestration framework of SimBricks, which already supplies key features for assembling and orchestrating full-system simulations. Our framework adds the features and abstractions that we covered in subsection 3.2.1 to provide practical large scale full-system network simulations. For this thesis we set the main focus on network simulators because they simulate many different components due to the fact that networks are very diverse systems. Therefore, network simulators offer a great opportunity for building an abstraction layer that enables us to compose large scale network simulations in the orchestration framework while also allowing us to build mixed-fidelity simulations and decompose bottleneck network simulators.

To this end, the presented framework builds an abstraction layer on top of network simulators, which consists mainly of two parts: one part that is integrated into the orchestration framework of SimBricks to construct and configure the full-system simulation and another part that is implemented in the network simulator which takes care of realizing and running the simulated network.

Figure 3.3 shows the general architecture of the framework. The user provides a configuration script that uses the orchestration framework to assemble and configure the simulation. To do this, the user first describes which network system he wants to simulate by creating a specification for this system. He then describes in the implementation how this specification is to be simulated. For the network simulators, the orchestration framework uses the abstraction layer of the network simulators to generate network descriptions that are given to the network simulator instances. The network

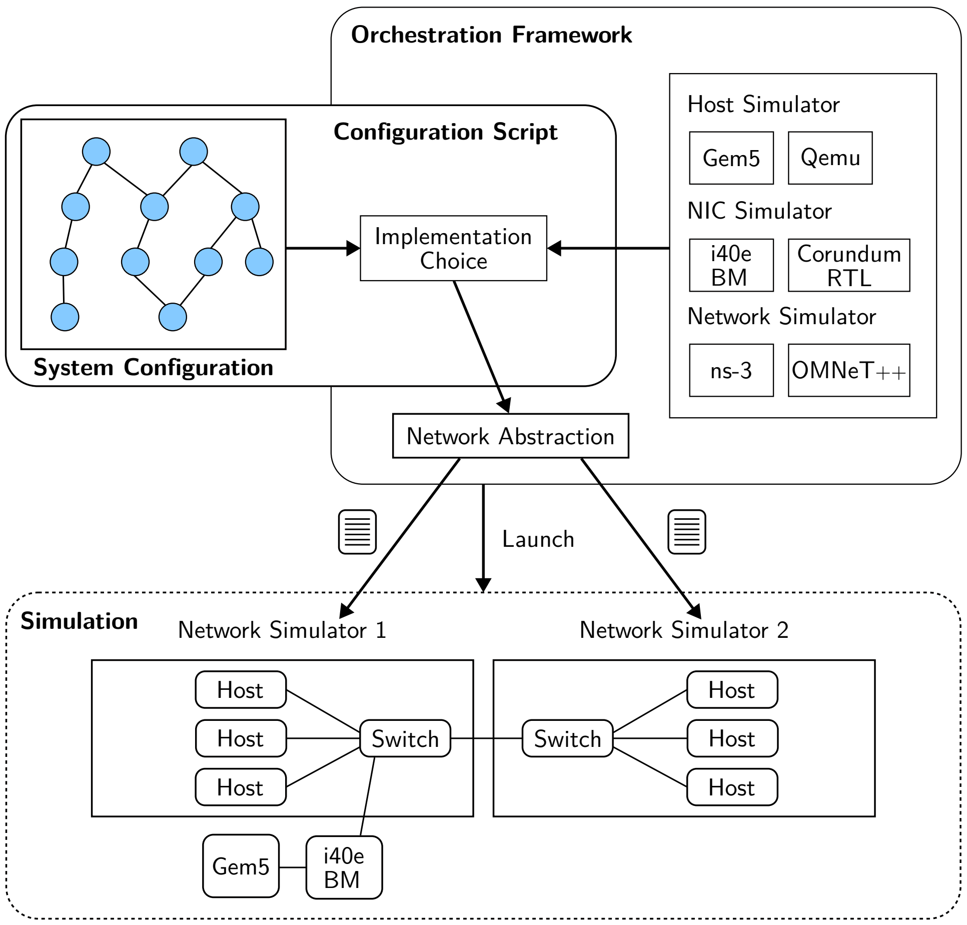


Figure 3.3: An overview of the general architecture of the presented framework.

simulators receive the descriptions and turn them into a network simulation, which they then execute. For the other simulators, the orchestration framework already provides the necessary capabilities to launch them. Finally, the orchestration framework takes care of launching all simulators with the necessary configuration parameters and makes sure to terminate them once the simulation has finished. This architecture enables us to easily configure and run large scale full-system network simulations, by providing a specification and an implementation choice for the desired network system within a single configuration script.

3.3 Network Simulator Abstraction Layer

Computer networks comprise many different components, like network switches, routers, or hosts, and we can use those components to build various network systems. This gives us a lot of freedom but also a lot of possibilities to build network simulations. Therefore, network simulators usually require us to assemble a specific network system within a program using the components that are offered by the simulator. However, in the context of our proposed framework that aims to overcome the issues of large scale full-system network simulations, directly configuring the network simulators becomes very cumbersome and time-consuming. For example, if we move components from a special component simulator to the network simulator in the course of a mixed-fidelity simulation, we have to adapt and reconfigure the network simulator. Similarly, if we want to decompose a bottleneck network simulator into several parts, this requires changes to the network simulator and even the addition and configuration of new simulator instances. We therefore want to move the configuration of the network simulators into our orchestration framework, where we can then easily configure mixed-fidelity simulations and decompose bottleneck simulators transparently for the underlying simulators.

To this end, we propose a network simulator abstraction layer. The abstraction layer introduces an abstraction of typical network systems in the form of a network description. In the orchestration framework the abstraction layer creates network descriptions according to the specification and implementation choices given by the user. The descriptions are subsequently given to the network simulators, which implement them into actual network simulations. By adding SimBricks adapters to the abstraction, it also lets us define connections to other simulators through SimBricks channels. This approach enables us to configure network simulators fully within our orchestration framework, providing together with the rest of the orchestration framework a practical and easy way to assemble and run large scale full-system network simulations.

3.4 Integration Into SimBricks

We integrate our framework into SimBricks in order to make use of SimBricks' ability to compose, configure and run full-system simulations, by combining several simulators into one overall simulation. Our framework extends SimBricks to overcome its limitations when it comes to large scale full-system network simulations. For this, we mainly implement the part of the network simulator abstraction layer that generates the network description, as part of the orchestration framework. This enables us to compose mixed-fidelity simulations by flexibly choosing which components of the simulated system should be simulated by a network simulator at a protocol level and which components should be simulated by a detailed component simulator. Additionally, the framework provides an easy way to decompose the network simulator into multiple partitions, which are in turn simulated in separate processes. Finally, the framework allows us to predefine network topologies enabling easy reuse of a topology in multiple network simulations.

3.4.1 Integrate Network Simulator Abstraction

The orchestration framework of SimBricks allows users to assemble and execute complex full-system simulations which involve multiple simulators. To this end, the orchestration framework provides descriptions for each simulator that is integrated into SimBricks. These descriptions are instantiated and configured in a experiment script and subsequently translated into invocations of the concrete simulators when the simulation is run. In order to integrate the network simulator abstraction into the orchestration framework, we add a set of building blocks that are used to assemble the network system independent of the concrete network simulator. Special network simulator descriptions take a network system assembled from the building blocks and turns it into a proper invocation of the network simulator, providing the network description in the right format.

However, to connect the network simulator to other simulators and thus be able to define full-system network simulations, we must also integrate the connection to other simulators in our network description. To do this, we add special components to our network description that describe the connection to another simulator. Those components get linked to a simulator instance so that the orchestration framework can provide both simulators with the necessary parameters in order for them to establish a connection through their SimBricks adapters during the simulation.

The SimBricks adapters in the supported network simulators use Ethernet as the underlying protocol, which is also the typical protocol at the boundaries of a network. This allows us to connect any other simulator that uses Ethernet SimBricks adapters, including NIC simulators, host simulators which also simulate the NIC internally, and other network simulators.

As a consequence, this design gives us a lot of flexibility in how we can connect other simulators to the network simulator to run full-system network simulations.

3.4.2 Mixed-Fidelity Network Simulations

Mixed-fidelity network simulations are easily implemented using the network simulator abstraction. The abstraction provides components that are directly simulated by a network simulator and components that connect the network simulator through a SimBricks channel to a dedicated detailed simulator. To move a component from the less accurate network simulator to a more detailed simulator, only one component in the network abstraction needs to be changed and linked to the dedicated simulator. The other direction works in the same way.

3.4.3 Decompose Network Topology

To decompose the network topology into multiple parts the network abstraction is split into partitions. Each partition is then given to a separate network simulator instance, which simulates the respective part of the network. The connections that run between two partitions are implemented as SimBricks channels between the corresponding simulator instances.

3.4.4 Predefined Network Topologies

The framework enables us to flexibly compose full-system network simulations by piecing together building blocks from the network abstraction and connecting other simulators. Although this gives us a lot of freedom, it might be tedious to stitch together the network topology for every simulation from scratch. Therefore we can build predefined network topologies that expose simple interfaces and configuration parameters using the components from the network abstraction. This enables us to quickly create network simulations based on a predefined topology. Since the network topology is still constructed from the same basic components, adapting and modifying it is still easily possible. We can additionally provide partitioning strategies that make it easy to parallelize and distribute the simulation among multiple network simulator instances.

3.5 Integration Into Network Simulator

We integrate the part of the network simulator abstraction layer that parses and realizes the network simulation into every network simulator that we want to support. For that, we have to implement each of the basic building blocks within the respective network simulator and provide sufficient glue code to combine the individual building blocks into a network simulation according to the network description. In order to connect the network simulator to other simulators via a SimBricks interface, we need to add a building block that wraps the SimBricks adapter to establish a connection, given that the adapter is already implemented. If an adapter for the simulator has not yet been implemented, we must of course implement this first.

The network simulator only needs to implement the network description that it receives. Everything else happens transparently for the network simulator, such as scaling the network simulation by decomposing it. The logic that takes care of how the full-system network simulation is assembled and generally executed is located in the orchestration framework.

Chapter 4

Implementation

The implementation is divided into two parts. One part is implemented in the Python orchestration framework of SimBricks and the other part is implemented in the network simulator. In this thesis we focus on the network simulator ns-3 [21] and implement the network simulator part of the framework only there. However, this part can also be implemented in a similar way for other network simulators.

4.1 Network Description

4.1.1 Component Categories

The network description specifies all components of the simulated system and how the components connect to each other. Although there are many different types of components, we can group them into categories. First, we have components that are responsible for connecting host or networks with each other and enabling communication between them, such as network switches and routers. Secondly, there are channels between connected components, for instance wired connections with copper cables or wireless connections with radio waves. These two categories form the backbone of a network and provide the structure of the network topology. Next, we have hosts that are connected to the network and applications that run on the hosts in order to generate and consume network traffic. Additionally, we use special categories that do not define network components, but let us configure the simulation, such as enabling logging or defining a global stop time for the simulation. One of those special categories are probes, which we can enable and configure for some of the simulated components. Probes let us record specific behavior of the network simulator in order to evaluate the simulation. Currently, we use the following categories for our network description.

