Synchronized Performance Evaluation Methodologies for Communication Systems

Von der Fakultät für Mathematik, Informatik und Naturwissenschaften der RWTH Aachen University zur Erlangung des akademischen Grades eines Doktors der Naturwissenschaften genehmigte Dissertation

vorgelegt von

Diplom-Informatiker

Elias David Weingärtner

aus Emmendingen

Berichter:
Prof. Dr.-Ing. Klaus Wehrle
Prof. Alejandro Buchmann, PhD.

Tag der mündlichen Prüfung: 19.6.2013

Reports on Communications and Distributed Systems

edited by
Prof. Dr.-Ing. Klaus Wehrle
Communication and Distributed Systems,
RWTH Aachen University

Volume 8

Elias David Weingärtner

Synchronized Performance Evaluation Methodologies for Communication Systems

Shaker Verlag
Aachen 2013

Bibliographic information published by the Deutsche Nationalbibliothek
The Deutsche Nationalbibliothek lists this publication in the Deutsche Nationalbibliografie; detailed bibliographic data are available in the Internet at http://dnb.d-nb.de.

Zugl.: D 82 (Diss. RWTH Aachen University, 2013)

ISBN 978-3-8440-2227-8
ISSN 2191-0863

Shaker Verlag GmbH • P.O. BOX 101818 • D-52018 Aachen
Phone: 0049/2407/9596-0 • Telefax: 0049/2407/9596-9
Internet: www.shaker.de • e-mail: info@shaker.de

Contents

Abstract

Researchers and developers typically rely on analytical techniques, simulation-based approaches or real-world experiments for the performance evaluation of communication systems. All of these methods have unique strengths. Network simulations are well suitable for conducting virtual experiments of very high scale; the deterministic and thus repeatable execution of simulations also make them an essential method for parameter studies. Performance evaluations carried out with real-world systems allow for investigations with the topmost possible level of realism and detail. Finally, analytical techniques abstract away from the technical environment of a communication system and hence enable the deduction of general evidence.

Unfortunately, all performance evaluation techniques also suffer from a number of individual shortcomings. The analytical formalization of complex communication protocols and their behavior is often very difficult. Network simulations model only the essential functionality of a communication system, making it problematic to apply this technique for analyzing resource usage or system-specific effects. Finally, evaluations carried out with real-world software prototypes are often perturbed by uncontrollable conditions in the environment. In addition, the high amount of hardware and manpower required for real-world performance evaluations of larger scale make such trials often very costly.

Hybrid performance evaluation methodologies are a promising approach for overcoming these issues. The concept of *network emulation* combines the flexibility and scalability of network simulation with the credibility and the level of detail associated with performance evaluations using real-world systems. A second important hybrid methodology is *hardware/network co-simulation*. The idea of this methodology is to integrate different simulation tools to bridge between their individual application domains, for instance network simulation and hardware modeling.

We contribute to the field of hybrid evaluation tools for communication systems in different ways. First, we introduce the concept of *Synchronized Hybrid Evaluation* that generalizes the idea of hybrid performance evaluation and thus is able to subsume a number of existing hybrid techniques. The second central cornerstone of our work is the synchronization and the virtualization of time. Existing network emulation frameworks require real-time capable simulations, which limits the application of network emulation to a rather narrow set of scenarios. Our work on *Synchronized Network Emulation* removes this burden by synchronizing the execution of the network simulation with virtualized communication systems. A further contribution to the field of network emulation is our work on *Device Driver-enabled Wireless Network Emulation*. It tightly integrates the simulated environment with the operating system context of the software prototype. As we further elaborate in this dissertation, both concepts and their respective implementations substantially extend the applicability of network emulation.

In addition, we contribute to the field of hybrid evaluation techniques with two additional frameworks. First, we propose the integration of a SystemC-based hardware simulator with a network simulation tool, aiming at a network centric design of embedded systems. Second, we show that virtualizing the progression of time and synchronizing the execution of virtualized software prototypes is helpful for building distributed debugging and monitoring tools.

Kurzfassung

Die Leistungsbewertung von Kommunikationssystemen wird derzeit meist entweder mit mathematisch-analytischen oder simulativen Methoden durchgeführt; ebenfalls sind Messungen mit realen Prototypen in Textnetzwerken (Test-Beds) verbreitet. All diese Ansätze besitzen individuelle Stärken: Die hohe Skalierbarkeit von Simulationen ermöglicht es, verteilte Kommunikationssysteme oder große Topologien zu untersuchen. Da Netzwerksimulationen eine deterministische Ausführung erlauben, ist es überdies möglich, Simulationsläufe beliebig zu wiederholen, was die Durchführung von Parameterstudien sehr vereinfacht. Im Gegensatz dazu liefern Messungen mit realen Prototypen Ergebnisse hoher Detailtreue und Güte. Mathematisch-Analytische Verfahren können generelle Aussagen über das Systemverhalten unabhängig von der konkreten Technologie liefern.

Ein großes Problem sind jedoch die individuellen Einschränkungen dieser Verfahren. So ist die mathematisch-analytische Formalisierung von komplexen Kommunikationssystemen sehr schwierig und oft nur eingeschränkt machbar. Der Abstraktionsgrad von Simulationen erschwert die Analyse von komplexem Systemverhalten, das sich beispielsweise aus der Interaktion mit dem Betriebssystem ergibt. Während Messungen mit realen Systemen solchen Einschränkungen nicht unterliegen, sind Experimente auf Testbeds typischerweise durch den hohen Hardwarebedarf sehr teuer. Ein großes Problem sind hier ebenfalls eine Vielzahl von schwer zu kontrollierenden externen Einflüssen, beispielsweise Hintergrundverkehr im Netzwerk oder eine nicht-deterministische Mobilität der Netzwerkknoten.

Ein Weg, den individuellen Schwächen dieser Verfahren zu begegnen, sind hybride Ansätze zur Leistungsbewertung. So vereint das Konzept der Netzwerkemulation die Stärken von Simulationen mit der Detailtreue bei Messungen mit realen Systemen. Ein weiterer solcher Ansatz ist das Konzept des Hardware-Software Co-Designs, in dem verschiedene Leistungsbewertungs- und Hardware-Entwurfsmethodiken in einem hybriden Werkzeug zusammengefasst werden.

In dieser Arbeit werden verschiedene Beiträge zu Forschungsfragestellungen im Bereich der hybriden Leistungsbewertungswerkzeuge besprochen. Das Konzept der *synchronisierten hybriden Evaluation* stellt eine Generalisierung hybrider Leistungsbewertungsverfahren dar und erlaubt es, eine Reihe von bestehenden hybriden Messverfahren zu subsubmieren. Das Hauptaugenmerk der Arbeit liegt auf einer daraus abgeleiteten Technologie, der *synchronisierten Netzwerkemulation*. Dieses Verfahren erlaubt es, Netzwerkemulationen mit Simulationen beliebiger Laufzeitkomplexität zu kombinieren und durchzuführen. Die *Gerätetreiber-basierte Netzwerkemulation für drahtlose Übertragungsverfahren (DDWNE)* ermöglicht es, den simulativen Kontext eng an die Ausführungsumgebung der realen Prototypen in einem Emulationsszenario zu binden.

In weiteren Verlauf werden zwei neue Evaluations-Werkzeuge vorgestellt. Zunächst wird ein Ansatz für die Netzwerk-zentrierte Entwicklung von eingebetteten Systemen diskutiert, das intern eine Hardware-Simulation auf Basis von SystemC mit einem Ereignis-basierten Simulator kombiniert. Abschließend wird ein Ansatz beschrieben, der mit Hilfe von Basistechnologien aus der synchronisierten Netzwerksimulation eine Umgebung für die verteilte Fehlersuche und Beobachtung der Ausführung von Kommunikationssystemen erlaubt.

Acknowledgments

Foremost I'd like to express my strong gratitude to my advisor, Klaus Wehrle. Six years ago, Klaus supported me in obtaining a PhD scholarship that enabled my journey with the fabulous COMSYS crew. He pointed me to a research problem that evolved into this doctoral thesis. Beyond that, I thank Klaus very much for creating an open-minded, creative and enjoyable environment. I will never forget the barbecues, skiing in Kleinwalsertal and all our intense conversations!

I'd also like to greatly thank Prof. Alejandro Buchmann not only for co-advising this dissertation, but also for all the exciting discussions we had during all the MAKI and QuaP2P-2 meetings in Darmstadt, Aachen and Kleinwalsertal.

This dissertation would not have been possible without the outstanding contributions by the highly talented and gifted students and colleagues I was lucky to work with. I'd like to give exceptional kudos to Florian Schmidt and Hendrik vom Lehn for their countless efforts with SliceTime. I deeply acknowledge the work of Marko Ritter for taming gdbserver-xen and Apache Thrift. René Glebke, Martin Lang and Alexander Hocks deserve a lot of praise for their patience with VODSim. I'd also like to thank Suraj Prabhakaran and Martin Lindner for additional SliceTime measurements. Finally, I'm very grateful for the fruitful cooperation with Stefan Schürmans, who introduced me to the concept of Hardware-Software Co-Design.

I owe the entire COMSYS team a big debt of gratitude. Tobi, I thank you so much for all the scientific and non-scientific discussions and for your exactness and diligence, not only when we've been writing papers together! Jó, I envy your endless curiosity and your enthusiasm! Raimondas and Georg, you have been both wonderful office mates. I have learned a lot from you! Ismet, thank you for being my No. 1 lifesaver by supplying me with coffee and snacks and for many great down-to-earth advices. I appreciate Petra, Ulrike and Rainer for being so helpful with all organizational issues I ran into - and Dirk and Janosch for taking over a lot of common duties in a self-forgetting manner! All of you have always been a wonderful source of inspiration and good reason to procrastinate. I will never forget three of you spontaneously performing “Bochum” by Herbert Grönemeyer in my office!

I am very thankful for the great support of my amazing friends. You have always been there to listen. You have always been great motivators, for example when we traveled the world together or simply by giving me a smile.

Finally, I'd like to express the strongest gratitude to my parents and my entire family. Rainer, Gertrud and Klaus Weingärtner have continuously supported my curiosity from an early age. I also deeply thank my brother Simon for letting me spend so much time on our C64 and all other family computers. This eventually made me become a computer scientist!

1 Introduction

One fundamental cornerstone of computer science is the goal of maximizing the efficiency and the reliability of information processing. In the context of computer networks this means that the transport of data has to be accomplished with the required and best achievable *performance*. In this dissertation we discuss new hybrid methodologies and tools for the performance analysis of computer networks.

In general the performance of a computer network is dependant on the complex interplay of network hardware (for example hosts, routers and switches) and software components like network protocols stacks that manage the exchange of data. In addition, a number of external influences, for instance interference in the case of wireless communication, directly affect network performance.

We term methodologies for the quantification of network performance *performance evaluation techniques*. One example of such a technique is network simulation.

1.1 Requirements for Evaluation Techniques

Performance evaluation studies in the domain of computer networks may range from a scalability study of an abstractly formulated network protocol to practical experiments, for example targetting the question how many requests a server can handle per second. Given this wide scope, there are also many and partially conflicting requirements for the performance evaluation in the domain of network systems:

- **Scalability:** Many performance evaluation studies investigate how well a protocol or communication system operates if it is deployed to an increasing number of hosts. For this reason, scalability studies require the used performance evaluation technique to be able to model a large number of end systems.
- **Controllability:** The overall performance of a network system is depending on a large and sometimes even unmanageable set of influences. Examples of

such are the system configuration, side effects such as varying cross traffic in the network or potential node mobility.

The goal of many evaluation studies is to explore how a certain system parameter affects the overall performance. Such *parameter studies* require the vast set of influences to be kept static in order to quantify the analyzed parameter's effect on the overall performance. The controllability describes how well an evaluation technique allows the user to command these influences.

- **Realism:** In many cases performance evaluation studies try to provide answers how well a specific mechanism, for example a network protocol or application, would operate in a real context, for example if deployed widely on actual Internet hosts. In order to be able to derive substantiated conclusions from a performance evaluation study, it is therefore important for the methodology to provide realistic investigation conditions.
- **Flexibility:** The specific demands for an actual performance evaluation study differ from case to case. For example, while one evaluation study might investigate the downloading performance of BitTorrent clients, some other study might analyze the routing efficiency of an ad-hoc routing protocol. A flexible performance evaluation technique is universal enough to cover a wide range of evaluation tasks and can be used for quantifying the performance of the system under test using a diverse set of performance metrics.
- **Ease of Use:** The focus of network research is mainly the development and the improvement of new and future communication systems. Therefore, it is crucial for a performance evaluation technique to be easy to apply in order to make the methodology accessible for a wide range of individuals.
- **Cost-Effectiveness:** An ideal performance evaluation technique requires both low hardware as well as few human resources for an evaluation to be conducted.

1.2 Performance Evaluation Techniques

Today, we mostly rely either on *real-world measurements*, *analytical methods* or *network simulations* for quantitative analyses of network systems.

1.2.1 Network Testbeds / Real-World Measurements

The most natural approach to conduct performance evaluations is to take *network measurements* either in a network like the Internet or in a specially deployed network testbed (cf. Sec. 2.1.2.2). We typically attribute a very high degree of realism to such measurements, as they are obtained in an actual networking environment.

The major drawbacks of this performance evaluation technique are the limited scalability and the restricted controllability of the experiment setting. First, even large-scale global testbeds such as PlanetLab [CCR$^+$03] range between few hundreds and less than two thousand nodes in size. Second, repeating measurement runs under equivalent environmental conditions is often complicated or even unfeasible due to an often insufficient isolation of the testbed environment.

1.2.2 Network Simulators

The core strengths of network simulation are scalability, flexibility and controllability. Modern network simulation tools like ns-3 [ns311] or OMNeT++ [VH08] are easily able to support performance evaluation studies with thousands of network nodes [Ril03, WvLW09]. By providing a rich set of software models of communication systems and network protocols, network simulators allow one to flexibly compose simulation scenarios that resemble almost any imaginable computer network. As the network is entirely modeled in software, all influences, even external influences such as node mobility, can be controlled in a deterministic fashion. Network simulations are entirely repeatable. Thus, they are often used for parameter studies.

The scalability of network simulation is a result of the underlying fundamental concept of abstraction. Abstraction is also the core source of shortcomings of this methodology. Simulation models only recreate the essential functional behavior of their real-world counterparts and disregard the system context of a network protocol and its run-time environment, for example the influence of an operating system. Such abstractions and simplifications may impact the validity of simulation-based research [FP01, FK03]. Hence, the degree of realism delivered by network simulations is limited and sometimes even questionable [KCC05]. We discuss the methodology of network simulation and according tools in detail later in Section 2.1.2.3.

1.2.3 Analytic Models

Analytic models explicitly describe the behavior and the performance of communication systems using mathematical and logical principles. Analytic models are well-suited for analyzing protocols without scalability constraints (for example due to simulation complexity or testbed size). The process of describing a system analytically (for example, using differential equations or queuing theory) is typically complicated. It also requires even stronger assumptions and simplifications than network simulation models [FP01]. A brief survey of analytical performance evaluation methodologies follows later in Section 2.1.2.1.

1.3 Hybrid Evaluation Methodologies

One way to cope with the shortcomings of network simulation, analytical methods and measurement-based studies is to form *hybrid evaluation methodologies* (cf. Fig. 1.1). They integrate different evaluation methods with the goal of mutually combining their distinct strengths. For example, Gu et al. have integrated a simulation with an analytic fluid model for the analysis of TCP in wide-area networks [GLT04].

Leaving analytic models aside, we focus on hybrid evaluation methods incorporating a network simulation with different executable representations of soft- and hardware systems. The most prominent such technique is *network emulation* [Fal99]. It brings together the flexibility of network simulation with the precision of measurements taken on real-world communication systems. For example, network emulation tools enable the investigation of software for wireless networks in a fully controlled and isolated software environment.

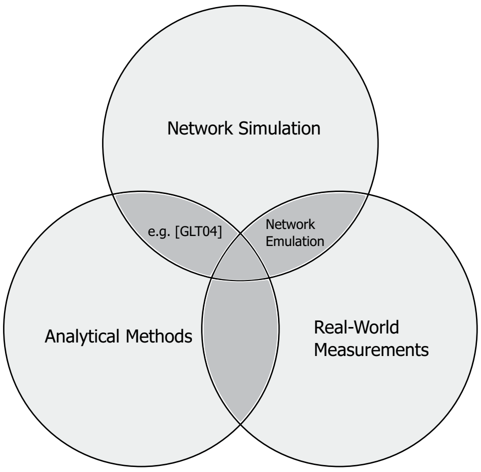


Figure 1.1 There are three basic classes of evaluation methodologies that are commonly applied for analyzing the performance of communication protocols and network systems, namely network simulation, analytical techniques and real-world measurements. Hybrid performance methodologies (see shaded areas) combine different singular techniques to overcome individual limitations of simulation, analytical and measurement-based approaches.

A second interesting hybrid method is *hardware/network* co-simulation. Here a full-system simulator that models an entire hardware platform is conjoined with a network simulation. One exemplary application area of this method is the co-design of software and hardware for networked embedded systems.

1.4 Challenges in Hybrid Evaluation

The development and application of hybrid evaluation frameworks, most particularly network emulation, faces different challenges. We now briefly discuss the ones that are in the focus of this dissertation.

1.4.1 Synchronization and Integration of Timing Domains

One fundamental issue, particularly with network emulation, is the integration of the different time representations that are used by simulation tools and communications software. Both network and hardware simulators mostly rely on the paradigm of discrete event-based simulation. Such a simulation consists of a series of discrete events with an associated event execution time. Once an event has been processed, the simulation time is advanced to the execution time of the next event. By contrast, software running on real hardware and a real operating system uses a uniformly progressing *wall-clock time*.

Existing network emulation frameworks pin the execution of simulation events to the corresponding wall-clock time. The network simulator then simply waits until the according point in wall-clock time is reached and executes the simulation event. This imposes the implicit assumption that the network simulation can be executed in real-time. However, if the simulation requires more events to be processed than the simulation machine can handle in real-time, the simulation starts lagging behind. Possible consequences are a flawed protocol behavior due to unwanted expirations of protocol timers and artifacts such as unpredictably varying or largely increased network latency.

The real-time requirement is a fundamental constraint of network emulation that directly limits its scalability and hinders the use of computationally complex simulation models. We further elaborate the synchronization problem in Section 2.1.5.

1.4.2 Interoperability of Evaluation Techniques and Tools

A second challenge associated with hybrid evaluation frameworks that incorporate evaluation techniques and tools from different classes is the interoperability between them. For example, abstract simulation models of a communication system typically omit functional properties or protocol fields like packet payload which are not required to carry out plain simulations. However, some of these omitted functionalities are vital for the interaction with other evaluation frameworks, for example hosts in a real-world testbed or a hardware simulator.

In order to increase the flexibility of hybrid evaluation frameworks we therefore need to develop mechanisms that enable the flexible composition and the exchange of data among a set of different evaluation frameworks and tools.

1.4.3 Realistic and Contextual Integration for Hybrid Evaluation Frameworks

Contemporary hybrid network evaluation frameworks primarily enable the communication and the interaction of nodes being modeled using either a network simulation, a hardware simulator or real-world systems. This can be achieved by providing adequate interfaces for the communication between them.

However, there is a set of scenarios where only enabling the communication between hosts modeled by different evaluation techniques is not sufficient. One example are emulation studies of wireless software for legacy operating systems in which a network simulator is used to model the environment, most notably the wireless channel or the node movement. In order to provide the wireless software with the illusion of being part of the modeled network, not only the network traffic needs to be interfaced with the software, but also the simulated context. Examples for such contextual information are signal strengths of received packets or a simulated localization information. In order to have the wireless software “perceive” this context, a tight integration of the simulated context with the software execution environment is required.

This example illustrates that we need mechanisms for closely interlinking the contexts modeled by different evaluation techniques in order to increase the overall degree of realism delivered by hybrid evaluation frameworks.

1.5 Research Questions

We derive three distinct research questions from the previous discussion of challenges and requirements faced by hybrid performance evaluation methods and tools. Throughout the remainder of this dissertation we aim at providing answers and elaborate discussions of these questions.

(Q1) How to increase the flexibility of hybrid performance evaluation?
We devise a novel architecture that enables the modular composition of hybrid evaluation tool chains using a set of reusable System Representation (SR) modules, namely network simulators, virtual machines and hardware simulations. All SRs communicate using two simple standard interfaces.

(Q2) How to improve the scalability of hybrid evaluation methods, in particular network emulation?
We explore a synchronization scheme that is capable of eliminating the requirement of real-time SR execution. This constraint is present in the vast majority of hybrid evaluation frameworks and so far has limited the scalability of network emulation.

(Q3) How to ease software testing using hybrid evaluation environments?
We develop technologies that enable the evaluation of real-world software in complex and fully simulated environments. In addition, we explore how we can employ the virtualization of time, which is a by-product of our synchronization approach, for evaluating communication software.

1.6 Contributions

Triggered by the aforementioned research questions this thesis discusses five contributions to the domain of hybrid performance evaluation for communication systems. Figure 1.2 visualizes the relationship between the research questions and our contributions C1-C5.

(C1) Synchronized Hybrid Evaluation

We propose the concept of Synchronized Hybrid Evaluation (SHE). It unifies different hybrid evaluation techniques, most notably network emulation and hardware/network co-simulation. The idea is to incorporate distinct types of *system representations* into a hybrid evaluation framework. A system representation, for example, can either be a real-world communication system, a network simulation or a full-system simulator. For the purpose of synchronization we provide a consistent progression of virtual continuous time to all

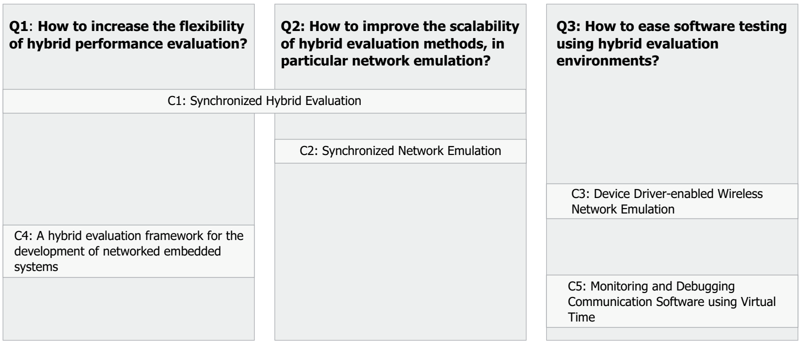

Q1: How to increase the flexibility of hybrid performance evaluation?
Q2: How to improve the scalability of hybrid evaluation methods, in particular network emulation?
Q3: How to ease software testing using hybrid evaluation environments?
C1: Synchronized Hybrid Evaluation
C2: Synchronized Network Emulation
C3: Device Driver-enabled Wireless Network Emulation
C4: A hybrid evaluation framework for the development of networked embedded systems
C5: Monitoring and Debugging Communication Software using Virtual Time

Figure 1.2 Illustruation how our contributions cover the stated research questions.

system representations. SHE fully decouples the progression of virtual time from wall-clock time. This enables the system representations to operate faster or slower than real-time without causing time drifts in the virtual time domain.

(C2) Synchronized Network Emulation [WSvL+11, Sch08]

With SliceTime we present a synchronized network emulation platform that allows the analysis of unmodified virtualized communication systems together with network simulations of any complexity. SliceTime tightly synchronizes the execution of virtual machines with a network simulation. As demonstrated later in this thesis by three use case scenarios, SliceTime enables large-scale simulation-based emulation scenarios to be conducted on legacy PC hardware.

(C3) Device Driver-enabled Wireless Network Emulation [WvLW11, vL10]

Our work on Device Driver-enabled Wireless Network Emulation (DDWNE) makes it possible to integrate a simulation context tightly with arbitrary communication software running on an unmodified operating system. Our corresponding implementation for 802.11 enables legacy software to interact with an ns-3 simulation using an interface that resembles a real wireless networking adapter.

(C4) A Hybrid Evaluation Framework for the Development of Networked Embedded Systems [SWK+10]

We present a hybrid methodology that eases the design of networked embedded systems; it makes uses of a network simulation to provide a simulated network context to a SystemC-based hardware model. This results in an increased flexibility of hardware/software co-design, as future networked embedded systems can be evaluated using the modeling versatility of network simulation tools.

(C5) Monitoring and Debugging Communication Software using Virtual Time [WRSW10, Rit09]

We show that the time synchronization of virtual machines can serve as a basic infrastructure for a distributed debugging and monitoring environment

[WRSW10, Rit09]. Our corresponding framework allows the analysis of kernel-level software in a fully isolated environment, using primitives such as distributed break points. Being more a by-product of this thesis, this result shows that the modularization of hybrid evaluation environments allows existing evaluation setups to be reshaped for new analysis purposes.

Relations of Contributions

The contributions C2 - C5 describe implementations of hybrid evaluation platforms that are derived from the concept of Synchronized Hybrid Evaluation (SHE) (Contribution C1). Being based on the same set of interfaces these software frameworks are interoparable with each other.

1.7 Thesis Organization

The remainder of this dissertation is organized as follows:

Chapter 2 outlines important fundamentals for the later discussions in this thesis. After an introduction to different techniques for the performance analysis of communication systems we provide an in-depth introduction to the concept of network emulation [Fal99]. In this context we further explain the timing problem of existing network emulation frameworks. In addition, Chapter 2 also provides an introduction to operating system virtualization methods, as this technology is an integral part of SliceTime. The chapter concludes with a brief introduction to hardware simulators, which are another technical building block used later in this thesis.

In Chapter 3 we describe the concept of Synchronized Hybrid Evaluation (SHE), our first contribution. It enables the flexible composition of hybrid evaluation platforms using reusable modules.

Chapter 4 presents the aforementioned contributions to the field of network emulation. We first elaborate the implementation of SliceTime (C2) and DDWNE (C3) and provide an in-depth analysis of their accuracy and their performance. We further showcase their applicability to complex emulation scenarios with three use case scenarios. The chapter also provides a comprehensive discussion of related work in the field of network emulation.

Chapter 5 discusses a hybrid co-simulation framework that is also derived from the SHE concept. This tool-chain (C4) facilitates the hardware/software co-design of networked embedded systems by integrating a network simulation with a full-system simulator.

Chapter 6 introduces our framework for distributed monitoring and debugging of communication systems (C5). It is an indirect by-product of our work on SliceTime and SHE.

We conclude this dissertation in Chapter 7. Here we also outline possible future research directions that are related to the challenges and the results discussed in this document.

2

Fundamentals and Background

Overview

The goal of this chapter is to elaborate important fundamentals and techniques that are relevant for later technical and conceptual discussions in this thesis. A strong emphasis here is placed on performance evaluation methods. In this regard we also explicate the synchronization problem that exists with most network emulation frameworks. We later provide a brief introduction to virtual machines and full-system simulators; these topics are important for the technical discussions related to SliceTime (cf. Chapter 4) and the subsequently described hybrid frameworks.

Structure of this Chapter

This chapter comprises three main parts. We first provide a introduction to the performance evaluation of computer networks. The second part gives an introduction to virtual machine technology and the Xen hypervisor. The third part summarizes important fundamentals from the domain of full-system simulation.

2.1 An Introduction to the Performance Evaluation of Computer Networks

"How fast is it able to transport data?" and *"How long does one have to wait until the data is delivered?"* are two typical questions colloquially expressing what the performance of any computer network is about. The performance of a computer network mainly corresponds to the efficiency of the data transport between two or more end systems. Hence, if we want to assess the performance of a computer network of arbitrary shape we need to evaluate the efficiency of the data transport. For this purpose we rely on *performance metrics* that enable the quantification of different

network performance characteristics, for example delay or network bandwidth. We discuss common performance metrics for computer networks later in this section.

On an abstract level the observed performance of a computer network depends on two general attributes, namely the network structure and different influences the network has to handle. The network structure describes which systems and infrastructure components, for example routers, are part of the network and how they are interconnected - to the latter we typically refer to as *network topology*.

There is a wide set of influences that directly impact the performance of a computer network with a given structure. A very prominent one is the *workload*, which for example specifies the amount and the kind of information a network has to transport. The workload also encompasses the established connections and the tasks are issued to the network by the individual end systems. Other influences with an impact on the network performance are the probability of transmission errors on the network links, the arrival and departure rate of network nodes and possibly node mobility.

It is noteworthy that both the network structure and the influencing factors may be of dynamic nature and hence are subject to change over time. Considering the Internet, it is a known fact that the network structure is constantly changing [DD08]. Measurement studies have also shown that the Internet traffic and hence its workload has changed over the past years [Wil01, BDF$^+$09].

With regard to performance evaluation this raises the question how to deal with dynamic network structures and varying external and internal influences. Typically, we are interested in determining how the network performance (measured using adequate metrics, see Sec. 2.1.1) depends on variabilities of the network structure or changing influences, for example an increased degree of node mobility or an alternative workload. For this reason, performance evaluation studies mostly discriminate between *parameters* and *factors* [Jai91]. Parameters are properties that are assumed to remain static during the experiment while factors are varied in a controlled fashion to investigate their influence on the network performance. The precise set of evaluation factors and parameters depends on the questions to be answered by the respective performance evaluation study.

Let us consider an example and look at the evaluation of a conceivable file sharing protocol that allows a group of users to mutually exchange data. It would be possible to define the performance of the system as average downloading time (the time a user has to wait for a file to be downloaded after it has been requested) and hence rely on this downloading time as performance metric. We then raise the question how the downloading time depends on the number of users concurrently requesting the file. In this case, the varied number of concurrent requesters would be a factor, while a static file size or a constant number of total nodes in the network would constitute evaluation parameters.

But how do we measure the performance of a communication system or service, for example such a file sharing service? Which metrics can be used to quantify the performance of such? The remainder of this section aims at providing further insights to these topics. First, we discuss fundamental performance metrics for computer networks and illustrate a general methodology for choosing performance metrics. In a second step we focus on different performance evaluation methodologies.

	Latency (RTT)	**Bandwidth**
ISDN (1 B-Channel)	200 ms [LW06]	64 kbit/s
Satellite Links	250ms [HL01] to 900ms [Cob11]	16 Kbit/s - 200 Mbit/s (downlink) 16 Kbit/s - 2 Mbit/s (uplink) [HL01]
RWTH Aachen Campus Network[1]	0.25 ms - 1 ms	6 MB/s - 30 MB/s
10 Gigabit Ethernet	8.9 μs [FBB+05]	7.6 Gbit/s [FBB+05]

Table 2.1 Magnitudes of latencies and bandwidths for different communication technologies.

2.1.1 Performance Metrics

Performance metrics facilitate the quantification of performance by mapping it to an adequate scale. A figurative example that illustrates this concept is an imaginative performance comparison of road vehicles. If we define the maximum speed a vehicle is able to achieve, the respective performance metric is the maximum kilometers per hour the vehicle is able to move at. If we would rather be interested in checking the engine power of one vehicle against another one, the respective performance unit would be kilowatt or alternatively horsepower. This example also demonstrates that there is mostly not one single performance metric that can be used to quantify the performance of a system as a whole. In the domain of computer networks, we distinguish between basic metrics, namely network latency and network throughput, and custom performance metrics which are used for evaluating specific services or applications. Custom performance metrics are also employed for quantifying the reliability and the resource usage of network systems. In the following, we discuss both categories separately.

2.1.1.1 Latency and Throughput

The most fundamental performance metrics that can be studied for any computer network and communication service are latency and throughput [PD03].

Network Latency

The *network latency* quantifies the time it takes a message to travel through a communication network, typically on a path between two systems. It is measured in a unit of time. The network latency can be measured unidirectionally in order to find out how long it takes to transport a packet through the network in one direction. The network latency is also often measured bidirectionally. This so-called *round-trip time* is equivalent to the time a packet needs from a host A to another host B in the network and back to A.

Table 2.1 lists the magnitudes of latencies for different communication technologies. It is apparent that we observe widely differing latencies for these technologies. The

[1]The magnitudes of RTTs and the throughputs for the RWTH Campus network have been determined by randomly sampling different servers using the ping and wget tools. The measurements were conducted for illustrative purposes and do not constitute a profound performance evaluation study of the network.

total travelling time is dependant on the number of links, their individual delays and the time the intermediary systems (for example routers and switches) need to process the packets. In addition, retransmissions due to packet loss may also contribute to the overall packet delay.

Network Bandwidth

The *network bandwidth* describes the amount of data a computer network or communication link is able to transport in a certain time interval. It is typically either specified in bits (bits/s) or bytes (B/s) per second.

The overall *throughput* between two end systems in packet-switched computer networks is mostly dependant on the bandwidth capacities of the individual links and the amount of concurrent network traffic. If only one end-to-end path is used for the communication between two end systems, the overall throughput never surpasses the bandwidth of the link with the lowest capacity, which we commonly refer to as *bottleneck*. In addition, the packet processing performance of middleboxes on the path, for examples routers, switches or firewalls, also has a direct influence on the overall throughput.

2.1.1.2 Selecting Custom Performance Metrics

It is quite obvious that not all performance evaluation studies in the context of computer networks solely evaluate network throughput or latency. For example, a developer of a P2P file sharing application might be rather interested in the mean downloading time, and developers of routing protocols might require to study path lengths or routing efficiency. In essence, these examples show that performance metrics need to be adequately chosen in order to provide insight about a certain evaluation goal. But how can we choose adequate performance metrics?

Jain [Jai91] has proposed an exciting approach for tackling this problem (cf. Fig. 2.1). He suggests to plan performance evaluation studies in a top-down manner by starting with a list of services of interest, e.g., web servers, routing daemons or file sharing services. For each request issued to one of these services, we might either receive a response or no response. If the response has been processed, we can further distinguish between correct outputs and invalid ones. For correct outputs we can measure so-called speed metrics. For example, we could measure the time a web server consumes to process and answer a request, which would correspond to latency. We could also investigate the number of requests the server is able to answer per second (throughput) or the CPU and memory consumed (resource utilization).

In the case the system delivers invalid and/or erroneous responses we also can evaluate its reliability. For this purpose it is advisable to trace down the error that causes the response to be invalid. We then can quantify the probabilities for these errors and the time between their occurrence. Such analyses are especially important for systems and services that cannot provide valid results under all circumstances, for instance dynamic routing services for mobile ad-hoc networks. Due to potentially continuous changes in the physical topology of the computer network, a routing service might deliver outdated routes that do not provide connectivity. For such

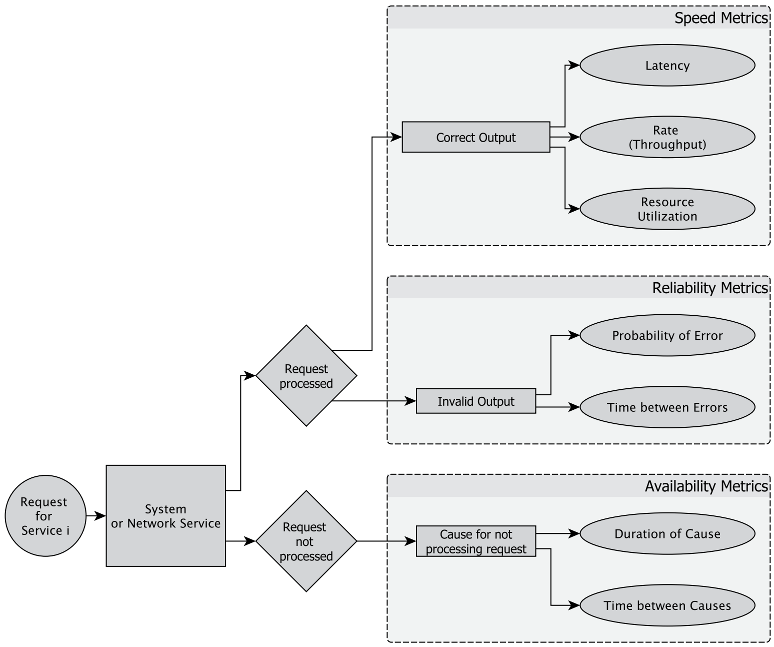


Figure 2.1 Taxonomy for the selection of performance metrics as devised by Jain [Jai91].

a service we could hence quantify the percentage of invalid routes delivered due to node mobility.

Finally, there is the possibility that the system under investigation does not deliver a response under certain circumstances at all. Such causes are for instance failure, overload or scheduled maintenance. So-called availability metrics quantify for example the percentage of time a system or service is available. Availability metrics are often used for ranking service providers like web hosting companies. A specific and famous availability metric in the hardware domain is the so-called Mean Time to Failure (MTTF). It is often used by hard disk vendors to promote the longevity of their products.

2.1.2 Performance Evaluation Methodologies

The performance evaluation of network protocols and communication systems is important at any point of the development cycle. For instance, designers of a new routing protocol might require an early performance analysis after the corresponding algorithms have been roughly sketched. In such a case, an implementation of the protocol is typically not yet available. However, researchers and developers are often also interested in measuring the performance of protocols, communication software or

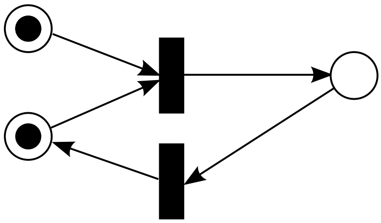


Figure 2.2 Petri nets are a mathematical technique for modeling distributed systems. A basic Petri net is a bipartite graph that contains two types of vertices, places (circles) and transitions (blocks). A transition executes if all incoming edges are supplied with tokens (black circles) at the origin. In such a case, a new token is created at the destinations of the outgoing edges.

network hardware at different implementation stages, ranging from proof-of-concept implementations to production systems.

Over the past decades a plethora of methodologies have been proposed for the performance evaluation of computer networks. As already sketched earlier in the introduction we can categorize them into analytical, simulative and measurement-based approaches. In the following, we first provide an overview of these three categories of performance evaluation methodologies. We later also discuss hybrid techniques that combine two different performance evaluation domains in order to overcome their individual limitations.

2.1.2.1 Analytical Performance Modeling Techniques

Analytic performance modeling is a collective term for abstract mathematical techniques that can be applied for modeling communication systems. Although none of them is of particular importance for the contributions made by this thesis we now briefly introduce common analytical techniques for the sake of argument. The following discussion is organized along a survey by Puigjaner [Pui03] and covers Petri nets, queuing networks and process algebra.

Petri Nets

One of the first analytic methods for describing the behavior of computer networks and distributed systems was introduced by Carl Adam Petri in the 1960s [Pet66]. Figure 2.2 shows a simple example. A Petri network is a directed graph, which contains two types of vertices, places (denoted by circles in the figure) and transition (black blocks); the nodes are interconnected using unidirectional edges. The state of such a network is given by the number and the location of tokens (black solid circles) that float around the places. The tokens move along the edges of the graph if the transition rules are satisfied. A transition executes if all places that are connected to it hold at least one token. In this case one token is removed from all the places linking to the transition, and one token is sent to the place the transition itself connects to.

Plain Petri nets cannot be applied for the quantitative performance evaluation of communication systems [Pui03], as they do not contain any timing information.

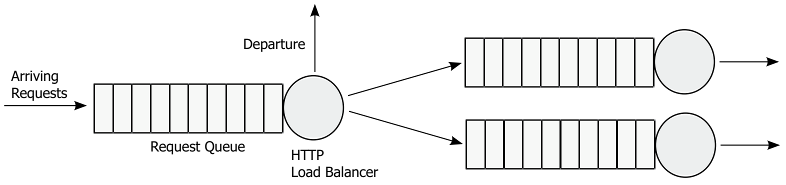


Figure 2.3 Figurative example of a queueing network: HTTP requests arrive at a load balancer which distributes the requests to two web servers for processing.

There are several proposals for solving this deficiency by adding timing annotations (e.g. [RH80, BD91]). These approaches are commonly known as Timed Petri nets.

There are several software packages available [BLPK07, Gra95, Lin95] that allow the convenient modeling and the execution of Petri net. Most of these tools also allow one to model advanced network types, for example Timed Petri nets or colored Petri nets, which contain tokens of different kinds.

Queueing Networks

Another analytic technique for the performance analysis of distributed systems are queueing networks [Jai91, LZGS84, Pui03]. The basic idea of this approach is to model systems as a set of queues with requests traveling from one queue to the next one. Figure 2.3 displays a simple example of a queueing network that models an HTTP server farm. A central load balancer accepts all incoming requests and forwards them to a set of slower web servers, for example in a round-robin fashion. The web servers successively process individual request queues. In addition, it is also possible that requests depart in between, for example, if the request is canceled.

A queueing network model allows one to analytically infer performance characteristics such as average processing time, throughput or the expected queue length. For this purpose different characteristics of the queues need to be specified, for example the arrival process of incoming requests and the service disciplines (First Come First Served, Last Come First Served, etc). Other parameters include the capacities of the different queues, the time it takes a server to process a request and so forth.

Depending on the structure of the queueing network different mathematical methods are then applied to derive performance estimations from such a model. A comprehensive description of according techniques can be found in [Jai91, LZGS84].

Process Algebras

Process Algebras [Bae05] are formal languages that facilitate the specification and the analysis of concurrent systems. For this purpose process algebras provide an algebraic syntax to specify the behavior, the composition and the message-based interaction of concurrently executing processes. Famous examples of process algebras are Communicating Sequential Processes (CSP) [Hoa78], π-Calculus [Mil99] and the Algebra of Communicating Processess (ACPs) [BK86]. All these process algebras facilitate logical reasoning on the specified concurrent system or network.

In order to apply this methodology for quantitative performance investigations different so-called stochastic process algebras (SPAs) have been proposed [CGHT07]. In essence all these SPAs extend the underlying process algebra with expressions for specifying the timing behavior or process mobility. Stochastic process algebras have been applied for different evaluation tasks, for example for studies of multimedia streaming [BBD01] or a DiffServ router [SB03].

2.1.2.2 Performance Evaluations using Network Testbeds

In many cases state-of-the-art network protocols and distributed applications are very sophisticated and it is very difficult to map the complex system behavior to an abstract analytical framework or simulation models. For this reason researchers and developers often directly create an executable system prototype to investigate a new network protocol or a new service by conducting a set of experiments. In the context of network protocols and distributed applications, a system prototype mostly corresponds to a software implementation. In certain cases, for example in the domain of wireless sensor networks, system prototypes often consist of both prototypical hard- and software.

This dissertation refers to computer networks that are employed for performance evaluation studies as network testbeds. Such testbeds diverge widely in their size, shape and mode of operation. For example, a simple network testbed may consist of a small set of machines that are interconnected using Ethernet and that are located in the same room of a university lab. Other network testbeds like PlanetLab [CCR$^+$03] comprise hundreds of nodes that are interconnected by public networks such as the Internet. While the small university testbed may be exclusively accessible to a small set of researchers, a global testbed like PlanetLab is often shared by many users that conduct different performance evaluation studies on the testbed at the same time. In addition, the non-evaluation traffic carried by the interconnecting public networks may also influence the evaluation carried out on such a testbed.

In the following we first identify different properties that can be used for the classification of network testbeds. In a second step we discuss a number of prominent network testbeds and relate them to these properties.

Properties of Network Testbeds

For the later classification of existing network testbeds we rely on the following properties:

- **Execution Context:** The execution context determines the type of system prototypes that can be evaluated on the network testbed. For example, if a network testbed consists of computers running the Linux operating system, the testbed can be used for evaluating the performance of Linux applications and Linux protocol implementations, but not for sensor network applications that are based on a different operating system and hence rely on an entirely different hardware platform.

- **Level of access:** The level of access describes to which extent the execution context can be used and customized for a measurement study. Testbeds that grant access to the entire system hardware do not put any restrictions on the software being run on the testbed. By contrast, testbeds like Pharos [FPS+11] only enable studies of specialized applications.

- **Node count:** The node count quantifies the number of execution contexts and hence the maximum number of system prototypes that can be deployed on the network testbed. This parameter is of special importance for scalability studies of system prototypes, as the number of system prototypes cannot be scaled out beyond the number of available execution contexts.

- **Topology Controllability:** Network testbeds differ in the degree of control with regard to the network topology. Some network testbeds like the UMIC mesh deployed at RWTH Aachen University provide a single fixed topology to all users. Other network testbeds like Emulab allow users to customize the network topology to better match the requirements of a performance evaluation study.

- **Mobility Support:** Most network testbeds only contain nodes with a static location. However, there are a few network testbeds such as Pharos featuring mobile nodes, which change their physical position over time.

- **Exclusivity of use:** Network testbeds differ in the way they provide their resources to the users conducting performance evaluation studies. We here distinguish between testbeds that are exclusively used by one user at a time and shared testbeds, in which multiple measurement studies are carried out at the same time.

- **Accessibility:** Most network testbeds are owned by a certain institution or organization and are often only available to affiliated researchers. One example for such a private testbed is the UMIC-Mesh at RWTH Aachen University. By contrast, public network testbeds are generally available to anyone who files an according application to use the testbed for a performance evaluation study.

A brief overview over existing network testbeds

In the following we provide a short overview over six common network testbeds that are currently employed for performance evaluation studies. Table 2.2 compares the testbeds discussed in the following, namely PlanetLab [CCR+03], MoteLab [WASW05], German Lab (G-Lab) [SGH+10], Emulab [HR12], Pharos [FPS+11] and the UMIC Mesh [ZGW+06].

[2]Emulab used to support mobile nodes, however according to the documentation available at `http://www.emulab.net/tutorial/mobilewireless.php3/` (accessed 8/2012) this support has been suspended.

[3]Motelab provides users exclusive use of the entire testbed during a scheduled time slot.

	PlanetLab	MoteLab	German Lab	EmuLab	Pharos	UMIC Mesh
Execution Context	Linux	TinyOS Applications and Components	Linux	Linux, FreeBSD & Windows Software	Proteus [Pai10] Robot Platform with proprietary API	Linux
Level of access	VM (full access)	Entire System Hardware	VM (full access)	mixed	Applications via custom API	Entire System Hardware
Node Count (08/2012)	1128	190	$\geq$170	$\sim$ 550	$\sim$ 20	50
Topology Control	Limited (Choosing nodes within slice is possible)	None (Static Topology)	Limited (Choosing nodes within slice)	Yes	Yes, with controllable mobility	None (Static Topology)
Mobility Support	No	No	No	ceased[2]	Yes	No
Exclusivity of Use	Shared Use	Exclusive Access[3]	Shared Use	Shared Use	Not known	Exclusive
Accessibility	PlanetLab Members	Public	G-Lab Members	Public	Private	Private

Table 2.2 Overview of different academic network testbeds.

PlanetLab [CCR+03] is a well-established research testbed designed for the analysis of new network services and distributed applications. It consists of 1100 network nodes around the globe. All PlanetLab hosts are connected to the Internet, and PlanetLab is frequently used to evaluate new Internet services and applications[4].

Performance evaluation studies conducted on PlanetLab suffer from two general issues [SPBP06]. First, results obtained from PlanetLab are not reproducible. The reason is that PlanetLab hosts are shared among multiple users and that the performance measurements are always dependant on uncontrollable network conditions such as background traffic or route changes. Second, the network interconnecting the PlanetLab hosts is not representing the Internet, as most PlanetLab nodes are hosted by academic institutions and hence are connected to high-speed research network backbones [BGP04].

The **MoteLab** [WASW05] testbed enables researchers to deploy TinyOS-based applications on a wireless sensor network located at Harvard University. The access to MoteLab is essentially granted using time slots, during which a researcher has potentially full and unrestricted access to all nodes. Hence, the testbed is only used by one user at the same time. By contrast, shared testbeds such as PlanetLab typically host multiple experiments at the same time.

A distributed research testbed that shares many similarities with PlanetLab is the **German Lab (G-Lab)** [SGH+10]. Around 190 nodes are deployed at six university campuses in Germany. The testbed uses the same software as PlanetLab for managing its infrastructure and allows for the testing of Linux networking software.

EmuLab [HR12] enables researchers and developers to evaluate communication software for different operating systems. It is able to remodel the topology and

[4]The PlanetLab website provides a comprehensive list of research publications that have made use of the testbed at `http://www.planet-lab.org/biblio` (accessed 08/2012).

the characteristics of widely diverging networks, ranging from Internet-wide deployments to local area networks. The major difference in comparison with PlanetLab and G-Lab are the datapaths among the nodes, which are fully isolated from other networks such as a campus LAN or the Internet. This yields a better degree of reproducibility. Emulab internally makes use of network emulation techniques to remodel the characteristics of the so-called target networks. We provide a more detailed discussion of Emulab later in Section 4.4.4.2.

One of the very few network testbeds that support performance evaluations with mobile nodes is **Pharos** [FPS+11]. It is a small-scale testbed that consists of multiple embedded systems on a mobile robotic platform whose movement can be programmed. This enables performance evaluations for mobile networks, for example mobile ad-hoc routing daemons.

There are numerous network testbeds operated by academic institutions, most of them are of small or medium size. One example for such a network testbed is the **UMIC Mesh** [ZGW+06] at RWTH Aachen University, which consists of around 50 nodes. Each node is equipped with a WiFi interface card and an Ethernet network interface. The network nodes are mostly used for performance evaluations of wireless mesh networking software.

Discussion: Advantages and Disadvantages of Performance Evaluations with System Prototypes and Network Testbeds

Performance evaluations carried out using system prototypes and network testbeds have a number of advantages in comparison with analytical methods and network simulations. The most prominent reason for implementing a system prototype is the high degree of authenticity, as the performance measurements are obtained under realistic conditions and are not distorted by disparities of a simulation model or analytical abstractions. A real-world working prototype is well able to demonstrate the feasibility of a new network protocol or service. Therefore, measurements obtained with system prototypes and network testbeds are typically also associated with a higher degree of credibility.

On the other hand it is by far not possible to conduct all performance evaluations solely using system prototypes and network testbeds. As we have just shown in our overview of established academic testbed facilities, most of them range from a few tens of nodes to above one thousand nodes in size. Although their capacity can be extended by using virtualization techniques (cf. Sec. 4.4.2) the limited size of current testbeds hinders their use for experiments of very high scale. One example are investigations of new P2P systems, for which researchers are often interested in conducting scalability analyses with thousands or ten thousands of nodes. A second shortcoming of network testbeds is the reduced reproducibility of performance evaluation results, especially if the network testbed is not isolated from other networks such as the Internet. Finally, if a performance evaluation experiment requires controlled mobility of a larger number of nodes it is difficult to employ network testbeds, because only a few provide mobility support.

2.1.2.3 Network Simulation

One of the most established methodologies for the performance analysis of communication protocols is network simulation. Network simulations reproduce an arbitrary computer network entirely in software. For this purpose, a network simulation employs a set of models that describe the operation of the network components, the communication channel and environmental factors such as node mobility. A network simulation model can be understood as a software component that describes the behavior of a certain building block of a communication network, typically using an imperative programming language.

Instead of creating such software reproductions from scratch we typically make use of so-called *network simulators*. A network simulator is a software program that is already equipped with a set of simulation models. A performance evaluation is then carried out by first composing a virtual communication network in the software environment. This step often involves the development of new models, as future network protocols a researcher might want to analyze are not yet available for the simulator. Finally, performance evaluations are carried out by executing the network simulation and by observing the behavior of the modeled network.

A fundamental concept of network simulation is abstraction. Simulation models usually only implement the most essential characteristics that are necessary to conduct a performance evaluation. For example, specialized network simulators for the analysis of P2P protocols such as PeerSim [MJ09] or PeerFactSim.KOM [SGR$^+$11] typically omit a detailed model of the transport layer in order to keep the simulation slim and simple. Such abstractions lead to a very high scalability of network simulators in comparison with network testbeds. We have shown earlier that recent network simulators are easily able to model thousands of nodes on a desktop PC [WSHW08], depending on the simulation complexity.

The third strength of network simulation is repeatability. As network simulators are software programs they can be implemented in a fully deterministic fashion. Even a potential randomized behavior of a network protocol or system, for example random back-off timers, can virtually be made deterministic by initializing the random number generator with the same random seed. The ability of conducting performance evaluation experiments in a deterministic way is helpful for parameter studies, which study the impact of a particular parameter on a distinct performance metric.

All network simulation tools share the requirement that they need to model the concurrent activity of a set of network nodes. There are two general ways how this is typically reflected by simulation environments. The first option is a concept named *process-based network simulation*. In this approach, every network node corresponds to a process or a thread. All these threads or processes are executed concurrently. The exchange of messages and network packets is then carried out at synchronization points between the processes. This approach is less common for network simulations. In fact, the activity-based mode of OMNeT++ [VH08] and SimPy [Mue] are rare examples for this methodology in the domain of computer network simulation. The second approach is *discrete event-based simulation*, which models concurrency using discrete events that are subsequently processed.

In the following, we first discuss important fundamentals of discrete event-based simulation. Then we provide a short overview of popular network simulators.

```
//Simulation Components
MessageScheduler scheduler
OrderedQueue queue
globaltime = 0 //Global Simulation Time

//Simulation of two hosts using a simple channel model
Host Node_A,Node_B
SimpleChannel channel

//Invoked if a simulated host receives a message
void Host.receiveMessage(Host Sender, String Message) {
    print globaltime + " " + Host " received " + Message + "from "
       + Sender
    channel.sendMessage(Sender,Destination,"Ping")
}

//Models the delayed transmission of data on a channel
void SimpleChannel.sendMessage(Sender,Destination, Message) {
    delay = 100ms
    arrival = globaltime + delay
    scheduler.scheduleMessage(arrival, Sender, Destination, Message
       )
}

//This method schedules a message to be delivered to the
   Destination at a certain arrival time
void MessageScheduler.scheduleMessage(arrival, Sender, Destination,
    Message) {
    MessageEvent event = new MessageEvent(arrival, Sender,
       Destination, Message)
    queue.insertEvent(arrival, event)
}

//The main event scheduler
void MessageScheduler.run() {
    while (queue NOT empty) {
    Message mev = queue.getEarliestMesssage()
    //Advance the global time to the arrival time of the next
       messsage
    globaltime = mev.getArrival()
    destination = mev.getDestination()
    message = mev.getMessage()
    sender = mev.getSender()
    //Execute Arrival at Receiver
    destination.receiveMessage(Sender, Message)
    }
}

void start {
    channel.sendMessage(Node_A,Node_B,"Ping")
    scheduler.run()
}
```

Listing 2.1 A exemplary discrete event-based simulator in pseudo code. It models two hosts that mutually exchange ping messages

Discrete Event-based Simulation

The underlying concept of most network simulation tools is discrete event-based network simulation. A queue that stores a set of discrete simulation events forms the core of all event-based simulators. The simulation events reflect all activities in the simulated network, for example communication messaging or node movement. The simulation queue stores these events ordered by the associated execution time. The network simulation is then simply processed by running through the event queue and by invoking a handler function for each event, which may in return spawn new simulation events.

Listing 2.1 shows a minimalistic network simulator in pseudo code, using a syntax that is reminiscent of C/C++. The simulated network consists of two network hosts that are connected using a simple channel. The communication channel has no bandwidth limitations, but delays the arrival of every message by 100ms. Both nodes simply send a “Ping” message back to the sender if such a message is received.

Our exemplary discrete event-based network simulation for modeling this scenario comprises three building blocks. The first building block is the host object that models a network node. It implements one sole method which replies to incoming messages with a “Ping”. The second building block is a simplistic channel model that enables the communication between different hosts. It delays the reception of messages at the destination by scheduling the arrival time 100 ms later than the current simulation time.

The core of our Ping-Simulator consists of two methods. The first one, `MessageScheduler.scheduleMessage()`, schedules the reception of a message at any host in the network. This is implemented by inserting a corresponding event in the central simulation queue at the planned time of message arrival. The second method, `MessageScheduler.run()`, puts the core behavior of this simple simulator into action. It essentially consists of a loop that always processes the first event in the simulation queue. As the queue is strictly ordered by the scheduled event execution time, the first event in the queue corresponds to the “next” event on the simulation time line. The actual processing of the event consists of two steps. First, the global simulation time is advanced to the execution time of the event. Second, the event is dispatched to an according event handler. In our simple example we assume the queue to contain only one type of event, namely `MessageEvent`. For this reason, the one and only event handler is the `host.receiveMessage()` method, which prints out received messages and sends a reply on the simulated channel. As there are only two hosts in the simulation, this results in always one event in the queue, and thus the simulation never terminates.

Moreover, every event-based network simulation requires an initial activity to be specified, which causes simulation events to be created. In our case this task is carried out by manually invoking `scheduleMessage()` before starting the scheduler. Otherwise, the event queue would be empty, which would result in an instant termination of the simulation program.

Contemporary Network Simulation Tools

Implementing an entire simulator for conducting a performance evaluation may occasionally be reasonable, for instance due to scalability requirements or because of a non-standard run-time environment of the simulator. However, in most cases we typically rely on existing network simulators for modeling computer networks. The main advantage of these tools is that they already provide a set of models for common protocols like TCP/IP. In the following we briefly introduce different common network simulation tools. We have also conducted a performance comparison of some of these simulators in our prior work [WvLW09].

ns-2

The well-known **ns-2** [MFF$^+$97, ns] simulator has been the de facto standard for network simulation over the past decade. Numerous network protocol models in different areas (e.g., wireless communication systems or transport protocols) have been developed for this simulator[5]. Network simulations for ns-2 are composed of C++ code, which is used to model the behavior of the simulation nodes, and oTcl scripts that control the simulation and specify further aspects, for instance the network topology. This design choice was originally made to avoid unnecessary recompilations if changes are made to the simulation setup [HRFR06]. Back in 1996 when the first version of ns-2 was released, this was a reasonable intent, as the frequent recompilation of C++ programs was indeed time-consuming and slowed down the research cycle. However, from today's perspective, the design of ns-2 trades off simulation performance for the saving of recompilations, which is questionable if one is interested in conducting scalable network simulations.

OMNeT++

Another well-established simulation tool is **OMNeT++** [Var01, VH08]. It is not a network simulator by definition, but a general purpose discrete event-based simulation framework. Yet it is mostly applied to the domain of network simulation, given the fact that with its INET package it provides a comprehensive collection of Internet protocol models. In addition, other model packages such as the OMNeT++ Mobility Framework and Castalia [Bou07] facilitate the simulation of mobile ad-hoc networks or wireless sensor networks.

OMNeT++ simulations consist of so-called *simple modules* which realize the atomic behavior of a model, for instance a particular protocol. Multiple simple modules can be linked together and form a *compound module*. For instance, multiple simple modules which provide protocol models can be combined into a compound module representing a host node. A network simulation in OMNeT++ is implemented itself as a compound module which comprehends other compound modules, like the ones which model host nodes. OMNeT++ rests upon C++ for the implementation of simple modules. However, the composition of these simple modules into compound modules and thus the setup of network simulations takes place in NED, the network description language of OMNeT++. NED is transparently rendered into C++ code

[5]A comprehensive list of models contributed to ns-2 is available at `http://nsnam.isi.edu/nsnam/index.php/Contributed_Code` (accessed 08/2012).

when the simulation is compiled as a whole. Moreover, NED supports the specification of variable parameters in the network description: For example the number of nodes in a network can be marked to be dynamic and later on be configured at run-time. In this case, the modules representing the nodes are dynamically instantiated by the simulator during execution. This feature is a direct consequence of the simulator's strict object-oriented design.

JiST/SWANS

A fresh approach in the context of network simulation is **JiST** ("Java in Simulation Time") [BHvR05], which in compliance with its name allows the implementation of network simulations in standard Java. It is mostly used in conjunction with SWANS[6], a simulator for mobile ad hoc networks built on top of JiST.

Network simulations in JiST are made up of entities which represent the network elements, for example nodes, with simulation events being formed by method invocations among those entities. The entities advance the simulation time independently by notifying the simulation core. While the code inside an entity is executed like any arbitrary Java program, only the interactions between the individual entities are carried out in simulation time. Thus, these interactions between entities correspond to synchronization points and facilitate the parallel execution of code at different entities, resulting in a potential performance gain. In order to execute the implementation in simulation time, JiST utilizes a custom dynamic Java class loader which dynamically rewrites the application's byte code.

ns-3

A rather new network simulator is **ns-3**[7]. The ns-3 simulator has been developed from scratch and differs widely from its predecessor, ns-2. The architectural design goals of ns-3 are high scalability, modularity, extensibility and the ability of applying the network simulator for emulation purposes [HRFR06]. Like its predecessor, ns-3 relies on C++ for the implementation of the simulation models. However, ns-3 no longer uses oTcl scripts to control the simulation, thus abandoning the problems which were introduced by the combination of C++ and oTcl in ns-2. Instead, network simulations in ns-3 can be implemented in pure C++, while parts of the simulation optionally can be realized using Python as well. An early performance evaluation of the ns-3 simulation core has shown that it holds performance characteristics that are comparable with JiST and OMNeT++ [WSHW08].

The ns-3 simulator has been especially designed for performance evaluation studies of Internet systems and services. As of 2012, ns-3 provides a rather comprehensive set of models. Besides the simulation of TCP/IP-based Internet Systems, ns-3 facilitates the analysis of LTE, 802.11 and WiMAX deployments. The investigation of mobile networks is backed by a set of mobility models and different routing protocol models such as Ad-hoc On-demand Distance Vector (AODV), Dynamic Source Routing (DSR) and Destination-Sequenced Distance Vector routing (DSDV).

[6] JiST/SWANS Website: `http://jist.ece.cornell.edu/` (accessed 12/2012)

[7] The ns-3 website available at `http://www.nsnam.org/` (accessed 08/2012) provides a rather comprehensive overview over the simulator and the bundled models.

One unique property of ns-3 is of major interest for our work. Instead of packet models that abstract from the actual bits and bytes that are transferred over the communication medium, ns-3 uses real-world packet formats. This enables the network simulator to be easily interfaced with real-world communication systems for hybrid performance evaluation experiments.

2.1.2.4 Limitations of Network Simulation

The high degree of flexibility and the ability to perform experiments of high scale make network simulation an essential methodology for the performance evaluation of network protocols and distributed systems. However, there are also a number of shortcomings and limitations associated with network simulation.

Most of these drawbacks stem from the fact that network simulators provide a custom event-based API for the implementation of simulation models. This simulation API typically differs widely from the operating system context that usually forms the execution environment of networking software. Except for specialized solutions such as ns-3 Direct Code Execution (DCE) [Lac10] these differences make it generally impossible to execute legacy networking software in a network simulator. Hence, existing networking applications and protocol implementations need to be manually ported and often reimplemented to a large degree in order to match the programming model of the simulation API.

A second issue connected to the disparity of the simulation API and the system context of networking software is that the use of a simulator limits performance studies to metrics that are reflected by the simulation models. For example, if one implements a transport protocol in a network simulator, metrics such as throughput or end-to-end latency can be well evaluated in the simulation domain. However, it is generally very difficult to evaluate complex system characteristics such as CPU and memory usage or energy efficiency using simulation, as they are mostly not modeled by simulators due to the high effort that is needed for accurately modeling the resource usage of a hardware platform.

2.1.3 Hybrid Performance Evaluation Methodologies

One solution to overcome the individual shortcomings of network simulation, testbed-based network measurements and analytical techniques are hybrid performance evaluation techniques. The core idea of such is to combine different techniques. This allows one to profit from the strengths that are associated with the incorporated tools and methodologies and to compensate for weaknesses of the individual techniques at the same time. We here focus on hybrid performance evaluation methods that amend a network simulator with either an analytical framework or real-world communication systems. To the latter approach we refer to as *network emulation*.

In the following, we first describe two examples of hybrid performance evaluation methodologies for communication systems. The first methodology is an example how analytic performance modeling can be integrated with network simulation. We then continue with an introduction to the concept of network emulation, which forms an important basis for the later course of this dissertation.

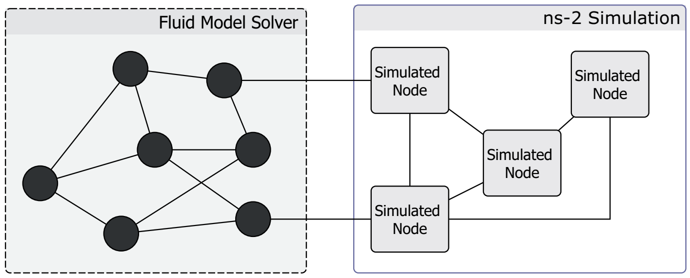


Figure 2.4 Gu et al. have proposed to integrate an analytical fluid model with a network simulation based on ns-2 [GLT04]. The framework aims at improving the scalability of the modeled network, as the output of the analytical model can be computed in an efficient way.

2.1.3.1 Integrating Fluid Models with Packet Simulation

With the goal of analyzing TCP in very large network topologies, Gu et al. have proposed to integrate a discrete event-based network simulator, precisely ns-2, with an abstract analytical TCP fluid model [GLT04]. The motivation behind this endeavour is that the used analytical fluid model scales well beyond discrete event based network simulations. In this regard it had earlier been reported by Liu et al. that computing the model output for a network size of 100 nodes is over 2100 times faster than a comparable discrete event-based simulation [LLPM$^+$03]. In effect, the proposed hybrid performance evaluation environment enables one to apply ns-2 to network sizes that are difficult to investigate using a network simulator alone.

Figure 2.4 illustrates this hybrid approach. The authors propose to amend a common ns-2 simulation, consisting of a set of nodes, with a fluid model that models a large-scale TCP network. The model consists of a set of differential equations. Network packets that are routed through the partition of the fluid-modeled subtopology are first conveyed to the fluid model domain. The output of the model solver is then used to determine the network traffic leaving the fluid model domain.

One interesting challenge in this hybrid framework is the interplay of the simulation packets with the fluid model. The authors propose two approaches to handle this mutuality. The assumption made by the first approach is that the simulation packets do not influence the traffic modeled in the fluid domain. Hence, the output of the fluid model can be directly computed and simulation events and packets can be scheduled accordingly. By contrast, the second approach enables mutual interactions between the simulation packets and the analytical fluid model. The core idea is to conduct two passes, with the first pass calculating the influence of the simulated packets on the analytical model and the second pass actually delivering the simulation results. This is feasible due to the deterministic nature of both the network simulation and the fluid model.

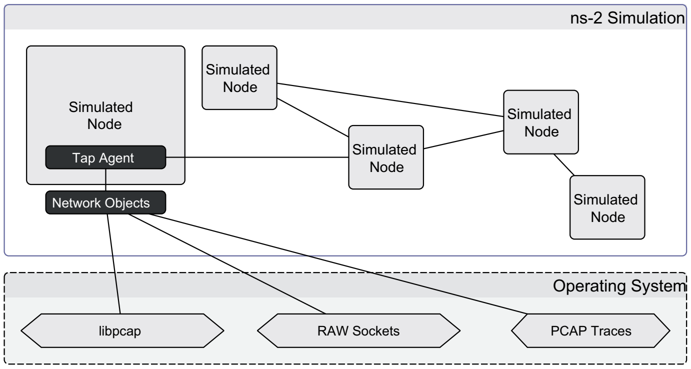


Figure 2.5 The emulation capabilities of ns-2 rely on so-called tap agents which translate between simulation packets and native packets from external sources. The access to real-world network traffic is handled by so-called network objects, which either rely on libpcap, raw sockets or trace files for this purpose.

2.1.3.2 Network Emulation using Discrete Event-based Simulation

One of the first hybrid evaluation frameworks for computer networks and communication systems was described by Kevin Fall in a paper entitled "*Network Emulation in the Vint/NS Simulator*" [Fal99]. Fall proposes the integration of a discrete event-based network simulator (ns) with real-world networking hosts for the main purpose of enabling software tests in an isolated and repeatable environment. Another strength of network emulation is the ability of simulating large portions of the network structure, for example network hosts and routers, purely in software. The integration of the network simulator with the real-world systems results in the systems perceiving the simulated part of the network as a real-world network.

For this purpose Fall describes an integration of the ns network simulator with UNIX-based networking systems. This implementation has later been merged into ns-2 and as of 2012 is still available with the current version (ns-2.35). Figure 2.5 shows the general architecture of the ns-2 emulation framework. External traffic sources, for example physical network hosts or simply networking applications, may be integrated with the network simulation either using RAW sockets, packet capturing based on libpcap[8] or using trace files in the PCAP format. These network sources are accessed by so-called network objects. The network objects forward incoming traffic to so-called TAP agents which are associated with a simulated network node. In a similar way, the network objects forward traffic received from the TAP agent to the external interface.

The TAP agent translates between simulation packets and real-world packet formats. This step is inevitable, as the packet formats internally used by ns-2 to model

[8]libpcap is packet capture library originally developed at the Lawrence Berkeley National Laboratory. It is an integral part of the TCPDUMP packet analyzer. Both TCPDUMP and libpcap are available at `http://www.tcpdump.org/` (accessed 08/2012).

network traffic differ widely from real-world packet structures. For this reason, every incoming packet has to be mapped to the according protocol model of ns-2 and vice versa for outgoing traffic. This process is complicated by the fact that most protocol models strongly abstract from their real-world counterparts, for example by disregarding packet payload at all. The disparity between the message formats used by ns-2 and real-world systems results in the need of developing a separate TAP agent for every protocol that is to be supported by the emulation framework.

Fall also identifies a core problem of this methodology. Except for packet traces, all systems connected to ns-2 using the network objects operate in wall-clock time. Hence, the simulation also has to process all packets it receives in real time, as otherwise the performance observed by the real systems would be deteriorated.

Network Emulation in ns-3

The more recent ns-3 [ns311, HRFR06] strongly improves the network emulation features in comparison with its ancestor. By far the most important difference of ns-2 and ns-3 is that ns-3 internally uses real-world packet formats at all layers. This eliminates the need of an explicit translation of packets using a TAP agent or similar technologies. In addition, this enables all protocol models of ns-3 to be transparently used for network emulation.

A second major improvement of ns-3 is its support for so-called TUN/TAP devices which are nowadays supported by nearly all UNIX and BSD-based operating systems. A TUN/TAP device provides a virtual network interface that is not bound to a real physical networking device such as an Ethernet adapter. Instead, all packets received on the TUN/TAP device are copied to a block device, and in a similar way, all data written to the block device is sent over the TUN/TAP interface. This greatly simplifies capturing networking data and allows one to isolate network emulation experiments from real-world computer networks.

Other Emulation Frameworks based on Discrete Event-based Simulators

In fact there are a number of discrete event-based network simulators like OMNeT++ [VH08] or JiST/SWANS [BHvR05] for which similar emulation features have been proposed and implemented. We provide an elaborate discussion of these tools later in Section 4.4. To put it in a nutshell, all these approaches are based on explicit network translation mechanisms for bridging the gap between the simulation and real-world systems; ns-3 is the only exception in this regard. In fact the superior emulation capabilities of ns-3 are the reason for it being the implementation basis for our implementation of synchronized network emulation (cf. Section 4.1).

2.1.4 A BitTorrent Model for Network Emulation

In the following, we briefly discuss the core design principles of VODSim [WGLW12, Gle11, Hoc12], a BitTorrent [Coh03] simulation model we have specially designed for network emulation. VODSim relies on the aforementioned ns-3 simulator and is fully interoperable with the emulation capabilities of ns-3. We will later resort to VODSim for one of our application studies of SliceTime in Chapter 4.

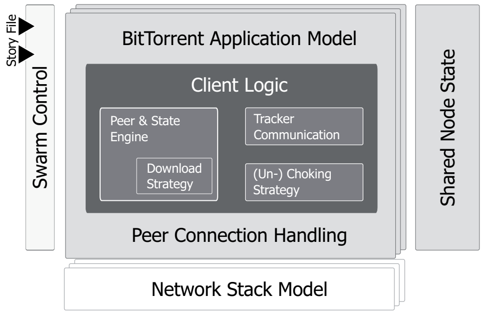


Figure 2.6 Conceptual overview of our BitTorrent client model: All distinctive features of a BitTorrent client are encapsulated in functional blocks, allowing for a flexible adaption of the client behavior.

The usefulness of any network simulator grows with every protocol and communication system it is able to model. Lately, ns-3 has significantly progressed in this regard, as it now provides models for a rich set of protocols at all layers of the protocol stack. However, regarding application layer models, one of the most prominent protocols in the Internet eco-system is missing: BitTorrent. BitTorrent [Coh03] is a P2P system originally designed for effectively sharing large amounts of data over the Internet. Over the past years, many extensions and changes to the original BitTorrent protocol have been proposed and implemented into client software. They range from performance improvements, such as Super-seeding [Hof08], to tracker-less operation that allows for a fully decentralized sharing of data using a Distributed Hash Table (DHT) [Loe08]. More lately, adapted BitTorrent clients have been proposed in the literature that alter the piece trading strategy (see below), for example in order to improve the own downloading performance [PIA$^+$07a] or to create Video-on-Demand services on top of the BitTorrent protocol [VIF06].