- **TopologyNode**: components that handle the traffic flow in the net-

```
--Category=[prefix-]key[(type)]:value;...
```

Figure 4.1: The format of the command line parameters providing the key-value pairs that describe the components.

work like network switches or routers.

- **TopologyChannel**: channels between TopologyNodes.
- **Network**: connection to another network simulated by a separate network simulator.
- **Host**: hosts in the network.
- **Application**: applications that run on the hosts and generate and consume network traffic.
- **Probe**: probe that allows us to record specific behavior of the network simulation for evaluation purposes.
- **Global**: global configuration parameters for the simulation, such as a global stop time.
- **Logging**: configures the logging capabilities of the network simulator.

We use the separate category Network to specify connections to another network simulator instance. However, we could move this to the category TopologyChannel, because it is also just an Ethernet connection through a SimBricks channel.

4.1.2 Network Description Format

We describe each component of the network through a set of key-value pairs. These key-value pairs provide all the necessary information that is required to instantiate and configure the component. They include fundamental information, such as the type of the component or the identifier for the component. Additionally, they provide specific configurations for each network component, for instance the latency of a link between two network switches.

In the case of ns-3, we pass the network description in the form of command line parameters. In order to parse the parameters, we make use of the command line parser that is provided by ns-3. Figure 4.1 shows the principal format of the command line parameters. We use the category to which the respective component belongs as the name for the parameter. This allows us to collect all components of a category in a separate list, while parsing the parameters. The value of the command line parameters is the list of key-value pairs, which we use for instantiating the components in the network simulator.

```
--Host=Id:/s1/host1;Type:SimpleNs3Host;Ip:10.1.0.2/24;...
```

Figure 4.2: An example for a command line parameter specifying a host component.

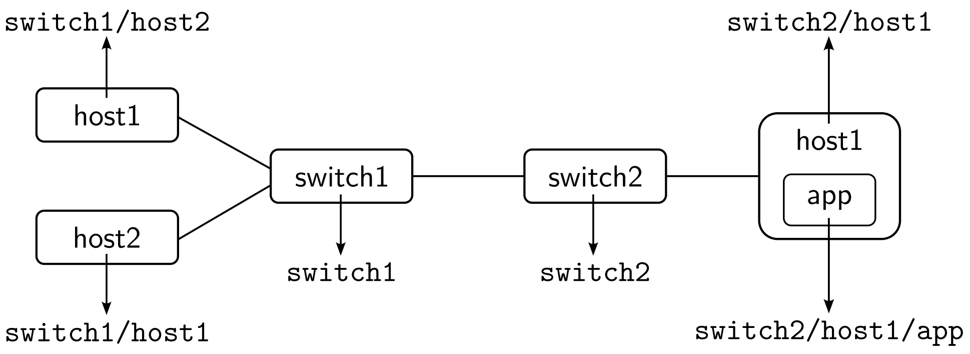


Figure 4.3: An example of a network topology with the IDs of the individual components

The format lets us additionally specify a prefix, a type or both for a key-value pair. The type helps the network simulator to convert the provided value to the data type that is need for this specific attribute. The prefix groups multiple key-value pairs, which enables us to flexibly set multiple attributes for a subcomponent of the specified component, like for example a queue which is part of the host. Both the prefix and the type are especially helpful, when we make use of the attribution system of ns-3, which we will explain in more detail in subsection 4.3.3.

An example for a concrete command line parameter is shown in Figure 4.2. The parameter specifies a host component, which is denoted by the category `Host`. The value of the command line parameter is the list of key-value pairs, including the ID and type of the component. In the example we also specify that the host should use the IPv4 address `10.1.0.2/24`.

Component IDs

Every component in the description receives a unique identifier, which also reflects where in the network the component is located. To this end, a component's ID is actually a path that specifies its position in the network which leads to a hierarchical description of the system.

Figure 4.3 shows an example for a small network topology with the IDs for each component. At the top-level are components that build the structure of a network, like network switches and links between them. In the example, those are the network switches `switch1` and `switch2` and the link with the ID `link`. Then, we have components that are connected to the top-level components, such as the hosts in Figure 4.3. Since `host1` is connected to `switch1`, we use the path `switch1/host1` as the full ID for this host. For the second host, that is connected to this switch, we choose the name `host2`. Because the switch is the same, the first part of the ID is also `switch1`, resulting in the full ID `switch1/host2`. The host that is connected to `switch2` is however also named `host1`. But this does not lead to conflicting IDs, since the full ID `switch2/host1` is again unique. Therefore, we are allowed to have the same name for multiple components, as long as they are at different locations resulting in unique IDs.

We continue the hierarchical structure by adding more levels to the path. An application, for example, runs on a host, resulting in an ID path length of 3. Thus, for the `app` in Figure 4.3 that runs on `host1` which is connected to `switch2`, we get the ID `switch2/host1/app`.

Component Type

The description of every component contains a key-value pair that specifies the concrete type of the respective component, where the type indicates which building block this is in the network simulator. There are, for example, various different applications that we can run on the simulated hosts and the type specifies which concrete application out of the available ones we want to use. With this information, the network simulator knows which concrete component it has to instantiate.

Component Configurations

For each component we can in principle give an arbitrary amount of key-value pairs as configuration parameters. The network simulator uses these parameters to configure the component during instantiation. The configuration parameters are specific to each component and depend on what the network simulator supports. For example, we can give a ping application the IP address of the destination, whereas we can give a link between two network switches the desired latency for that link. Some components may require mandatory configuration parameters without which the component cannot be instantiated.

4.2 Implementation in SimBricks

4.2.1 Current Limitations

The orchestration framework does not yet support a clean separation between specification and implementation for the components. However, we provide the network abstraction as a first step towards the separation between specification and implementation, especially in regard to full-system network simulations. Starting with an abstraction for the network simulator

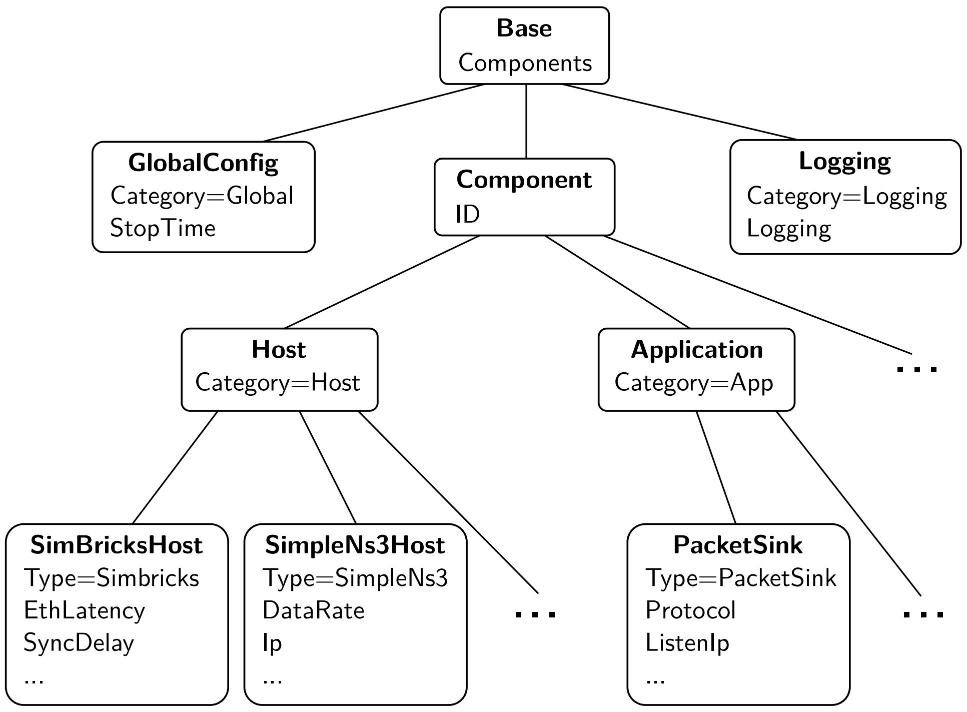


Figure 4.4: The class hierarchy used to implement the specification of a network system.

makes sense, because it allows us to already create an abstract specification of a network system. Later, we can further extend this to integrate other simulators into the specification and implementation abstraction.

4.2.2 Network Specification

As part of our framework we integrate a set of classes into the orchestration framework of SimBricks which allow us to specify a network system. From this specification, we can then generate one or more network descriptions. For this, we implement a class for each supported component in Python, which contains fields for configuring the component and a function to generate this component's description. We use inheritance as shown in Figure 4.4 to specify common attributes only once and with that reduce the implementation effort. For example, the class Base implements the logic of creating a string representation of the command line parameter that we give to ns-3 and the class Host sets the category to "Host", so that all the specific hosts use the correct command line parameter. We also see that the special categories GlobalConfig and Logging do not inherit from the Component class, because they are not actually components in the network.

Through the class Base every component has a list of components, which is used to capture the hierarchy of the network system. This means that when we create an application to run on a host, we add the application component to the host component. Similarly, we add host components to TopologyNodes, for example. This allows us to capture how the components are connected to each other, thereby creating the network structure. An exception here are the TopologyChannels, which, like TopologyNodes, are located at the top level of the hierarchy. We must explicitly give these the TopologyNodes that the channel is supposed to connect with each other.

This approach further enables us to generate the path-like full IDs automatically. Instead of specifying the full ID manually while creating the components, we give each component a name. After we have specified the full network system, we can combine the individual names of the components into the full ID. This can easily be done by traversing the system hierarchy and recursively constructing the IDs.

In order to specify a connection through a SimBricks channel to another simulator, we include special components in the network specification. Currently, we implement two components that let us connect the network simulator to another simulator: SimbricksHost is meant to connect a host that is simulated by a separate simulator, such as the architectural simulator gem5, to the network, whereas NetworkSimbricks is meant to connect another network simulator instance to the specified network. Both components require us to add the respective object from the orchestration framework, which represents the simulator that we want to connect. The orchestration can subsequently set up the SimBricks adapters on both ends to establish a SimBricks channel between the simulators. Since both components are basically wrappers around the same SimBricks adapter using the same protocol, we could also unify this in one component. However, using the more explicit components, makes the network specification abstraction a bit cleaner, because it tells us what type of component is on the other end of the SimBricks channel.

4.2.3 Integration as Simulator

The orchestration framework implements for every supported simulator a class that can be instantiated and added to the full-system simulation in the configuration script. The classes usually also let the user configure the simulator by providing configuration parameters. In order to integrate our network specification into the orchestration framework, so that we can use it to configure and run a network simulator given the specification, we add a separate class to the orchestration framework. This class receives the network specification and takes care of launching the network simulator, in our case ns-3, with the correctly formatted network description that is generated by the specification. Additionally, the class provides plumbing specifically for the orchestration framework to correctly launch and establish

```
class DumbbellTopology(Topology):

    def __init__(self):
        self.left_switch = SwitchNode("_leftSwitch")
        self.right_switch = SwitchNode("_rightSwitch")
        self.link = SimpleChannel("_link")
        self.link.left_node = self.left_switch
        self.link.right_node = self.right_switch

    def add_left_component(self, component):
        self.left_switch.add_component(component)

    def add_right_component(self, component):
        self.right_switch.add_component(component)

    @property
    def data_rate(self):
        return self.link.data_rate

    @data_rate.setter
    def data_rate(self, data_rate):
        self.link.data_rate = data_rate

    ...
```

Figure 4.5: Definition of a dumbbell topology using two switches and a bottleneck link between them. The topology implementation provides functions for adding components and setting attributes.

connections to other simulators for full-system simulations.