This variety in existing BitTorrent systems shows that an according model for ns-3 should not only model a specific BitTorrent client, but instead should allow for the easy replication of the behavior of different BitTorrent systems.

The implementation of our BitTorrent client model is highly modular (cf. Appendix A.2) and allows for easily adapting, customizing and extending its design and its behavior. In order to reflect the design principles of ns-3, almost all interactions between components of VODSim are implemented using specific simulation events, callbacks and callback handlers. For a more elaborate discussion of VODSim and its implementation see [WGLW12, Gle11, Hoc12].

2.1.4.1 Conceptual Design of VODSim

Corresponding to the architecture of BitTorrent, our ns-3 model in fact consists of two models, a BitTorrent client model and a BitTorrent tracker model.

BitTorrent Client Model

Figure 2.6 displays the high-level architecture of our BitTorrent client model. The model consists of the core BitTorrent model that sits on top of the network simulator's network stack model and two singletons that govern the *swarm control* and hold the *shared node state*.

Swarm Control

The swarm control component is responsible for the configuration and the execution of a BitTorrent simulation. For this purpose, it parses a so-called *story file*. The story file consists of a set of commands in a Domain-specific Language (DSL). The DSL facilitates the specification of the simulation setup and the activity of the BitTorrent clients during the simulation run. Hence, the user of the BitTorrent simulation uses the story file for example to instruct a set of nodes to start requesting a file at a certain point in time. Other commands in the story file allow one to configure the piece selection strategy used by the clients, the user to set the initial distribution of pieces in the swarm or to hand over a .torrent meta-info file to the simulation.

Shared Node State

A major design goal of our BitTorrent simulation model is to enable large-scale BitTorrent simulations with hundreds or thousands of simulated clients. In order to achieve this goal we reduce the individual state space required for each node by storing redundant node state information only once. The most significant state information in this regard is the data payload shared by the swarm; it may range from a few megabytes to a couple of gigabytes. While using an actual payload would not be required for a pure BitTorrent simulation, we use real-world payloads to enable emulation studies with BitTorrent. Storing the data payload at one central component dramatically reduces the memory footprint of one simulated node and hence the memory requirements of the entire simulation. We later evaluate the memory requirements of our model at greater detail.

BitTorrent Application Model

The BitTorrent client model aims at reproducing different BitTorrent systems and software clients. For this reason we have abstained from "hacking" together a monolithic BitTorrent client. The design of the BitTorrent model instead is highly modular and compositions of different functional units define the actual client logic of the model. It contains the following sub-components:

- The **Peer & State Engine** keeps track of the pieces a client has retrieved and provides this information to the other functional units. The peer and request engine also maintains state information for the peers a client is aware of. For example, it stores state information describing if these clients are choked or if an interest message has been sent to them.

- **Tracker Communication Unit:** This unit carries out all interaction with the BitTorrent tracker, mostly in order to obtain a number of peers that participate in the swarm. For the actual communication, the tracker communication unit encapsulates a HTTP client and adequate sub-components to parse the meta-information it retrieves from the tracker. It provides interaction stubs for other methods of peer discovery, for instance a DHT.

- The **Download Strategy Component** controls which parts of the file are downloaded next from other BitTorrent clients. Classic BitTorrent file sharing clients typically use a *rarest-first* strategy. As the global availability of the pieces is not known, the clients count how many of the peers they are connected to possess each piece. This information is then used to request the least popular pieces with the goal of maximizing the availability of each piece.

  The encapsulation of the download strategy into an own logical unit is important to support other schemes in our model: The *pure sequential* strategy downloads the parts of the file one after another. This scheme is a very naive strategy for the delivery of streaming media. However, the architecture of our strategy control unit allows more sophisticated streaming strategies such as Give-To-Get [MPM+08] to be retrofitted in the future.

- **Choking/Unchoking Strategy:** The choking/unchoking strategy is responsible for blocking/unblocking other clients from downloading pieces. In the classic BT file-sharing service this decision is largely dependent on the amount of data a remote peer has already sent to the client, resulting in a *tit-for-tat* trading scheme. However, the modularization of the choking/unchoking behavior allows for arbitrary strategies to be implemented, like randomly (un)-choking remote peers or basing this decision on recommendations from other peers.

- The **Peer Connection Handling Unit** implements the Peer Wire Protocol (PWP) and exchanges both payload data and status messages with other peers. It uses the socket layer abstraction of the underlying network simulator - in our case ns-3.

BitTorrent Tracker Model

In addition to the client model, we also created a rather straightforward BitTorrent tracker model. Once a node joins the swarm, it registers at the tracker. The BT tracker model stores the client information, most importantly its IP address and the peer ID, in a local data structure. It then sends a tracker response to the client, which, among different status information, contains a list with the IDs of other peers in the swarm, their IP addresses and the TCP ports on which they are listening.

As all communication tasks are handled using HTTP, our tracker model also incorporates a respective HTTP sub-component. This sub-component provides basic mechanisms for sending GET requests and for parsing HTTP headers; it provides its functionality to upper layers using corresponding callbacks. In the following, we omit a further discussion of the tracker model's implementation due to its rather straightforward design and implementation.

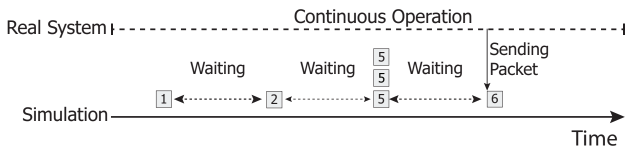


Figure 2.7 Real-time network emulation tools align the simulation events to the wall-clock time and simply perform active waiting between the simulation events.

2.1.5 The Synchronization Problem of Network Emulation

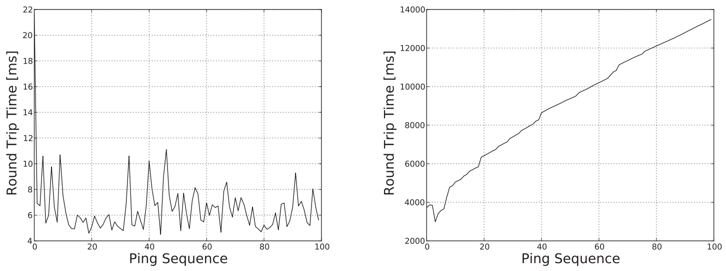
A core problem of all network emulation tools is the need of integrating the two entirely diverse time domains and timing concepts used by discrete event-based network simulations and real-world systems. Discrete event-based network simulations operate on a fully virtual time line (cf. Sec. 2.1.2.3); the time is directly advanced between the simulation events. By contrast, real-systems use wall-clock time that constantly progresses in a strong monotonic way.

All real-time network emulation frameworks rely on the same technique in order to align these two timing concepts with each other. For this purpose the scheduling component of the network simulator is exchanged with a so-called real-time simulation scheduler. The basic operation of such a scheduler is depicted in Figure 2.7. A real-time scheduler does not process all events directly after each other. Instead it aligns the synchronization events with the wall-clock time and performs active waiting between the simulation events. If a network packet is received by the discrete event-based simulation it instantaneously schedules an corresponding simulation event. The basic assumption behind this approach is that the computer running the simulation is powerful enough to process all events in real-time.

However, there is a large number of network simulations that are not able to be executed in real-time, simply due to the vast number of events created by the simulation or by complex calculations inside the model code. In such a scenario, the real-time scheduler cannot keep up with real-time and starts falling behind.

But what happens in a real-time network emulation scenario if the simulation cannot keep up with real-time? How is this reflected in actual measurements? In order to illustrate these effects we have conducted a small experiment. We used the real-time emulation capabilities of ns-3 and simulated a simple network with 4 hosts and CSMA channels. Using network emulation, we attached a Linux host to this simulation and collected 100 ICMP Echo replies ("Pings"). Figure 2.8(a) shows the ping sequence for the normal case. We obtained round trip times ranging mostly between 4 ms and 10 ms, which matches the setup of the simulated network.

In order to introduce overload conditions into this simulation scenario, we artificially slowed down the processing speed of ns-3. For this purpose we delayed the processing of every simulation event by 10 ms using an according `usleep` statement in the scheduler. Figure 2.8(b) shows the outcome of this experiment. Although no changes were applied to the simulated network topology we observed a strong increase in the



(a) Undisturbed Measurements (b) Faulty RTTs due to overloaded simulation

Figure 2.8 Simulation Overload results in increasing round trip times, as the network simulator processes simulation events at a lower rate than they are created due to the communication with an external host.

measured round-trip times. The reason for this faulty behavior is that the externally received ICMP echo requests create more events than the simulator is able to process in real-time. This results in a constantly growing event queue, because ns-3 does not drop any packet by default and enqueues every incoming packet. The growing length of the event queue leads to a steadily increasing processing time for each simulation event. This is reflected by the continually swelling round trip times. In summary, this example demonstrates that so-called *overloaded simulations* in the context of network emulation directly may cause erroneous performance measurements.

One way to circumvent simulation overload is increasing the event throughput of the network simulator with the goal of making the simulation execute in real-time. One method is upgrading the processing power of the computer that executes the simulation, for example by using a faster CPU or by improving the performance of the simulation program. A second alternative consists in parallelizing the simulation program. In fact, Kiddle has shown that parallelized network simulations are a viable method for improving the scalability of a real-time emulator [KSU05]. However, we argue that this approach lacks generality because parallel processing can only scale to the degree of possible parallelism within the simulation. In addition, the amount of hardware needed for real-time execution rapidly grows with the simulation complexity, making this option inaccessible for many researchers. For these reasons it is impossible to make arbitrary network simulations real-time capable.

There is just one conceptual solution for making network simulations of arbitrary complexity usable for network emulation. We need to turn the tables and synchronize the execution of the real-world systems with the network simulation. If the network simulation is not real-time capable, the real-world communication systems attached to it need to be slowed down in order to prevent them from drifting away in time. To the best of our knowledge, there are just three network emulation frameworks that implement such a capability, namely SVEET [ELL09], SliceTime [WSvL$^+$11] and TimeSync [SPL$^+$12]. We discuss all these tools later in this thesis.

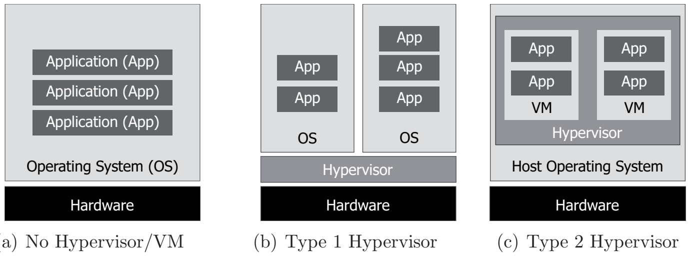


Figure 2.9 Comparison of different approaches for system virtualization. The provision of virtual machines that execute a guest operating system and a set of applications is carried out using so-called hypervisors. The first type of hypervisors operate directly on the hardware. Type 2 hypervisors are executed on top of a host operating systems.

2.2 Virtual Machines

Modern network emulation testbeds and frameworks, for example Emulab [HR12], VirtualMesh [SGB09], SVEET [ELL09] and SliceTime [WSvL$^+$11], often rely on Virtual Machine (VM) technology for increasing the number of hosts in a network testbed or to solve the problem of overloaded network simulations. We will later explain such solutions in greater detail.

Figure 2.9 compares a traditional setup of a computing system with two approaches for the provisioning of virtual machines. On a classical personal computer (cf. Fig. 2.9(a)) the operating system runs directly on top of the hardware. Concurrent activities, for example the simultaneous execution of different applications, are solely managed by the operating system, for instance by spawning and by scheduling a set of processes for each application.

VMs enable the concurrent execution of multiple operating systems on one physical computer. In fact, a virtual machine encompasses an entire operating system, consisting of a kernel, device drivers and a set of applications. The major difference in comparison with a classical PC setup is that the operating system running in the VM does not have exclusive access to the system hardware.

The access to the system hardware, the management of multiple virtual machines and their execution is carried out by a so-called **hypervisor**, synonymously also called **virtual machine monitor**. Goldberg [Gol73] has identified two general concepts for the implementations of hypervisors and virtual machines. So-called Type 1 hypervisors (cf. Fig. 2.9(b)) operate directly on top on the hardware. For this reason they are also referred to as *bare metal hypervisors*. Contemporary virtualization frameworks that belong to this category are Xen [BDF$^+$03, Chi08], Microsoft HyperV [mic] and VMware ESX/ESXi [VMW]. Type 1 virtual machine monitors typically execute one privileged guest system that serves as host operating system and is able to fully control the behavior of the hypervisor.

By contrast, Type 2 hypervisors (cf. Fig. 2.9(c)) such as Parallels[9] or VirtualBox [Wat08] operate on top of a host operating system. From a user's perspective, such virtual machine monitors form a normal application that models an entire computer on which an operating system of choice may be installed.

2.2.1 Implementation Aspects of System Virtualization

In order to virtualize an operating system, the hypervisor needs to provide an interface that closely resembles the system hardware. A second vital requirement is that the virtual machine monitor retains control of the computer hardware at any point in time. The hardware of a computer consists of a Central Processing Unit (CPU), physical memory and peripherals such as disk drives or input devices; to the latter we refer to as Input/Output (I/O) devices. In the following we discuss important aspects of system virtualization for each type of these hardware components.

2.2.1.1 CPU Virtualization

All hypervisors directly execute the code of the operating system running inside a VM directly on the host CPU. This takes away the need of emulating a CPU in software and makes a very efficient execution of the guest operating system possible. The core concept used for implementing such a direct execution of VMs is *de-privileging* [AA06].

Most CPU architectures support different privilege levels to allow only the kernel of the operating system and device drivers to perform crucial operations on the CPU. For example, the x86 processor architecture provides four privilege levels, referred to as rings 0, 1, 2 and ring 3 (cf. Fig. 2.10). Ring 0 is the level with the highest privilege. Software running at this level is able to directly access and to configure the entire system hardware. For this reason, the kernel of a x86 operating system normally operates at ring 0. In order to prevent normal software applications to perform sensitive instructions, they normally operate at lower privilege levels. In fact, most x86 operating systems use just two privilege levels, ring 0 for the kernel and ring 3 for applications. Switching between different privilege levels takes place using software interrupts or so-called call gates; newer processors also implement new instructions such as `SYSENTER` for this purpose [Gar06].

The key idea behind classical de-privileging is to move the operating system kernel to the – normally unused – ring 1 and to execute the hypervisor in ring 0 instead (cf. Fig. 2.10(b)). Any sensitive instruction executed by the kernel in ring 1 then makes the system automatically switch to the hypervisor context in ring 0, where an exception handler can be used to *trap* that instruction. After trapping it, the hypervisor performs an emulation of the hardware operation that would be caused normally by the instruction and then returns to the execution of the kernel in ring 1. This enables the hypervisor to retain full control over the hardware and hence makes it possible to coordinate the concurrent access of multiple guest systems to the system hardware.

[9]Parallels is a commercial type 2 hypervisor for Windows and Mac OS X. More information about these products can be found at `http://www.parallels.com/` (accessed 10/2012).

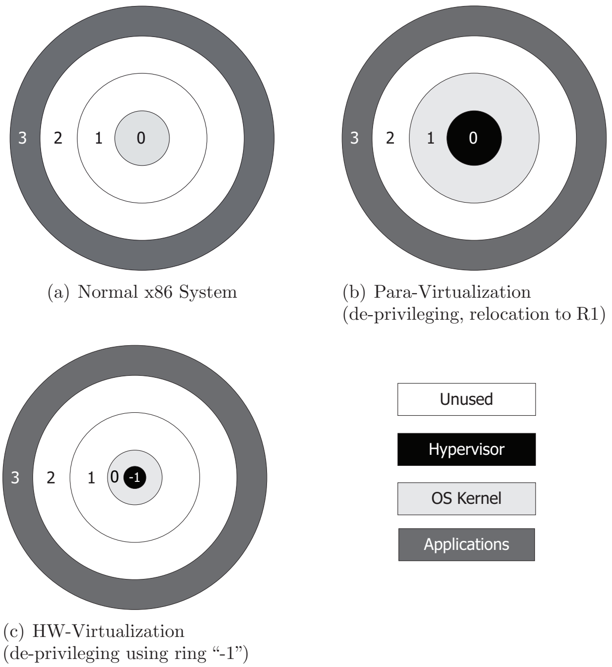

3
2
1
0
(a) Normal x86 System
3
2
1
0
(b) Para-Virtualization (de-privileging, relocation to R1)
3
2
1
0
-1
(c) HW-Virtualization (de-privileging using ring "-1")
Unused
Hypervisor
OS Kernel
Applications

Figure 2.10 Usage of protection rings for different kinds of virtualization (x86 platform).

Unfortunately, implementing such a *trap-and-emulate* hypervisor is typically not straightforward, as many CPU architectures like the x86 platform were originally not designed to be virtualized. This results in a number of difficulties, for example in a set of CPU instructions that do not raise an exception if executed outside ring 0 [RG05]. This is traditionally solved either using binary code rewriting [AA06] of the operating system code running in ring 1 or using para-virtualization. The idea of para-virtualization is to modify the kernel of the guest operating system to effectively execute in ring 1. Hence, all sensitive invocations are replaced with adequate calls to the hypervisor which in effect removes the need of trapping instructions at all. However, a disadvantage of para-virtualization is the need of manually modifying the code of the operating system kernel.

As virtualization has become a rather common principle in modern computing systems, CPU manufacturers such as AMD and Intel have incorporated different virtualization extensions such as AMD SVM [amd12] and Intel VT [NSL$^+$06] into their processors a couple of years ago. Essentially, they enable the guest operating system kernel to remain in ring 0 by adding an additional priviledge level (ring "-1") that holds the hypervisor (see Fig. 2.10(c)).

2.2.1.2 Memory virtualization

In order to execute multiple guest operating systems concurrently, the hypervisor needs to provide each of them with the illusion of possessing an exclusive portion of system memory. In addition, the hypervisor needs to isolate the memory allocated to one VM from the other VMs in order to prevent the VM's memory to be compromised for example due to faulty guest kernels or due to malicious code.

Hypervisors implement memory virtualization by maintaining a shadow page table for each VM. Every time the guest operating system modifies its page table, the hypervisor traps this change and creates a mapping in the shadow page table that points to the actual physical memory location. The use of the shadow page table enables the hypervisor to retain absolute control over physical memory allocated by virtual machines [RG05]. As trapping the access to the page table and managing shadow page tables purely in software is rather time-consuming, AMD and Intel have introduced hardware support for memory virtualization in recent versions of their processors [NSL$^+$06, amd08].

2.2.1.3 I/O Virtualization

In order to isolate a VM from the system hardware, a hypervisor also needs to virtualize the access to I/O devices such as disk drives and peripherals that are connected using different bus systems, for example PCI, USB and SCSI. As most peripherals require specialized device drivers, most contemporary hypervisors make use of the device drivers available on the host operating system. A second challenge for device I/O is that virtualization leads to disparities in the memory addresses used by the guest operating system and the actual physically used addresses. This makes it necessary for the hypervisor to perform address translations for device I/O operations, for example Direct Memory Access (DMA) transfers. As this is a rather complicated and critical task regarding I/O performance, recent AMD and Intel processors have also incorporated features that enable a better virtualization of device operations [amd12, NSL$^+$06].

2.2.2 The Xen Hypervisor

In the following, we briefly illustrate the core characteristics of Xen [BDF$^+$03, Chi08], a very established hypervisor for the x86 platform. It serves as basis of the SliceTime implementation (cf. Sec. 4.1).

Figure 2.11 displays a conceptual overview of a Xen configuration that virtualizes two guest operating systems. Xen belongs to the class of type 1 hypervisors and hence operates directly on top of the hardware. Xen distinguishes between two types of virtual machines, which are called *domains*. The first type is the so-called *domain 0*, also called control domain. It is a privileged virtual machine. All administrative tasks regarding virtualization are carried out on domain 0. Most notably, domain 0 is the only VM which can invoke new VMs itself, or suspend or stop their execution. In fact, if a computer running Xen is booted, it first starts the Xen hypervisor, which invokes domain 0 right away.

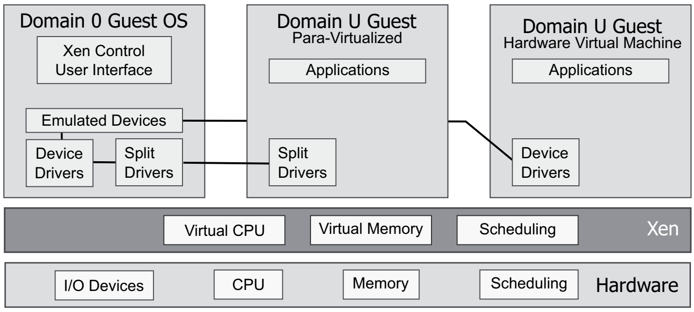


Figure 2.11 Conceptual example for a common Xen installation [Chi08]: Xen is a type 1 hypervisor. It facilitates executing both para-virtualized and hardware-virtualized domains.

The second type of Xen domains is called "domain U". It is used to execute actual guest VMs. Xen supports two types of such guest domains:

Para-Virtualized Domains

So-called *para-virtualized* domains execute an operating system kernel that was specially adapted to work with Xen. The core idea of para-virtualization is to replace system calls that require expensive trapping with so-called *hypercalls*, which are handled by the hypervisor instead. This cooperative form of virtualization enables a very efficient virtualization of many guest systems and is the main reason why Xen popularized para-virtualization in the early 2000s. The major drawback of para-virtualized domains is the need of specially adapted operating system kernels. At the time of writing this thesis, according operating system kernels for Linux, NetBSD, FreeBSD and Open Solaris are available[10].

Para-virtualized domains make use of the device drivers of the domain 0 to interact with I/O devices such as disk drives and network interfaces. For this purpose, Xen implements a so-called split driver model. The guest domain only implements a stub that provides a lightweight front-end enabling applications running on the guest to access the respective hardware. All low-level hardware interactions are handled by the other half that resides in domain 0. The interaction and exchange of data between domains is carried out using shared memory and asynchronous event notifications.

Hardware-assisted Virtual Machines (HVMs)

So-called HVM domains, which have been introduced in version 3.0 of Xen, enable the virtualization of unmodified guest operating systems. HVM domains require a CPU on the host system that supports either AMD SVM [amd12] or Intel VT [NSL+06].

In order to enable an unmodified guest operating system to perform I/O operations in a virtualized environment, Xen emulates a set of hardware devices such as a

[10]The Xen online documentation at `http://wiki.xen.org/` (accessed 10/2012) provides a comprehensive overview over possible guest systems.

networking card and a graphics adapter in software. This emulation of system devices is required to allow legacy device drivers that are bundeled with the guest operating system to make use of the hardware of the host system. Xen resorts to the device models of QEMU [Bel05] for this purpose. One issue that is associated with the emulation of hardware devices is the reduced I/O performance in comparison with para-virtualized domains.

Domain Scheduling

A component being of particular interest for the implementation of SliceTime (cf. Sec. 4.1) is the scheduling subsystem of Xen. In essence, SliceTime relies on a modified Xen scheduler to eliminate the simulation overload problem of network emulation.

The Xen scheduler is a central component that is part of the hypervisor layer. Its task is the assignment of CPU time to the domains, and it also manages the mapping of Virtual Central Processing Units (VCPUs) to physical CPUs. A core functionality of all Xen schedulers is the ability of influencing the amount of CPU time each domain receives, which enables the prioritization of guest domains against each other. Since its original development, different schedulers have been proposed and implemented for Xen [Chi08]. The two most widespread ones are the following:

- The *Credit scheduler* distributes the CPU time among the guest domains in fair shares, based on weights and optional CPU usage bounds that are associated with every guest domain. The Credit scheduler is work-conserving, which means that it achieves a host CPU utilization of 100% as long as any guest domain is not idle.
- The later discussed SliceTime implementation is based on the *Simple Earliest Deadline First (SEDF)* scheduler. The standard SEDF scheduler periodically executes every domain for a time slice of N milliseconds every M milliseconds. As N and M can be configured separately for each domain, this can be used to establish a prioritization of domains as well.

From a developer's perspective, Xen allows additional schedulers to be added to the system using the scheduler interface. More information about this interface and internals regarding Xen scheduling can be found in [Chi08].

2.3 Hardware Simulation and Full-System Simulators

A universal approach for the virtualization of a computing system is a simulation environment that reproduces the behavior of a *hardware platform* and its peripherals entirely in software. Such *full-system simulators* enable the software development for future systems without the need of the hardware to be physically available. Simulating the entire hardware makes it also possible to analyze low-level performance characteristics such as cache efficiency or to investigate operating system kernels in a fully deterministic environment.

Later in this thesis we make use of such a hardware simulator to implement a hybrid framework for the development of networked embedded systems. We therefore now briefly introduce two such frameworks that play a role in the later course of this document, namely Simics [MCE$^+$02] and SystemC [VvRBM96, cow].

2.3.1 The Simics Full-System Simulator

A rather well-known full-system simulator is Simics. Its core strength is the provided support for a wide range of target platforms and target devices. It provides models for many processor architectures, for instance x86, AMD64, MIPS and different ARM platforms. In combination with models for peripherals such as disk drives, network cards and graphic cards[11], Simics enables the deterministic execution of unmodified operating systems on fully simulated hardware [MCE$^+$02, EE06].

Simics enables the accuracy and the performance of its simulation to be adjusted using different simulation modes. If a high simulation performance is required, Simics implements a just-in-time compiler to execute portions of the target system natively on the simulation host [EE06]. The other simulation modes trade execution performance for accuracy and allow the detailed simulation of low-level hardware characteristics like caching behavior or CPU timing.

In addition, Simics is able to simulate entire networks of computing systems [Eng09]. This feature enables fully deterministic investigations of networked systems using legacy operating systems and applications. We later discuss such an application in Section 5.5.1.2. In addition, Simics can be applied for modeling network nodes at different levels of abstractions. Certainly, Simics can be used to model network hosts at a very high detail using a complete hardware simulation. However, it is also possible to implement abstract simulation models which only implement the functional behavior of a network protocol. Such abstract Simics models closely resemble the level of abstraction of network simulators such as ns-3.

At first glance, the full-network simulation features of Simics seem to weaken the motivation of network emulation. However, there are two reasons why Simics is not able to replace simulation-based network emulators in most cases. First, a major strength of network emulation frameworks, for example the ns-3 network emulation capabilities, is the possibility to resort to the collection of models that is available for the simulator. Simics, however, does only provide abstract protocol models for the most essential network protocols like IP, ICMP and DNS [EKMR05]. The second reason is scalability. While Simics facilitates the detailed investigation of networked systems, simulating the entire hardware of a very large computer network is rather computationally expensive and may easily outgrow the available resources on the simulation computer. By contrast, we later show in Section 4.3 that frameworks such as SliceTime can be used to investigate scenarios with 1000s and even 10000s of networked nodes.

[11]A full list of target platforms and supported peripheral devices is available at the Simics product website at `http://www.windriver.com/products/simics/` (accessed 12/2012).

2.3.2 SystemC

SystemC [GLMS02] is a modelling and simulation framework that is widely used for the design and the exploration of new hardware systems. SystemC applies the paradigm of event-based simulation for modeling concurrent activities. The SystemC library itself consists of a set of macros and C++ classes and thus can be compiled using standard software development tools.

2.3.2.1 Modeling Components of SystemC

A SystemC model consists of a set of building blocks which conjointly reproduce the desired system behavior. **Modules** are the basic building blocks of a SystemC model composition. They typically encapsulate the functionality of a self-contained (hardware) unit, for example a processor core. A SystemC module typically consists of the following sub-components [GLMS02]:

- A set of **ports** allows the module to communicate with the environment and other SystemC modules.
- **Processes** describe the actual functionality of the module. Multiple processes are executed concurrently.
- The potential inclusion of other SystemC **modules** allows the organization of a SystemC model in a hierarchical fashion.
- Internal data structures that hold the **state information** of the module.

Channels are used to enable the communication between different modules. SystemC provides a set of primitive channels like hardware signals or FIFO channels. In addition, SystemC facilitates the implementation of hierarchical channels for the purpose of modeling more complex communication mechanisms, for example bus systems. So-called **interfaces** are used to express how channels can be accessed. This is necessary to avoid ports to be connected to incompatible channels. SystemC **events** allow the specification of certain conditions, which trigger processes if the conditions are met. This enables a SystemC model to react to changes in a process or on a channel, for example due to data input.

The implementation of a SystemC model is carried out by writing a C++ simulation program and by including the header file `systemc.h`. The declaration of modules takes place using specialized SystemC macros, which all share the prefix `SC_`.

2.3.2.2 A brief SystemC example

Listing 2.2 shows an example of a simple SystemC module that implements a multiplication of two implementation numbers. The simple multiplier module has three ports. Two input ports x and y are used to deliver integer numbers to the module, and port c is an Integer output port that will deliver the result. The macro `SC_CTOR` declares a constructor for the module. Once the module is instantiated, `SC_METHOD` registers the actual computation method with the SystemC scheduler. Finally, the statement `sensitive` makes the process react if there are changes on the x and y ports, thus triggering a new computation once the input changes.

```
SC_MODULE(Multiplier) {

sc_in<int>  x;
sc_in<int>  y;
sc_out<int> c;

    void multiply() {
        c = x * y;
    }

SC_CTOR(Multiplier){
    SC_METHOD(multiply)
    sensitive << x << y;
}

};
```

Listing 2.2 A simple SystemC module that models an integer multiplier (adapted from [GLMS02])

2.3.2.3 Advanced SystemC Modelling Frameworks

In order to facilitate the development and the analysis of hardware platforms, many advanced SystemC modeling frameworks have been proposed over the past years. Integrated Development Environments such as gSysC [EAHC05] facilitate SystemC models to be composed in a graphical environment, and integrated visualization and debugging mechanisms ease the investigation of complex hardware designs. In addition, tools like the CoWare Virtual Platform Architect optionally provide the user with so-called IP-blocks[12] for existing components such as CPUs. Such special SystemC components are distributed as compiled binaries in order to mask the underlying hardware design since they are protected as intellectual property.

2.4 Interim Summary

Within this chapter we have laid out the fundamentals for the later course of this dissertation. We have reviewed different aspects and methodologies related to the performance evaluation of computer networks and communication systems. In this regard, we have put an emphasis on network simulation and network emulation, which forms the starting point for our contributions to the field of hybrid evaluation (see Chapter 3). Our discussion of Virtual Machines (VMs) and Full-System Simulators (FSSs) has introduced basic principles that are of interest for the later technical discussions.

[12]In this case, IP denotes Intellectual Property. Throughout this document, the abbreviation IP stands for *Internet Protocol*.

3

Synchronized Hybrid Evaluation

Overview

With Synchronized Hybrid Evaluation (SHE) we now propose a modular architecture for hybrid evaluation platforms of communication systems. The foremost goal of SHE is to eliminate the time synchronization problem (cf. Sec. 2.1.5). All frameworks later discussed in Chapter 4, 5 and Chapter 6 of this dissertation are derived from the SHE concept.

Figure 3.1 shows an exemplary SHE setup. The core idea behind SHE is to form hybrid evaluation frameworks by composing different so-called *System Representation (SR)* modules. SR modules model the behavior of one or more communication systems at different levels of abstraction:

- **Virtual Machines (VMs)** execute arbitrary communication software systems. Hence, VM system representations facilitate the incorporation of unmodified communication software into a SHE setup.
- **Discrete Event-based Network Simulations (NETSIMs)** recreate the behavior of an entire network of communication systems at a high level of abstraction. This enables the use of NETSIMs to provide a large-scale and deterministic network environment for other SRs.
- **Full-System Simulators (FSSs)** enable the close exploration of protocol stacks and their interaction with system hardware. A FSS simulates hardware components or even full hardware architectures, for example embedded networking hardware such as home routers or set-top boxes, in a very detailed manner (typically at the cycle or the instruction level).

The cornerstone of every SHE composition is the **Synchronization Component (SYNC)**[1]. It centrally synchronizes the execution of all SRs in a SHE composition.

[1]We also refer to this component synonymously as the *synchronizer* throughout this document.

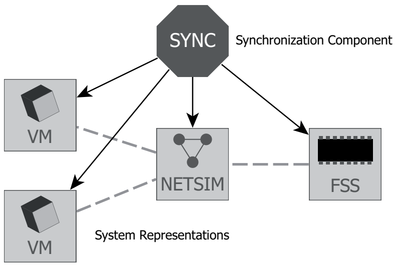


Figure 3.1 Synchronized Hybrid Evaluation (SHE) enables the flexible composition of hybrid network evaluation environments by combining different system representation (SR) components. A central synchronization component aligns the execution of the different system representations. The SHE setup depicted here integrates three different types of SRs: A set of virtual machines (VM), a network simulation (NETSIM) and a full-system simulator (FSS).

As discussed earlier in Section 2.1.5 a major issue of hybrid evaluation frameworks is the inequality of the time domains that are used by the different system representations. While VMs operate in real-time (wall-clock time), both NETSIMs and FSSs provide a virtual timeline to the communication systems they model. The progression on these individual timelines is not tied to the wall-clock time in any way. In effect, NETSIMs and FSSs typically operate faster or slower than real-time which may cause problems if they are incorporated with systems that execute in real-time. In order to relieve all SRs from the constraint of having to operate in real-time, the SYNC supplies all system representations with a consistent virtual progression of time that is decoupled from wall-clock time.

Communication Interfaces

In a SHE setup, we distinguish between two different communication flows. The Time Control Interface (TCI) is used to attach the system representations to the SYNC component. The Packet Exchange Interface (PEI) interconnects the different system representations. In essence the PEI forms a virtual network that carries all network traffic exchanged between the SRs.

Structure of this Chapter

We first elaborate this synchronization process and the operation of the SYNC component. (cf. Sec. 3.1). Afterwards, we discuss general aspects and requirements for SRs (cf. Sec. 3.1.2) and the interfaces used for synchronization and data exchange between different SR modules (see Sec. 3.1.3 and Sec. 3.3). We finally describe a number of different SHE compositions and show how earlier hybrid evaluation methodologies can be subsumed under the SHE concept (cf. Sec. 3.4). A detailed discussion of state-of-the-art hybrid evaluation frameworks follows later in Chapter 4, Chapter 5 and Chapter 6 which all describe specific SHE configurations. We hence defer a comparison of related approaches with SHE to these chapters.

3.1 Time Synchronization

The time synchronization in a SHE setup prevents the system representations to drift apart in time. We now first discuss important assumptions and requirements for the synchronization of different system representations before we closely describe the chosen synchronization approach and its characteristics.

3.1.1 Execution of System Representations: Assumptions

In order to keep the possible set of evaluation tools and techniques to be used as system representation as large as possible, it is important not to impose too many requirements on these "SR candidates". However, we base our design on the following set of assumptions that forms the common denominator for the execution of all system representations in a SHE setup:

- **No Predictability of Future SR Execution:** We generally assume that we can not predict the future behavior of an SR and the systems it models. This implies that any prospective communication activities (e.g., sending a network packet) or system events (e.g., a communication system suspends itself or becomes active again) are unknown at any prior point in time.

  This constraint is admittedly not present for some kinds of system representations, most notably discrete event-based simulations. Here, the future execution behavior can be investigated by iterating over the simulation event queue. The inspection of this so-called *look-ahead* is often used to optimize the synchronization of multiple parallel simulation processes [Fuj88].

  Similar techniques are employed in event-based hardware simulators for accelerating their execution, for instance by detecting idle phases [MCE$^+$02, EAW10]. However, for other system representations such as VMs hosting legacy operating systems and communications software, predicting the future run-time behavior is impossible. The execution of an operating system is typically non-deterministic because of a complex interweaving of concurrent processes and events, such as network and user input, threading or interrupt handling [KDC05].

  In order to facilitate such system representations to be included into a SHE setup, we abstain from requiring any execution predictability from the SRs.

- **Monotonic Progression of Time:** We make the assumption that the time at all system representations progresses monotonically. Hence, a system representation must not remigrate to an earlier execution state on its timeline.

  This assumption is made because of different reasons. A local rollback performed at one SR would require all other SRs to be also rolled back to exactly the same prior point in time; otherwise a time drift between the rolled-back SR and non-rolled back SRs would be the straight consequence. In order to support such a functionality, Distributed Coordinated Check-Pointing (DCC) [EAWJ02] of the entire SHE setup would be needed. As we discuss in

Section 3.1.2 we aim at synchronization accuracies well below 1ms[2]. We expect DCC to be impractical and very costly for such high check-pointing frequencies. Additionally, implementing according rollback functionality is difficult for different types of system representations. For example, state-of-the-art check-pointing frameworks for VMs achieve snapshotting intervals down to 20 ms [ZDJ+10, CLM+08]. This snapshotting granularity, however, is not sufficient for supporting the desired level of synchronization accuracy.

In summary, assuming a monotonic advancement of time lowers the implementation demands the system representations are confronted with.

- **Controllability of SR time sources:** One of the key concepts of Synchronized Hybrid Evaluation is that we provide all system representations with a consistent virtual progression of time that is decoupled from wall-clock time. In order to allow the SYNC component to centrally manage this progression we assume that we can fully control the system representations' internal timing infrastructure. As we discuss later in Section 3.2 this typically requires changes to the SR's scheduling subsystem.
- **Run-time Controllability:** In order to implement the synchronous execution of different system representations basic control primitives such as `start`, `stop`, `pause` and `resume` are needed. We assume that these actions can be performed at any point in time. We also expect each system representation to provide access to corresponding functionality via a software API or assume them to be easily extendable with such an interface.

3.1.2 Synchronization Requirements

We now discuss different high-level requirements for the synchronization of SRs.

3.1.2.1 Synchronization Accuracy

Two of the most important questions related to Synchronized Hybrid Evaluation are the following: *Can we tolerate time differences between different SRs? What is the required level of synchronization accuracy?*

The first question can be answered clearly affirmatively. The core motivation of a SHE setup is modeling a network consisting of different communication systems. The operation of the vast majority of communication systems, for example Internet hosts, exclusively depends on communication messaging, which may happen at any point in time. By contrast, almost all communication systems and network protocols do not rely on a globally synchronized time. The time reported by the local clocks of communication systems usually differs. Most network protocols and communication systems hence are designed to operate independently of such clock disparities.

Hence, we do not aim at synchronizing the local clocks of the different system representations. As we have discussed in Section 2.1.5, different execution speeds that

[2]This magnitude of synchronization accuracies is imposed by the typical end-to-end delay found in modern communication networks (cf. Table 2.1).

are mutually observed by non-synchronized system representations are the more likely cause for deteriorated performance measurements in hybrid evaluation frameworks. Hence, the synchronization scheme instead has to align the execution speeds of the different system representations in order to avoid relative time drifts. For this purpose, an according synchronization algorithm guides the execution of all participating system representations on one common virtual timeline and makes sure that the system representations do not drift apart. The second question regarding the required level of synchronization therefore has to be rephrased to question how much time drift between the system representations can be tolerated.

The synchronization accuracy needs to be chosen in a way that it limits the time drift between SRs considerably below the minimum end-to-end delay in the network as modeled by the SHE composition. The reason is the following: If system representations differ in their run-time performance, this inequality in execution speed leads to increased communication delays being observed by the "faster" SR. As the communication delay between two SRs is always bounded below by the delay in the modeled network, additional delays introduced by time drifts would only cause significant artifacts if they are larger than the network delay.

3.1.2.2 Synchronization Transparency

In order to obtain valid measurements in a SHE setup, it is vital that the synchronization process has ideally no or at most a marginal impact on performance measurements conducted using the system representations. Many performance metrics such as throughput or RTT are directly dependent on time measurements taken on the SRs. As synchronizing the execution speed of different SRs requires decoupling their timeline from wall-clock time and potentially altering their perception of time, this is a crucial issue for the implementation of SR components.

3.1.2.3 Preservation of Temporal Causality

The temporal causality of events, for example the exchange of network packets, must not be flawed by the synchronization process. Otherwise, the valid operation of basic protocol mechanisms such as handshakes at the end hosts could be easily disrupted. As we assume the system representations to progress only monotonically, this risk is largely taken away; however, this also implies that the synchronization scheme must not reset the execution of a SR to an earlier state either (cf. Sec. 3.1.1).

3.1.3 The Synchronization Scheme

The timing issues found in a non-synchronized network emulation framework stem from the problem that different system representations independently progress on individual timelines at different speeds (cf. Sec. 2.1.5). In a SHE setup, the SYNC component synchronizes the execution speeds of all system representations.

In order to carry out this synchronization the SYNC component needs to implement a suitable synchronization algorithm. For this task we consider a list of "candidate algorithms". Many of these algorithms originate from the research domain of Parallel Discrete Event-based Simulation (PDES) [Fuj90]. PDES aims at the synchronization of an event-based network simulation that is split across multiple parallel logical processes that work on individual event queues.

3.1.3.1 Classes of Synchronization Algorithm Candidates

According to the candidate algorithms, we differentiate between two classes of synchronization schemes, optimistic and conservative synchronization schemes.

- **Optimistic synchronization schemes** have been extensively studied in the PDES domain. The core idea is to execute the parallel simulation in a speculative fashion. Hence, synchronization errors may still occur with this class of algorithms. Optimistic schemes aim at detecting synchronization errors and they take different actions to recover from occasional violations. In the research domain of PDES, the so-called Time Warp algorithm [JS85] uses snapshotting to roll back a parallelized simulation to a consistent and error-free global state. Carothers et al. instead propose to recover from synchronization errors using reverse computation [CPF99], which allows one to partially undo the execution of a computer program. However, as we assume system representations not to support arbitrary roll-backs (cf. Sec. 3.1.1) and as we expect SRs to solely progress monotonically on the virtual time line, such schemes cannot be applied for the synchronization of SRs in a SHE setup.

  In the research domain of network emulation, it has been repeatedly proposed to dilate the time progression of virtual machines to extend the capacities of emulation testbeds [GMHR08, GVV08, GKN+04]. Sveet [ELL09] uses time dilation to statically slow down a VM to match the execution performance of a network simulation. We provide an elaborate comparison with these tools in the related work section (cf. Sec. 4.4). However, the bottom line is that all these tools only execute an emulation scenario in a speculative fashion. None of them is able to conceptually prevent time drifts.

- **Conservative synchronization schemes** prevent time drifts between the synchronized entities. For example, a well-known conservative algorithm from the PDES domain is the so-called null-message algorithm by Chandy and Misra [CM79]. The null-message algorithm maintains a global order of events in a distributed fashion by exchanging so-called null messages. A null message tells other simulation instances until which point in the future they may safely proceed. This information is obtained by inspecting the future events in the simulation queue (look-ahead). While determining this look-ahead is possible for SRs such as network simulations (cf. 3.1.1), we cannot predict the future execution behavior of other SR types, most notably virtual machines.

In effect, this limits the choice of a synchronization algorithm for Synchronized Hybrid Evaluation to a scheme which neither makes assumptions about the future behavior nor requires regular snapshots to be taken.

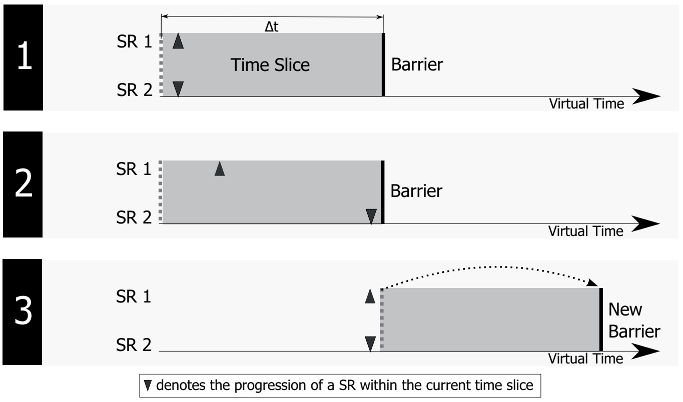


Figure 3.2 Different steps of the barrier algorithm (depicted for two SRs): ① Both SRs start at the beginning of a new time slice. ② The execution of both SRs is blocked until the completion of the time slice. ③ All SRs have completed the time slice; the barrier is advanced.

3.1.3.2 Barrier Synchronization

Synchronized Hybrid Evaluation uses a scheme we refer to as *barrier synchronization* for aligning the execution speeds of different SRs. The algorithm is related to Conservative Time Windows (CTW) [Lub89] in the PDES field. The core idea of the barrier synchronization approach is to supply all SRs with tiny bits of virtual run-time. Figure 3.2 shows the synchronization of two SRs via the barrier synchronization algorithm. It allows every synchronized SR to run for a certain amount of time, the so-called *time slice*, after which it blocks until all other SRs also have reached the barrier. At this point, the barrier is lifted, and a new future barrier is set up until which the execution progresses. This way, all SRs are guided on one common virtual time line that is discretized by the time slices. As the execution of all SRs is always bounded by a barrier, the time drift between them and the jitter on the shared timeline is limited to the size of one time slice at all times. Consequently, the synchronization accuracy is directly given by the size of the time slice.

Timeline Progression

Figure 3.3 illustrates how two barrier-synchronized SRs progress on the virtual time line in comparison to their real-world execution. In this hypothetical example the time slice size Δt is set to 1ms, and the real-world execution time of the SRs for Δt is assumed to randomly vary between 0.2 ms and 2 ms. Hence, both SRs execute faster or slower than real time. The TS labels at the right denote the time slice on the virtual time line (opposite of y-axis). We refer to the real-world time it takes for all SRs to complete a time slice as a *synchronization period*. These periods are highlighted on the opposite of the x-axis.

Three important properties of the barrier synchronization scheme are observable in this plot: Firstly, the suspension of the SR execution at the end of the time slice causes the time to stand still until the slower SR has also completed the current

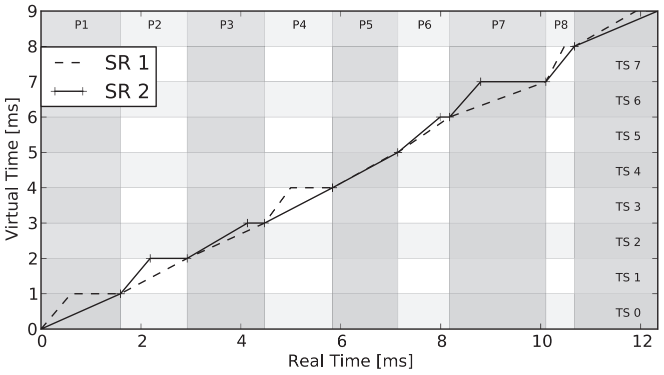


Figure 3.3 Exemplary visualization of the time progression at two SRs that are synchronized using the barrier synchronization scheme.

slice. This leads to a step-like time advancement pattern if the execution speeds of the SRs differ widely. Secondly, we see that the barrier-synchronization algorithm effectively decouples the virtual timeline from real-world time: the time slices always take the specified 1ms on the virtual time line, while the corresponding periods span largely different real-world intervals. Thirdly, as the bounds of the time slices on the virtual timeline form mandatory synchronization points for all participating system representations, the time drift is limited to the size of a slice in the virtual time domain. This property also holds if the SRs are not synchronously started at the beginning of a time slice, which is assumed in the depicted example.

Time Slices of Uniform Size

A further relevant aspect of our synchronization scheme is that we only use time slices of uniform size. Hence, it forms a static parameter that is configured before a SHE measurement run. This decision was made because we assume that the execution speed of a SR may vary arbitrarily. In other words, we believe the real-world duration of a synchronization period to be independent from prior synchronization periods.

This assumption is motivated by the following gedankenexperiment: A system representation might be fully idle until a certain time t_{action} on the virtual time line and hence not require much computational resources. Consequently, the SR is able to complete the time slices before t_{action} faster than real-time. However, when arriving at t_{action} the SR might switch from fully idle to a state where it has to perform longstanding computations, leading to a very slow progression on the virtual time line. For example, such a behavior may occur with a network simulation, for which at t_{action} a larger number of simulated hosts synchronously starts to put heavy load on the simulated network. Such discontinuities in the execution performances of the system representations essentially make it difficult to enforce strict synchronization bounds if the time slice size would be altered dynamically. We therefore confine ourselves to uniform time slices.

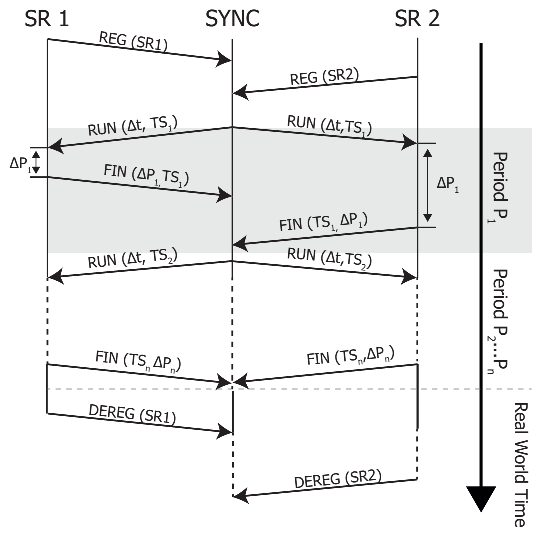


Figure 3.4 This protocol diagram illustrates how our synchronization protocol controls the execution of two system representations on the virtual timeline.

3.1.4 The Synchronization Protocol

The synchronization component uses a straightforward synchronization protocol to assign the time slices to the system representations. Its mode of operation is depicted in Figure 3.4. The synchronization protocol comprehends four different message types:

1. The `REG` messages registers a system representation at the synchronization component using a unique identifier. Once the SR has sent the `REG` message, it blocks its execution and waits for the next time slice to be assigned. A system representation may register at a SYNC component at any point in time. The SYNC component waits for all registered SRs to complete their time slice before assigning the next one.

2. The SYNC component broadcasts so-called `RUN` messages to all registered SRs. The `RUN` message specifies the duration of the time slice, Δt. It also contains a strongly monotonic sequence number (TS_n) that defines the current period. The sequence numbers are required in order to enable the system representations to distinguish different periods. This is important, as `RUN` messages may be rebroadcasted by the SYNC component if a corresponding FIN message is not received within a certain time interval (time-out).

3. When a system representation has completed the execution of a time slice, it sends a `FIN` message to the synchronization component. Besides the ID of the completed time slice (TS_n), the `FIN` also contains a second value (ΔP_n) that

provides information to the SYNC component how much real-time it took the SR to execute the time slice.

4. The `DEREG` message is used to deregister a SR at the synchronization process. After receiving such a message, the execution of the system representation is decoupled from the SHE setup. Hence, the SYNC component does not wait any more for the deregistered SR to report back finished time slices after receiving a `DEREG` message.

It is worth mentioning that the synchronization protocol makes a few assumptions about the network interconnecting the SYNC component and the SRs. Firstly, we assume that the interconnecting network of the SHE setup is a local area network (LAN) that is fully controlled by the SHE setup maintainer. We also expect the network to be isolated from other computer networks and that the system representations and the synchronization component are the sole communication peers. Using an isolated and private network for example makes it possible to abstain from incorporating more complex authentication schemes into the protocol.

A second important requirement is that the underlying communication technology supports broadcasting or multicasting. We hence assume that the run permissions are delivered to all SRs roughly at the same time. In the case of IP (as used by our implementation discussed in Chapter 4) the use of broadcasting implies that all system representations and the SYNC component belong to the same broadcasting domain or multicast group, respectively.

3.1.4.1 Messaging Overhead

An important preliminary consideration for later implementations of SHE is the messaging overhead caused by the synchronization protocol. Therefore we now quantify the number of messages (MC) that is exchanged between a set of SRs ($\{S\}$)and the SYNC component for a given interval of virtual run-time (T_v).

$$MC = \frac{T_v}{\Delta t} * (1 + |S|) \tag{3.1}$$

The first factor of the product in Equation 3.1 amounts to the number of periods in the considered time interval T_v; it can simply be obtained by dividing the interval on the virtual time line by the time slice size Δt. The second compound factor corresponds to the number of messages sent by the individual components in an SHE setup during one synchronization period. The SYNC component sends one broadcast message to all SRs, which report back the completion of the time slice to the SYNC component using individual messages. Hence the total number of messages sent by the components in each period equals $(1 + |S|$.

If we, for example, calculate the message count MC for a setup with two system representations, $T_v = 1s$ and a typical time slice size $\Delta t = 100\mu s$, we obtain $MC = \frac{1s}{10^{-4}s} * (1 + 2) = 30.000$ messages. Hence, every second of progression on the virtual time line requires this amount of messages to be exchanged between SYNC and SR components.

SR Type	Execution Model	System Scalability	Deterministic/ Repeatable Execution
Network Simulation	Simulation Models	Many Systems	Yes
Virtual Machines	Real Applications + Real OS	Few VMs on 1 physical Machine	No
Full-System Simulators	System Hardware + partially OS/Apps	Few systems	Yes

Table 3.1 Important high-level properties of different system representation types.

3.2 System Representation Components

As discussed in Chapter 1, a major reason for setting up hybrid evaluation frameworks is to benefit from the different and sometimes orthogonal strengths of different evaluation technologies. In the concept of Synchronized Hybrid Evaluation, we subsume network simulations, virtual machines and hardware simulators under the term *system representations*, as they all recreate the behavior of one or multiple communication systems at different levels of abstraction. We now first discuss general aspects and properties of system representations before elaborating the specific types of SRs considered in this document.

3.2.1 Characteristics of System Representations

From a conceptual perspective, a major reason that different system representations are able to amend each other stems from the fact that they hold widely different properties. Table 3.1 categorizes the classes of system representations considered in this thesis by three categories: Execution Model, System Scalability and the question if the systems are deterministic and thus enable repeatable executions.

3.2.1.1 Execution Model

The execution model essentially describes how a system representation technology reproduces the operation of the system it models. Here, we basically differentiate between two different classes of SRs. The first class consists of virtual machines and partially hardware simulators that execute an unmodified communication system or model communication hardware. For this purpose the system representation has to recreate the execution context of the communication system, for example the Operating System (OS) environment.

The second class of SRs on the contrary aims at indirectly reproducing the functionality of a system, e.g., by solely imitating its communication behavior. By reducing the system representation to a concise model, we obtain a model that is indeed less precise but also has a much lower overhead. Network simulations are the most prominent representative in this class of system representations, however, hardware simulators often also employ abstract models that neglect the detailed behavior of system components or peripherals.

3.2.1.2 System Scalability

The system scalability describes the magnitude of systems the SR is able to model. Virtual machine-based system representations and full system simulators are usually only able to model one or at most a few complete systems at full detail or with full native performance on one physical computer. The reason is that they have to reproduce the entire execution context of the communication system that is investigated (for example by simulating the detailed hardware behavior).

By contrast, network simulation-based SRs use abstract models to represent an entire set of communication systems. These abstract models typically only encompass the basic functionality that is needed to reproduce the functional behavior of a communication system. In comparison with VMs and FSSs, simulations are rather lightweight in terms of computational complexity. For this reason, modern network simulation tools are easily able to model many hundreds to thousands of systems on present hardware [WvLW09].

3.2.1.3 Deterministic/Repeatable Execution

A major difference between the different types of system representations is the ability of reproducing the behavior of a system deterministically. In this context, determinism refers to the possibility of exactly repeating the execution of an SR in a way that its execution path is identical compared to a prior run. Common virtual machine monitors (VMMs) such as Xen [BDF+03] or VirtualBox [Wat08] only provide a non-deterministic execution of the guest system. The reason is that standard VMMs execute most of the instructions of the guest system natively on the CPU of the host machine. Hence, there is no complete isolation of the guest system's operation from the non-deterministic execution of the VMM host. This yields to a non-deterministic execution of the guest system as well, because side effects due to the shared CPU usage (for instance cache misses, flushed instruction pipelines or the execution of interrupt handlers) may directly influence the execution flow of the guest system.

In contrast to that, both full system simulations and event-based network simulators feature a fully deterministic execution of the system(s) they model. The reason is that both hardware simulations and event-based network simulators model the systems entirely in software. While these system models may also involve randomness, an identical run of the SR can simply be achieved by using an identical seed for the initialization of the random number generator.

3.2.2 Virtual Machines (VMs)

In traditional network emulation environments like the original EmuLab [WLS+02] or the CMU Network emulator [JS04, JS05] physical machines are used for executing real-world communications software. By contrast, in a synchronized hybrid evaluation setup, virtual machines replace physical computers. The reason for using VMs instead of real-world machines are two requirements that are imposed by the synchronization scheme:

- **Full access and control over time sources:** An operating system uses different time sources for a wide range of tasks, from implementing low-level timers (e.g., these used by protocols to detect time outs) to providing user-land applications with the current wall-clock time [Sta01]. These time sources are part of the system hardware. For example, commonly used time sources[3] in the present x86 systems are the Real-time Clock (RTC), the Time Stamp Counter (TSC), the Programmable Interval Timer (PIT) and the High Precision Event Timer (HPET). One cornerstone of Synchronized Hybrid Evaluation is to supply the synchronized system representations with a virtual progression of time. Hence, in order to virtualize the time progression of an operating system, we need to virtualize its time sources. This can either be achieved by changing the time sources accessed by the operating systems' kernel or by virtualizing the hardware timers.

  The first option would require access to the source code of the kernel, which might be not available for closed-source operating systems. Such a solution would demand changes to be applied individually for different operating systems or even kernel versions.

  In contrast to that, the virtualization of hardware time sources allows unchanged operating systems to be supplied with a virtual progression of time. In order to technically carry out this virtualization of hardware time sources, we use a virtual machine monitor to separate the operating system from the underlying physical hardware.

- **Transparent suspension of OS execution:** Our synchronization scheme requires the execution of the SR to be suspended after the current time slice has been finished. Hence, we need a mechanism to transparently freeze the execution of an operating system and the software running on it. As the execution of an OS is usually a non-deterministic conglomerate of different concurrent processes and events, manually implementing lockdowns in an OS kernel can be expected to be cumbersome.

  Instead, the most straightforward way to achieve a transparent suspension is to pull back the execution of the OS from the CPU entirely when the time slice has been consumed. This, similarly to the virtualization of time, requires an indirection layer between the CPU and the operating system. In the case of synchronized hybrid evaluation, this indirection layer is formed by a hypervisor.

It is absolutely vital for the correct operation of VM-based SRs that the real-world time gaps that are introduced by the synchronization scheme are not observable by the guest system running inside the VM. Similarly, these frequent interruptions in the VM's execution flow should not perturb performance measurements carried out using the VM. We subsume these requirements under the term *synchronization transparency* (cf. Sec. 3.1.2.2). As stated earlier, VMs are not fully isolated from the host system that concurrently executes the synchronization infrastructure and potentially also other virtual machines. Hence, achieving a perfect synchronization transparency is impossible. For our VM-based SR, we later quantify the impact of synchronization on VM performance in Section 4.1.

[3]A good and concise overview over time sources available in present x86 systems was given by Vojtech Pavlik (Novell Suse) in his post on the Linux Kernel Mailing List [Pav05].

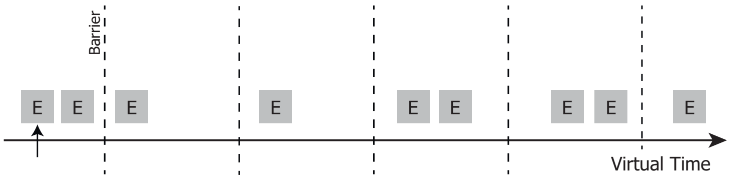


Figure 3.5 A network simulator can serve as system representation using a modified event scheduler that checks if the next event resides in the current time slice. If yes, it is processed. Otherwise, the simulation blocks until the assignment of the next slice.

Most established open-source virtual machine monitors like Xen [BDF$^+$03], VirtualBox [Wat08], Kernel-based Virtual Machine (KVM) [Kiv07] or qemu [Bel05] may serve as implementation basis for a VM-based system representation. From a technical perspective, proprietary VMMs such as Microsoft HyperV [mic] or VMWare ESX [VMW] might also be used for such an endeavour. However, as implementing a VM-based SR requires changes to the core of the hypervisor, this option is not viable for most researchers and developers. Hence, in the following discussion we restrict ourselves to VMM software for which full source code is available.

Open source virtual machine monitors, depending on their internal architecture, are more or less suitable for implementing VM-based SRs. The reason is that VMMs differ widely in their scheduling and timekeeping subsystems. For example, the type 2 hypervisor VirtualBox[4] offloads the scheduling entirely to the operating system that executes the VMM. Moreover, accesses to time sources are forwarded to the underlying OS. By contrast, other Virtual Machine Monitors, especially native hypervisors such as Xen and VMWare ESX, feature a scheduling component that centrally manages the execution of the VMs on the host machine's CPU. The Xen hypervisor, which forms the basis for our VM-SR implementation, allows for the integration of customized scheduling algorithms and components. It also provides suitable interfaces for altering the virtual machines' time perception. We later provide an elaborate discussion of this implementation in Section 4.1.

3.2.3 Discrete Event-based Network Simulation

In a Synchronized Hybrid Evaluation scenario, network simulation system representations are used to model the computer network that interconnects the different system representations. Hence, their task in contrast to other SRs such as virtual machines is not to represent a single system, but a large set of communication systems. The network simulator models all other elements of the computer network, ranging from communication channels over middleboxes to contextual properties such as mobility.

From a conceptual standpoint, most network simulators (for example OMNeT++ or ns-3, for a broader discussion see Section 2.1.2.3) may serve as implementation basis

[4]This architectural properties of VirtualBox were obtained from a code review of the source code of VirtualBox (version 4.12).

for this type of system representation. The task of integrating a network simulation framework into a SHE setup as a system representation is twofold. Firstly, we need to adapt the network simulator's scheduler to follow the barrier synchronization algorithm. Secondly, we have to establish interoperability between the network protocol models used by the simulation and the other system representations of the SHE setup.

3.2.3.1 Scheduling

Adapting the event scheduling of an event-based network simulator in order to make it follow the barrier synchronization scheme is rather straightforward (cf. Fig. 3.5).

Recall that an event-based network simulator maintains a list of all scheduled events ordered by the time of execution. Usually, the simulation simply processes these events sequentially until the event queue is empty. In synchronized network emulation, the scheduler checks if the next event's time of execution resides in the current time-slice. If this is the case, the event is executed. If not, the event scheduler notifies the synchronization component. The next event is processed after the barrier has been shifted past the execution time of the event.

3.2.3.2 Protocol Model Interoperability

In fact, the larger challenge connected to the use of a discrete event based network simulation tool as system representation is the establishment of interoperability between the simulator's network protocol models and the implementations or simulations executed on other SRs. The main reason for this is that network simulators mostly use internal representations for network packets that differ widely from their real-world counterparts. For example, OMNeT++ [VH08] models network packets using an object-oriented scheme. Network packets corresponding to a particular network protocol are created by instantiating an according message object. Packet encapsulation is expressed using pointers that organize these objects into a hierarchy. The actual communication between simulated network hosts in OMNeT++ is then modeled by passing pointers to these objects among the simulated hosts.

It is apparent that this approach for modeling network protocols differs widely from real-world communications systems that exchange binary packets. Hence, network simulators need to implement mechanisms to convert between the internal representation of network protocol packets and the interchange format used for the communication among the system representations. Typically, we assume this to be a common MAC layer protocol. We later discuss the conceptual aspects of the communication between different system representations in Section 3.3.

3.2.3.3 Gateway Nodes

One particularity of network simulations in comparison to other system representations is that they model not only a set of systems but also a network topology that interconnects the simulated nodes. These network topologies may be of arbitrary shape. Examples are rather simple grid or dumbbell topologies that are typically used for early evaluation studies; for the creation of large-scale scenarios we often resort to network topology generators such as INET [JCJ00] or BRITE [MLMB01]. Naturally, these topologies are self-contained if the simulation operates on its own.

This is not the case if a network simulation is used as system representation in a SHE setup. We therefore need to specify the location in the simulated topology at which the traffic originating at the other system representations is connected with the simulated network. This takes place at the so-called Gateway Nodes (GNs) that are part of the simulated network topology. The gateway nodes form the logical endpoints for the packet exchange interface. Every network packet that arrives from an attached SR at the gateway node is injected into the simulated network. Similarly, network packets that arrive at the GN in the simulated network are transferred to the system representation that is attached to the gateway node using the packet exchange interface.

3.2.4 Full-System Simulators

The objective of Full-System Simulators (FSSs) such as Simics [MCE+02], CoWare Virtual Platform [cow], SimOS [RHWG95] or Mambo [BEG+04] is the deterministic simulation of entire hardware systems at the instruction or even the cycle level (cf. Sec. 2.3). For this purpose, these full system simulators (FSSs) completely emulate the behavior of a CPU, system caches, the memory, storage devices and other peripherals. Simics even can be used to model an entire computer network [EKMR05]. Many full system simulators model the hardware to a degree that enables the execution of operating systems and device drivers on the simulated hardware. Therefore, one of their main applications is cross-platform development of low-level software (e.g. device drivers or OS kernels) for future hardware.

Another important strength of these tools is the unsurpassed degree of control they offer to developers and researchers. Full system simulators allow one to pause the execution at any point in time in order to investigate the system state, for instance the content of CPU registers or its instruction pipeline. While this high level of controllability satisfies many wishes and requirements of system software developers, the high overhead of full system simulation hinders their use for investigating large-scale computer networks. Therefore, we now discuss the technical implications of integrating them into a synchronized hybrid evaluation setup.

3.2.4.1 Scheduling

In general, most full system simulators are well prepared for the potential use as system representation. One reason is that their normal operation already requires fine granular scheduling, and many full system simulators eventually provide mechanisms to execute the compound system model for an exact amount of time or a corresponding number of cycles. If this is the case, adapting the FSS to the barrier synchronization scheme essentially boils down to implementing a client module that transcribes the time slices received using the synchronization protocols into corresponding calls to the API of the full system simulator. If the FSS does not provide an explicit API, the synchronization can still be implemented at a lower level: most

FSSs are implemented using an event-based architecture, and hence an according synchronization scheduler may be implemented similar as discussed for event-based simulation tools. In fact, our implementation of a hybrid evaluation platform that integrates a FSS with a network simulator (cf. Sec. 5) implements the synchronization on top of SystemC [Sys06], which is an event-based simulation library used for modeling different types of hardware, for example CPUs or FPGAs.

3.2.4.2 Interoperability with other SRs

In contrast to discrete event-based network simulations, establishing the interoperability of network protocols running on a FSS-based system representation with other SRs is rather straightforward, as a full-system simulator models a system at a comparably high level of detail.

As discussed in the following section, we mostly consider the SRs to be interfaced at the MAC level. While the exact way of establishing this connectivity between the FSS and different SR types depends on the actual implementation of the FSS-based SR, it can be considered to be straightforward. The reason is that most networked systems construct packets for the MAC layer either entirely in software or at the "upper layers" of the system's hardware architecture. Hence, this functionality is mostly contained in the used hardware model provided by the FSS or in the software executed on top of the simulated hardware. This generally spares the need of implementing a message translation from the SR to the commonly used protocol for the data transfer among different system representations.

3.2.5 Other System Representation Types

While this thesis focuses on virtual machines, event-based networked simulations and full system simulators as technologies to represent a networked system in a SHE setup, other imaginable types of system representations could be incorporated into a SHE setup in an analogous way. One example for such an unstudied system representation type are special-purpose simulators, for example in the domain of P2P systems (e.g. [SGR+11, BHK09, MJ09]) and wireless sensor networks (e.g. [LLWC03, Bou07]).

Integrating a new SR type into a SHE setup consists of implementing a client for the synchronization protocol and a scheduler into the execution environment of the SR. In addition, the exchange of data needs to be interfaced to the other SR's. Naturally, the development efforts connected to such an adaptation directly depend on the soft-/hardware architecture of the SR and its level of abstraction.

3.2.6 Hybrid System Representations

So far, we have merely looked at monolithic system representations where one functional unit (for instance a virtual machine) models one or a set of systems. However, a system representation can also be hybridly formed using different representation technologies. This idea is illustrated using an example in Figure 3.6, in which a

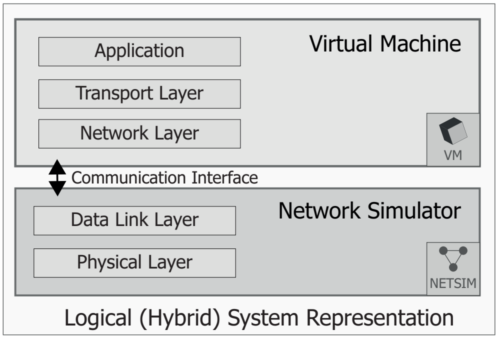


Figure 3.6 An exemplary hybrid system representation that uses a VM for the representation of the upper layers in a network stack. The lower layers are modeled using a network simulator. A communication interface integrates both partial system representations into a logical one.

network simulation and a VM cooperatively form a logical, hybrid system representation. The network stack that provides the basis for the modeled communication system is split across the two, and a communication interface is used to integrate both partial system representations.

One motivation behind such hybrid system representations is to overcome certain limitations that are bound to specific SR types. By moving parts of a system modeled by a SR for example into a network simulation, we can reduce the amount of non-determinism regarding the execution of the SR. This is particularly useful if the behavior of an SR's subsystem is heavily non-deterministic and/or highly dependent on environmental factors, for example, the wireless channel. In fact, the concept we later propose as Device Driver-enabled Wireless Network Emulation (DDWNE) (cf.Sec. 4.2) can be regarded as a hybrid system representation.

3.3 Packet Exchange between System Representations

The task of the packet exchange interface (PEI) is to enable the actual exchange of network packets between the different system representations. Technically, it corresponds to a network tunnel that is used to transport network packets between the different SRs. We now elaborate important conceptual aspects of this interface and give reasons for our design choices.

3.3.1 Non-Synchronized Packet Exchange

Looking at the high-level architecture of synchronized hybrid evaluation, (cf. Fig 3.1), it becomes obvious that the transport of network packets between the system representations is decoupled from the synchronization process. Hence, we cannot give any guarantees about the number of time slices it takes for a network packet to travel from one SR to another. Instead the end-to-end delay between two SRs is only dependent on the SR's implementation and the delay introduced by the packet exchange interface. As discussed in the following, this design choice was made for different reasons.

There are different hypothetical ways how a deterministic delay of the PEI could be realized. One option would be the introduction of so-called "transport cycles" that would alternate with the execution of the time slices. During a transport cycle, the execution of all SRs would be halted, and the SRs would download fresh network packets from each other. At the end of a transport cycle, all SRs would inject these network packets into the systems they model and resume their execution. We abstain from such a solution for two reasons: First, the introduction of periodic transport cycles would increase the real-world run-time of a SHE scenario, as all SRs would be required to pause their execution during the transport cycles. The second reason is that we expect a pull-based interface for the exchange of network packets to be difficult to implement for SR types like virtual machines at which determining the availability of new network packets would require access to system internals like the network stack.

We mentioned earlier that we assume a SHE setup to be deployed on a local area network. Contemporary LANs based on Fiber optics or Gigabit Ethernet typically achieve end-to-end delays in the order of a few tens up to a few hundreds of microseconds. Hence, the delay artifacts added to measurements by the PEI in a SHE setup can be expected to be of this magnitude. Given the fact that we foresee SHE primarily to be applied for the evaluation of network systems for wide area networks like the Internet, we presume these artifacts to be of sparse significance, as the end-to-end delays mostly range from at least a couple of milliseconds to tens of milliseconds in the envisioned use-case scenarios.

3.3.2 Integration at the MAC layer

In a Synchronized Hybrid Evaluation framework we integrate the communication of the different system representations at the MAC layer. Hence, the SRs exchange MAC frames in a common format, for example IEEE 802.3 Ethernet or IEEE 802.11 frames, over the packet exchange interface.

From a conceptual point of view, the only requirement regarding the packet exchange interface is that the system representations agree on a common network protocol and a common frame format. This is not particular for the concept of SHE. Likewise, different network emulation frameworks have proposed to integrate network simulations with physical systems at different layers. For example, Avvenuti and Veccio [AV06] propose a network emulation framework at the application layer by providing an alternate socket library that redirects all communication over the socket to an emulation engine. As a direct consequence, this emulation toolkit is limited to serve for evaluations of protocols that operate on top of the socket layer. Other network emulation frameworks such as wtun [Sei08] bridge the simulation with the systems attached at the IP layer. While such frameworks provide the opportunity to examine both custom application and transport protocols, they still obstruct emulation studies that rely on routing protocols other than IP.

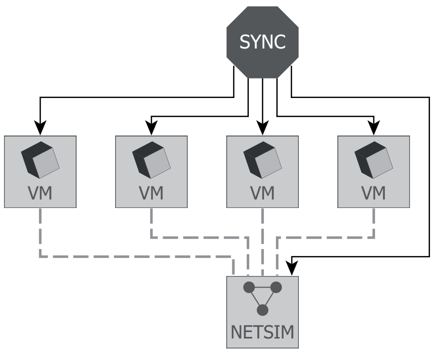


Figure 3.7 A SHE setup for synchronized network emulation comprises two types of SRs, a set of virtual machines and at least one network simulation.