4.2.4 Predefined Network Topologies

With the network abstractions, we now also have the possibility to define the basic structure of a network topology so that we can use it in later simulations. For that, we assemble the backbone of the network with components from the categories TopologyNode and TopologyChannel. Additionally, we provide functions for adding hosts and possibly other components to create a full network simulation from the defined topology. However, since we are implementing this in Python and given the flexibility of the network specification, we have a lot of freedom in how we can define the topology by using Python language features and providing the user functions and attributes to configure and instantiate the topology.

As one simple example, we provide in our framework the definition of a dumbbell topology. Figure 4.5 shows an excerpt of the implementation in our framework. The constructor of the topology creates two switches and a bottleneck link that connects the switches. Furthermore, the implementation includes two functions for attaching components such as hosts to the topology. One function adds the components to the first switch and the other one adds them to the second switch. Finally, the topology definition also provides some configuration attributes, for instance for setting the data rate and latency of the bottleneck link.

4.2.5 Mixed-Fidelity Simulations

The network abstraction enables us to implement mixed-fidelity simulations fully within the orchestration framework. By choosing between different component types in the network specification, we are able to either simulate the component in the network simulator or via a dedicated component simulator, that implements a more detailed model for the respective component. In the current implementation we focused on host components in regard to mixed-fidelity simulations. The network specification therefore offers a host component that gets simulated directly in the network simulator and a host component that uses the SimBricks adapter to connect a dedicated host simulator. Since we make this selection in the orchestration framework, we can easily decide for each host component with which accuracy we want to simulate it, which is transparent for the underlying simulators.

Figure 4.6 exemplifies how to create a simple mixed-fidelity simulation using the orchestration framework with the help of the network specification. In the network specification, we specify that we want to simulate three hosts with ns-3 and one host with dedicated simulators, which in this case is the combination of a NIC simulator and an architectural simulator. For that, we link the SimBricks host in the specification to a simulator object of the orchestration framework. When running the simulation, the orchestration framework launches three simulator instances and establishes SimBricks channels between them using the SimBricks adapters.

4.2.6 Decomposing Network Simulator

Decomposing the network simulator into multiple separate simulator instances is similar to defining mixed-fidelity simulations. In order to decompose the simulator, we split the network specification along TopologyChannels into multiple partitions, where each partition gets simulated by a separate instance of the network simulator. To obtain the same network topology as before, we need to replace all TopologyChannels that now run between two instances with SimBricks channels. For this, the network specification offers a component to denote a connection to another network

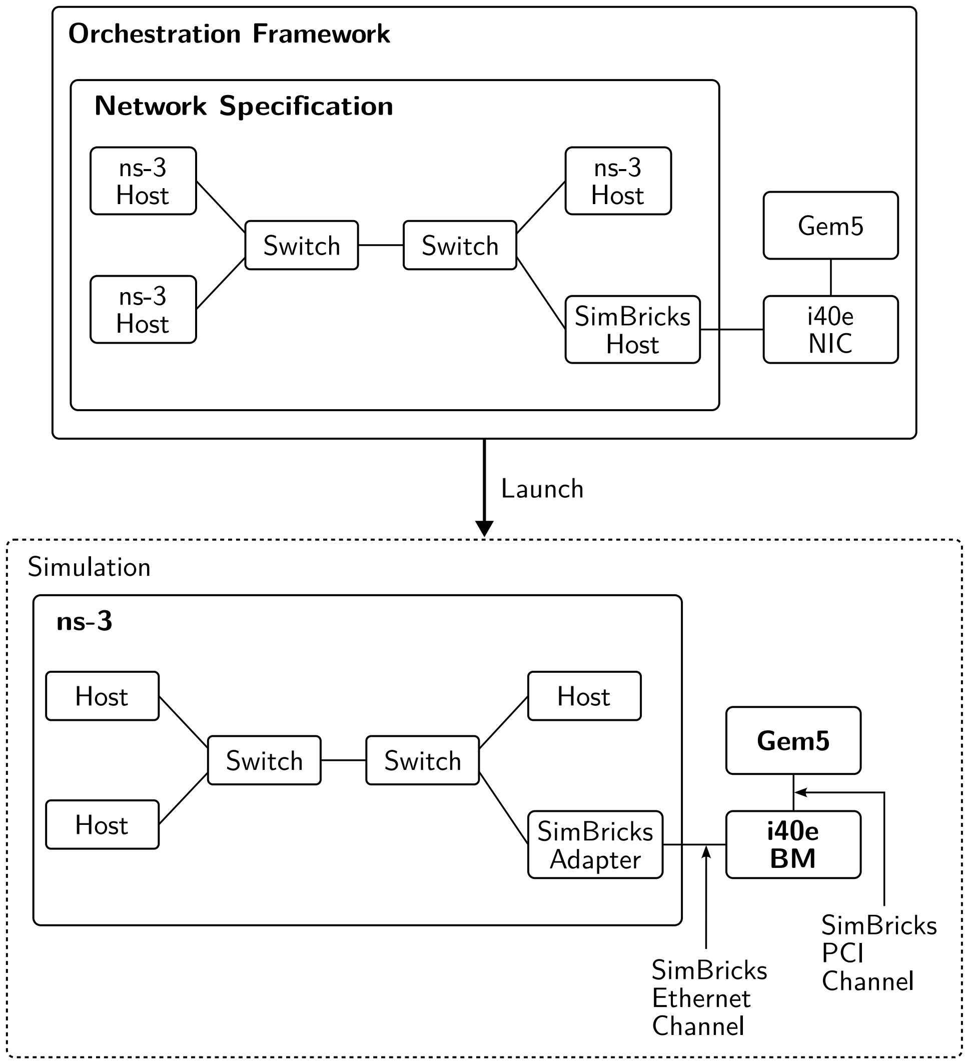


Figure 4.6: An example for assembling a mixed-fidelity simulation using the network specification, which is turned into a mixed-fidelity full-system simulation.

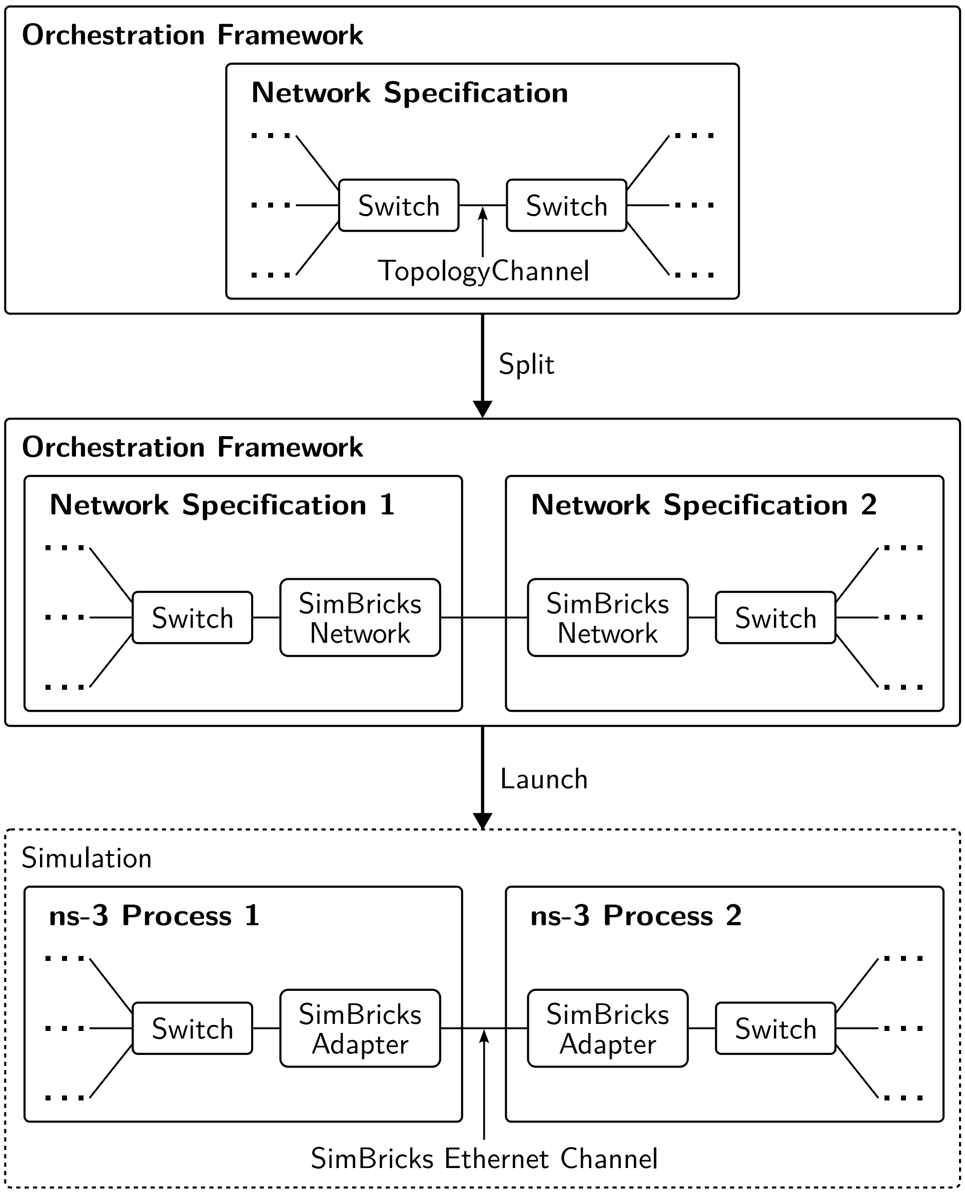


Figure 4.7: Split a network specification at a TopologyChannel into two smaller specifications introducing SimBricks Network components to replace the TopologyChannel. Each specification is simulated by a separate ns-3 process using SimBricks adapters to establish a SimBricks Ethernet channel.

simulator instance. This component also uses the SimBricks adapter, which means that internally this is basically the same as connecting to a dedicated simulator for mixed-fidelity experiments.

Implementing the specification in Python gives us again the advantage, that we can easily implement a function that takes the specification of a large network and splits it up into smaller parts according to a partitioning strategy, while automatically replacing affected TopologyChannels with SimBricks channels. In particular, we can implement such functions for the predefined network topologies so that users can easily parallelize them to speed up the simulation.

Figure 4.7 shows an example for decomposing a network simulator by splitting up the specification into two smaller specifications. In order to replace the TopologyChannel that connects the two switches, we introduce on each side a SimBricks Network component. When running the simulation, each specification is simulated in a separate ns-3 process where the SimBricks Network components are implemented by SimBricks adapters to establish the necessary SimBricks Ethernet channel.