The reason for our design decision to integrate the SRs using a common MAC layer protocol is that it marks the lowest possible level of abstraction beyond the physical exchange of data between communication systems. Building an emulation framework that operates at the MAC layer offers the highest level of generality without crossing the hardware-software boundary of communication systems.

3.4 Types of Synchronized Hybrid Evaluation setups

Given the fact that all system representations are equipped with mutually compatible interfaces for packet exchange and synchronization, the SHE concept allows for creating different types of hybrid evaluation setups by simply plugging the respective SR types together. We now illustrate different SHE setups and how they relate to existing hybrid evaluation methodologies.

3.4.1 Synchronized Network Emulation

The first class of SHE setups we term *Synchronized Network Emulation* (cf. Fig. 3.7). A SHE composition for synchronized network emulation comprises two types of system representations, a set of virtual machines and one network simulation that models the interconnecting network between the VMs. The main motivation behind synchronized network emulation as described in our prior work [WSHW08, WSHW09] is to overcome the problem of simulation overload that exists in classical, non-synchronized network emulation tools like [Fal99]. Simulation overload restricts these tools to network simulations that execute in real-time, thus hindering their applicability to setups with network simulations of high complexity (see also Sec. 1 and Sec. 2.1.3.2).

From a conceptual point of view, Synchronized Network Emulation shares some similarities with other network emulation frameworks that employ virtual machines.

DieCast [GVV08] dilates the progression of time in a network testbed in order to increase the capacity of physical network testbeds. The same authors also have demonstrated that altering the time perception of a VM allows for the emulation of high-speed network links for which no hardware is available at present [GYM+06]. DONE [BVB06], TimeJails [GMHR08] and SVEET! [ELL09] all synchronize the execution of VMs, and in the latter case, also a network simulation to enable network emulation scenarios that do not operate in real-time. As we later discuss in the corresponding related work (cf. Sec. 4.4), all these frameworks unfortunately only implement rather rough optimistic synchronization schemes that are unable to prevent synchronization errors.

Later in this dissertation we provide an elaborate discussion of our framework for Synchronized Network Emulation, *SliceTime* (cf. Chapter 4). SliceTime allows one to setup network emulation scenarios in which the network simulation operates only at a fraction of real-time. We will later show that this allows one to setup network emulation scenarios with thousands of nodes or scenarios using computationally complex models that couldn't be used for emulation setups previously.

3.4.2 Synchronized Hardware/Network Co-Simulation

The second kind of SHE compositions we closely consider in this thesis are setups for the synchronized execution of one or more instances of a full-system simulator coupled to a network simulator (cf. Fig. 3.8). This allows the networked system, whose system hardware and system software behavior is modeled using a full-system simulator, to interact with a set of simulated communication systems.

The main motivation behind such SHE configurations is to provide a platform for rapid prototyping and the co-design of networking hardware and corresponding software, for example network stacks for embedded devices. While full-system simulators like Simics [MCE+02] also allow for the implementation of purely functional models of network protocols, the reason to amend full system simulators with a network simulator is given by the different specializations of both types of tools. Full-system simulators typically provide a very detailed model for one or more hardware architectures, but they are usually not equipped with many models of network protocols and respective applications. By contrast, network simulators exactly provide this functionality, but fall far short if it comes to the simulation of low-level system behavior, for example bus access or system caching behavior. It is noteworthy that all used system representations are constituted by different types of simulators. Hence, this form of Synchronized Hybrid Evaluation setups conceptually falls into the category of *co-simulation*; the notion of co-simulation describes the integration of different simulation tools in order to obtain a comprehensive model of a system that is difficult to come up with using only a singular simulation environment. The application of co-simulation for the purpose of co-designing embedded network hardware and software that involves a network simulator and a hardware simulation has been earlier conceived by Fummi et al. [FPM+04]. In the context of full-system simulation, co-simulation has been proposed for different tasks, for example to improve the timing accuracy of respective simulations [MHW02].

In Chapter 5 we discuss our corresponding SHE framework for the co-simulation of a full-system simulation based on SystemC and arbitrary ns-3 network simulations.

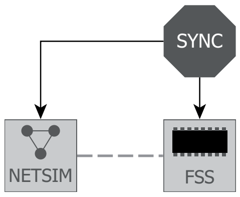


Figure 3.8 Synchronized Hardware/Network Co-Simulation frameworks contain two types of SRs, namely at least one network simulation and a set of hardware simulators.

Later in this document, we also provide a more elaborate discussion of how this framework relates to existing tools in the domain of software/hardware co-design for networked systems.

3.4.3 Other Types of Compositions

In addition to the previously discussed Synchronized Hybrid Evaluation (SHE) configuration types there are other options for SHE configurations that are sketched in the following.

3.4.3.1 Synchronized Hybrid Evaluation Compositions with more than two types of System Representations

Both *Synchronized Network Emulation* and *Synchronized Hardware/Network Co-Simulation* each comprise two different types of system representations. As all these SRs, however, share the same interfaces for synchronization (*time control interface*) and for the exchange of packet data (*packet exchange interface*), it is of course possible to setup more complex scenarios that incorporate all possible types of system representations for which according implementations are available.

One such configuration is illustrated in the introductory example (cf. Fig. 3.1). Such a SHE setup might be beneficial for instance to aid interoperability testing between legacy software (executed in a VM) and a future embedded networked hardware device; the network simulator could be used to model an interconnecting wide-area network or the mobility of the communication systems in such a scenario.

While such setups demonstrate the conceptual flexibility of SHE, there is only a narrow set of use cases that really require hybrid evaluation frameworks that comprehend more than two types of SRs at the same time. In addition, every additional type of SR increases the complexity of a SHE scenario, both regarding its manageability and its technical operation.

For these reasons, we do not provide an elaborate discussion of more complex SHE configurations in this document, as almost all emulation or co-simulation tasks can be projected to a SHE base scenario that contains one network simulation and either a set of VMs or one or multiple full system simulators.

3.4.3.2 Homogeneous Configurations

Technically, it is also possible to compose SHE configurations that only contain one singular type of system representation and the synchronizer. We refer to this type of setup as *homogeneous configurations*. In the strict sense homogeneous conjunctions of SRs and one SYNC do not form a hybrid evaluation framework, as all systems are represented using the same technology or tool.

For the set of SRs discussed in this dissertation there are three different imaginable types of homogeneous configurations:

1. **Synchronized Full System simulators:** The synchronization of different full-system simulator instances may be used to investigate the communication between multiple fully simulated hardware devices. Such a scenario is very similar to a Synchronized Hardware/Network Co-Simulation configuration with more than one full-system simulator attached to the network simulation. However, in the homogeneous case, a detailed model of the interconnecting network would be missing. Moreover, most full system simulation tools (Simics [EE06], Mambo [BEG+04]) already provide mechanisms to synchronize the execution of multiple of their instances, which obviates the need for this type of homogeneous configuration. Because of these reasons we omit a detailed discussion of the homogeneous synchronized coupling of full system simulators in this document.

2. **Synchronized Network Simulations**: The homogeneous conjunction of network simulations can be used to split a large simulated network topology into smaller subtopologies of which each is modeled using one separate network simulation SR. The data exchange between the subtopologies can be organized using the PEI. In such a setup the gateway nodes form the "overlapping" rendezvous points between the individual subtopologies. The synchronization of the SRs is used to integrate the run-time behavior of the different network simulation SRs.

   Such homogeneous simulation scenarios form a special case of parallelized discrete event-based network simulations. While the possibility of setting up parallelized network simulations using homogeneous configurations is a nice byproduct, it is conceptually clearly inferior compared to classical approaches for parallelized network simulation. First, we cannot guarantee a absolute global order of events, as our simulation is based on time intervals rather than individual simulation event timings. Second, the bandwidth limitations of the PEI (e.g. 1 Gbit/s for Gigabit Ethernet) puts a strict capacity limit on the links between different subtopologies. As the gateway nodes, however, might not only forward packets of one system but many simulated hosts, it is imaginable that the simulation traffic exceeds the PEI's capacity, leading to a deteriorated performance between different subtopologies. Third, splitting the global topology into subtopologies and organizing them into separate simulations has to be carried out manually by the developer.

   Existing network simulation tools such as ns-3 or OMNeT++ provide much richer and more flexible methods for this purpose. For these reasons, we disregard homogeneous simulation setups in the following. However, the possibility of splitting up a simulation topology might be helpful for SHE setups in which the network simulation outgrows the capacity of the simulation host machine.

3. **Synchronized execution of VMs:** A homogeneous configuration just comprising the synchronizer and a number of VMs is technically possible but rather pointless if it is used on its own. The reason is that the VMs operate in real-time. Hence, there is no need for synchronization or the virtualization of time.

   However, the situation dramatically changes if the execution of individual VMs is expected to be manually paused, e.g. by invoking an according command provided by the virtual machine monitor's control interface. The prevalence of synchronized execution will then yield to a transparent suspension of all VMs at the end of the current time slice. As we later discuss in Chapter 5 we take advantage of this circumstance in order to implement a time-synchronized framework for debugging and monitoring a set of virtual machines.

3.5 Interim Summary

In this chapter we have discussed the concept of Synchronized Hybrid Evaluation (SHE), a unified approach for the setup of hybrid evaluation platforms. The core idea is to construct such hybrid evaluation platforms using a set of reusable *system representation (SR)* modules. The execution of the SRs is conservatively synchronized in order to relieve the SRs from operating in real time. For instance, this allows emulation setups in which the network simulation operates only at a fraction of real-time; our corresponding implementation, *SliceTime*, is discussed in the upcoming section. In the following we will also pick up the conceptual implications on actual SHE implementations as sketched in this chapter. For example, one of the most crucial impacts of the barrier synchronization scheme is its high messaging volume, yielding to the need of efficient message processing in a SHE setup.

4

Synchronized Emulation Frameworks

Overview

In this chapter we present two emulation frameworks that are derived from the previously discussed Synchronized Hybrid Evaluation (SHE) concept. Both frameworks directly improve the general flexibility of network emulation by extending the range of scenarios to which this methodology can be applied.

- *SliceTime* [WSvL$^+$11, Sch08] implements Synchronized Network Emulation (cf. Section 3.4.1). By eliminating the requirement for the simulation to operate in real-time, SliceTime enables network emulation scenarios that include a discrete event-based network simulation with topologies of any size and any run-time complexity.

- *Device Driver-enabled Wireless Network Emulation (DDWNE)* [WvLW11, vL10] enables legacy wireless software to be tested in fully controlled and isolated environment. The software communicates over a wireless channel, which is modeled by a discrete event-based network simulator. The simulator models the wireless channel, the MAC layer and contextual factors such as node mobility. As we discuss later in more detail, DDWNE forms an implementation of so-called hybrid system representations as introduced earlier in Section 3.2.6.

Structure of this Chapter

We first discuss the design and the implementation for both SliceTime (cf. Sec. 4.1) and DDWNE (cf. Sec. 4.2). In a second step we apply these emulation frameworks to three use case scenarios to demonstrate their suitability for real-world performance studies. We conclude this chapter with an elaborate discussion of related work and compare our work with these state-of-the-art approaches.

4.1 SliceTime: A Platform for Scalable and Accurate Network Emulation

The concept of *network emulation* (cf. Sec. 2.1.3.2) brings together the flexibility of discrete event-based network simulations with the precision of evaluations using real-world testbeds. The core idea of this concept is to connect an event-based simulation modeling an arbitrary computer network to a real-world software prototype. Traffic from the prototype is fed to the simulation and vice versa. This way, the prototype can be evaluated in any network that can be modeled by the simulator.

One fundamental issue of network emulation is the inequality of the time representations used by event-based simulations and software prototypes (cf. Section 2.1.5). Existing implementations of network emulation pin the execution of simulation events to the corresponding wall-clock time. Unfortunately, this approach is only useful if the simulation can be executed in real time. Otherwise, a simulation without sufficient computational resources will lag behind and thus be unable to deliver packets on time.

Speeding up the simulation to make it real-time capable is the first obvious option and the traditionally applied method to mitigate this problem. However, this approach is often very costly in terms of hardware requirements or even infeasible for models of very high computational complexity.

With SliceTime [WSvL^{+}11] we follow an orthogonal approach and aim at reducing the cost of precise network emulation by designing a system with *fixed hardware demands* but with *variable execution time* (real-time or slower).

4.1.1 Conceptual Design

In accordance to the Synchronized Network Emulation (SNE) concept (cf. Sec. 3.4.1) SliceTime incorporates three main components:

1. **Virtual Machines (VMs)** form the first type of system representations in a Synchronized Hybrid Evaluation (SHE) setup. Their task is to encapsulate the software prototype to be integrated with the network simulation (cf. Sec. 3.2.2). We consider a prototype to be an instance of any operating system (OS) that carries arbitrary network protocol implementations or applications. The virtualization of OS instances hosting software prototypes disassociates their execution from the system hardware and hence allows for obtaining full control over their run-time behavior.

2. The key task of the **network simulation** (cf. Sec. 3.2.3) is to model the network that connects the virtual machines. For this purpose it implements the communication protocols that are used by the VM prototypes. This enables the network simulation to act as system representation by providing a set of simulated hosts that interact with the VMs.

3. The **synchronization component** centrally coordinates a SliceTime setup. Its task is to manage the synchronous execution of the network simulation

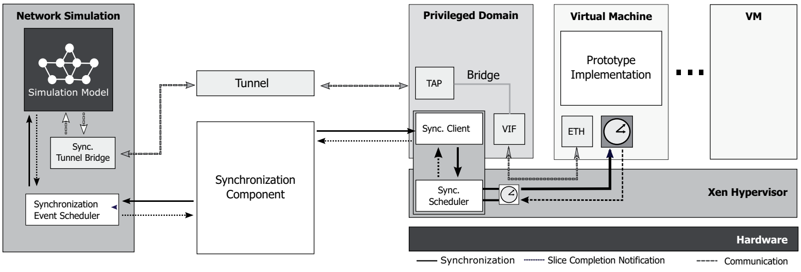


Figure 4.1 SliceTime consists of a central synchronization unit, at least one network simulation based on ns-3 and one or more Xen hypervisor systems serving as the VM infrastructure.

and the attached virtual machines. It implements the barrier synchronization algorithm (see Section 3.1.3.2) and delivers the time slices to the VMs using the according synchronization protocol.

4.1.2 Implementation of SliceTime

In the following we discuss our implementation of the individual SliceTime components and their interaction (cf. Fig. 4.1). The synchronizer delivers the synchronization information over the *timing control interface* to both system representations, the VMs and the network simulation. A tunnel that carries Ethernet frames from the VMs to the simulation and vice versa serves as our Packet Exchange Interface (PEI). The VM implementation is based on the Xen hypervisor and executes multiple instances of guest domains which host an operating system and a prototype implementation. Our implementation uses the ns-3 network simulator to model the network to which the VMs are connected. For this purpose we extend the existing emulation framework of ns-3 for synchronized network emulation.

4.1.2.1 Synchronization Component

The synchronization component (also referred to as *synchronizer* in the following) is implemented as a user-space application. Its main purpose is to implement the timing control interface. The synchronization component assigns discrete slices of run-time to the simulation and to the virtual machines using the barrier synchronization algorithm.

In addition to the synchronization coordination, the synchronizer also manages the set of synchronized components. In particular, it allows peers to join and to leave the synchronization during run-time. This allows to run certain tasks (e.g., booting and configuring a virtual machine and the hosted software prototype) outside the synchronized setup.

4.1.2.2 Time Control Interface (TCI)

As we have earlier outlined in Section 3.2 it is vital to limit the delays and the overhead caused by synchronization signaling and message parsing in order to keep the overhead of the synchronization as low as possible.

For these reasons we implemented the synchronization protocol (cf. Sec. 3.1.4) based on UDP. The assignment of time slices to all synchronized peers is carried out using UDP broadcasts, while the remaining communication, such as signaling time slice completion, takes place using unicast datagrams. Moreover, the UDP packets have a fixed structure and only carry the synchronization information in binary form, thus keeping both the packet size and the parsing complexity at a low level.

4.1.2.3 Virtual Machines

We use the Xen hypervisor (cf. Sec. 2.2.2) and its scheduling mechanisms as the basis of our work. Xen is a virtual machine monitor for x86 CPUs. The hypervisor itself takes care of memory management and scheduling, while hardware access is delegated to a special privileged virtual machine (or *domain*, in Xen's parlance) running a modified Linux kernel. As the first domain that is started during booting, this privileged VM is often referred to as dom0.

SliceTime uses Xen Hardware-assisted Virtual Machine (HVM) domains for virtualizing operating systems and software prototypes. In contrast to para-virtualization, HVM domains do not require the kernel of the guest system to be modified for virtualization. This allows any x86 OS, also closed source operating systems such as Microsoft Windows, to be incorporated into a SliceTime setup.

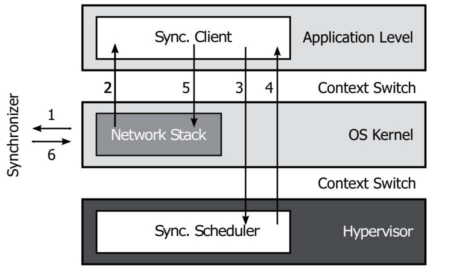
We now describe the main parts of our work in more detail: a) the Packet Exchange Interface (PEI) to couple virtual machines and the simulator, b) the synchronization client that interfaces with the synchronization component, and c) the changes necessary to transparently interrupt and restart the VM to align its execution speed to the run-time performance of the simulator.

Packet Exchange Interface

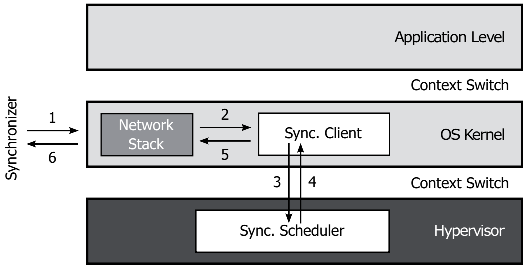
For the data communication between VMs and the simulation, it is important to note that every VM can have one or several virtual network interfaces. The packets on these interfaces can be intercepted at domain 0.

We bridge the virtual interface in the dom0 with a TAP device[1] and redirect all Ethernet packets from the VM to the computer running the simulation. Conversely, all Ethernet frames received from the simulation are fed back to the virtual machine in the same way. The transport between the VM and the simulation host is carried out by wrapping the packets into UDP datagrams.

[1]TAP devices are virtual network interfaces that are realized entirely in software using a kernel device driver. If the operating system kernel is instructed to create a TAP device, it also spawns a corresponding block device. All network packets sent to the TAP interface can be obtained from the block device. Conversely, any network packet written to the block device is delivered to TAP device, from which it can be obtained using a socket. Detailed Information about TAP devices is available at `http://vtun.sourceforge.net/tun/` (accessed 11/2012).



(a) Potential application-level synchronization (4 Context Switches)



(b) Actual SliceTime implementation (2 Context Switches)

Figure 4.2 Packet flow for a potential application-level synchronization vs. the kernel-based SliceTime implementation. Moving the synchronization client to the kernel level strongly reduces the amount of context switches, as the system does not need to switch between the application and the kernel context.

The Xen Synchronization Client

To keep the VM in sync with the communication, the synchronization component communicates with a synchronization client on the machine running Xen. Because of the potentially high number of synchronization messages (depending on the size of the chosen time slices), the performance of the synchronization clients is crucial to the overall performance of the system. For this reason, the client was implemented as a Linux kernel module. This is especially beneficial because Xen delegates hardware access to the privileged domain dom0. As illustrated in Figure 4.2, the implementation in kernel space of the privileged domain saves half of the otherwise necessary context switches for communication and our VM implementation. Since context switches (between user space, kernel space, and, in addition here, hypervisor context) are expensive operations, halving the number of them has a very noticeable impact on the overall performance.

The client communicates with the synchronization component via UDP datagrams as described in Section 4.1.2.2. It then instructs Xen's scheduler via a hypercall (the domain-hypervisor equivalent of a user-kernel system call) to start the synchronized domain for the amount of time specified by the synchronizer. The client also registers an interrupt handler to a virtual interrupt, that is, an interrupt that can be raised by the hypervisor. When the synchronized domain has finished its assigned time slice, the interrupt is raised, the client's handler is executed, and the kernel module sends a corresponding message to the synchronizer. This interrupt-based signaling ensures a prompt processing by the involved entities.

Extensions of the Xen Hypervisor

The other tasks necessary for our synchronization scheme are carried out within the Xen hypervisor. The most vital requirement for our VM implementation is the ability to precisely start and stop the VM's operation according to the time slices assigned by the synchronization component. However, since operating systems have ways to detect the passing of time via hardware support (real time clocks, hardware timers etc.), simply stopping and restarting the VM will not lead to the desired effect. It will still be aware of the passing of time while it was stopped, and therefore, operations that depend on time information (e.g., time-outs of TCP connections) will still occur at the wrong times. Therefore, to reach transparency, it is not only necessary to be able to start and stop VMs accurately, but also to provide a consistent and steady perception of time for the VM. Hence, all time sources of the VM must be controlled and adjusted in the hypervisor.

Precice Execution of Xen domains for the duration of a time slice

In order to start and stop VMs and in order to run them for a precise amount of time, we extended the sEDF (simple earliest deadline first) scheduler that is part of the Xen hypervisor. Schedulers in Xen schedule VMs in a similar fashion to an operating system's scheduler. In particular, the sEDF maintains periodical deadlines for each domain, and an amount of time the domain has to be executed up to that deadline. To manage the domains, it utilizes several queues. A run queue contains all domains that still need to run some time until their next deadline; once this constraint is fulfilled, a domain migrates to the wait queue until it reaches its deadline, at which point it rejoins the run queue with a new deadline and required execution time.

However, the synchronized domains have to be kept outside this periodical scheme, because they must only be scheduled when the synchronization component issues the instruction to do so. Therefore, we introduced another queue, the sync queue, which works as a replacement of the wait queue for synchronized domains. These domains stay on that queue until they are to be scheduled again, then migrate to the run queue, and back to the sync queue afterwards. This way, synchronized domains can be kept outside the normal scheduling on non-synchronized domains. Hence non-synchronized domains may coexist with synchronized domains on the same physical machine.

One issue that originally impaired precise timing in the low microsecond range was rooted in the original implementation of the Xen scheduling subsystem. The Xen scheduler assumes itself to run instantly, not consuming any time. Therefore, a time stamp at the beginning of the execution of the scheduling loop was taken. This was considered the point of time the next scheduled domain was started. Therefore, time spent in the scheduler was attributed to the domain chosen for execution. We changed this to take a time-stamp before the context switch to the domain.

This causes the time spent in the scheduler not to be attributed to any domain, therefore increasing accuracy. In addition, our modified sEDF scheduler records overall assigned run-time and adjusts itself to the small (generally sub-microsecond) inaccuracies that are inherent to Xen's timer management and lead to slightly early or late returns from the scheduled VM to the hypervisor.

Timekeeping

To reach the second goal, that is, masking the passing of time from VMs while being stopped, different changes had to be applied to the Xen hypervisor. In fact, one of the reasons we decided to use a virtualization approach for SliceTime was the specific characteristic of decoupling a virtualized operating system from the hardware it, under normal circumstances, directly interfaces with. This way, we can modify the information that the OS receives from the hardware time sources, and therefore reach our goal of masking the passing of time.

To facilitate this masking, we have to amend the two main sources of time keeping: time counters and interrupt timers. The modified scheduler records timestamps whenever a domain is scheduled and unscheduled. This allows us to keep track of the total amount of time the domain was not running since the start of the synchronization. This delta value is subtracted from the counter that domains use to measure the passing of time; in the case of Xen and HVM domains, this measurement is chiefly based on the time stamp counter (TSC), a CPU register whose value increases at regular intervals. Modern CPUs with hardware virtualization support allow the virtualization of the TSC, which allows us to change its value as realized by the VM by subtracting the delta value. This way, the TSC progresses in a linear fashion, even if the domain is unscheduled for extended amounts of time.

Timers, the second source of time keeping, must also appear to act as if the domain was running continuously. To facilitate this, the same scheduler timestamps are used to keep track of the time the domain was last unscheduled. Whenever a domain is unscheduled, all timers that belong to it are stopped; in particular, all timers that belong to the virtualized hardware timers such as the Real-time Clock (RTC) and the one provided by the Advanced Programmable Interrupt Controller (APIC). When the domain is rescheduled again, the time delta since the last unscheduling is added to the expiry time of all timers, after which they are reactivated. This way, timers expire at the correct point of virtual time, upholding the notion of linearly progressing time.

4.1.2.4 Network Simulation

SliceTime relies on ns-3 in contrast to our preliminary work [WSHW08, WSHW09] in which OMNeT++ was used as network simulator. The modular design of ns-3 facilitates the integration of the additional components as needed by SliceTime. Our timing control and the communication interfaces are implemented as completely separate components. These components are not intermingled with existing code.

There are some similarities between the SliceTime simulation components and the emulation features already provided by ns-3 (cf. Sec. 2.1.3.2). Both have to synchronize the event execution to an external time source. For the existing emulation implementation of ns-3 this is the wall-clock time. In the case of SliceTime the synchronizer acts as external time source. The so-called simulator implementations in ns-3 are responsible for scheduling, unqueuing and executing events. There is one which does this in a standard manner and another one for real-time simulations (i.e., synchronized to wall clock time). Which of these is used is determined by setting a global variable in the simulation setup.

We added a third simulator implementation that connects arbitrary ns-3 simulations to the Time Control Interface (TCI). The simulation registers at the synchronizer before its actual run begins. Similarly, the simulation deregisters itself at the synchronizer after all events have been executed. Upon the execution of an event, our implementation checks whether its associated simulation time is in the current time slice. If this is not the case, it sends a finish message to the synchronizer and waits for the barrier being shifted. The actual communication with the synchronizer is encapsulated in a helper class which holds a socket, provides methods to establish and tear down a connection and to exchange the synchronization messages. Another modification is the provision of a method which schedules an event in the current time slice. This is needed because the regular scheduling methods only provide the time of the last executed event, which can be wrong in case of network packets arriving from outside the simulation.

The ns-3 simulator already provides two mechanisms for data communication with external systems. Both can be used with real-time simulations and synchronized emulation. The so-called emulation net device works like any ns-3 network device, but instead of being attached to a simulated channel, it is attached to a real network device of the system running the simulation. In contrast to this the so-called tap bridge attaches to an existing ns-3 network device and uses a virtual tap device on the host system. With both mechanisms, packets received on the host system are scheduled in the simulation and packets received in the simulation are injected into the host system.

Besides supporting these existing two ways, we added a *synchronized tunnel bridge*. It implements the Packet Exchange Interface and connects the simulation to a remote endpoint. The endpoint is usually formed by a VM, however the tunnel protocol could also be used to interconnect different instances of ns-3. Again the actual communication is encapsulated in a helper class. This is not only to keep the bridge itself small, but also to reduce the number of sockets needed. In a scenario where multiple tunnel bridges are installed inside a simulation it is sufficient to have one instance of this helper class. Outgoing packets are sent through its socket to a destination specified by the bridge sending the packet. Incoming packets are dispatched by an identifier included in our tunnel protocol and then scheduled as an event in the corresponding bridge to which the sender of the packet is connected. Since incoming packets are not triggered by an event inside the simulation but can occur at any time, there is a separate thread running which uses a blocking receive call on the socket. This technique has the advantage to avoid polling and is also used by the emulation net device and the tap bridge.

4.1.3 Evaluation of SliceTime

We now examine the achievable accuracy of SliceTime. First, we look into the timing precision and the accuracy of the perceived throughput. Later on, we also measure the performance impact introduced by the synchronization process on the general run-time performance. We further investigate how it affects the perceived CPU performance on a VM. All experiments were carried out in a testbed of four Dell Optiplex 960 PCs, each equipped with a 3 GHz Intel Core2 Quad CPU and 8 GB of RAM, either executing our VM implementation based on Xen or ns-3 with our synchronization extensions. The PCs were interconnected using Gigabit Ethernet. Regarding the VMs, we used Linux 2.6.18-xen for the control domain as well as the guest domains.

Most importantly, SliceTime needs to produce valid results for any run-time behavior of both the simulation and the VMs attached. For this purpose, we investigate two performance metrics at different levels of synchronization accuracy. The round-trip time (RTT) between a simulation and a VM as well as the TCP throughput of two VMs which are communicating using TCP over a simulated network.

4.1.3.1 Timing Accuracy

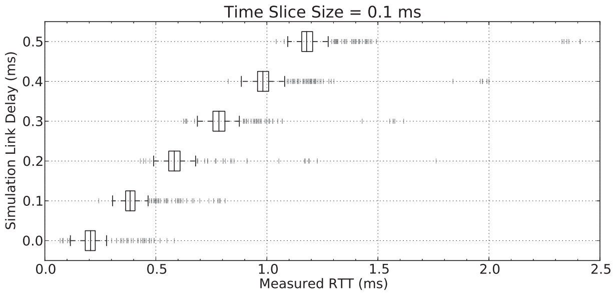
In our first emulation experiment, we captured 1500 ICMP Echo replies (Pings) between a VM and a simulated host for different simulated link delays and time slice sizes. In order to obtain a reference scenario, we additionally recorded 1500 Pings between two Linux PCs that were interconnected using Gigabit Ethernet. We used NetEm [Hem05] (cf. Sec. 4.4) to model the link delays between the two PCs in the reference scenario.

Integration of Timing Domains

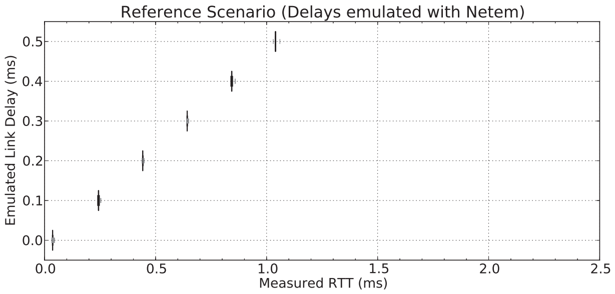
Figure 4.3 compares the RTT distributions obtained from the SliceTime setup for a fixed time slice size of 0.1 ms with the corresponding reference measurements. We visualize the RTT distributions using standard box plots. The boxes are bounded by the upper and lower quartile of the corresponding RTT distribution. A box represents the middle 50% of the RTT measurements and its width is given by the Interquartile Range (IQR). The whiskers visualize the lowest and the highest RTT measured within an interval of 1.5 IQR.

If no simulation delay is present, most RTTs fall into a small range around 0.2 ms. We term this the *base delay* and it comprises time for processing and packet propagation. At all other simulation delays, the median and the RTT distributions are correctly shifted by the sum of twice the simulated link delay.

In comparison with reference measurements (see Fig. 4.3(b)) two observations can be made. The first noteworthy fact is that the base delay of the SliceTime setup (approximately 0.2 ms) is higher than the one observed in the reference scenario (around 0.04 ms). We explain this disparity with the increased complexity of the packet flow in the SliceTime set-up. Here, a network packet has to traverse a VM's network stack, the network stack of the domain 0 hosting the VM, the tunnel interface and



(a) RTTs for different simulated link delays: the simulated delays are correctly perceived by the VM.



(b) Reference Scenario

Figure 4.3 We evaluated the timing accuracy of SliceTime by analyzing the RTT distributions of 1500 ICMP echo replies using different time slice sizes and simulation delays.

finally the simulated link. By contrast, the PCs in the reference setup were directly interconnected using Gigabit Ethernet, resulting in a smaller end-to-end delay.

Second, the RTTs obtained from the SliceTime setup show a larger variance and more outliers than the reference measurements. We attribute these deviations to the increased number of non-deterministic sources in the SliceTime set-up, for example caused by the tunnel implementation and the time needed to traverse the different network stacks in the packet flow.

While this evaluation shows that SliceTime does not perfectly reproduce the reference scenario, we however emphasize that the RTT disparities are considerably smaller than one millisecond. Delay artifacts of this size are mostly not significant for the main application scenarios of Slice Time, namely applications for wide area networks or Internet Services. The reason is that RTTs typically range between a few milliseconds to tens of milliseconds in such scenarios, resulting in a negligible impact of such comparatively small delays.

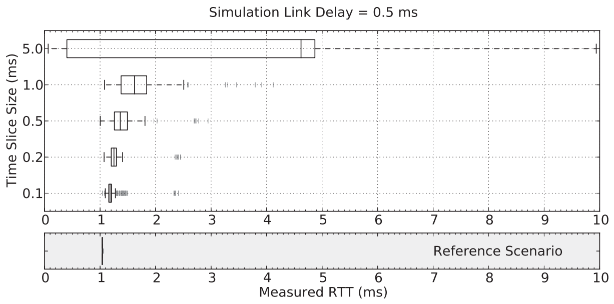


Figure 4.4 RTT distributions for different time slice sizes: smaller time slices lead to less variance in the measured RTTs and improve the approximation in comparison with the reference measurement.

Impact of Time Slice Size

Figure 4.4 displays the relation of the chosen time slice size and the resulting RTT distributions for a fixed simulated link delay of 0.5 ms and a variable time slice size. The figure also depicts the corresponding reference RTT distribution. The results obtained from the SliceTime setup well approximate the reference values for increasing synchronization accuracies and the variance in the measurements decreases. We observe the same behavior for all measured time slice sizes (cf. Appendix A.1).

First, this result clearly demonstrates that a higher synchronization accuracy directly impacts the accuracy of the measurements themselves. Second, we see that it is important to choose the time slice size considerably smaller than the simulated link delay. Hence, the correct choice of the adequate slice size is a crucial parameter of SliceTime. For the simulation of many wide area network scenarios (e.g., Internet services) time slices in the range between 0.1 ms and 2 ms are sufficient.

4.1.3.2 Throughput Accuracy

We now evaluate the accuracy of our implementation regarding the network throughput perceived by the VMs. For this purpose we use a small ns-3 simulation, consisting of one IP node to which two gateway nodes are attached using CSMA/CD channels. To each of those two gateway nodes, one VM is connected. Using the netperf [JCS] `TCP_STREAM` benchmark, we measured the throughput between both VMs. Figure 4.5 shows the results for different simulated channel bandwidths and time slice sizes. The data points are averages over 10 netperf runs, with every run lasting 20 seconds.

Most notably, the synchronization is transparent to the VMs in terms of perceived TCP bandwidth, as the time slice size has practically no influence on the measured TCP throughput. In addition, the throughput measured on the VMs very well reflects the simulated channel bandwidth. On average, the measured throughput on the VMs is 5.4% lower than the simulated link capacity.

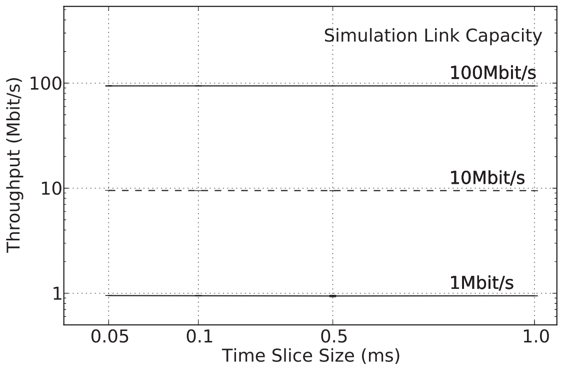


Figure 4.5 Network throughput at different time slice sizes: the synchronization does not affect the throughput perceived by the VMs. The measured throughput on the VMs corresponds to the simulated link capacity.

4.1.3.3 Synchronization Overhead

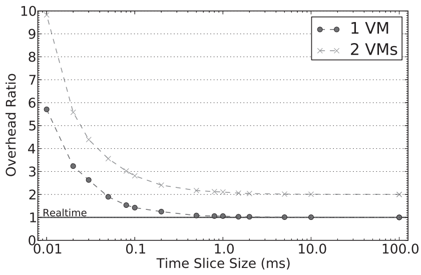
Because synchronized VMs are not operating in real-time, we now analyze the overhead in terms of actual run-time penalties introduced by the synchronization. We measured the real-time duration for 120 seconds of logical time issued to the VMs by the synchronizer. All VMs were executed on the same physical machine. We calculated the Overhead Ratio (OR) by dividing the consumed real-time by the logical run-time.

Figure 4.6(a) displays the OR of one and two VMs for varying time slice sizes. Up to a size of 0.5 ms, the synchronization overhead remains below 10%, which is still close to real-time behavior. For smaller slice sizes, VMs need to be suspended and unpaused more frequently, and the messaging overhead increases. This leads to a higher OR.

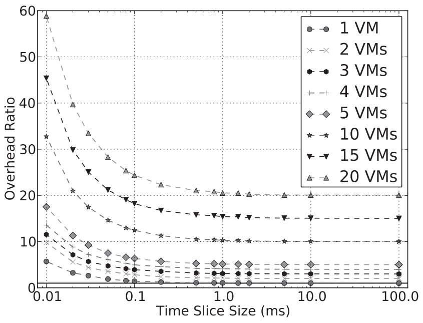
The parallel execution of several VMs per physical machine is not the main objective of our work. Nevertheless, our implementation facilitates such configurations. Figure 4.6(b) shows the OR also for a higher number of VMs. The increase of the OR is linear in the number of VMs for all time slice sizes. This is a direct consequence of our scheduling policy. Even if a system is equipped with multiple processors or cores, VMs are always executed in a pure sequential order. We made this conservative decision, as an interleaved execution of VMs may lead to unwanted artifacts due to a shared use of system resources, for example network bandwidth or hard disk I/O.

4.1.3.4 CPU Performance Transparency

One of the major reasons for the run-time efficiency of SliceTime is given by the fact that the VMs, once scheduled, are executed natively on the host machine instead of a full simulation of system hardware. While we have previously shown that the



(a) 1 to 2 synchronized virtual machines



(b) 1 to 20 synchronized virtual machines

Figure 4.6 Overhead introduced at the VM at different synchronization levels: we observe less than 10% of run-time overhead for time slices greater than 0.5 ms. The overhead is linear in the number of VMs on one physical machine.

integration with the network simulation is accurate in terms of timing and network bandwidth, we now investigate the transparency of our VM implementation regarding the perceived CPU performance within a VM. In an ideal case, the perceived CPU performance of a VM would be invariant at different levels of synchronization accuracy.

In order to quantify the CPU performance of a VM, we executed CoreMark [CRM] inside the synchronized VM. CoreMark is a synthetic benchmark for CPUs and microprocessors recently made available by the Embedded Microprocessor Benchmark Consortium. It performs different standard operations, such as Cyclic Redundancy Check (CRC) calculations and matrix manipulations, and outputs a single CPU performance score. Figure 4.7 shows the CoreMark score for different time slice sizes. Most notably, the CPU performance is rather stable above time slices of 0.2 ms. For a time slice size of 0.1 ms, the impact of the synchronization still is less than 5%. However, for smaller values, the CPU performance decreases rapidly. At the highest measured accuracy level (0.01 ms), the CoreMark score drops to about 73% of the score of an unsynchronized VM on the same hardware.

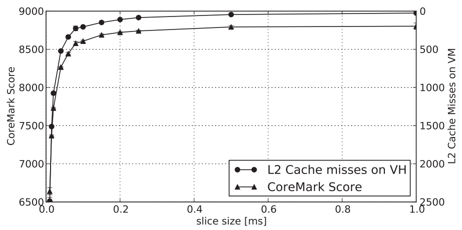


Figure 4.7 CoreMark CPU Benchmark score at different time slice sizes: For smaller time slices, the CPU performance of a VM decreases due to an increased amount of L2 cache misses. Please note the inverted y-axis on the right.

We further investigated this effect using OProfile [opr] and its XenoProf [MST$^+$05] extension. By concurrently executing OProfile in the control domain while CoreMark was running inside the VM, we were able to trace internal CPU events caused by the VM. This way, we identified an increased amount of L2 cache misses to cause the observed performance degradation. As shown in Figure 4.7, the number of L2 cache misses is negatively correlated to the measured CoreMark scores. For smaller time slices, the CPU needs to be switched more frequently between the execution of the VM and the control domain, thus decreasing the efficiency of L2 caching. Although this is a conceptual issue, we argue that the effect is negligible for time slices down to 0.1 ms. This means that for the vast amount of application scenarios that will use larger slices, this minimal performance reduction will have no negative influence on the produced results.

4.1.4 Interim Summary

Up to this point we have discussed the implementation of SliceTime and we have provided an in-depth analysis of its performance properties. SliceTime synchronizes the run-time execution of the ns-3 network simulator and Xen based virtual machines with an accuracy down to 10 μs. For many purposes larger time slices between 0.1 ms and 0.5 ms are absolutely sufficient, resulting in a moderate synchronization overhead. Later in this chapter, we will resort back to SliceTime for setting up two emulation use cases that demonstrate the applicability of SliceTime to complex network emulation scenarios.

4.2 Device Driver-enabled Wireless Network Emulation

Before we demonstrate how SliceTime can be applied for conducting network emulation scenarios with non-realtime simulations, we first introduce a novel methodology for the emulation of wireless networks. This concept that we term *Device Driver-enabled Wireless Network Emulation (DDWNE)* [WvLW11] uses both a network simulator and a real-world software system to cooperatively model one or a set of

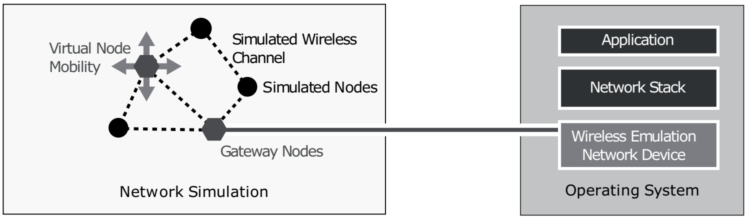


Figure 4.8 Conceptual overview of our wireless emulation approach: an instance of an entire operating system is integrated with the network simulation using a custom wireless network device driver. Traffic between the simulation and the device driver is exchanged at so-called gateway nodes. The network simulation models the wireless channel, other fully simulated nodes as well as potentially virtual node movement.

wireless communication systems. Hence, DDWNE puts the idea of hybrid system representations as earlier discussed in Section 3.2.6 into effect.

DDWNE integrates the wireless software with a network simulation by providing it with a virtual wireless network interface that behaves like a real wireless networking card, but instead handles the transmission and reception of data using a network simulation. This way, any real-world wireless networking software as well as routing and transport protocol implementations can be investigated inside a fully simulation-controlled environment. The simulation models the MAC and PHY layers, the wireless channel, potential node movement as well as other fully simulated nodes.

In this section, we also present a discussion of a DDWNE emulation framework for IEEE 802.11 (cf. Sec. 4.2.2). Our DDWNE implementation for 802.11 allows any networking software for Linux making use of 802.11 wireless networks to be evaluated within ns-3 simulation scenarios of any kind. The corresponding evaluation in Section 4.2.3 shows that our framework accurately integrates the ns-3 Wi-Fi models with real-world networking software for Linux at the MAC layer, both according to latency and network bandwidth.

4.2.1 DDWNE Architecture

Figure 4.8 shows a high-level view of our architecture for the emulation of wireless networks. Corresponding to the underlying concept of network emulation (cf. Chapter 2), our framework consists of two main components: The first is a host operating system (OS) that executes the wireless software to be investigated. We consider the wireless software to be any program or service that makes use of a wireless networking device. The second component is a discrete event-based network simulation, which models a virtual wireless network containing both simulated nodes and so-called gateway nodes. The gateway nodes connect the simulation domain with the real-world software prototypes.

4.2.1.1 Real-World Software for Wireless Networks

Our architecture integrates entire instances of operating systems executing the wireless software under investigation into the emulation setup. The most important cor-

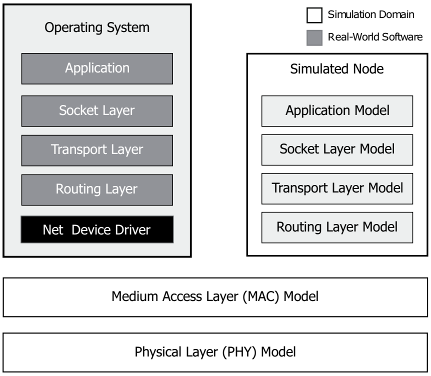


Figure 4.9 Stack organization in a device-driver enabled emulation scenario. We integrate the OS with the network simulation at the MAC layer. The network simulation models the MAC and the PHY layers; for fully simulated nodes it provides the entire protocol stack.

nerstone in our architecture is a special device driver providing the wireless software with MAC-layer connectivity to the simulated wireless network. In the following we explain why this design decisions was made.

First, the major design requirement for our emulation architecture is to enable the incorporation of arbitrary wireless software into an emulation scenario. Any software making use of wireless communication, for instance ad-hoc routing protocol daemons, VoIP applications or legacy operating system applications, should be able to be included into the wireless emulation setup. For this reason we need to provide the wireless software with its genuine environment that is of course the operating system context for which the software was developed. Second, we generally assume that modifying the software for the inclusion into an emulation framework is not possible. Hence, we generally do not require source code to be available. This is the case for many commercial applications.

Interfacing the wireless software with the simulation is generally possible at different levels of the protocol stack. One option would be to provide an alternative socket layer for the wireless software to link against like in EmuSocket [AV06]. However, this constrains one to rely on TCP/IP for all means of wireless communication, and thus, investigating custom routing or transport protocol implementations becomes impossible. Similar problems hold for the interception at the Internet Protocol (IP) layer (e.g. [Sei08]), which require the wireless software to use IP for communication.

Figure 4.9 displays how we address this problem. The common language of all nodes in the simulation is the protocol used for wireless communication at the MAC layer. All MAC and PHY layer behavior is therefore modeled by the network simulator.

Our architecture employs a custom driver behaving like a real-world wireless networking device to embed arbitrary software into a wireless emulation scenario. Besides sending and receiving data from other nodes, it also implements device specific actions such as scanning for access points, depending on the emulated wireless communication technology. Consequently, any protocol stack or application that is capable of accessing network interfaces can be transparently used for wireless network emulation. This neither requires source code changes nor any other additional effort such as recompilations or relinking the code.

4.2.1.2 Network Simulation

The overall task of the network simulation is to model a wireless network, consisting of so-called *gateway nodes* and optional fully simulated nodes. The gateway nodes are stand-ins for the real-world software inside the simulation's virtual network topology.

The core functionality of the *gateway nodes* is to relay traffic originating at real-world wireless software over the simulated wireless channel and vice versa. To enable the communication between gateway nodes and other nodes in the simulation, the gateway nodes only implement the physical and medium access control layers of the simulation.

Besides incurring the pure communication actions for the software prototypes, the gateway nodes also implement other functionalities usually carried out by wireless communication hardware. One example is reading Received Signal Strength Indicator (RSSI) values. In the real world, RSSI values indicate the signal strength associated with received packets. Typically such values are exported to the operating system and the software using an interface at the device driver. For the emulation case the RSSI values are the outcome of simulation models. In order to enable real-world software to access such "simulated" environment parameters, the driver bridges important parameters and properties of the wireless simulation model with the API of the host operating system. Similarly, the gateway nodes map other commands to corresponding actions in the simulation, for instance a request to scan for Access Points (APs). Hence our architecture not only emulates the wireless communication characteristics but also the operating system interface of respective typical networking hardware.

It is also noteworthy that a major reason to rely on an event-based network simulator as an emulation engine is its capability of modeling additional environmental behavior in a deterministic way. Most notably this concept allows the simulator to implement *virtual mobility* support. Network emulation with virtual mobility allows one to investigate how real-software prototypes and their performance are affected by influences due to deterministic node movement. Similarly, the network simulation may also implement *simulated nodes*, for instance access points in the context of 802.11 networks or simulated hosts forming an arbitrary background network. This enables emulation scenarios to scale up to a larger degree in terms of node count or emulations containing network components that are not available otherwise.

4.2.1.3 Message Exchange

A crucial part of every device driver-enabled wireless emulation framework is the message exchange between the gateway nodes and the device drivers that are associated with them. We assume that the network simulation and the wireless emulation device driver are typically executed on separate installations of an operating system.

For example, if two physical machines are used, one might host the network simulation while the other runs the emulation device driver and the wireless software. The use of virtual machines makes it is also possible to run the network simulation as well as multiple OS installations with the driver on one physical computer.

The first important requirement regarding the communication between both components is *low latency*. As the execution of the operating system hosting the driver and the network simulation is usually not tightly integrated, the message exchange scheme directly influences the communication delay perceived by the wireless software. Hence, a low messaging latency is vital to avoid potentially significant performance loss regarding round trip times and end-to-end throughput between a gateway node and another node in the simulation. A second challenge is the adequate reproduction of *buffering* behavior. While network simulations mostly assume unlimited transmission queues, the capacity of transmission buffers found in real-world network devices is strictly limited. In order to obtain realistic performance measurements, for instance regarding the throughput measured on the emulated network device, we also need to emulate buffer capacity.

According to communication between the device driver and the gateway node, the following forms of message exchange between driver and simulation can be differentiated in a device driver-enabled wireless emulation tool-chain:

- **Driver (un-)registration**
  We define the wireless simulation to be the "master" component at which the device drivers may register and unregister at any point in time. This implies that the network simulation that models the environment for the gateway nodes has always to be instantiated prior to the wireless device drivers.

- **Exchange of data frames**
  The main type of message exchange is the transmission of data frames from the gateway node to the emulation device driver. Similarly, data packets delivered to the driver need to be transferred to the gateway node where they are injected into the simulated wireless network.

- **Status update notifications**
  The wireless driver needs to provide statistics and status information such as RSSI values. As this information is only available at the wireless communication stack of the gateway node, a data exchange mechanism for status information needs to be established.

- **Network hardware configuration and commands**
  As the wireless software might invoke certain commands typically carried out by the wireless networking hardware, the messaging scheme needs to forward them to the gateway node.

As we later discuss in Section 4.2.2.3, an actual implementation of the data exchange between driver and real-world simulation might also require the proactive transfer of information, e.g. to enable the timely access to status information.

4.2.1.4 Scalability

Like with any network emulation framework an important aspect is the degree of achievable scalability. More specifically, the question is how many emulated hosts can be modeled by the simulation (*simulation scalability*) and how many real systems can be attached to it (*emulation scalability*).

The *emulation scalability* is heavily dependent on the actual implementation of the gateway nodes and the message exchange with the device drivers. One important factor is the accumulated traffic between all gateway nodes and the corresponding driver instances. It has to be kept within the bounds of available communication resources of the computer executing the simulation. In addition, each gateway node requires state information that is in the same magnitude as the one required by a simulated host. From our experience with our 802.11 implementation of DDWNE, however, we have learned that emulation scalability is not really an issue. Early experiments had shown that our framework is easily capable of handling a couple of dozens of driver instances at the same time.

Instead, for classic network emulation frameworks the *simulation scalability* would be the bigger issue. Given the computational complexity of wireless channel models and the detailed simulation of the MAC layer, the resource demands of wireless simulations grow fast with the number of simulated hosts and gateway nodes. Hence, many simulations of wireless networks are not real-time capable. This would cause problems for traditional emulation frameworks. The concept of synchronized network emulation and our according implementation SliceTime (cf. Sec. 4.1) exactly solves this problem. DDWNE can be considered as a hybrid system representation type (see Section 3.2.6) that can be seemlessly used together with SliceTime. Hence, DDWNE scenarios are able to rely on wireless simulations that do not operate in real-time.

4.2.2 Implementation of DDWNE for 802.11

We now describe the implementation of a device driver-enabled wireless emulation framework for 802.11 (Wi-Fi) networks. Although we focus on 802.11 in the following, the concept of device driver-enabled wireless network emulation proposed in Section 4.2.1 may also applied to different communication technologies like Bluetooth or Zigbee.

Our 802.11 wireless emulation framework encompasses the following components:

- We rely on ns-3 for the simulation of the 802.11 network. We extended the 802.11 model of ns-3 with an implementation of an **802.11 gateway node** to enable the 802.11 model to be used for network emulation. In order to support further typical features of 802.11 such as scanning for access points (APs), only minor changes had to be applied to the model itself.

- We implemented a custom **Wi-Fi device driver** for Linux as a loadable kernel module. It provides all functionality of a common wireless network device and supports the Linux Wireless Extensions. Thus any protocol implementation and Linux application can seamlessly be incorporated into a Wireless emulation setup.

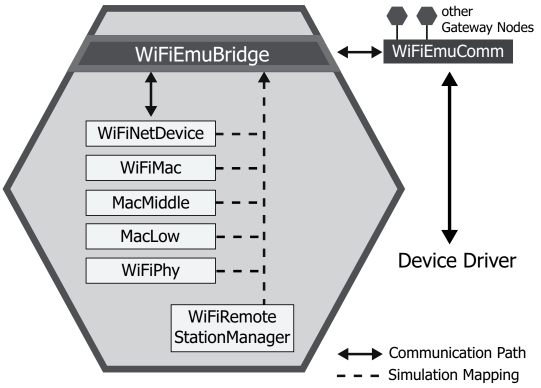


Figure 4.10 Structure of our ns-3 gateway node implementation: The WifiEmuBridge integrates the data transfer with the ns-3 Wi-Fi stack and maps simulation properties and actions to the model. The actual data communication with the device driver is carried out by one singleton object.

- We designed a lightweight and flexible **message exchange** protocol to integrate the functionalities of the driver and the gateway nodes.

In the following we discuss important aspects regarding the implementation of the individual DDWNE 802.11 components.

4.2.2.1 The 802.11 Model of ns-3

The 802.11 network model used in ns-3 originates from the Wi-Fi model of an earlier discrete event-based simulation tool named *Yet Another Network Simulator (YANS)* [LH06]. Recent versions of ns-3 include detailed MAC layer and PHY layer simulation models for 802.11a and 802.11b networks. ns-3 supports the simulation of infrastructure as well as the investigation of ad hoc scenarios. The ad hoc network implementation, however, is not complete, as it currently only sends the data frames themselves and does not contain management operations. Still, it can be used for the simulation of ad hoc networks, even though the behavior is not fully compliant with real 802.11 networks. The complete Wi-Fi model consists of several classes which form a stack of sub-layers.

The 802.11 Gateway Node

The 802.11 gateway node bridges the logic of the ns-3 Wi-Fi model with the Wi-Fi device driver. Figure 4.10 illustrates the module composition that forms a gateway node in our 802.11 framework. Due to the clean design of ns-3, we were able to implement the simulation part of our emulation framework by just adding two essential classes: The `WifiEmuBridge` module and the `WifiEmuComm` adapter. The latter centrally manages the data exchange between multiple gateway nodes and associated Wi-Fi device drivers.

802.11 property	Description
RSSI	Received Signal Strength Indicator
Operation Mode	802.11 Infrastructure, Monitor or Ad Hoc mode
PHY standard	802.11 standard in use: a,b
Data Rate	The current data rate of the interface, eg. 54Mbit
SSID	The Service Set Identifier (SSID) of the access point the gateway node is currently associated to
BSSID	The Basic Service Set Identification (BSSID) of the network the gateway node currently belongs to
Channel	The number of the 802.11 channel currently used

Table 4.1 Different 802.11 status values and statistics are supported by our Wi-Fi emulation framework. They are either accessible to the Wi-Fi software via RadioTap headers or the common API defined by the Linux Wireless Extensions.

The `WifiEmuBridge` module is the central cornerstone of our gateway node implementation. In order to enable the wireless software to send data over the simulated 802.11 network, it receives raw data frames via the `WifiEmuComm` adapter that originate from the Wi-Fi device driver (cf. Figure 4.9). In a similar fashion 802.11 data frames received on the simulated Wi-Fi channel are relayed to the driver.

During the instantiation of the gateway node, the `WifiEmuBridge` module uses callbacks to register at the different sublayers of the ns-3 802.11 model. This is required to gather status information and statistics from lower layers during an emulation run. Table 4.1 lists all the 802.11 status values that are supported by our Wi-Fi emulation framework. All of these are made accessible to the Wifi software via the Wi-Fi emulation device driver, either by implementing respective Linux Wireless Extension calls or using RadioTap[2] headers.

The `WifiEmuComm` adapter centrally manages the data exchange between multiple gateway nodes and associated Wi-Fi device drivers. It is implemented as a singleton object, which is instantiated only once for the entire emulation scenario. If data frames are received from a Wi-Fi emulation device driver, `WifiEmuComm` dispatches them to the corresponding gateway node using an identifier sent along with the data frame. This design decision was made mainly for the reason of decreasing the complexity of the gateway node implementation. A second helper class encapsulates the low-level communication. This enables alternative implementations of the message exchange mechanism between `WifiEmuComm` and the device driver.

We further emphasize that our Wi-Fi emulation extensions only require minor changes to the ns-3 802.11 model, such as the addition of a few callbacks to access 802.11 status values. One important extension is scanning support; it is required to get common Wi-Fi software such as `iwlist` working. In order to enable scanning in a Wi-Fi emulation scenario, we extended an early prototype by Gustavo Carneiro [Car09] and incorporated it into our implementation.

[2]Radiotap headers are optional 802.11 frame headers that can be used to exchange driver information with userspace applications. A comprehensive overview of this technology can be found at `http://www.radiotap.org/` (accessed 11/2012).

4.2.2.2 Wi-Fi Emulation Device Driver

The network driver which is part of our 802.11 emulation framework is implemented for the Linux operating system. The open nature of Linux makes it suitable to form the basis of a device-driver enabled wireless emulation tool-chain, as all parts of the system can be easily inspected and modified. However, from a conceptual point of view additional device drivers could also be implemented in an analogous way for any operating system providing general support for network communication.

Since the main goal of this driver is to represent the simulated wireless network card, it has to interact with the Linux network stack exactly like a driver of a regular wireless network interface. The Linux networking interfaces work as follows [WPR+04]: During initialization or when hardware is found, a network card driver registers itself at the networking subsystem, providing a list of function pointers. These functions are called later during the execution by the networking subsystem to pass data which has to be sent, to retrieve statistics or to start and stop the network card. In turn, the network driver can call functions of the networking subsystem to start and stop its sending queue or to transfer received packets.

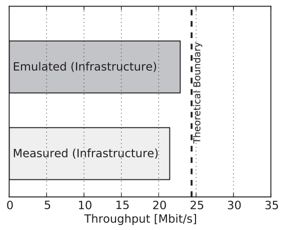
While this general network card interface already allows the driver to exchange network packets with the networking subsystem, it does not support any wireless network card specific features. For this purpose, the *wireless extensions* [Wirb] (wext) are added on top of this interface. Through a number of additional pointers to functions provided by the network driver, the Linux kernel can set or get additional parameters or retrieve statistics of the wireless network card.

4.2.2.3 Message Exchange

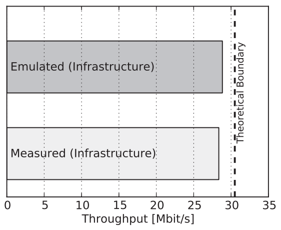
In order to integrate the 802.11 model and our gateway node implementation of ns-3 with the emulation Wi-Fi device driver, we have implemented a lightweight messaging interface that supports all communication primitives discussed in Section 4.2.1.3. In order to fulfill the low latency requirement, we need to keep the delays caused by message processing as low as possible. For this reason we developed a straightforward lightweight UDP protocol that embeds both data frames as well as status information in binary form. This enables a rather efficient conversion of data structures using static typecasts in contrast to protocols that would introduce a far higher messaging complexity, for example protocols based on XML messages.

To provide efficient access to statistics and status information such as RSSI and BSSID, the gateway nodes push changes of this information to the device driver using our messaging interface. This decision was made for performance reasons, as the wireless interface of Linux splits access to 802.11 status messages into a series of system calls. By pushing all status information to the driver, all such requests can be answered locally without further interactions with the gateway nodes. In contrast to that, a pure polling approach would require a much higher amount of interaction between the emulation Wi-Fi driver and the network simulation and thus would introduce a higher messaging overhead.

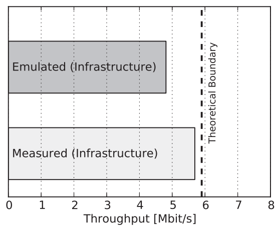
In addition, we also equipped the driver with a *virtual transmission buffer* to emulate the limited capacity of sending queues found in real 802.11 network cards. If this



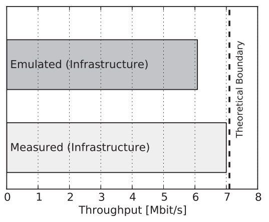
(a) TCP, 802.11a



(b) UDP, 802.11a



(c) TCP, 802.11b



(d) UDP, 802.11b

Figure 4.11 Throughput of emulated 802.11 compared with real-world measurements. Our 802.11 emulation framework reaches realistic throughput performance for both UDP and TCP.

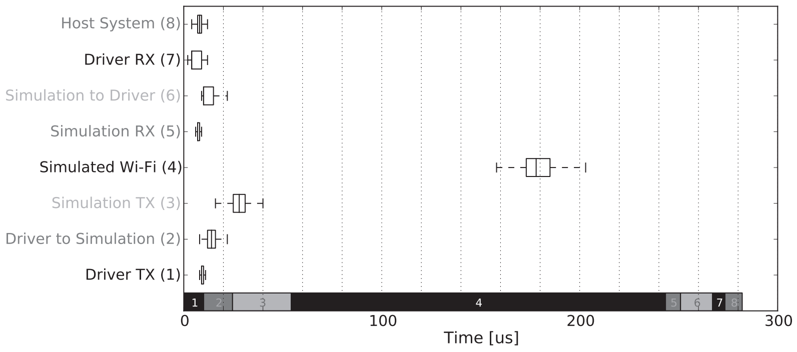
feature is enabled, the device driver counts the number of bytes transferred to the gateway node. After the gateway node has sent the data on the simulated Wi-Fi channel, it instructs the driver to subtract the number of transmitted bytes from the counter again. Hence, the counter amounts to the number of bytes that are currently waiting to be sent. If this counter exceeds a distinct threshold, the virtual buffer is full and the device driver blocks the network stack from sending more frames.

4.2.3 Evaluation

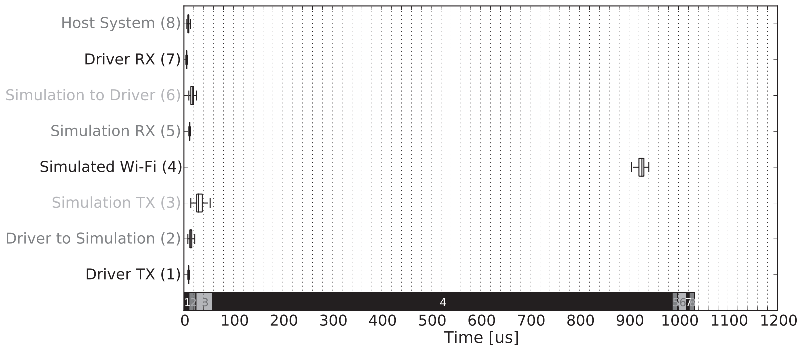
We now evaluate the accuracy of our driver-enabled 802.11 emulation framework regarding throughput and end-to-end latency. Later we also investigate the timing behavior of our driver-based integration of the ns-3 Wi-Fi model with Linux more precisely.

4.2.3.1 802.11 Throughput

We first analyze the throughput between two hosts in an 802.11 emulation scenario. Both nodes communicate with each other over a simulated ns-3 Wi-Fi channel. For this experiment, we used a Xen [BDF+03] hypervisor with two virtual machines

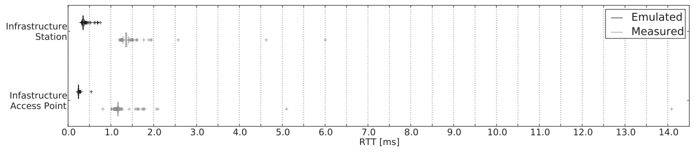


(a) 802.11a

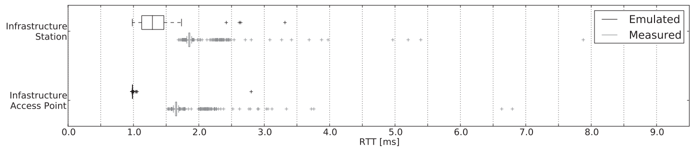


(b) 802.11b

Figure 4.12 Timing analysis of a DDWNE emulation set-up: The largest amount of the RTTs between the host system and the simulation is caused by the ns-3 Wi-Fi model and not by the interaction between device driver and gateway node. The bar below the boxplot shows the accumulated average delay of the individual communication actions.



(a) 802.11a



(b) 802.11b

Figure 4.13 Comparison of round trip times of 802.11 measured in a real experiment and in an emulated network. (Whisker length: 1.5 IQR).

(VM) hosting Linux and the emulation driver as well as one VM that executed the Wi-Fi emulation framework based on ns-3. Figure 4.11 compares the throughput for both emulated 802.11a and 802.11b with real-world 802.11 measurements in infrastructure mode. All real-world measurements were obtained on a plain meadow (low interference) using two MacBooks running Linux and a Linksys WRT610N wireless router that serves as access point. The `TCP_STREAM` and `UDP_STREAM` tests of netperf [JCS] were used to measure the throughput for both the emulated as well-as the real-world 802.11 networks. The upper bounds are taken from [Ath03].

For both 802.11 sub-standards and investigated transport protocols, our Wi-Fi emulation framework produces realistic throughput measurements in the right magnitude. Regarding 802.11a the throughput obtained using the emulation is slightly higher than the one measured in the real-world. It is the other way around for 802.11b: the measurements taken in the real-world outperform those of the emulated scenario. Such discrepancies according to the 802.11 throughput are well-known and not a specific property of our 802.11 emulation framework. For example, the measurements presented in [GBST08, Ath03] show that the achieved throughput in 802.11 networks may be strongly influenced by the Wi-Fi hardware used.

4.2.3.2 802.11 Round Trip Times

Analogous to the throughput measurements we now compare the round trip time (RTT) between two hosts in an emulated and a real-world scenario. Figure 4.13(a) and 4.13(b) display the RTT distributions for 802.11a and 802.11b. For each wireless standard, we depict two RTT distributions. The distribution *Infrastructure Station* shows the RTTs between one notebook and the other end host, while *Infrastructure Access Point* displays the RTT distribution between one notebook and the access point.

Most notably the round trip times taken using our emulation framework are constantly lower than the RTTs measured in reality. We regard this difference as natural disparity caused by the abstractions of the ns-3 802.11 simulation model from the real-world behavior of Wi-Fi.

The level of abstraction of the ns-3 Wi-Fi model for good reason increases at lower layers, as it is the case for most wireless simulation models. For instance, the Yans channel model [LH06] only approximates the typical delays of the 802.11 channel access. Other sources of delay, for example those imposed by the design of Wi-Fi hardware are also not reflected by the 802.11 model, as their implications on performance evaluations of network protocols and applications are mostly irrelevant. The potential delay differences between a Wi-Fi emulation setup and a corresponding real-world scenario can be easily compensated, e.g. by introducing additional static or random delays in the 802.11 channel model.

4.2.3.3 Timing Analysis

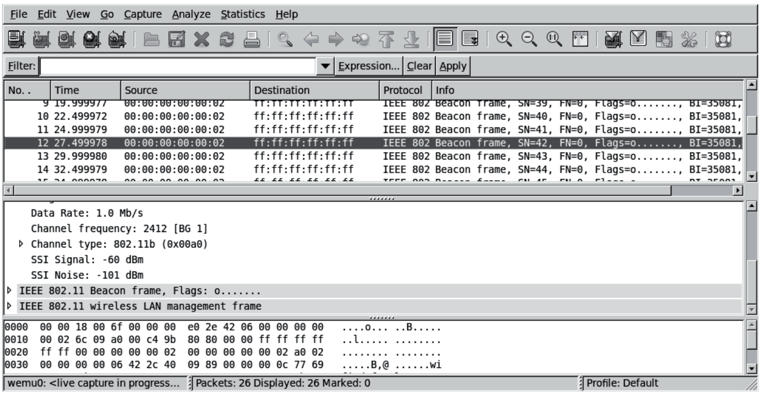
We now investigate the timing behavior of our 802.11 emulation framework in more detail. Figure 4.12 breaks down the round trip times between a simulated host and a Linux VM into the individual communication actions between the Linux device driver and the ns-3 802.11 models. The boxplots visualize the absolute delay distributions of the individual communication actions. The bar at the bottom of the plot shows how the average delays of the individual communication actions accumulate the total round trip time.

By far the largest fraction of the RTT is constituted by the ns-3 Wi-Fi simulation (denoted by `Simulated WiFi`), which also contains the time for accessing and data transmission on the simulated 802.11 channel. According to the communication actions introduced by our 802.11 emulation framework, `Simulation TX` accounts for the largest part of the delay overhead. It encompasses all delays caused by message processing inside the *WiFiEmuBridge* component when a packet is relayed over the simulated wireless channel from the gateway node. Altogether, the delay overhead caused by this and the other communication actions introduced by our 802.11 emulation framework amounts to less than a third of the overall communication delay. This fraction is certainly not negligible, however we have previously shown in Section 4.2.3.2 that our framework constantly achieves lower RTTs than a comparable real-world 802.11 deployment.

4.2.4 Compatibility with unmodified Wireless Software

A main motivation behind the concept of device-driver enabled wireless network emulation and our corresponding 802.11 emulation framework is to ease the investigation of arbitrary and unmodified wireless networking software in a fully simulated network. We now briefly demonstrate this capability.

Once the ns-3 Wi-Fi simulation is running and the driver has been initialized (see Appendix A.3), any networking software may access the emulated 802.11 network device. Figure 4.14(a) displays a screen-shot of the Wireshark network protocol analyzer monitoring an 802.11 infrastructure network modeled by ns-3. Here Wireshark



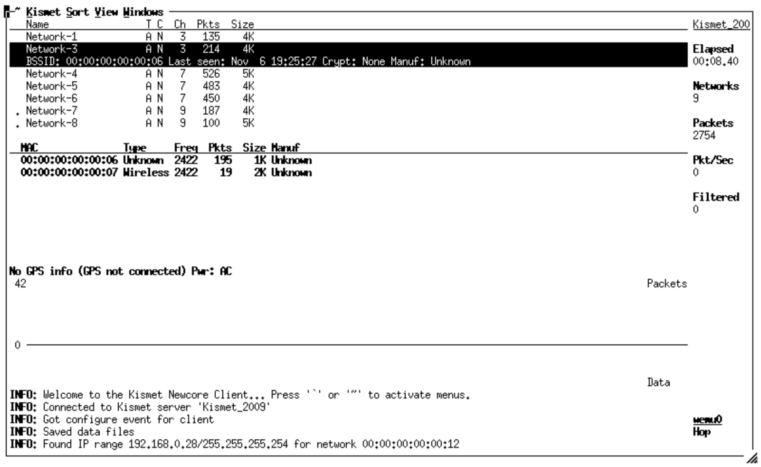
```
No. .  Time        Source              Destination         Protocol  Info
    9 19.999977   00:00:00:00:00:02   ff:ff:ff:ff:ff:ff   IEEE 802 Beacon frame, SN=39, FN=0, Flags=o......., BI=35081,
   10 22.499972   00:00:00:00:00:02   ff:ff:ff:ff:ff:ff   IEEE 802 Beacon frame, SN=40, FN=0, Flags=o......., BI=35081,
   11 24.999979   00:00:00:00:00:02   ff:ff:ff:ff:ff:ff   IEEE 802 Beacon frame, SN=41, FN=0, Flags=o......., BI=35081,
   12 27.499978   00:00:00:00:00:02   ff:ff:ff:ff:ff:ff   IEEE 802 Beacon frame, SN=42, FN=0, Flags=o......., BI=35081,
   13 29.999980   00:00:00:00:00:02   ff:ff:ff:ff:ff:ff   IEEE 802 Beacon frame, SN=43, FN=0, Flags=o......., BI=35081,
   14 32.499979   00:00:00:00:00:02   ff:ff:ff:ff:ff:ff   IEEE 802 Beacon frame, SN=44, FN=0, Flags=o......., BI=35081,

    Data Rate: 1.0 Mb/s
    Channel frequency: 2412 [BG 1]
  ▷ Channel type: 802.11b (0x00a0)
    SSI Signal: -60 dBm
    SSI Noise: -101 dBm
▷ IEEE 802.11 Beacon frame, Flags: o.......
▷ IEEE 802.11 wireless LAN management frame

0000  00 00 18 00 6f 00 00 00  e0 2e 42 06 00 00 00 00   ....o... ..B.....
0010  00 02 6c 09 a0 00 c4 9b  80 80 00 00 ff ff ff ff   ..l..... ........
0020  ff ff 00 00 00 00 00 02  00 00 00 00 00 02 a0 02   ........ ........
0030  00 00 00 00 06 42 2c 40  09 89 00 00 00 0c 77 69   .....B,@ ......wi

wemu0: <live capture in progress...   Packets: 26 Displayed: 26 Marked: 0   Profile: Default
```

(a) Wireshark



```
  Name                  T C  Ch  Pkts  Size                                   Kismet_200
  Network-1             A N   3   135    4K
  Network-3             A N   3   214    4K                                   Elapsed
  BSSID: 00:00:00:00:00:06 Last seen: Nov  6 19:25:27 Crypt: None Manuf: Unknown   00:08.40
  Network-4             A N   7   526    5K
  Network-5             A N   7   483    4K                                   Networks
  Network-6             A N   7   450    4K                                   9
. Network-7             A N   9   187    4K
. Network-8             A N   9   100    5K                                   Packets
                                                                            2754
 MAC               Type     Freq  Pkts  Size Manuf
 00:00:00:00:00:06 Unknown  2422   195    1K Unknown                        Pkt/Sec
 00:00:00:00:00:07 Wireless 2422    19    2K Unknown                        0

                                                                            Filtered
                                                                            0

No GPS info (GPS not connected) Pwr: AC
 42                                                              Packets

 0 ____________________________________________________________

                                                              Data
INFO: Welcome to the Kismet Newcore Client... Press '`' or '~' to activate menus.
INFO: Connected to Kismet server 'Kismet_2009'
INFO: Got configure event for client                                        wemu0
INFO: Saved data files                                                      Hop
INFO: Found IP range 192.168.0.28/255.255.255.254 for network 00:00:00:00:00:12
```

(b) Kismet

Figure 4.14 Our 802.11 network emulation framework enables arbitrary unmodified networking software for Linux to be investigated in a Wi-Fi scenario modeled by ns-3.

is used to examine the RadioTap header of a packet received from an access point. The parameters shown correspond to the state descriptors of the MAC layer of the gateway nodes.