4.3 Implementation in Ns-3

For this thesis, we implement the network simulator part of the network abstraction in ns-3. It is implemented as a new module in ns-3, which is written in C++. The module provides implementations for the components that are supported by our network specification, using the components provided by ns-3. Additionally, the module implements logic to parse the network description and assembling the simulation.

4.3.1 Processing Network Description

Ns-3 receives the network description in the form of command line parameters, and we use the command line parser that is integrated into ns-3 to parse them. Since the network description can become very large and there is a limit on the maximum length of a command on typical systems such as Linux, we can also pass the description via a file. In this case, we specify the path to the file that contains the description using a special command line parameter and parse the parameters from this file using the command line parser. While parsing the description, we split up the list of key-value pairs and store them in a map for each component. The mapping from keys to values allows us to easily access them later, when creating the components. We also store all parsed components of one category, which we described in subsection 4.1.1, in a list.

To assemble the network simulation from the description, we need to create the components in ns-3 and connect them correctly. To make this step easier, we create the components in the order specified by the ID paths. We start with the components on the highest level that have an ID path length of one, then we create the components with an ID path length of two, and so on. This makes sense because the components from a lower level must be connected or added to components from the level above, which is easy when the components from the higher levels already exist. In practice, this means that we first create TopologyNodes, such as switches, and then TopologyChannels, which connect the TopologyNodes. Afterwards, we create the hosts that are connected to the TopologyNodes and then the applications which will be added to the hosts. We proceed in a similar way for the remaining component types. After we have assembled the network system given by the description, we can run the actual simulation.

Of course, we also process the special components that specify global configurations or logging and configure the network simulation accordingly.

4.3.2 Network Components

We implement the network components using the building blocks that are provided by ns-3. Some of our components are just simple wrappers around components in ns-3, such as the applications. The wrapper uses the key-value pairs that are provided by the network description to configure the ns-3 component. For more complex components, our implementation uses multiple building blocks from ns-3 in order to realize the component in the simulation. Furthermore, every component implementation provides necessary code that allows us to connect the component to its parent.

In ns-3 there are two basic components that are used for most of the physical components: Node and NetDevice. The Node is used for any device in the network, like hosts or network switches and the NetDevice handles sending and receiving of network packets while it is attached to a Node. Together with channels between NetDevices and software stacks that are aggregated to the Nodes, we can implement a basic set of components that is sufficient to create network simulations. For example, a network switch is implemented as a ns-3 Node containing a special BridgeNetDevice which implements the functionality of a simple switch. Figure 4.8 shows a simple network system specifying the components of the abstraction and the ns-3 components that are used for the implementation. Note that we do not show all ns-3 components involved for reasons of clarity, however some of the components actually include further components. As we have already mentioned, the *SwitchNode* is implemented via a ns-3 Node containing a BridgeNetDevice. We attach hosts to the *SwitchNode* by adding and connecting NetDevices which connect to the actual host. In the case of the *SimpleNs3Host*, we have a Node containing a SimpleNetDevice which is connected through a SimpleChannel to another SimpleNetDevice. This whole construct is implemented by the *SimpleNs3Host* component. The chosen abstraction makes sense when we look at the *SimBricksHost*, which is implemented via the Sim-

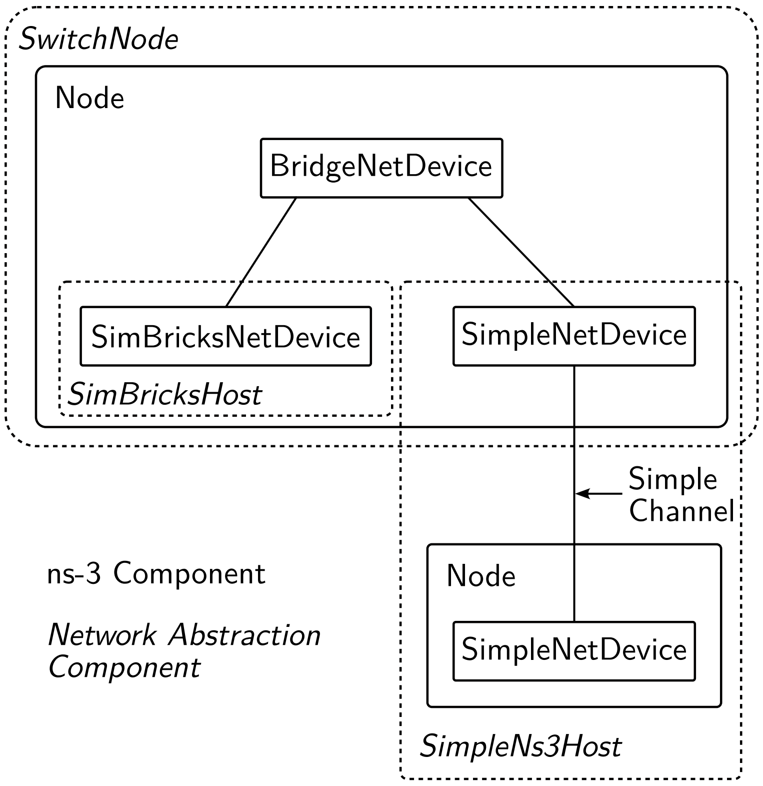


Figure 4.8: Overview of a simple network system showing both the components of the network abstraction and the ns-3 components that are used to realize the abstraction components.

BricksNetDevice. The SimBricksNetDevice is a NetDevice that wraps the SimBricks adapter enabling communication through the SimBricks channel. Therefore, the SimBricksNetDevice also connects another host, simulated by a dedicated simulator, through a channel, giving us basically the same abstraction as for the *SimpleNs3Host*.

For each component of our network abstraction, we implement a separate class in our ns-3 module. To simplify the process of assembling the network system, we establish a class hierarchy that closely resembles the class hierarchy of our network specification, which is shown in Figure 4.4. To capture the hierarchy of the specified network, every component maintains a map that contains mappings for its child components with the name of the child component as the key. This design allows us to traverse the network system along a given path, which enables us, for example, to find the parent of a component given its ID path. We leverage this while creating the components in order to connect and add a component to its parent.

4.3.3 Setting Attributes

In addition to creating the network components, we also need to configure them. To this end, we mostly leverage the attribute system of ns-3, which can be used to provide a set of attributes for each component. We set and access the attributes through keys in the form of strings. While creating a component in our framework, we can look for specific key-value pairs and either use them to set specific attributes or carry out more complex processing and configuration of the component. For the remaining key-value pairs we try to set them automatically through the attribute system. To do this, we use the key and the value of the key-value pair also as the key and the value for the attribute. This allows us to support many attributes without having to implement special handlers for them, reducing implementation effort. We only need to make sure to set the correct keys in the network description so that ns-3 understands them.

Note that we give the value for an attribute always as a string. This means that in many cases ns-3 needs to convert the string to a different type that is required for this attribute. For example, if the type of the attribute is an integer, the value string has to be converted to an integer first, before we can set the attribute. Although ns-3 is able to convert the string representation of the value for many types, we sometimes need to use a custom type conversion, because there is no conversion available for this specific type or our string representation of the value does not fit to the conversion provided by ns-3. In case we need to use our own type conversion, we provide a type with our key-value pair in the network description. While setting the component attributes, we check for each key-value pair whether it provides a type. If a type was found, we convert the string representation of the value first using our own conversion and then use the converted value to set the attribute.

As we have already noted before, many of the network components that are defined by our abstraction are actually implemented in ns-3 using multiple ns-3 components. When we want to automatically set the component attributes based on the key-value pairs that are given by the network description, we do no longer know which key-value pair is meant for which ns-3 component. To overcome this issue, we prefix the keys of the key-value pairs with an additional identifier that denotes the ns-3 component to which it refers. This allows us to automatically configure multiple ns-3 components.

4.3.4 Attaching Probes

In order to evaluate and gain some insight into the network simulation, we need to capture the behavior of the simulation. To do this, we can attach probes to network components in our network specification, which is supposed to record a specific behavior of the respective component. In ns-3 we implement probes via TraceSources, which are callbacks that get triggered at specific events, for example when a NetDevice received a packet. Currently, we support a small set of probes that we can attach to specific components. For their implementation we provide callback functions that write the captured data into a file for later evaluation.

Chapter 5

Experimental Evaluation

In our evaluation, we use the data center network protocol Homa [20] to demonstrate how we can easily realize network simulations with our framework and use techniques such as mixed-fidelity simulations and splitting up the network topology to perform large scale simulations. In contrast to well established transport protocols such as TCP [4] which focus on large messages and aim to achieve maximal throughput, Homa focuses on small messages in data center networks and tries to keep latencies for those messages as low as possible. Traditional protocols are often connection-based and sequential which facilitates head-of-line blocking leading to inconsistent and high latencies especially for small messages. Homa improves tail latency for small messages by approximating SRPT (shortest remaining processing time first) through using prioritization for messages and a receiver-driven approach where the receiver informs the sender what and how much data to send. To this end, Homa is connectionless, message-based, and uses priority queues provided by modern network switches.

There exists an implementation of the Homa transport protocol in ns-3 [9], which we will use to generate Homa traffic in a network simulated by ns-3. Additionally, there exists a Linux kernel module implementation of Homa [10, 22], which also serves as the current reference implementation of the protocol. In our experiments we run an unmodified version of the kernel module on a Linux host simulated by an architectural simulator.

5.1 Experimental Setup

We run our experiments on a machine with 256 GB RAM and two Intel Xeon Gold 6336Y CPUs with 24 physical cores each, giving a total of 48 physical cores. In order to keep the required resources low, we limit the size of the simulations so that they fit on a single machine, but choose large enough simulations that suffice for the evaluations. While we can distribute the simulation among multiple machines using SimBricks proxies, we have

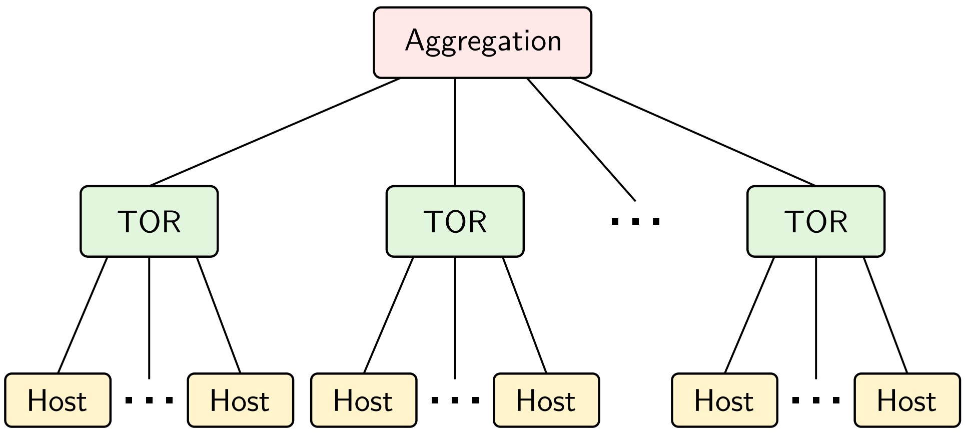


Figure 5.1: The basic topology used for the experiments in this evaluation.

not found the need to do so for this evaluation.