Figure 4.14(b) shows an unmodified version of the Kismet [Ker] wireless network scanner monitoring the simulated Wi-Fi network. The topology contains a number of access points and wireless stations, for which Kismet displays the MAC addresses corresponding to the ns-3 simulation scenario. The main purpose of Kismet is to passively scan for 802.11 stations. It internally makes use of the Linux wireless extensions to implement Wi-Fi scanning and for gathering miscellaneous 802.11 information. The fact that our 802.11 framework is able to execute programs such as Kismet in an entirely simulated context demonstrates its ability to provide an investigation platform for wireless software that requires deep interaction with the wireless network device driver. Hence we expect our 802.11 emulation framework to be especially supportive for the analysis of ad hoc routing protocol implementations or Wi-Fi network management software.

4.2.5 Interim Summary

In this section we have introduced the concept of Device Driver-Enabled Wireless Network Emulation (DDWNE) that bridges a full operating system instance with a network simulator at the MAC layer. For this purpose, a custom device driver is used to provide a network interface to the OS that behaves like a real-world network hardware device. All communication, however, is carried out using a simulated channel that is modeled by the network simulator. The evaluation of our corresponding implementation for 802.11 has shown that this implementation is able to provide a platform for realistic emulation studies for Linux wireless software.

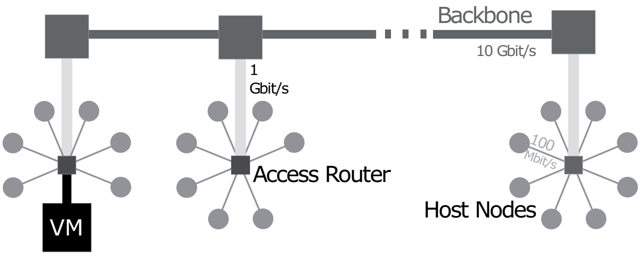
In the following, we will show that our 802.11 implementation for DDWNE ties in nicely with SliceTime and hence forms a comprehensive tool chain for complex wireless emulation studies.

4.3 Application Studies using SliceTime and DDWNE

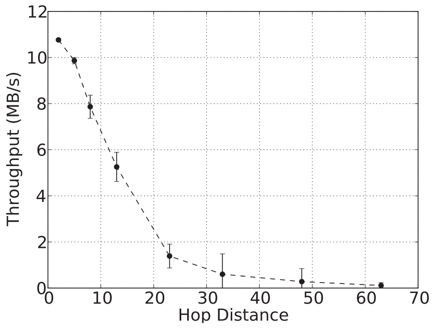
Up to this point, we have provided an in-depth discussion of SliceTime and Device Driver-enabled Wireless Network Emulation (DDWNE). In this section, we utilize these frameworks for three case scenarios. First, we apply SliceTime to an emulation example that involves a very large HTTP network topology. Second, we use SliceTime jointly with DDWNE to reproduce a large scale outdoor study of the AODV routing protocol entirely in software. The third use case shows how VODSim (cf. Sec. 2.1.4) in cooperation with SliceTime can be applied for the analysis of legacy BitTorrent software.

4.3.1 HTTP Use Case

A core motivation of our work is to enable large-scale network emulation setups on custom hardware. In order to analyze our framework in this direction we first



(a) Simple P2P Network: The simulation consisted of one VM and 15000 simulated nodes (60 backbone nodes with 250 host nodes each).



(b) Throughput between VM and Hosts at different simulated hopcounts.

Figure 4.15 Our first use cases applies SliceTime to a HTTP scenario that involves 15000 simulated hosts and one virtual machine.

applied it to a large-scale WAN scenario in which 15000 simulated nodes exchange data in a P2P-like fashion. Due to the simulation size and event load, the whole setup executes about 15 times slower than real-time. For this experiment we used just two of the four testbed PCs (cf. Sec. 4.1.3). One PC hosted the VMs and the synchronizer. The simulation was executed on the second computer.

Figure 4.15(a) illustrates the two-tier topology we used, consisting of 60 interlinked backbone nodes, to which 250 host nodes each are attached via an access router. All host nodes act both as HTTP servers and HTTP clients, requesting a random number of 64kb data blocks from each other. To one of the access routers we connect one VM that runs a standard Linux distribution. The synchronization accuracy was set to 0.1 ms. Using the standard `curl` tool we measured the HTTP throughput between the VM and the simulated hosts at different hop distances (cf. Fig. 4.15(b)).

The observation of the throughput decreasing for higher hop counts is expected and rather straightforward. However, our point here is a different one: First, we achieve valid and consistent measurements on the VM despite both the simulation and the VM operating only at a fraction of wall-clock time. Second, this simple example demonstrates that SliceTime is able to model large-scale network environments at moderate hardware costs, as the vast majority of the network topology is simulated using a discrete event-based network simulator.

4.3.2 AODV routing daemon study

We now investigate the applicability of SliceTime for testing and analyzing 802.11 communications software. We used SliceTime to remodel the AODV[3] part of a real-world field test [GKN+04] in which different Mobile Ad-Hoc Network (MANET) routing protocol implementations were evaluated. In the original experiment volunteers on an athletic field carried around 33 laptops running an AODV daemon. The AODV routing daemon used the 802.11b ad-hoc demo mode for link layer communication. During the experiment the mobile nodes recorded both routing and traffic statistics as well as GPS traces to log the node mobility.

The authors have published corresponding trace files at the CRAWDAD repository [GKN+06]. These trace files allow for computing high level performance metrics for AODV, such as hop count distributions and packet delivery ratios. Unfortunately, basic RTT or throughput measurements have neither been published by the authors nor are available as trace files.

Experiment Design

To remodel the original experiment entirely in software using SliceTime we set up 33 VMs executing the AODV software bundled with the trace files from CRAWDAD. The AODV daemon was configured to use the virtual WiFi NetDevice of DDWNE (cf. Sec. 4.2). We implemented a corresponding simulation scenario in ns-3, which used the ns-3 log distance propagation loss model and random fading for modeling the wireless channel. In addition we extended ns-3 with a mobility model that reproduces the nodes' mobility according to the GPS traces. We only used one of our testbed machines for this experiment. It hosted all 33 VMs, the synchronizer and the ns-3 simulation. The synchronization accuracy was configured to 0.5 ms.

Comparison of real-world measurements with results obtained from SliceTime

Figure 4.16 compares the AODV hopcount distributions of received packets for the real-world data as reported as reported by [GKN+04] and the corresponding remodeled scenario. The hop counts measured using SliceTime are very close to the observations obtained from the real-world field test. We also determined the average Packet Delivery Ratio (PDR) for the real-world experiment and the emulated scenario (cf. Figure 4.17). In the experiment conducted by Gray et al. the PDR amounts to 42.10%[4]. In our remodeled scenario the average PDR amounts to 46.39%.

[3]Ad-hoc On-demand Distance Vector (AODV) [PBRD03] is a mobile ad-hoc routing protocol. It belongs to the class of reactive routing protocols.

[4]We calculated this percentage from the CRAWDAD trace files, as the original publication only visualizes the PDR (see Figure 3 in [GKN+04]) and does not state a precise number.

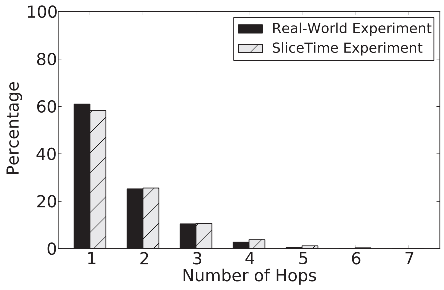


Figure 4.16 Real-World AODV experiment vs. remodeled SliceTime scenario: the hop count distribution of received packets obtained from the AODV scenario remodeled with SliceTime well matches the hopcounts measured in the real-world scenario.

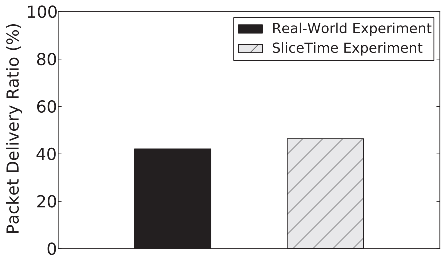


Figure 4.17 The remodeled SliceTime scenario properly approximates the total AODV Packet Delivery Ratio of the real-world field test as reported by [GKN+04].

These differences in the packet delivery ratio and the hop count distributions demonstrate that there is no perfect match between the original and the remodeled scenario. For example, the different hopcount distributions indicate that AODV used different paths for routing packets between the nodes. There will always be differences between real-world measurements and observations taken with emulation platforms. This is a direct consequence of the disparity between the real world and the environment modeled in software. The 802.11 model of ns-3, for example, is relatively sophisticated and quite accurately reproduces the behavior of the 802.11 MAC and PHY layers. However, there are many factors that are not considered by our remodeled scenario, like antenna characteristics or even a hypothetical nearby microwave oven that could have influenced the real-world measurements.

This use case shows that SliceTime is well able to provide a testing environment for 802.11 software that delivers results being close to the reference scenario. Repeating real-world experiments like the one conducted by Gray [GKN+04] is costly and often challenging due to continually changing conditions, for example, regarding the wireless channel. By contrast, SliceTime allows one to arbitrarily modify and rerun WiFi software experiments at the push of a button. SliceTime is also cost effective compared to the hardware costs and manpower requirements of the original

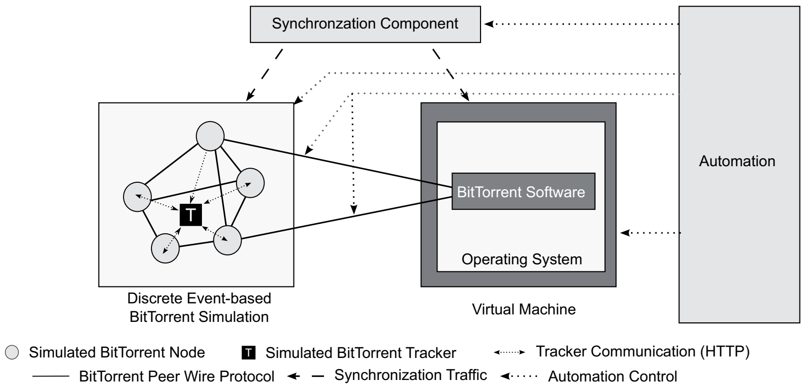

Synchronzation Component
Automation
BitTorrent Software
Operating System
Discrete Event-based BitTorrent Simulation
Virtual Machine
T
Simulated BitTorrent Node
Simulated BitTorrent Tracker
Tracker Communication (HTTP)
BitTorrent Peer Wire Protocol
Synchronization Traffic
Automation Control

Figure 4.18 Overview of the BitTorrent evaluation framework.

experiment. While the original field test involved around 40 volunteers and the same number of laptops, with SliceTime the same experiment can be conducted on one desktop PC.

4.3.3 A Hybrid Evaluation Platform for BitTorrent systems

The integration of VODSim (cf. Sec. 2.1.4) with SliceTime enables the construction of an emulation framework for legacy BitTorrent (BT) software. The vision behind this concept is a BitTorrent evaluation platform that enables the analysis of real-world BitTorrent implementations in a fully isolated and repeatable networking environment. The second goal is supporting both open and closed source BitTorrent implementations for different operating systems.

Figure 4.18 displays the conceptual overview of the scenario. In the same way as all synchronized network emulation setups (cf. Section 3.4.1) we rely on a network simulation based on ns-3 and VODSim to model an entire BitTorrent swarm, consisting of simulated BT clients and one simulated tracker. The legacy BitTorrent software is hosted using our SliceTime VM implementation.

An additional *automation component* coordinates the execution of the entire framework. In essence, it is responsible for instantiating the network simulation, the VM and the synchronizer in the correct order. It also establishes the data transfer between the simulation and the VM and facilitates fully-autonomous batch runs. Although the automation component is not required for carrying out BT emulation experiments, it is greatly helpful due to the potentially long simulation run-time. The automation component is implemented using a set of shell scripts which internally rely on the Xen command line tools and ssh to perform the required tasks.

4.3.3.1 A Synthetic Benchmark of BitTorrent software

We now demonstrate the application of our BitTorrent evaluation platform for the analysis of BitTorrent software by conducting a performance benchmark of different software clients. In the following we first provide a brief overview of the investigated clients. In a second step we describe the benchmark setup at greater detail before discussing the actual results at the end of this section.

BitTorrent Software used in the Benchmark

In our performance benchmark we consider the following client implementations.

- **Vuze** [vuz], previously named Azureus, is one of the most popular BitTorrent clients. The first version of Azureus was released in 2003 as open source software under the GPL license. Since Vuze is based on Java, it can be executed on a wide range of platforms. In our benchmark we execute Vuze 4.3.0.6 on a Debian Squeeze VM using OpenJDK 1.6.0_18/IceTea6 1.8.13.

- **Deluge** [del] is another well-established BitTorrent client that is available for different operating systems. In contrast to Vuze, Deluge is written in C++ and hence is natively compiled for the operating system it executes on. Internally, Deluge makes use of the libtorrent[5] software library. Libtorrent is a comprehensive software API that provides almost any required functionality to implement a BitTorrent client. It encompasses a very robust implementation of both the tracker and the peer-wire protocol as well as sophisticated piece selection strategies. In addition, it implements many BitTorrent extensions such as tracker-less operation or super-seeding. Besides Deluge, numerous BT clients internally also make use of libtorrent. In order to demonstrate the applicability of our framework to different operating systems we include both the Windows and the Linux implementations of Deluge in our benchmark. More specifically, our measurements were taken using the Deluge versions 1.2.3 (Linux) and 1.2.5 (Windows).

- **BitTyrant** [PIA+07b] is a strategic BitTorrent client. It demonstrates that advanced sharing strategies can be used to dramatically improve the downloading performance of a client. In essence, BitTyrant optimizes the upload/download ratio of the data exchange with other peers. It also employs advanced choking and unchoking mechanisms. We include BitTyrant in our synthetic benchmark driven by the question if BitTyrant is able to achieve lower downloading times in comparison to Vuze and Deluge.

  BitTyrant is based on the 2.5 version of Azureus and hence inherits most architectural characteristics of Azureus/Vuze as described above. In our performance Benchmark we deploy BitTyrant on the same system environment as Vuze.

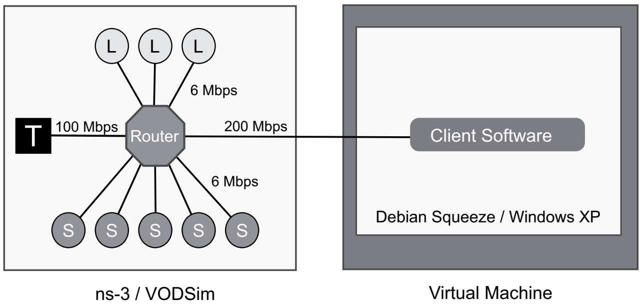


Figure 4.19 The scenario used for the benchmark: ns-3 and VODSim are used to model a straightforward star topology that interconnects a set of seeders, a number of leechers, one tracker and the virtual machine. The synchronizer is omitted in this illustration.

Benchmarking Scenario

The goal of our synthetic benchmark is to investigate how different BitTorrent clients deal with different ratios of seeders and leechers in a BitTorrent swarm. Figure 4.19 illustrates the setup of our benchmark scenario. Using an adequate story file, VODSim is instructed to construct a star topology that interconnects all BT nodes, the tracker and the virtual machine which executes either Vuze, BitTyrant or Azureus. The BT nodes are connected to the router using 6 Mbps links. The tracker is connected to the router using a simulated 100 Mbps link. The link that connects the VM with the simulated topology has a bandwith capacity of 200 Mbps. We use a fixed number of simulated nodes of $n = 100$ for all measurement runs. All simulated nodes simultaneously join the simulated BitTorrent swarm at the beginning of the simulation, resembling a flash crowd scenario. The parameter varied across different measurement runs is the ratio of seeders versus leechers in the swarm.

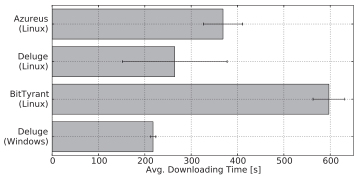
Performance Metric: Downloading Time

We quantify the performance of a BitTorrent client using *downloading time* as base performance metric. As the performance measurements differ between the runs due to a non-deterministic execution of the BitTorrent client on the VM, we conduct five measurement runs for each BitTorrent client and each seeder/leecher ratio.

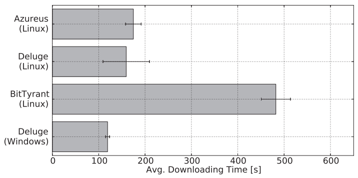
Benchmark Result

The results of the BitTorrent client benchmark are depicted in Figure 4.20. The shown downloading times are averages over five measurement runs for each client with error bars denoting the respective standard deviation. As expected, all clients require less downloading time if more seeders become available. In this regard, the Windows version of Deluge and Azureus rather quickly converge to their maximum performance, as the average downloading time does not increase any further for

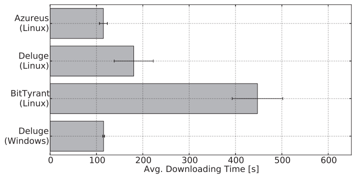
[5]More Information about the libtorrent library and its features can be found at `http://www.libtorrent.org/` (accessed 12/2012).



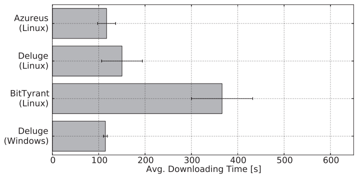
(a) 10 Seeders, 90 Leechers



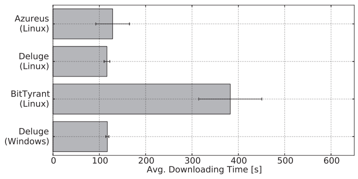
(b) 30 Seeders, 70 Leechers



(c) 50 Seeders, 50 Leechers



(d) 70 Seeders, 30 Leechers



(e) 90 Seeders, 10 Leechers

Figure 4.20 We used our emulation framework to measure the downloading speed of three different BitTorrent clients for five different ratios of seeders and leechers. Interestingly, all contemporary BitTorrent clients outperform BitTyrant which originally had been developed to optimize its own downloading performance [PIA$^+$07b].

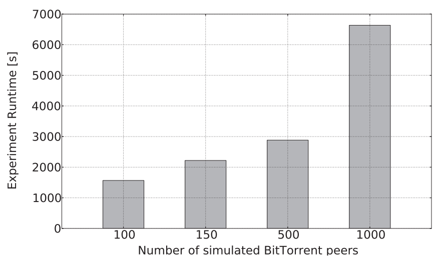


Figure 4.21 Experiment run-time for the BitTorrent emulation experiment.

seeder ratios higher than 50%. Comparing Azureus with Deluge, both clients show a very similar downloading performance in our experiment.

There are two observations in these measurements that require further clarification: The first interesting effect is that Deluge for Windows tends to download the torrent faster than its Linux counterpart. This indicates that there are disparities in the implementations between both operating systems, which are reflected by slight performance differences.

Second, it is striking that BitTyrant performs significantly worse than the other clients in all measurement runs. This is particularly interesting, as BitTyrant is a strategic client which has been proven to be able to dramatically optimize its own performance using advanced trading strategies [PIA$^+$07b]. BitTyrant, however, is not able to reproduce this behavior in our emulation experiment. We have no definite answer for this result. However, we attribute this observation either to incompatibilities of BitTyrant with VODSim or to the fact that BitTyrant has not been maintained since 2007. As we have used recent versions of all other clients, they might have incorporated advanced piece selection strategies, traffic shaping or better choking strategies as well, which might allow them to outperform BitTyrant.

Our benchmark scenario indeed models a very specialized case and is not representative for all existing BitTorrent swarms in any way. For example, all simulated clients implement exactly the same behavior for piece selection and choking; as already mentioned, a certain client software might either suffer or benefit from this circumstance. One way to compensate for such effects would be the introduction of more heterogeneity to the behavior of the simulated BitTorrent clients, for example by using a random distribution of different strategies.

Scalabilty of BitTorrent Emulation

Besides the actual outcome of the BitTorrent benchmark we also measured the resource usage in order to investigate the scalability of our platform. For this purpose we relied on the same benchmarking scenario as before and measured the experiment run-time and the memory consumption for different BitTorrent swarm sizes. All the given numbers were obtained using Deluge as client deployed on a Linux virtual machine.

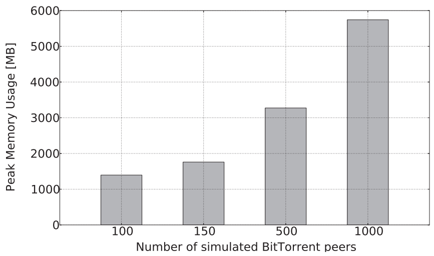


Figure 4.22 Peak Memory used by the simulation for different node counts.

Experiment Runtime

Figure 4.21 compares the runtime for simulated BitTorrent swarm sizes between 100 and 1000 nodes. The experiment run-time grows almost linearly for the investigated swarm sizes. In this regard it is important to mention that the overall run-time is governed by the network simulation. The VM itself would be able to execute in real-time, but it needs to be slowed down to match the execution speed of the simulation, resulting in a higher degree of slow-down of the VM for bigger BitTorrent swarm sizes.

Memory usage

Likewise, the memory required by our framework is dependent on the memory consumption of the BitTorrent simulation, as the VM always requires the same and statically configured amount of RAM. Figure 4.22 shows the peak memory allocated by the BitTorrent simulation for BT swarm sizes of 100, 150, 500 and 1000 nodes. The memory needed increases almost proportionally to the node count. Even for 1000 nodes, the peak memory consumption stays below 6GB of RAM.

Discussion of BitTorrent emulation results

From these numbers we conclude that legacy BitTorrent software can be well evaluated with our framework for simulated swarms of medium size. In fact, the only limiting hardware resource is the amount of memory that is available on the simulation machine. A second issue is the slow down caused by the growing complexity of the network simulation. It is not uncommon for simulated swarm sizes of 500 or 1000 nodes that one simulated second takes 10 or even 100 seconds to compute in real-time. Hence, one second on the VM will be stretched to this duration. This makes measurement runs sometimes tedious; however, our automation framework enables BT studies to be carried out without any supervision. In addition, the design decision to trade variable experiment run-time for constant hardware requirements makes our framework rather cost-effective if one is patient enough to use it.

4.4 Related Work

The history of network emulation dates back to the late 1980s, when this concept was first proposed as a new methodology for the evaluation of networked systems [Bac87, BY88, DSYB90]. Since then, quite a few network emulation frameworks for different target applications have been proposed. For example, there are network emulation tools specifically targeting wireless networks ([JS05, KGM+01, SGB09]) while others have been designed for the analysis of a specific protocol (e.g SVEET! [ELL09]).

Despite its diversity, the entire set of network emulators shares one common property: A *network emulation engine* (NEE) mimics the propagation of packets between the clients that are interconnected via the NEE. Hence, the capabilities of a network emulation framework are directly dependant on the design and the implementation of the NEE. In the following discussion of these approaches we distinguish between four different categories of network emulation engines.

- Most network emulation environments use a custom simulation engine to reproduce network effects such as end-to-end latency, bandwidth limitations or virtual mobility in software. We refer to these emulation engines as *Link Emulation Engines.*

- The idea behind *virtualization-based emulation engines* is to execute a number of virtualized communication clients in parallel. For this purpose, they employ different virtualization techniques. Most of them use link emulation engines to model the network topology that interconnects the client instances.

- A special type of software network emulation engines are frameworks that rely on an event-driven simulator such as ns-2, OMNeT++ or ns-3 for the implementation of the NEE. We refer to this category as *Network Simulator-based Emulation Engines.* All the frameworks presented in this dissertation so far belong to this category.

- *Hardware-based Network Emulation* relies on dedicated emulation hardware or on a special emulation testbed to model the interconnecting network between the clients. Typical examples of hardware-based emulators are Emulab [WLS+02] or the Carnegie Mellon Network Emulator [JS05, JS04].

In the following, we will discuss these categories of network emulators separately; however it is noteworthy that there is a strong overlap between them. For example, Emulab uses Dummynet [Riz97], a specific emulation engine, to model the behavior of wide area links. In addition, it is noteworthy to mention that almost none of the frameworks implements a time synchronization scheme that aligns the execution of the clients with the NEE. The only rare exceptions to our knowledge are Time Jails [GMHR08], a specific emulation engine, and SVEET [ELL09], a simulator-based one. We discuss Time Jails and SVEET at greater detail later in this section and compare their properties with our work.

4.4.1 Link Emulation Engines

In general, Link Emulation Engines (LEEs) constitute a piece of software that merely influences the communication between the clients it interconnects. A LEE affects the clients' communication in different ways, for example by dropping network packets, by delaying, duplicating or corrupting them. Table 4.2 provides a concise comparison of the tools that are discussed in the following.

4.4.1.1 Basic Link Emulation Engines

Conceptually, such emulation engines can be integrated at different layers of the TCP/IP stack. Emulation tools such as Delayline [IP94] and EmuSocket [AV06] implement link emulation at the socket layer. Essentially, the standard socket functions of the Berkeley Socket API [SFR04] such as `send` are relinked to use particular sending functions of an emulation library. Today, this approach is less common because it has different shortcomings. First, this approach requires access to the source code of the software running on the client, as it otherwise is not possible to change the sending and receiving functions. Second, their application is limited to applications that use standard transport protocols like UDP and TCP that are supported by the respective emulation socket library. Finally, as the entire emulation takes place at the application layer, such emulation frameworks can not be used to investigate lower layer protocol implementations, for example new routing or transport protocols.

A more common and more flexible approach is to integrate the LEE into the communication between the clients at the MAC layer. This gives one the opportunity to investigate the influences of network effects such as packet loss on arbitrary routing, transport and application protocols. Emulation tools that have popularized this approach are Dummynet [Riz97, CR10], NetEm [Hem05] and NIST Net [CS03]. As these frameworks are integrated into the operating system kernel and operate in real-time, their accuracy depends on the respective kernel implementation.

Dummynet

Dummynet is a flexible link emulation engine that is available for different Unix-based operating systems, Mac OS X as well as for Windows and the OpenWRT Linux distribution for wireless routers. It allows one to specify bandwidth limitations, network delays and packet dropping behavior for incoming and outgoing TCP/IP network traffic. Using IP address filters, it is also possible to specify different link characteristics for different communication end points. The implementation of Dummynet is based on a set of queues, so-called pipes in the context of Dummynet. To apply network emulation to packets, such pipes can be applied to both incoming and outgoing network traffic. If a packet arrives at a pipe, it is first inserted into a queue whose departure rate corresponds to the bandwidth that was specified for the respective pipe. Afterwards, the network packet is delayed for the specified network delay before the packet is reinjected into the network stack.

Name	Target System	Delay	Drop	Duplication	Reordering	Traffic Shaping	Note
Delayline [IP94]	Unix	✓	✓				Requires source code access
EmuSocket [AV06]	Java Software	✓	✓				Requires source code access
Dummynet [Riz97, CR10]	Linux, FreeBSD, Mac OS X, Windows	✓	✓			✓	Used by Emulab [WLS+02, HR12]
NetEm [Hem05]	Linux 2.6	✓	✓	✓	✓	✓	Also supports modeling of packet corruption
Nist Net [CS03]	Linux	✓	✓	✓	✓	✓	Supports “imitating” a target network
NEMAN [PP05]	Linux		✓				Mobility Support; Nodes are either connected to the topology or not.
TUM Nist Net [RSW03]	Linux	✓	✓	✓	✓	✓	Mobility Support; Provides GUI.

Table 4.2 Comparison of different link emulation tools.

NetEm

NetEm [Hem05] is a link emulator for Linux which pretty much resembles Dummnynet according to its feature set. It has been integrated into the Linux kernel since version 2.6, and in comparison to Dummnynet supports additional network effects, most notably packet reordering, packet duplication and packet corruption. NetEm is integrated into the general Linux Traffic Control (tc) module and thus can be controlled with the `tc` command on most contemporary Linux systems. Its implementation is based on a rather sophisticated system of different queues and packet schedulers [Alm99].

NIST Net

NIST Net [CS03] has a distinctive feature that sets it apart from plain link emulators such as NetEM, Dummynet or simpler and earlier approaches such as ONE [AO97]. NIST Net allows one to use external modules for calculating statistics that are e.g. applied for delaying network traffic or for dropping packets. This enables researchers and developers for example to delay network packets according to a delay table that was obtained from prior network measurements, thus opening up the potential for calibrating the emulation engine.

4.4.1.2 Wireless Link Emulators

Investigating wireless networking software, systems and respective protocols in real-world deployments is cumbersome due to the continuously changing channel conditions. For this reason, the potential of network emulation for the evaluation of such has been recognized early in the research community, as network emulation is able to deliver a controlled environment for the evaluation of wireless networking systems.

While basic link emulators already are able to model the phenomena that are prevalent in wireless networks, for example packet errors or different latencies, it is difficult to adjust the emulators' settings to resemble a certain scenario. In fact, emulation engines such as the previously discussed emulators NetEm [Hem05], NIST Net [CS03] and Dummynet [Riz97, CR10] often form the basis of wireless network emulators [Sly07, RSW03, BNM+09]. In addition, wireless link emulators such as Seawind [KGM+01] have been proposed in the literature. We omit Seawind in the following discussion, as it very much resembles the discussed network emulators according to its feature set.

Advanced wireless network emulation tools such as NEMAN [PP05] and TUM NIST Net [RSW03] are able to project events such as node mobility onto the behavior of an underlying link emulator. For this purpose, the emulators implement mobility models and wireless propagation models. The real-time output of these models is then fed into the emulation. NEMAN implements this projection in an on-off-fashion, which means that the outcome of the model simply computes if nodes are able to communicate or not. TUM Nist NET is more sophisticated in this regard, as it dynamically adjusts link properties such as delay or packet error rate based on the models' outcome. It is also noteworthy that these emulation tools provide a graphical user interface (GUI) to control the network topology and the nodes' movement.

4.4.1.3 Comparison with our work

The main difference between link emulation tools and our frameworks, most notably SliceTime and DDWNE, is that link emulators only reproduce the characteristics of the communication path. Link emulators all rely on the existence of real clients. Clients in this regard are ordinary computer programs that are executed either on physical host or inside a virtual machine. In contrast to that, our synchronized evaluation frameworks allow the provision of simulated hosts that are modeled by a network simulator. This is useful if one is interested in investigating the behavior of real-world client software in a large-scale simulated networking context. We have demonstrated this capability of SliceTime in our simple P2P use case (cf. Sec. 4.3.1) and more importantly with its application for investigating BitTorrent software (cf. 4.3.3).

Network emulators that are based on discrete-event based simulators, for example ns-3, can also be applied for pure link emulation. This can be achieved simply by modeling the links in the network simulation and by attaching two or more gateway nodes to the simulation. If traffic originating at real clients is then simply passed through the simulation, such an emulation scenario equals the functionality of a plain link emulation engine. As our frameworks, particularly SliceTime, are based on ns-3, they inherit this capability. This yields to the question if it makes sense to perform link emulation using our frameworks. The answer to this question is twofold.

If the topology is rather straightforward and just adds e.g. static delays or packet loss to a couple of links, it is more convenient to use a link emulator than applying ns-3 and SliceTime to such a scenario. Why? Link emulation engines such as NetEm or Dummynet can be configured easily with a couple of Unix shell commands in order to model a particular link behavior. By contrast setting up such a scenario in ns-3 is more complex, as it involves writing a C++ simulation program. Moreover, such ns-3 simulation scenarios mostly can be executed in real-time and hence supersede the need of applying the SliceTime synchronization features.

However, the situation changes if the scenario requires the emulation of dynamic link characteristics, e.g. in wireless scenario due to node mobility or because of channel models that incorporate wireless effects such as fading or interference. Wireless link emulators need to compute the output of auxiliary propagation and mobility models in real-time; the output of these models is then used to adjust the link properties. This limits the models' computational complexity and hence their accuracy. Network simulators such as ns-3 incorporate more accurate wireless channel models.

As SliceTime eliminates the need of the simulation to execute in real-time, it can be used for setting up very complex link emulation scenarios that incorporate computationally expensive propagation and mobility models. In fact, in our AODV use case scenario (cf. Section 4.3.2) ns-3 together with SliceTime was acting as link emulation engine. Ns-3 was used to model the mobility and the complex channel dynamics between the 40 software clients. Although not being investigated in this thesis, this would enable the integration of highly accurate propagation models of high computational complexity, for example raytracing based models such as [SW06], into an emulation setup.

4.4.2 Virtualization-based Emulation Engines

Evaluating networking software and communication systems on physical testbeds is traditionally costly in terms of hardware requirements. This has led to different virtualization-based network emulation tools that virtualize an entire communication network and execute it on one host or a small set of physical machines.

4.4.2.1 Virtualization-based Emulation in Real-Time

Most of these frameworks execute the virtualized emulation scenario in real-time. In the following, we briefly survey important tools of this category (cf. Table 4.3).

ENTRAPID

ENTRAPID [HSK99] is a prominent example for this category of network emulators. It runs as a user-space process that instantiates so-called Virtualized Network Kernels (VNKs). Each VNK replicates the functionality of a 4.4 BSD-kernel[6]. In the ENTRAPID framework, applications and application level protocols are implemented using so-called virtualized processes. Multiple virtualized processes can be executed on top of one virtualized network kernel. The topology between different VNKs is modeled using "wires" that resemble communication links. Propagation delays and bandwidth restrictions can be associated with every wire. In addition, the

[6]BSD (Berkeley Software Distribution) is a former UNIX OS that was developed at the University of California Berkeley in 1977. While the development of the original BSD has stalled, derivatives such as FreeBSD are still common today.

Name	Operating System	Virtualization Approach	Note
GNUNet Emulation [EG11]	Linux	Virtualized GNUNet processes	Specially designed for the evaluation of GNUNet
P2PLab [NR08]	FreeBSD	Virtualized FreeBSD processes	Uses Dummynet for link layer emulation
ENTRAPID [HSK99]	BSD	Virtualized BSD kernels	Supports topologies with link emulation
IMUNES [ZM04, Zec03]	FreeBSD	Virtualized FreeBSD network stacks	Support for legacy applications
CORE [ADH$^+$08]	FreeBSD, Linux	Virtualized FreeBSD network stacks, Linux Containers	Rich GUI, Support for wireless networks, mobility models
NetKit [PR08]	Linux	User Mode Linux	

Table 4.3 Comparison of different real-time emulation frameworks making use of virtualization.

framework allows the inclusion of external processes that run on the same machine as ENTRAPID into the virtualized topology.