To simulate the network we use a ns-3 implementation that includes SimBricks adapters, the framework presented in this thesis, and the Homa transport protocol. We use qemu with instruction counting for time synchronization to simulate a Linux hosts in detail and run the Homa kernel module. The simulated hosts use a single core, and they are configured with 4 GHz clock frequency and 2 GB of main memory. We use the NIC simulator `i40e_bm` that is integrated into SimBricks which provides a behavioral model for the Intel X710 NIC. An instance of the NIC simulator connects via PCI to a host simulated by qemu and via Ethernet to the network simulated by ns-3.

For our experiments we use the topology that is shown in Figure 5.1, which is also used in a similar form for the evaluation of Homa using network simulators. The topology consists of racks consisting of hosts and a Top-Of-Rack (TOR) switch and an aggregation switch that connects to all TOR switches. Each link has a latency of 250 ns and the links between TOR and aggregation switch have a bandwidth of 10 Gbps. The original topology uses four aggregation switches where each aggregation switch connects to every TOR switch with 40 Gbps links, resulting in multiple available paths between two hosts. However, the BridgeNetDevice that we use to implement a network switch in ns-3 can not handle multiple paths correctly, resulting in switching loops creating broadcast storms. We therefore use only one aggregation switch but in turn increase the bandwidth of the links between aggregation switch and TOR switches to 160 Gbps. The evaluation of Homa that was carried out on physical testbeds uses a simpler topology that connects all hosts to a single switch. During our evaluation we found however that the two topologies lead to only slightly different results. Since the topology shown in Figure 5.1 also allows us to decompose and parallelize the network, we opt to use it throughout our experiments. A network consisting of a single switch would not allow us to split up the network simulator, because this would require us to decompose a single network component, which we not support.

5.2 Full-System Simulation Is Necessary

When we want to evaluate the Homa transport protocol, we have a few different options to do so. On the one hand, we could use a physical testbed using the Linux kernel implementation of the protocol. But this is often not feasible because it requires a physical system to be available. On the other hand, we can resort to simulation, for example using a network simulator such as ns-3. However, the ns-3 implementation of the Homa protocol comes with some drawbacks and differs from the Linux kernel module implementation which serves as the reference implementation. Using full-system simulation enables us to combine the network simulator with an architectural simulator such as qemu that is able to execute a full Linux operating system including kernel modules. Thus, full-system simulation allows us to run the Homa kernel module in simulation while also modelling the network part at the same time. Therefore, if we want to use an implementation that closely matches the reference without the effort of manually adapting the ns-3 implementation, we need to use full-system simulation.

For example, for the evaluation of Homa Montazeri et al. used different workloads [20]. One of the workloads represents the network traffic of a Hadoop cluster at Facebook [26], which contains many messages that are smaller than the size of a single network packet. However, the ns-3 implementation currently only supports sending messages with sizes that are multiples of the size of a packet, which means that the smallest supported message has the size of a full network packet. The same limitation does not exist for the Homa kernel module. If we want to run this workload without changing the implementation of the Homa protocol, we need to use the kernel module.

In Figure 5.2 we compare the evaluation of the previously mentioned workload on a physical testbed with our full-system simulation, where the results for the physical testbed are taken from [22]. For our simulation, we use the topology shown in Figure 5.1 with 20 hosts evenly distributed across four racks. Every host serves as both a client and a sever with each of them randomly sending messages to all hosts except themselves. We simulate all hosts with qemu using the Homa kernel module and evaluate the measured slowdown for each message size that occurs in the workload. The slowdown tells us how much more time it took to successfully transmit the message compared to an optimal case with no traffic in the network. It is calculated by dividing the measured transmission time by the best-case transmission time. We see that the general trend of the slowdown curves is

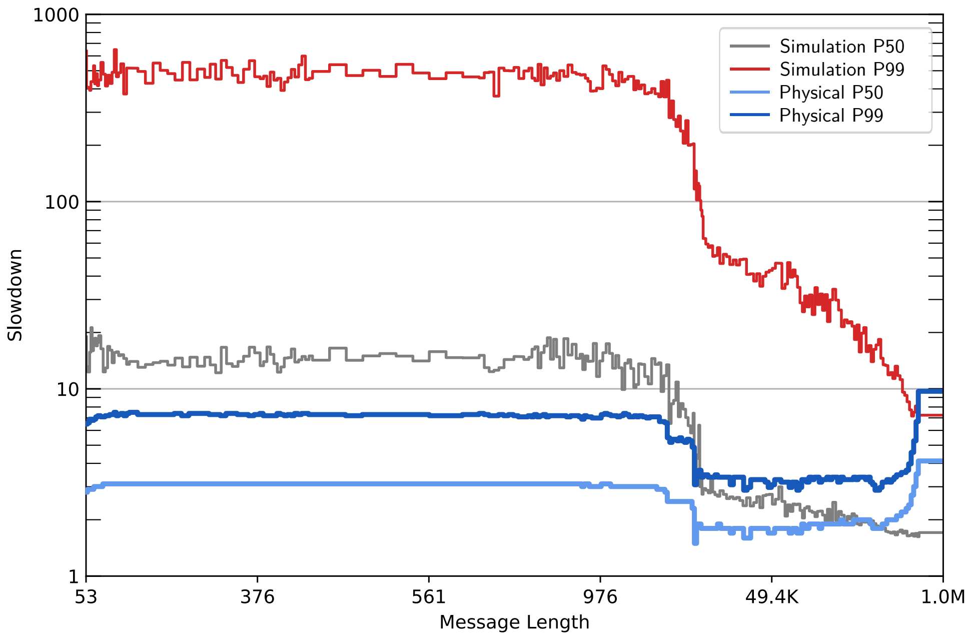


Figure 5.2: Median and 99th-percentile slowdowns for Homa traffic evaluated with full-system simulation and a physical testbed. The x-axis is linear in the number of sent messages with the respective size. The results of the physical testbed are taken from [22].

```python
for i in range(self.params['n_racks']):
    sw = SwitchNode(f'_tor{i}')
    sw.mtu = self.params['mtu']
    self.tor_switches.append(sw)
    self.switches.append(sw)
```

Figure 5.3: Creating a given amount of TOR switches in a loop.

similar for both evaluation platforms. However, the full-system simulation incurs high transmission times for some messages, which leads to a slightly higher median and a very high 99th-percentile slowdown compared to the physical testbed. Unfortunately, we have not been able yet to find the root cause of this issue. Nevertheless, this does not detract from the significance of the other results in this evaluation.

5.3 Composing Network Simulations Is Easy

We show that it is easy to compose large scale network simulations within the orchestration framework using the framework presented in this thesis. It requires only little code changes to configure mixed-fidelity simulations or to decompose and parallelize a network simulator. In the following, we want to analyze how much implementation effort it requires to compose and configure those simulations.

In the course of this evaluation, we implemented the topology from Figure 5.1 as a predefined topology in our framework that uses parameters to instantiate different configurations. This enables us to reuse the topology throughout the experiments that we conduct and instantiate systems of different sizes by choosing the number of racks and hosts per rack. The basic structure of the network spanning the switches and the links between them is implemented in roughly 40 lines of code independent of the concrete size of the topology. Since many of the components in the network are almost the same as other components, we can use loops so that we basically only have to implement it once and then generate an arbitrary number of the components. The TOR switches, for example, are generated as shown in Figure 5.3 given the number of racks. In addition to that, the predefined topology provides functions for adding hosts simulated in ns-3 and hosts simulated by a dedicated component simulator. Furthermore, this allows us to provide help functions, for instance a function that fills up the topology with hosts simulated in ns-3 running a specific application.

Thanks to the predefined network topology, we can conduct mixed-fidelity simulations with little effort. Currently, to use different simulators for hosts we need to instantiate and configure different host components of

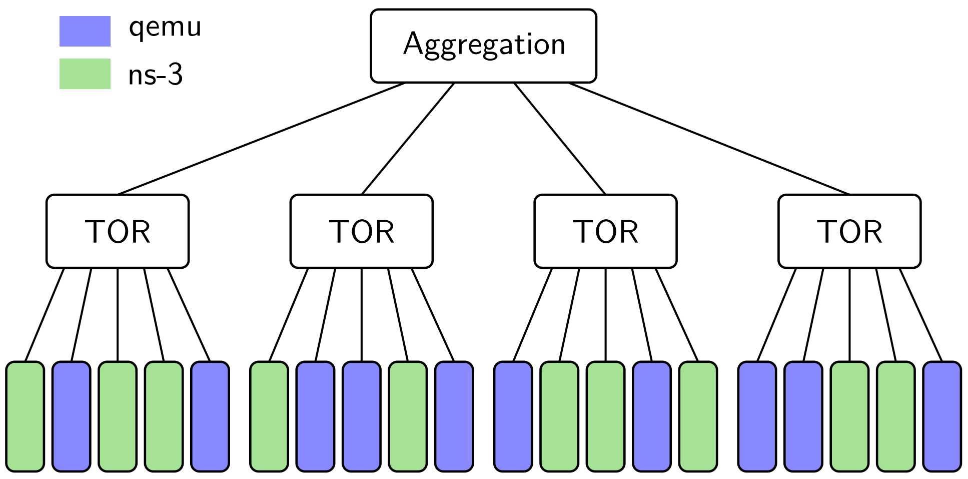


Figure 5.4: An example for a mixed-fidelity configuration using 10 qemu hosts and 10 ns-3 hosts.

the orchestration framework or the network abstraction, but we can then add the host to the topology by calling an appropriate function. The amount of code that is required to create and configure a host depends on the exact use case, but it can be as little as 10 lines. Therefore, switching hosts from a detailed architectural simulator to a network simulator that simulates them at the protocol level, takes only a few code changes. Furthermore, with the use of loops or functions, switching components from one simulator to a different one boils down to changing only a few lines of code or possibly even one line of code. This is possible because we carry out all the required changes in the orchestration framework, and we do not have to configure a simulator directly.

We have a similar case for decomposing network simulators. Given the predefined network topology, we can partition it by assigning the switches to different partitions. A generic function takes the partitions and distributes them across separate network simulator instances while replacing links with SimBricks adapters. In order to split the network topology we only need to provide a partitioning strategy. Simple partitioning strategies can be expressed in a single line of code, but more complex strategies usually also take only a few lines of code. Thus, decomposing the network simulator is effectively done within a handful lines of code.

5.4 Mixed-Fidelity Simulations Are Cheap

Next, we want to show that mixed-fidelity simulations help to reduce the amount of compute resources that are necessary to run the simulation. For that, we run five different configurations of mixed-fidelity simulations that

Configuration (qemu/ns-3)	#Processes	Simulation Time [minutes]
20/0	41	64
15/5	31	72
10/10	21	74
6/14	13	75
2/18	5	71

Table 5.1: The required number of simulator processes and the simulation time for different mixed-fidelity configurations.

use varying amounts of detailed qemu hosts which record the data for the evaluation and ns-3 hosts which generate Homa background traffic. We use again a total of 20 hosts evenly distributed among four racks. To set up a mixed-fidelity configuration, we first assign the desired number of qemu hosts randomly to the network topology. Then, we fill up the remaining spots in the topology with ns-3 hosts. Figure 5.4 shows an example for how a mixed-fidelity configuration using 10 qemu hosts and 10 ns-3 hosts could look like. For the extreme cases, we have one configuration that uses 20 qemu hosts but no ns-3 hosts and one configuration with only two qemu hosts communicating and 18 ns-3 hosts. For the other configurations we choose 15, 10, and 6 qemu hosts.

Because of the different implementations of the Homa protocol in ns-3 and the kernel module, we limit communication to hosts of the same type. This means, that all the qemu hosts communicate with each other and all the ns-3 hosts communicate with each other, but there is no cross-communication between the two.

The required resources for the mixed-fidelity configurations are summarized in Table 5.1. The table shows for each configuration the number of needed processes and how long the simulation took to complete. The number of processes directly translates to the required number of physical cores, because SimBricks adapters use busy polling. We can easily calculate the amount of processes one of these mixed-fidelity simulation spawns. Each qemu host launches an instance of the qemu simulator and additionally an instance of a NIC simulator. The network is always simulated in a single process for the given configurations. Therefore, reducing the number of qemu hosts by one, saves two simulator processes, which means that reducing the amount of qemu hosts also reduces the required physical cores. The smallest configuration using two qemu hosts requires only 5 physical cores so that it could be easily executed on a modern laptop. The runtime of the simulation is similar for all configurations and the differences seem to be regular fluctuations. Note, that the total runtime of a synchronized simulation is determined by its slowest simulator, which means that adding more hosts to the network simulator could make it the bottleneck. In order to analyze this further we would need to profile the simulators such that we obtain information about how busy a simulator is and how much time it spends waiting for other simulators. However, this goes beyond the evaluation point that we want to make here.

5.5 Mixed-Fidelity Simulations Are Still Accurate

section 5.4 showed that mixed-fidelity simulations help to keep the amount of required compute resources low. Now we want to show that they still provide sufficient accuracy. To do this, we use the same mixed-fidelity configurations as in section 5.4. However, this time we will evaluate the communication between the qemu hosts by looking again at the slowdowns for the different message lengths.

Figure 5.5 shows the measured median and 99th-percentile slowdowns for the different mixed-fidelity configuration. Note that we omit the configuration for 15 qemu hosts and 5 ns-3 hosts out of brevity, however the results are very similar to the configuration with 20 qemu hosts and no ns-3 hosts. We observe that with an increasing amount of qemu hosts the measured slowdowns look more stable and smooth compared to configurations that use more ns-3 hosts. This is because there are more detailed qemu hosts communicating with each other and recording the data for the evaluation. But we also see that the general shape of the graphs remains the same for all configurations. Therefore, configurations with only a few qemu host, which require fewer resources as we showed in section 5.4, are still accurate enough to observe the general trend of the slowdown curves. For more stable and accurate results, we have to increase the number of qemu hosts requiring more compute resources. Overall, this gives us a trade-off between compute resources and accuracy that we can decide based on our needs. For example, a configuration that lies in the middle of the two extremes could be a good option to keep the required compute resources low while offering decent accuracy.

5.6 Simulations Scale by Decomposing Simulators

Finally, we want to show that we can scale the simulation by decomposing and parallelizing bottleneck simulators. To this end, we focus on decomposing the network simulator by partitioning the network topology and simulating each partition in its own network simulator instance. As already explained in section 5.3 our framework enables us to define and use partitioning strategies within a few lines of code, which makes parallelizing the network simulator an easy endeavor. In this evaluation, we show two cases where we can reduce the total simulation time by splitting the network topology. For both cases we analyze the three partitioning strategies

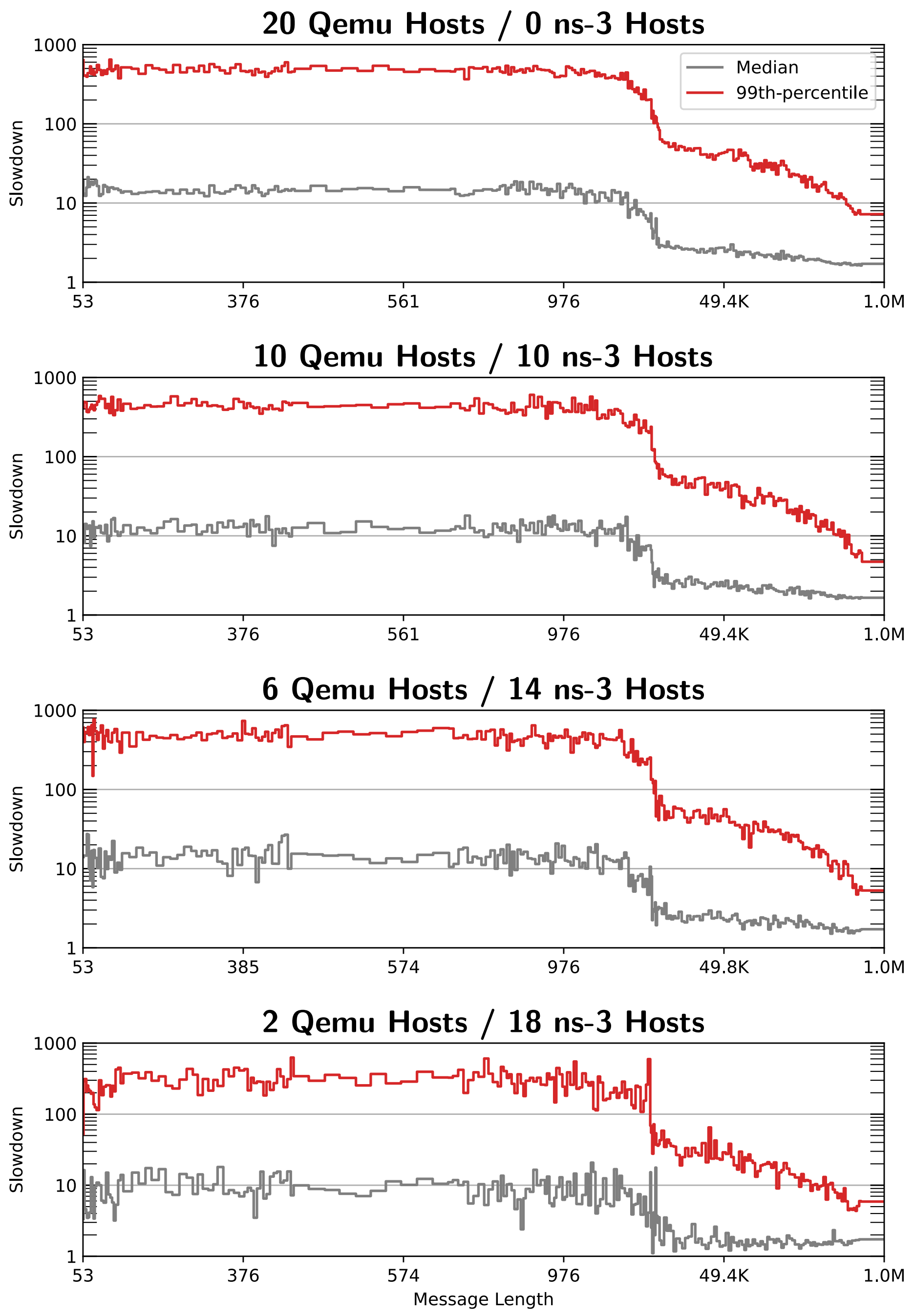
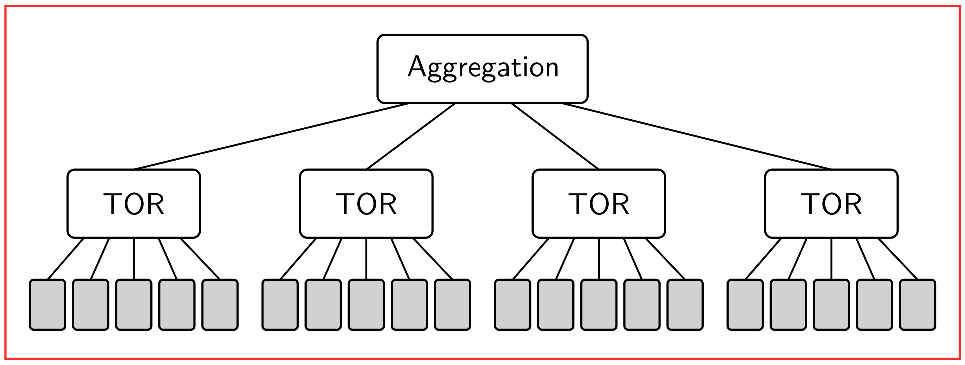
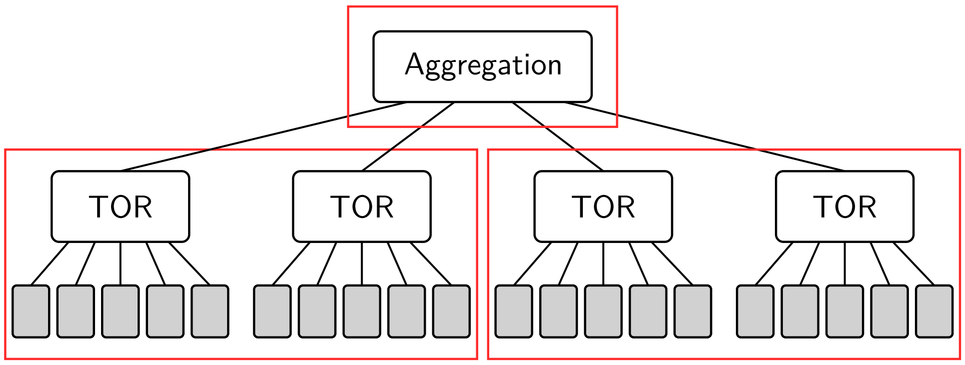


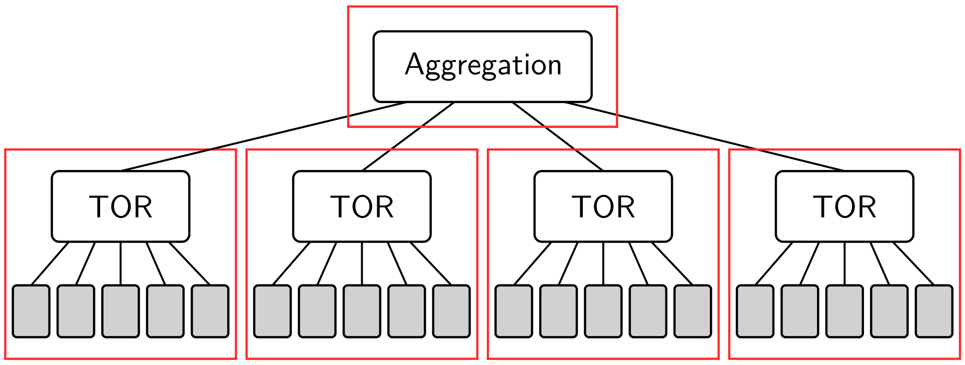
Figure 5.5: Median and 99th-percentile slowdowns measured for different mixed-fidelity configurations.



(a) Partitioning Strategy 1



(b) Partitioning Strategy 2



(c) Partitioning Strategy 3

Figure 5.6: Three partitioning strategies for the network topology that is used for the evaluation. The partitions are marked with red boxes.

Partitioning Strategy	#Processes	Simulation Time [minutes]
1	41	102
2	43	77
3	45	62

Table 5.2: The required number of processes and the simulation time for three different partitioning strategies of a full-system network simulation with 20 hosts. All hosts are simulated with qemu.

Partitioning Strategy	#Processes	Simulation Time [minutes]
1	1	436
2	3	220
3	10	44

Table 5.3: The required number of processes and the simulation time for three different partitioning strategies of a network simulation with 144 hosts. All hosts are simulated in ns-3.

that are shown in Figure 5.6. The first and simplest partitioning defines one partition for the entire network. The second strategy uses three partitions, with the aggregation switches in one and half of the racks in each of the other two partitions. The last partitioning puts each rack in one partition and the aggregation switches in a separate one.

For the first case, we look at a full-system network simulation with 20 hosts distributed among four racks generating Homa traffic where each of them is simulated by qemu. The network simulator models only the switches and the links between them. We run for each partitioning strategy one simulation with the results being presented in Table 5.2. For each configuration, we need 40 processes to simulate the hosts. Depending on the selected partitioning, we then need either one, three or five additional processes for the network. In regard to the simulation time, we observe that the simulation finishes in less time the more partitions we use for the network. Although the network simulator merely models switches and links, it has to communicate and synchronize with all 40 hosts using SimBricks channels. The heavy synchronization with a single simulator therefore slows down the whole simulation leading to longer simulation times. Splitting the network among multiple network simulator instances reduces the number of simulators each instance has to communicate and synchronize with. Thus, the simulation time decreases by parallelizing the network simulator.

For the second case, we look at a network simulation that simulates all hosts in ns-3 and does not use any dedicated simulator. For this, we choose a larger setup with nine racks containing 16 hosts each, giving a total of 144 hosts. We run again three simulations for the different partitioning

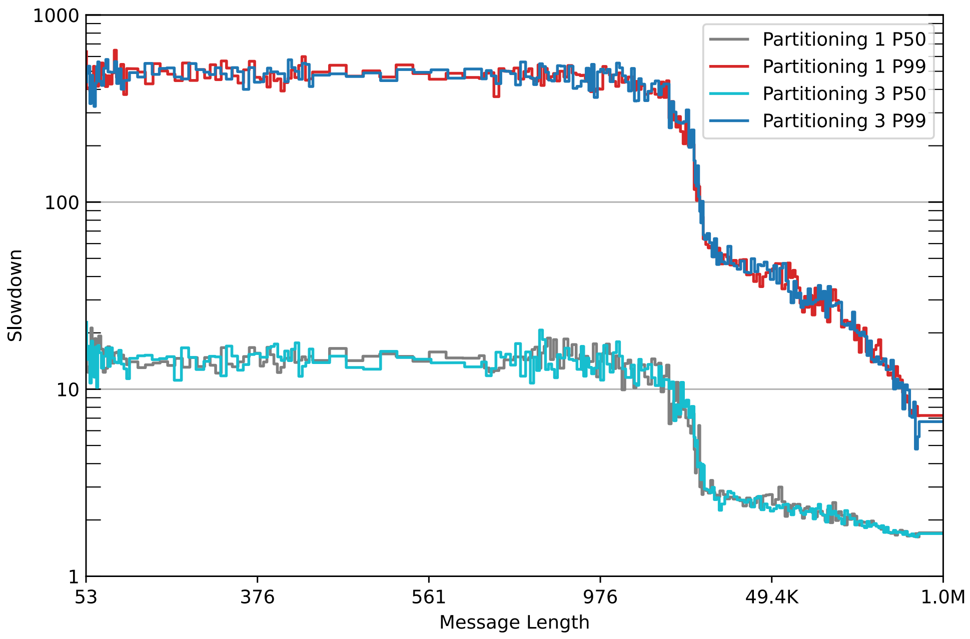


Figure 5.7: Median and 99th-percentile slowdowns for two partitioning strategies of the same simulated system.

strategies and present the results in Table 5.3. As in the previous case, we need one and three network instances respectively for the first two configurations. Since we have now nine racks, we need ten processes for the third partitioning strategy. Once more we observe that the simulation time decreases when increasing the number of partitions. However, this time the reason is not communication and synchronization through SimBricks channels, but the network simulator has to carry out a lot of computation. By splitting the topology among multiple simulator instances, we effectively parallelize and distribute the computation across multiple processes. This leads to a decrease of the simulation time while increasing the number of required physical cores.

Finally, we show that parallelizing the network simulator does not affect the results of the simulation. To this end, we compare the measured slowdowns for partitioning strategies one and three using our network topology with 20 qemu hosts and no ns-3 hosts. As we can see from Figure 5.7 the curves for the median and the 99th-percentile slowdowns are basically the same for both partitioning strategies. Therefore, parallelizing the simulation by decomposing the network simulator does not influence the overall behavior of the simulation.

Chapter 6

Related Work

6.1 Discrete Event Network Simulator

The predominant simulator type for network simulators is the discrete event simulator [11, 24, 28]. These simulators typically model the network at the packet level using discrete events, resulting in a good representation of the network behavior in regard to how packets are processed and move through the network. However, while they are good at capturing the network behavior at the packet level, they do not implement detailed models for devices and hosts in order to capture the architectural behavior of them. This prevents us from capturing the full end-to-end behavior of the system. We therefore use modular simulation to combine multiple different simulator into a full-system simulation enabling end-to-end evaluations. Furthermore, discrete event simulators keep events in an event queue sorted by timestamps and process them sequentially. The sequential processing results in increasing simulation time when increasing the size of the simulated system. Because of this, some network simulators enable users to parallelize the simulation by providing a solution that is specific to the simulator's design. For example the network simulator ns-3 [24] partitions the network and distributes it across multiple processes. To enable communication between the separate processes, ns-3 uses the Message Passing Interface (MPI) [19]. However, the parallelization capabilities offered by network simulators often scale poorly [32] and they are specific to the respective network simulator. With our framework, we provide parallelization that is transparent to the underlying simulators and uses efficient SimBricks channels for communication and synchronization.

6.2 Physical Testbeds

Physical testbeds enable a system to be evaluated in a real environment, giving representative results. But one of the big disadvantages is that physical testbeds are often not available, especially on a large scale. However, there are approaches to change this, for instance Pantheon [30], PlanetLab [5] or Emulab/Netbed [29]. They provide shared physical testbeds that enable researchers to evaluate, for example, transport protocols or congestion control algorithms in real network systems. For that, they maintain network nodes or clusters in realistic systems that can be used to execute custom workloads and network software stacks. For example, Pantheon consists of a realistic testbed comprised of network nodes distributed all over the world in order to evaluate network systems end-to-end on real network paths. Furthermore, these platforms frequently provide additional tooling that makes configuring and using the provided testbeds easier. Although these physical testbeds provide a variety of different network systems, such as wired or wireless networks, we still have only limited control over the system. In case we need a specific network topology or specialized hardware, those testbeds fail to meet our requirements which makes them not helpful. However, with our framework we leverage modular simulation to provide large scale full-system network simulations. This enables us to evaluate network systems end-to-end, while giving the flexibility of modelling a wide variety of systems.

6.3 Simulation With Physical System

One drawback of simulation is that it typically takes a lot of implementation effort to replicate the behavior of the system. This is already the case if we want to model the functional behavior and provide the same interfaces as the real system so that we can run unmodified software, for example. Combinations of physical and simulated or virtual systems can provide a solution to this problem. Mininet [15] uses lightweight virtualization with Linux namespaces to run hosts, software models of switches and virtual Ethernet connections. It models the functional behavior of the system and runs interactively, which is often also described as emulation. Dummynet [25] combines a physical system with simulation by intercepting an application's network communication at the protocol layer and simulating the behavior of the network, such as bandwidth limitations and latencies. This allows running unmodified applications interacting with the real network protocols while giving the opportunity to freely model the network. Similarly, the ns-3 extension Direct Code Execution (DCE) [27] enables us to run almost unmodified applications and network protocols basically natively on the machine in conjunction with the network simulator ns-3. For that DCE intercepts, like Dummynet, the network communication and sends it through the simulated network. By using modular simulation that combines multiple simulators into one simulation, we can leverage appropriate simulators that are able to run unmodified applications and software stacks, such as architectural simulators [2, 23]. Leveraging the physical system provides a fast way to obtain a functional model of a component, but it does not provide deep insight and prohibits exact performance measurements, which are provided by end-to-end simulation.

6.4 Network Performance Estimator

Apart from simulation and physical testbeds, theoretical models [6] can be used to estimate specific behaviors of the network system such as throughput and latency. Theoretical models allow us to compute metrics even for large scale networks within a few minutes, where packet-level simulators such as ns-3 and OMNeT++ take hours or days. However, they only calculate key metrics of the system and do not offer the same visibility as network simulators. Instead of theoretical models other solutions leverage machine learning models to decrease the computational effort that is required for network simulations [13, 31, 32]. By learning the behavior of some parts of the systems using machine learning models and subsequently replacing the respective part with the learned model in the simulation, the runtime of large scale network simulations can decrease significantly. However, this approach can not provide end-to-end application performance. Additionally, it is trained for one specific network configuration so that changing the configuration typically requires re-learning the model, making it inflexible to use.

Chapter 7

Future Work

7.1 Separation Between Specification and Implementation

In our design, we proposed to split the definition of a full-system simulation into a specification and an implementation. The specification should describe the system that we want to simulate, whereas the implementation specifies how we simulate it. The network abstraction that we propose in this thesis is a first step into the direction of separating specification and implementation by providing a mostly simulator-independent network specification. In a second step, we decide how we want to simulate the specified network, for example by choosing which network simulator to use. However, the current implementation of the orchestration framework does not yet fully support this separation for the rest of the system. Instead, many components of the orchestration framework are used as the specification for a system component while also defining how this component is simulated. In the future we want to implement this separation for all components of the orchestration framework and provide it as a general abstraction in SimBricks. This will make it even easier and more practical to define and assemble large scale full-system network simulations. The separation will allow us to define the system once and afterwards let us select different concrete implementations depending on our needs, such as simulation detail or simulation time.

7.2 Extend Network Abstraction

Currently, the network abstraction supports only fundamental components that are sufficient to define basic networks systems. Typical network simulators offer a rich library of network components, different types of communication channels, and advanced features, such as modelling transmission errors. Therefore, we want to further extend the network abstraction in the future, in order to support more features of the network simulators. To do this, we need to expand our network specification in the orchestration framework so that we are able to specify a greater variety of network systems. At the same time, we have to add support for more components and advanced features to the part of our framework that is implemented in the network simulator to be able to use them in our network abstraction.

7.3 Extend Network Simulator Support

As of right now, we only support the network simulator ns-3. In the future we want to add more network simulators to our framework. For that, we need to implement the network abstraction in each network simulator that we want to support. However, the current implementation of the network specification and the generation of the network description were mainly designed with the goal of supporting ns-3. This means, that we might have to adapt the network specification and the format of the description in order to support other network simulators. For this, we could add another abstraction layer between the network specification and the simulator, which takes care of generating a description that the respective simulator is able to understand. This abstraction layer might also have to “translate” parts of the specification to accommodate for the different feature sets and abstractions of the various network simulators.

Chapter 8

Conclusion

In this thesis we presented a framework that enables practical large scale full-system network simulations. It addresses the shortcomings of full-system simulation for large scale networks that are high needs for compute resources and long simulation times. The framework separates the specification and the implementation of the simulated system in order to provide a clean abstraction between the description of the system and its implementation. With the main focus on network simulations, it provides an abstraction for network simulators that allows the network to be specified independently of the simulator. The abstraction also enables the easy creation of mixed-fidelity simulations and splitting the network topology into multiple partitions, each simulated by a separate network simulator instance. Through the abstraction the simulation is assembled and configured without having to configure the respective simulators directly. In order to combine multiple simulators into a full-system simulation, the framework integrates into SimBricks. In the evaluation we were able to show how easy it is to assemble full-system network simulations and configure mixed-fidelity and parallelized simulations using the framework. Furthermore, we have shown that mixed-fidelity simulations and decomposing bottleneck simulators reduce the required amount of compute resources and simulation time, while still providing accurate results.

Bibliography

[1] Mohammad Alizadeh et al. “Data center TCP (DCTCP).” In: *Proceedings of the ACM SIGCOMM 2010 Conference*. SIGCOMM ’10. New Delhi, India: Association for Computing Machinery, 2010, pp. 63–74. ISBN: 9781450302012. DOI: 10.1145/1851182.1851192. URL: https://doi.org/10.1145/1851182.1851192.

[2] Nathan Binkert et al. “The gem5 simulator.” In: *SIGARCH Comput. Archit. News* 39.2 (Aug. 2011), pp. 1–7. ISSN: 0163-5964. DOI: 10.1145/2024716.2024718. URL: https://doi.org/10.1145/2024716.2024718.

[3] Pat Bosshart et al. “P4: programming protocol-independent packet processors.” In: *SIGCOMM Comput. Commun. Rev.* 44.3 (July 2014), pp. 87–95. ISSN: 0146-4833. DOI: 10.1145/2656877.2656890. URL: https://doi.org/10.1145/2656877.2656890.

[4] V. Cerf and R. Kahn. “A Protocol for Packet Network Intercommunication.” In: *IEEE Transactions on Communications* 22.5 (1974), pp. 637–648. DOI: 10.1109/TCOM.1974.1092259.

[5] Brent Chun et al. “Planetlab: an overlay testbed for broad-coverage services.” In: *ACM SIGCOMM Computer Communication Review* 33.3 (2003), pp. 3–12.

[6] Florin Ciucu and Jens Schmitt. “Perspectives on network calculus: no free lunch, but still good value.” In: *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*. SIGCOMM ’12. Helsinki, Finland: Association for Computing Machinery, 2012, pp. 311–322. ISBN: 9781450314190. DOI: 10.1145/2342356.2342426. URL: https://doi.org/10.1145/2342356.2342426.

[7] Faeze Faghih et al. “SmartNICs in the Cloud: The Why, What and How of In-network Processing for Data-Intensive Applications.” In: *Companion of the 2024 International Conference on Management of Data*. SIGMOD/PODS ’24. Santiago AA, Chile: Association for Computing Machinery, 2024, pp. 556–560. ISBN: 9798400704222. DOI:

10.1145/3626246.3654690. URL: https://doi.org/10.1145/3626246.3654690.

[8] Daniel Firestone et al. “Azure Accelerated Networking: SmartNICs in the Public Cloud.” In: *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. Renton, WA: USENIX Association, Apr. 2018, pp. 51–66. ISBN: 978-1-939133-01-4. URL: https://www.usenix.org/conference/nsdi18/presentation/firestone.

[9] Serhat Arslan. *HomaL4Protocol-ns-3 - NS3 implementation of Homa Transport Protocol*. https://github.com/serhatarslan-hub/HomaL4Protocol-ns-3. Retrieved Nov 16, 2024. 2024.

[10] John Ousterhout and HomaModule Contributors. *HomaModule - A Linux kernel module that implements the Homa transport protocol*. https://github.com/PlatformLab/HomaModule. Retrieved Nov 16, 2024. 2024.

[11] htsim Authors. *htsim Network Simulator*. https://github.com/Broadcom/csg-htsim. Retrieved Nov 16, 2024. 2024.

[12] INET Authors. *INET Framework*. https://inet.omnetpp.org/. Retrieved Nov 16, 2024. 2024.

[13] Charles W. Kazer et al. “Fast Network Simulation Through Approximation or: How Blind Men Can Describe Elephants.” In: *Proceedings of the 17th ACM Workshop on Hot Topics in Networks*. HotNets ’18. Redmond, WA, USA: Association for Computing Machinery, 2018, pp. 141–147. ISBN: 9781450361200. DOI: 10.1145/3286062.3286083. URL: https://doi.org/10.1145/3286062.3286083.

[14] Elie F. Kfoury et al. “A Comprehensive Survey on SmartNICs: Architectures, Development Models, Applications, and Research Directions.” In: *IEEE Access* 12 (2024), pp. 107297–107336. DOI: 10.1109/ACCESS.2024.3437203.

[15] Bob Lantz, Brandon Heller, and Nick McKeown. “A network in a laptop: rapid prototyping for software-defined networks.” In: *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*. Hotnets-IX. Monterey, California: Association for Computing Machinery, 2010. ISBN: 9781450304092. DOI: 10.1145/1868447.1868466. URL: https://doi.org/10.1145/1868447.1868466.

[16] Hejing Li, Jialin Li, and Antoine Kaufmann. “SimBricks: end-to-end network system evaluation with modular simulation.” In: *Proceedings of the ACM SIGCOMM 2022 Conference*. SIGCOMM ’22. Amsterdam, Netherlands: Association for Computing Machinery, 2022, pp. 380–396. ISBN: 9781450394208. DOI: 10.1145/3544216.3544253. URL: https://doi.org/10.1145/3544216.3544253.

[17] Hejing Li et al. *SplitSim: Large-Scale Simulations for Evaluating Network Systems Research.* 2024. arXiv: 2402.05312 [cs.NI]. URL: https://arxiv.org/abs/2402.05312.

[18] Nick McKeown et al. "OpenFlow: enabling innovation in campus networks." In: *SIGCOMM Comput. Commun. Rev.* 38.2 (Mar. 2008), pp. 69–74. ISSN: 0146-4833. DOI: 10.1145/1355734.1355746. URL: https://doi.org/10.1145/1355734.1355746.

[19] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard Version 4.1.* Nov. 2023. URL: https://www.mpi-forum.org/docs/mpi-4.1/mpi41-report.pdf.

[20] Behnam Montazeri et al. "Homa: a receiver-driven low-latency transport protocol using network priorities." In: *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication.* SIGCOMM '18. Budapest, Hungary: Association for Computing Machinery, 2018, pp. 221–235. ISBN: 9781450355674. DOI: 10.1145/3230543.3230564. URL: https://doi.org/10.1145/3230543.3230564.

[21] nsnam. *ns-3 | a discrete-event network simulator for internet systems.* https://www.nsnam.org/. Retrieved Nov 16, 2024. 2024.

[22] John Ousterhout. "A Linux Kernel Implementation of the Homa Transport Protocol." In: *2021 USENIX Annual Technical Conference (USENIX ATC 21).* USENIX Association, July 2021, pp. 99–115. ISBN: 978-1-939133-23-6. URL: https://www.usenix.org/conference/atc21/presentation/ousterhout.

[23] QEMU Authors. *QEMU – the FAST! processor emulator.* https://www.qemu.org/. Retrieved Nov 16, 2024. 2024.

[24] George F Riley and Thomas R Henderson. "The ns-3 network simulator." In: *Modeling and tools for network simulation.* Springer, 2010, pp. 15–34.

[25] Luigi Rizzo. "Dummynet: a simple approach to the evaluation of network protocols." In: *SIGCOMM Comput. Commun. Rev.* 27.1 (Jan. 1997), pp. 31–41. ISSN: 0146-4833. DOI: 10.1145/251007.251012. URL: https://doi.org/10.1145/251007.251012.

[26] Arjun Roy et al. "Inside the Social Network's (Datacenter) Network." In: *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication.* SIGCOMM '15. London, United Kingdom: Association for Computing Machinery, 2015, pp. 123–137. ISBN: 9781450335423. DOI: 10.1145/2785956.2787472. URL: https://doi.org/10.1145/2785956.2787472.

[27] Hajime Tazaki et al. “Direct code execution: revisiting library OS architecture for reproducible network experiments.” In: *Proceedings of the Ninth ACM Conference on Emerging Networking Experiments and Technologies*. CoNEXT ’13. Santa Barbara, California, USA: Association for Computing Machinery, 2013, pp. 217–228. ISBN: 9781450321013. DOI: `10.1145/2535372.2535374`. URL: `https://doi.org/10.1145/2535372.2535374`.

[28] András Varga and Rudolf Hornig. “An overview of the OMNeT++ simulation environment.” In: *1st International ICST Conference on Simulation Tools and Techniques for Communications, Networks and Systems*. 2010.

[29] Brian White et al. “An integrated experimental environment for distributed systems and networks.” In: *SIGOPS Oper. Syst. Rev.* 36.SI (Dec. 2003), pp. 255–270. ISSN: 0163-5980. DOI: `10.1145/844128.844152`. URL: `https://doi.org/10.1145/844128.844152`.

[30] Francis Y. Yan et al. “Pantheon: the training ground for Internet congestion-control research.” In: *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. Boston, MA: USENIX Association, July 2018, pp. 731–743. ISBN: 978-1-939133-01-4. URL: `https://www.usenix.org/conference/atc18/presentation/yan-francis`.

[31] Qingqing Yang et al. “DeepQueueNet: towards scalable and generalized network performance estimation with packet-level visibility.” In: *Proceedings of the ACM SIGCOMM 2022 Conference*. SIGCOMM ’22. Amsterdam, Netherlands: Association for Computing Machinery, 2022, pp. 441–457. ISBN: 9781450394208. DOI: `10.1145/3544216.3544248`. URL: `https://doi.org/10.1145/3544216.3544248`.

[32] Qizhen Zhang et al. “MimicNet: fast performance estimates for data center networks with machine learning.” In: *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*. SIGCOMM ’21. Virtual Event, USA: Association for Computing Machinery, 2021, pp. 287–304. ISBN: 9781450383837. DOI: `10.1145/3452296.3472926`. URL: `https://doi.org/10.1145/3452296.3472926`.