Virtualization-based Emulators for P2P Networks

The idea of using virtualized networking applications for emulations has recently also been proposed for the analysis of P2P networks. Evans and Grothoff [EG11] have implemented an emulation library for GNUnet, a free software framework for developing P2P applications. The emulation library allows one to execute a large set of GNUnet-based P2P applications in a virtualized topology using one or multiple physical hosts. The emulated application instances are executed as individual processes on top of the host's operating system. P2PLab [NR08] is another emulation framework that is based on the concurrent execution of P2P application processes. It is implemented by applying slight changes to the FreeBSD C library in order to isolate the TCP/IP traffic of the applications against each other. P2PLab relies on the earlier discussed Dummynet framework for the emulation of links.

IMUNES

IMUNES [ZM04, Zec03] enables the instantiation of multiple independent network stacks on a FreeBSD system. This approach allows the emulation of a network with different hosts by creating a set of independent network stacks first. Multiple virtual clients are then subsequently modeled by a set of application processes of which each uses a distinct network stack instance. Virtual network bridges among these network stack instances serve as primitive for reproducing network links and topologies. The major advantage of IMUNES in comparison with frameworks like ENTRAPID is its support for legacy network applications.

CORE

The original version of the CORE [ADH$^+$08] network emulator was implemented on top of IMUNES and extends its basic emulation features in different ways. First, CORE has added support for wireless networks and mobility by implementing corresponding models that project their output onto the network topology.

Other improvements include the ability of controlling physical hosts, support for scripting the emulation scenario and a rich GUI that enables a flexible control of the real-time emulation. In its later releases CORE has also started to use Linux Containers [Men07] as an alternative framework for process virtualization. Hence, it also can be executed on Linux and doesn't require IMUNES or FreeBSD any more.

NetKit

NetKit [PR08] is a network emulation framework that makes use of User Mode Linux [Dik01] to model a network of Linux systems on a single host. All systems are executed concurrently as UML instances. In contrast to frameworks such as P2PLab, Entrapid or IMUNES, Netkit virtualizes an entire operating system with a dedicated filesystem. Hence, its virtualization overhead is higher. Still, the authors report that they were able to execute 100 VMs on a single workstation PC.

Discussion and Comparison with our work

The strength of all these frameworks is their rather lightweight virtualization approach, which enables the real-time execution of a rather large number of virtualized clients on a physical host. Evans and Grothoff report [EG11] that they were able to emulate a DHT similar to Kademlia [MM02] with 80000 peers using this framework using a cluster of 32 machines. This corresponds to the concurrent execution of 2500 P2P applications on a physical machine. By contrast to that, the largest number of virtualized clients we have run on a single host is 33 (cf. Section 4.3.2). This difference in virtualization scalability is a direct consequence from the fact that a Xen-based virtualization of an entire OS is far more costly in terms of system resources.

However, the scalability of virtualization-based emulation engines is also strictly limited by the capacity of the host machine. If the memory requirements of the virtualized networking applications or the required CPU time outgrow the resources available of the host machine, either swapping memory to the hard disk or an overload on the system's process scheduler will be the straight consequence. In such situations the observed networking performance of the systems may be deteriorated because the execution of them is lagging. For these reasons, such system overload may strongly damage the emulation accuracy and thus has to be avoided.

Another shortcoming of these frameworks is their limited flexibility if compared with SliceTime. Tools such as ENTRAPID and IMUNES only provide support for virtualized application processes and thus cannot be applied for the evaluation of kernel-level software.

4.4.2.2 Virtualization-based Emulation with Virtual Time

The use of time virtualization has been proposed for different reasons in the domain of network emulation. Table 4.4 lists such emulation frameworks. As these tools constitute important related work we now discuss them separately in further detail.

Name	Main Goal	Virtualization Technology	Synchronization Scheme
Time-Warped Network Emulation [GYM+06]	Investigate software on high-speed network interfaces	Xen	TDF
DieCast [GVV08]	Emulation of large-scale networks on limited hardware resources	Xen	TDF
TVEE [GMHR08]	Scalable network emulation using virtualized applications	Xen + OpenVZ	Adaptive TDF scheme
dONE [Ber06, BVB06]	Scalable network emulation using virtualized applications	Custom application virtualization library	Conservative scheme based on hierarchic time sources

Table 4.4 Comparison of different emulation environments that employ time virtualization.

Time-Warped Network Emulation

Gupta et al. [GYM+06] have proposed a rather unique concept for network emulation. *Time-warped network emulation* enables the evaluation of unmodified applications and operating systems on communication links that are magnitudes faster than available physical communication hardware. In order to achieve this goal the communication systems are virtualized using the Xen hypervisor. The authors decouple the VM's progression of time and scale the clocks' speed upwards or downwards using a so-called *time-dilation factor (TDF)*. For example, a TDF equaling to 3 means that the wall-clock progresses three times faster than the time on the virtual machine. In effect, this leads to the virtual machine and the software running on it to perceive a network bandwidth that is about three times higher than the physical bandwidth of the communication link it uses.

While the motivation of time-warped network emulation differs greatly from our work, the implementation of Gupta et al. shares some similiarities with our SliceTime framework, as both are based on the Xen hypervisor and both tweak the timer management of Xen. However, the major difference between SliceTime and time-warped network emulation is that our framework executes the VM at its native execution speed during the time slice, whereas Gupta et al.'s tool-chain statically throttles or speeds up the time progression on the VM in order to achieve the desired scaling of network performance. Hence, their virtual machines progress through time continuously (with a scaling factor to the VM's clock sources applied) while in SliceTime the VMs are advanced through the virtual time line in a on-off fashion.

On a side note it shall be mentioned that a behavior similar to the one produced by time-warped network emulation could also be observed with early versions of SliceTime. The early implementation of our framework was based on Xen paravirtualized machines (PVMs), in contrast to the hardware virtualized machines (HVMs) we presently rely on. As many of the network operations for PVM domains are handled by the privileged control domain 0 and thus are executed outside the virtualized time progression, we observed that the network throughputs of PVMs were increasing for smaller time slices. In order to circumvent these effects, we implemented a traffic shaper into the PVM's networking back-end at this point. As these effects, however, do not occur when HVM domains are used, these early changes became obsolete and where thus removed from SliceTime later on.

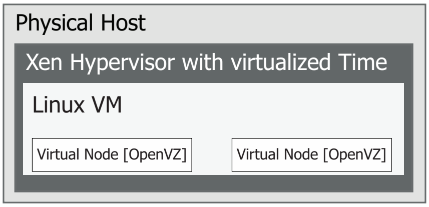


Figure 4.23 The TVEE emulation framework combines the concepts of application virtualization, operating system virtualization and time virtualization to improve the scalability of network emulation.

DieCast

With DieCast [GVV08] Gupta et al. have also introduced a direct successor of time-warped network emulation. In addition to their prior work DieCast also includes models to scale the perceived system resources, such as hard disk performance and CPU resources using an according scale factor. The purpose of this endeavour is to model entire setups of networked systems using a set of restricted physical hardware resources by trading off execution time for hardware costs. The effect is that one is able to multiply the capacity of a testbed by the scale factor. For example, a scale factor of 5 would allow one to run five times as many machines on the testbed with each machine perceiving still the full capacity of hardware resources in terms of CPU, networking and I/O performance. Due to the time virtualization, this experiment would run five times slower than real-time.

While the motivation of DieCast is largely different in direct comparison with SliceTime the authors of DieCast observe and make use of effects that are also apparent in SliceTime setups. Most notably, the authors also have shown that tampering with the time progression of a Xen VM impacts its perceived CPU performance; for this reason they proposed an according scale model to retain the perceived CPU when the clock speed is scaled. SliceTime does not implement such mechanisms, and for this reason, the synchronized virtual machines do observe impacts on the CPU performance (cf. Sec. 4.1.3.4). However, as we have shown, the CPU performance degradations we observed are well tolerable for time slice sizes of 0.1 ms and can be neglected for time slizes $\geq$ 0.5ms. In direct comparison with DieCast, it is also noteworthy that we neglect the disk I/O performance, as it is not the limiting factor for the vast majority of networking applications.

The Time Virtualized Emulation Environment (TVEE)

With TVEE [GMHR08] Grau et al. have developed a comprehensive framework that uses time virtualization for extending the scalability of virtualization-based network emulation. TVEE virtualizes communication clients using a two container hierarchy (cf. Fig. 4.23). First, TVEE relies on the Xen hypervisor to create a Linux-based virtual machine. Second, the virtual machine itself uses OpenVZ [ope] to encapsulate virtual communication nodes into separate containers.

In order to extend the capacity of such a setup, Grau et al. employ time virtualization at the VM level to multiplex system resources akin the previously discussed DieCast. In TVEE, each VM is slowed down by applying a TDF factor that is chosen proportionally according to the CPU time consumed by the virtual machine during a certain epoch. Dynamically adjusting the TDF allows the framework to achieve a better system utilization in comparison to a conservatively chosen static TDF, which would lead to system under-utilization and thus also to an increased execution time of the emulation scenario.

The continuation of this work [GKN+04] adds support for cluster-based deployments of TVEE, in which a set of physical hosts executes a number of virtual machines each. Like in the original TVEE all VMs also govern a number of virtual nodes. For synchronizing the progression of time among the cluster of hosts, the authors monitor the CPU load of all VMs on all physical hosts using a central load monitor. The TDF is then chosen according the min/max values of the measured CPU loads.

TVEE shares many similarities with Time Jails and DieCast, as it also uses time dilation to enhance the capacity of a physical host regarding the concurrent execution of a number of virtual clients. Conceptually, the most striking alikeness between TVEE and SliceTime is the epoch based TDF adaption scheme. TVEE uses discrete epochs for which the CPU load is measured, and the TDF is then used to match the execution speed of the VMs to the available CPU resources. Essentially, this means that TVEE uses an optimistic synchronization scheme that aligns the execution speed of the VM's and the hosted virtual nodes in best effort fashion. As a consequence the synchronization scheme of TVEE cannot guarantee an avoidance of drift among the VMs. SliceTime's barrier synchronization scheme, however, is conservative and the drift among the systems is bounded by the time slice size. Moreover, SliceTime does not alter the execution speed of the VMs. The execution of the VM is simply blocked after the time slice has been consumed, and the VM is subsequently unblocked if the next time slice is assigned. Hence, the observed slowdown in a SliceTime setup does not stem from a use of a TDF, but from an on/off-alike execution pattern.

The biggest difference between TVEE and SliceTime is that TVEE solely relies on virtual nodes for the purpose of modeling communication clients. In contrast to that, SliceTime is a network simulation-based emulation tool, which allows communication clients additionally to be modeled by the network simulator. Using a network simulator also makes it possible for parts of the emulation scenario to be of deterministic nature, which is not feasible for systems such TVEE and Diecast that are exclusively based on native code execution.

The Distributed Open Network Emulator (dONE)

The distributed open network emulator (dONE) [Ber06, BVB06] integrates an event-based parellelized network simulation based on MPI [SOHL+96] with virtualized software clients. For virtualizing the software clients the authors rely on a custom virtualization library and a custom socket emulation layer, which reproduces the behavior of BSD 4.4 sockets.

In order to synchronize the execution of virtualized clients, the authors introduce the concept of *relativistic time*. Basically this time synchronization scheme is based on the application of two time dilation factors to two clocks. The first clock, the global relativistic time, is a time source which controls the execution of all virtualized software clients. It progresses continuously and slower than real-time; it is scaled down by the first TDF. The second TDF, the local relativistic time, throttles the execution of the local virtualized clients individually in order to compensate for local time drifts among them. Both dilated time sources are controlled by a so-called coordination processor which is also connected via MPI. As both TDFs are of the throtteling kind, the global relativistic time constitutes an upper bound beyond which no virtualized client can have progressed. For this reason, the authors classify this scheme into the category of conservative synchronization algorithms. Unfortunately, the authors are rather inexplicit regarding dONE and its synchronization scheme in different ways. First, it is not quite clear how the concept of relativistic time is implemented in the MPI context. Second, the authors regrettably surrender an investigation of the timing accuracy of the chosen synchronization approach; hence, it is unclear to some extent how well dONE actually is working.

There are several strong differences and similarities if one compares SliceTime with dONE. The first obvious common ground is that both are in the need of synchronizing the execution of an event-based architecture with the continuous progression of actual communication software. Both SliceTime and dONE rely on conservative synchronization schemes, which however differ significantly. dONE relies on two hierarchic time sources which provide a strict upper time bound beyond which no software client may progress. According to the authors the synchronization scheme allows the global relativistic time to be advanced without explicit messaging. In contrast to that, a cornerstone of SliceTime's barrier synchronization protocol is the distinct provision of each time slice. In our evaluation we have shown that our synchronization scheme adds a moderate amount of overhead for time slices equalling to or greater than 0.1 ms. Moreover, our RTT analyses have demonstrated that the barrier synchronization scheme efficiently synchronizes the VM and the simulation.

While the exchange of network packets between different virtualized nodes in dONE is handled using a discrete event-based interface (MPI), all clients along the lines of DieCast, TVEE, ENTRAPID and time-warped network emulation are constituted by actual (virtualized) software clients. In contrast to that, SliceTime encompasses ns-3, an event based general purpose network simulator. This allows larger amounts of clients to be modeled by the network simulator. We have shown this capability of SliceTime in Section 4.3.1.

4.4.3 Network Simulator-based Emulation Engines

Besides custom emulation engines, there are also a few network emulation tools that are based on discrete event-based simulators (cf. Table 4.5). In the following we discuss simulator-based emulation frameworks and relate them to our work. It shall also be mentioned that the emulation features of ns-3 (cf. Section 2.1.3.2) also belong to this category.

Name	**Simulation Engine**	**Protocol Support**	**Synchronization**
ns/ns-2 emulation [Fal99]	ns/ns-2	ARP, ICMP, IP, TCP, UDP	none (real-time)
wtun [Sei07, Sei08]	ns/ns-2	IP-based, custom routing protocols	none (real-time)
IP-TNE [BSUU00]	Custom PDES simulation kernel	ARP, ICMP, IP, TCP, UDP	none (real-time)
JiST/Mobnet [KBHS07]	JiST/SWANS	IP-based protocols	none (real-time)
INET Emulation Framework [TRR08]	OMNeT++	ARP, IP, STCP	none (real-time)
VirtualMesh [SGB09]	OMNeT++	802.11b, ARP, ICMP, IP, TCP, UDP	none (real-time)
Maya [ZJTB04]	Qualnet	ARP, ICMP, IP, TCP, UDP	none (real-time)
SVEET [ELL09]	SSFNet	ARP, ICMP, IP, TCP, UDP	TDF-based
SliceTime	**ns-3**	**All routing-, transport- and application-level protocols supported by ns-3**	**Conservative barrier synchronization**
TimeSync [SPL+12]	QualNet	ARP, ICMP, IGMP, IP, UDP	Adaptive TDF scheme

Table 4.5 Comparison of different simulation-based emulation environments with SliceTime.

4.4.3.1 ns and ns-2

The network simulator ns and its successor ns-2 have been the predominantly used network simulators over the last two decades. From early on, ns and ns-2 have been supporting network emulation [Fal99]. In contrast to pure link emulators or virtualization based network emulation tools, the emulation features of ns and ns-2 allow one to model the infrastructure of arbitrary computer networks, i.e. network links, switches and fully simulated hosts (cf. Chapter 2.1.3.2). Hence the simulator may provide a physical host or communication software running on the same host as the simulator with the illusion of being connected to an entire network, which however is entirely modeled by ns-2.

There are two major striking differences between network emulations based on ns-2 and ones that are set up using ns-3 and SliceTime. Firstly, in contrast to ns-2 both ns and ns-2 do not use real-world packet formats. Second, for the purpose of synchronizing the execution of the network simulation with real-world communication clients, ns and ns-2 exclusively rely on a real-time scheduler like the one of ns-3 that pins the execution of simulation events to the corresponding wall-clock time (cf. Section 2.1.5). Mahrenholz and Ivanov [MI04] have developed a more accurate real-time scheduler for ns-2. The authors report that these changes were necessary to improve the emulation accuracy of ns-2 in order to facilitate studies of wireless networks. However, the strict use of a real-time scheduler results in ns and ns-2 directly being vulnerable to overload conditions which are caused by not real-time capable simulations.

wtun: Wireless emulation for ns-2

Tim Seipold [Sei07, Sei08] has extended ns-2 in order to increase its flexibility according to the emulation of wireless networks. More specifically, Seipold has equipped ns-2 with a so-called wireless tun (wtun) interface that is governed by the simulation. The wtun interface allows real-world applications running on the same machine as ns-2 to deliver and to exchange layer 3 packets, typically IP, with the network simulation. In addition, Seipold has added emulation support for dynamic addressing to ns-2, which is important if it comes to the analysis of ad-hoc routing protocol daemons that modify the IP routing table of the system. In essence, Seipold achieves this goal by enabling ns-2 to change the configuration of the wtun interface, for example in order to change the IP address of this interface.

From a functional point of view, both Seipold's work and DDWNE pursue very similar objectives. Both provide a wireless networking interface that acts as a front-end of a network simulation, which models a wireless network using ns-2 or ns-3, respectively. The biggest difference between both tools is that wtun bridges the simulation and the communication software at the routing layer (IP), while DDWNE provides an emulated 802.11 MAC layer interface to the operating system instance on which the kernel module is deployed. As the routing layer employs state information such as node address and routing tables, wtun has to synchronize this state information between the simulation and the wtun interface. Seipold has made the design decision to declare ns-2 to be *master*; hence all configuration changes and state information such as address changes are populated from the simulation to the wtun interface. By contrast, DDWNE integrates a ns-3 based wireless network simulation with a network interface at the MAC layer. This spares DDWNE from synchronizing complex routing state information, as all layers above (routing, transport, application) are fully implemented at the operating system context. In addition, DDWNE does neither deputize the simulation nor kernel module providing the 802.11 interface to act as definitive master. Instead, the kernel module pushes configuration changes, for example issued by the user using the `iwconfig` tool, to the simulation, while the simulation posts contextual state information like RSSI values to the kernel module. This state information can then be retrieved from the wireless interface or by using the wireless extensions of the Linux kernel.

4.4.3.2 Internet Protocol Traffic and Network Emulator (IP-TNE)

IP-TNE [BSUU00] is an exciting research network emulator implemented on top of a parallel discrete event-based network simulation (PDES). The use of PDES allows the authors to speed up the execution of the network simulation in order to increase the scalability of the emulation engine and to allow for the use of more complex simulation models.

IP-TNE uses an own simulation kernel. Multiple instances of these simulation kernels are used to model a set of logical processes (LP). Each LP processes an individual event queue, and all LPs are synchronized using a variant [XUSC99] of the well-known Null-Message [CM79] Algorithm.

A shortcoming of IP-TNE that hinders the application of this emulator is its limited support for network protocols on the simulator side, as it only supports IP, ICMP and UDP as well as ARP for name resolution. However, this can be explained with the core motivation of this emulator that in fact consists in analyzing how the scalability of network emulation can be improved by making use of PDES. Kiddle et al. have demonstrated [KSU05] that IP-TNE well achieves this goal. By incorporating additional techniques such as fluid models, it is reported to scale up to network topologies of up to 200000 nodes if executed on a computer with 128 processors.

In direct comparison with SliceTime, IP-TNE follows a rather inverse approach. In order to achieve a high scalability of the network emulation, IP-TNE uses parallelization techniques to achieve a real-time execution of complex network simulation scenarios. By contrast to that, SliceTime slows down the execution speed of the virtual machines in order to match it to a (arbitrarily) slowly progressing network simulation. Both approaches have their advantages and shortcomings. IP-TNE always executes a network emulation in real-time, resulting in shorter experiment run-times and a perfect suitability for analyzing interactive applications, for example multimedia streaming. By contrast to that, SliceTime experiments may only operate at a small fraction of real-time, which sometimes makes it difficult to incorporate user interaction. However, there always will be scenarios that outgrow the computational capacity of the computer that executes IP-TNE; hence, it puts fixed bounds on the possible simulation complexity. SliceTime, on the other hand, works well with slow simulations that only operate at a fraction of real-time, resulting in non-realtime experiment execution. There is no definite answer which approach is superior for actual network emulation applications. However, it is always possible to slow down a VM by almost any degree while arbitrarily speeding up the execution of network simulations is generally impossible.

4.4.3.3 JiST/Mobnet

JiST/MobNet [KBHS07] is a network emulation tool based on JiST [BHvR05] and the Scalable Wireless Ad-Hoc Network Simulator (SWANS) [Bar04].

JiST is an advanced simulation framework that uses a dynamic classloader and binary code rewriting to execute Java programs in simulated time. JiST contains a very efficient simulation core that outperforms a number of older network simulators, among them ns-2, in terms of computation time and memory consumption [WvLW09, BHvR05]. SWANS implements a wireless network simulator on top of JiST. It contains a basic set of models for the physical, network, link and transport layers; it also provides different application and mobility models.

JiST/MobNet extends JiST/SWANS in different ways and turns it into a real-time emulation engine. In a similar fashion to Seipold's wtun, JiST/SWANS spawns virtual tun interfaces to bridge external networking clients with the simulation environment. JiST/SWANS allows one to inject packets from a real network into the simulation. For this purpose it relies on the pcap library of WireShark [Wira].

Being another representative of the class of real-time simulation based network emulators, JiST/Mobnet also suffers from the possible problem of simulation overload. In comparison with our frameworks SliceTime and DDWNE, the most apparent similarity is the motivation of analyzing wireless ad-hoc network software using network emulation. JiST/MobNet is quite well equipped for this purpose, as the underlying

SWANS provides many useful simulation models. In addition, the prevalence of mobility models would allow one to conduct network emulation experiments using virtual mobility. Unfortunately, the authors of JiST/MobNet have not published results from an actual emulation study with JiST/MobNet which renders its application potential rather unclear.

4.4.3.4 Network emulation frameworks based on OMNeT++

The OMNeT++ [VH08] simulation framework (cf. Sec. 2.1.2.3) forms the basis of different network emulators. One of the reasons why OMNeT++ is suited for network emulation is a real-time scheduler that is already bundled with the standard software distribution package of the simulator. Along the lines of the real-time schedulers developed for ns-2 and ns-3, it also pins the execution of events to the corresponding wall-clock time and pauses the execution between events if needed. Needless to say all real-time emulations based on OMNeT++ suffer from the problem of possible simulation overload. However, it is noteworthy that the real-time scheduler of OMNeT++ supports scaling its execution speed similar to using a time-dilation factor. For this reason, it would be rather straightforward to match the execution speed of an OMNeT++ simulation with a VM by throttling the VM using a static TDF. However, to the best of our knowledge there is no network emulation framework for OMNeT++ that currently makes use of time dilation for the purpose of synchronizing the simulators execution to an external entity, for example, a VM.

Despite its real-time scheduling features, OMNeT++ itself is merely a general event-based simulation framework whose software distribution does not contain specialized network simulation models. Moreover, OMNeT++ uses message objects and pointers among these to model network packets and packet encapsulation respectively. The transmission of packets between hosts is correspondingly simulated by passing pointers to these message objects. For these reasons, network emulation tools based on OMNeT++ all have to implement a set of network protocol models and a translator to bridge the disparity between real-world packet formats and the internal messaging system of OMNeT++. In the following we now discuss different emulation tools based on OMNeT++.

Emulation using the INET framework

Tuexen et al. have proposed to amend the INET framework with an external interface that allows one to connect real-world hosts to the simulation [TRR08]. The INET framework[7] is a comprehensive suite of network protocol models, ranging from wireless propagation models over many protocols of the TCP/IP stack to different application layer protocols such as HTTP.

The authors propose to achieve an integration using the libpcap packet capture library for retrieving packets. The translation between OMNeT's message formats and actual network packets takes place at a central serialization component. It needs to implement packet serialization for every network protocol to be supported by the emulation engine. Unfortunately Tuexen et al. do not explicitly state which

[7]The INET framework is available online at `http://inet.omnetpp.org/` (accessed 12/2012)

protocols are supported by their packet serializer. However they demonstrate a working emulation scenario that involves a SCTP connection between a real and a simulated host. From this it can be deducted that packet serialization was at least implemented for ARP, IP and SCTP.

4.4.3.5 VirtualMesh

VirtualMesh [SGB09] is an exciting framework for emulating wireless mesh networks using OMNeT++. More specifically, the authors aim at evaluating actual networking software in a fully simulated wireless network whose behavior is modeled using the 802.11 models of the INET framework. Like DDWNE, VirtualMesh integrates the network simulation with the software to be evaluated at the MAC layer. VirtualMesh implements this functionality by spawning an ordinary TAP networking device. In a very similar way to our PEI, MAC layer frames originating at the TAP device are sent to the OMNeT++ simulation and vice versa. In order to enable the software running on VirtualMesh to access different configuration parameters that are normally available through interfaces like the Linux Wireless extensions, VirtualMesh provides a specialized user space daemon. It implements an API for the wireless software and allows it to set wireless configuration parameters, for example it may be used to associate the system with an access point. Hence, all software that needs to control the configuration settings of the simulated wireless interface has to be changed; for this reason, the authors supply a modified version of `iwconfig` to interact with the simulated interface. However, normal networking software may be executed in a VirtualMesh emulation scenario without any changes. The authors report that they have successfully operated ssh, ftp and scp on top of VirtualMesh.

In fact, VirtualMesh provides almost the equivalent functionality of DDWNE, as both allow one to execute real-world communications software in a wireless network, of which the packet propagation and the MAC layer is entirely modeled by a discrete-event based network simulator. However, there are three distinct differences that set DDWNE apart from VirtualMesh. First, DDWNE fully emulates the operating system front-end of a wireless networking interface. This way, any legacy software, for example unmodified versions of Kismet of `iwconfig`, can be executed on top of the simulated wireless network. In contrast to that, VirtualMesh amends ordinary TAP devices with an additional configuration interface, which results in mandatory changes to all kinds of software that need access to configuration parameters of the wireless interface. Second, DDWNE benefits from the strong emphasis of ns-3 on emulation features and message compatibility. The ns-3 framework guarantees real-world packet formats by design. Therefore, all software executed on top of a wireless interface spawned by DDWNE is able to interact with its simulation counterpart.

By contrast, the authors of VirtualMesh have extended the 802.11 models of the INET framework to enable one to pass traffic through the simulation. The interaction of simulated hosts with real-world software, however, may still require additional packet translations and serializations to be implemented. Finally, VirtualMesh is strictly limited in terms of scalability, as it is bound to real-time execution. DDWNE is fully interoperable with SliceTime, allowing it to be incorporated with wireless simulations that execute much slower than real-time.

4.4.3.6 Maya

One of the most universal approaches in the field of network emulation and network modeling is Maya [ZJTB04]. Maya integrates three different performance evaluation methodologies in one hybrid framework:

- Discrete event-based simulation models based on QualNet[8] are used to model an IP-based network topology, similar to network emulation scenarios in ns-2 or ns-3.
- The fluid model by Liu et al. [LLPM+03] is employed for modeling large-scale TCP networks.
- A physical testbed infrastructure allows real-world software to be tested in a network provided by both analytical and event-based models.

The core difficulty Maya has to handle is the integration of these techniques. The authors use an event-based subsystem for this purpose, which eases the integration of existing event-based simulation models. In order to integrate the analytical model with the physical testbed, important properties of the network traffic, for instance packet sizes or IP addresses are extracted and fed to the fluid model. The output of the fluid model is then used to schedule according simulation events.

Comparison of Maya with our work

The great strength and contribution of Maya for sure is its integration of techniques from all three domains of performance evaluation, as earlier discussed in Chapter 2. To the best of our knowledge, Maya is the only hybrid evaluation framework that is able to integrate physical testbeds with an analytical performance model.

The biggest difference between Maya and SliceTime is that Maya operates strictly in real-time, resulting in the need in for computing both the output of the fluid model and the event-based simulators in real-time or faster. Like with all real-time emulation frameworks built on top of an event-based system this limits the scalability of the framework. In contrast to that, SliceTime and all other SHE frameworks do not require any system representation to be real-tim e capable. Hence, our frameworks do not suffer from such scalability constraints.

4.4.3.7 Scalable Virtualized Evaluation Environment for TCP (SVEET)

To the best of our knowledge, the first emulation environment based on a discrete event-based simulator that incorporates time virtualization is SVEET [ELL09]. SVEET is an emulation environment specially tailored towards the analysis of TCP implementations.

[8]QualNet is a commercial network simulator. More information is available at `http://web.scalable-networks.com/content/qualnet` (accessed 02/2013).

The network simulator serving as implementation basis for SVEET is PRIME[9], which itself builds on the earlier SSFNET [CNO99]. PRIME extends SSFNET in different ways. Most notably, PRIME adds a real-time scheduler that also supports scaling its execution speed in similar way to OMNeT++. For this reason, PRIME enables the execution of a discrete event-based network simulation in real-time or at a specified fraction of it. Besides this scaling support, PRIME also adds a number of protocols to SSFNET, ranging from link layer models to ones of application layer protocols.

SVEET relies on Xen-based Linux VMs as its second building block. In a similar way to our Packet Exchange Interface, the authors establish a tunnel between the network simulation and the virtual machines.In contrast to SliceTime, which wraps MAC frames in UDP packets, SVEET uses OpenVPN [Bau10] to tunnel IP packets between the simulation and the virtual machines. As PRIME uses custom message formats inside the simulation domain, an according translation is performed by the simulator for incoming and outgoing traffic.

In contrast to all other discrete event-based simulation frameworks except for SliceTime with ns-3, SVEET incorporates a mechanism to circumvent simulation overload situations. The authors of SVEET propose to use a static TDF factor to slow down both the VM and the network simulation to a fraction of wall-clock time at which the simulation safely executes without an occurrence of overload. For example, if a network simulation executes between two and five times slower than real-time, the execution of the VM and the simulation is throttled by a factor of five. In order to implement this feature for Xen-based VMs, the authors resort to code from Gupta et al. [GYM$^+$06].

This approach of synchronizing the execution speed of the VM with the run-time performance of the network simulation has one major drawback. The static Time Dilation Factor (TDF) used for throttling the execution speed of the setup needs to be specified beforehand. Determining a suitable TDF is very delicate, as predicting the future execution speed of a network simulation is difficult if not generally impossible. This is especially a problem because the execution performance of a network simulation is also depending on the traffic that is passed to it by the VM.

But what is a "suitable" TDF? First of all, if the TDF is chosen in a too conservative fashion, both the VM and the network simulation are slowed down to a small fraction of their actually possible execution performance. Such overly conservative time dilation factors lead to a potentially strong resource under-utilization that entails a rather sluggish execution of the entire emulation setup. If the TDF, however, is chosen in a too optimistic fashion, a mismatch in execution performance and possible simulation overload are the straight consequence. As the performance of a network simulation is rarely constant in terms of execution speed, SVEET always results either in possible simulation overload if the TDF is poorly chosen or a varying degree of resource under-utilization.

Due to these reasons, the approach of SVEET can be categorized as a best effort synchronization scheme that is not able to prevent or to recover from mismatches in execution speed. By contrast, the barrier algorithm of SliceTime guarantees the

[9]More information on PRIME is available online at `http://www.primessf.net/` (accessed 12/2012).

time drift between the VM and the network simulation to be strictly bounded; it is a conservative synchronization algorithm. Moreover, SliceTime allows the network simulation and the VM to progress at the maximal possible execution speed during the assigned time slice. The observed slowdown of a SliceTime setup hence is not caused by the application of a TDF. In fact it a consequence of a “faster” system representation such as a VM being regularly blocked at the end of a time slice.

4.4.3.8 TimeSync

A very recent emulation framework that synchronizes the execution of an event-based network simulation with software prototypes is TimeSync [SPL$^+$12]. It was published after SliceTime and shares some similarities with our work.

Along the lines of SliceTime, the foremost goal of TimeSync is to enable emulation studies employing complex scenarios based on a general purpose simulator (in this case QualNet) that are not real-time capable. The authors tackle this problem by providing a common and virtualized progression of time to a set of Xen-based VMs and to the simulation. However, the synchronization approach of TimeSync differs significantly from our work.

The authors of TimeSync do not rely on a dedicated synchronization unit. Instead, they apply a master-slave scheme by declaring the simulation to be the global time controller that governs the time progression on the VMs.

The synchronization between the simulation and the VMs is based on a dynamically adjusted slowdown factor that is pushed to the VMs at regular intervals. The slowdown factor is determined by sampling the execution performance of the network simulation. Hence, we categorize the used synchronization algorithm as a dynamic TDF-scheme, similar to the one proposed by dONE [BVB06] (cf. Sec. 4.4.2.2).

Like with the earlier discussed approaches that make use of dynamic TDFs, the most crucial parameter in a TimeSync setup is the sampling frequency that determines the slowdown of the network simulation. If it is chosen too coarsely the TDF may be adjusted not early enough to react to sudden and potentially drastic changes in the simulation performance. By contrast, SliceTime employs a fine-grained lockstep synchronization algorithm, which strongly bounds the time drift to the size of one time slice.

4.4.4 Hardware-based Network Emulation

All network emulation engines discussed up to this point are entirely based on software, no matter if the emulation engine reproduces only the behavior of a link or an entire computer network. In the following, we discuss network emulation approaches based on specialized hardware and distinct emulation testbeds (cf. Table 4.6).

4.4.4.1 The Carnegie Mellon University (CMU) Wireless Emulator

The CMU Wireless Emulator [JS05] is a hardware-based link emulation engine for wireless networked devices. In contrast to all other emulation frameworks discussed

Name	Operation Purpose	Supported Endsystems	Scalability
CMU Emulator [JS05]	System evaluation using a fully emulated wireless channel	PCs using WiFi cards and GNU software radios [Blo04]	15 nodes
Emulab [WLS+02]	Emulation testbed for the evaluation of IP-based network services and applications	Linux, FreeBSD & Windows nodes	$\geq$ 550 nodes
ModelNet [VYW+02]	Software suite for using a server cluster as emulation testbed	FreeBSD, Linux	100 node deployment reported by authors

Table 4.6 Overview of different hardware-centric emulation platforms.

in this thesis, the CMU emulator reproduces the behavior of wireless links at the physical layer. Hence, it allows one to evaluate actual MAC-layer implementations, for example 802.11 devices, in fully controlled environment.

In order to achieve this goal, the CMU Emulator has to be connected with the systems to be evaluated at the physical layer. It is tailored towards 802.11 networks. The CMU emulator is connected to wireless communication systems by redirecting their antenna output to the emulation engine. This task is carried out by so-called Radio Frequency (RF) nodes. Inside a RF node, the radio signal is first routed through a frequency divider to obtain a Low Frequency (LF). This LF signal is then digitized using an A/D converter. The samples obtained from the A/D converter are then passed to the actual emulation engine. Similarly, samples received from the emulation engine are restored by the RF nodes to the original wireless frequency range by running them through an D/A converter and a corresponding frequency multiplier.

The core processing of the radio samples obtained from the RF nodes happens at a central emulation engine. As this emulation engine has to reproduce the behavior of the shared wireless medium in real-time, the authors resort to specialized hardware for this task. More specifically, the emulation engine of the CMU emulator is a complex digital signal processor (DSP) based on FPGAs. The DSP engine is able to model effects like fading, path loss and signal fluctuations due to node mobility in real-time.

In addition to the RF nodes and the DSP-based emulation engine, the CMU emulator also contains a high-level emulation controller. It allows one to specify different aspects of the emulator's behavior, for example node movement or the nodes' application behavior. For this purpose, it provides both a GUI and a scripting interface.

Apparently, the CMU emulator differs from our frameworks SliceTime and DDWNE in many ways. First of all, it is noteworthy that the CMU emulator operates at the PHY layer and hence allows one to investigate MAC layer implementations, no matter if they are realized in hard- or software. In contrast to that, DDWNE uses a simulated 802.11 MAC layer to bridge the communication among different nodes. Hence, effects like the interference of other wireless technologies like Bluetooth with a data transfer between two VMs are difficult to model with our framework; such studies can be carried out using the CMU emulator. One downside, however, of

```
set ns [new Simulator]  #Instantiate simulation unit
source tb_compat.tcl    #Include Emulab core library
set node1 [$ns node]    #Define two nodes and one router
set node2 [$ns node]
set router [$ns node]

#Configure links
$ns duplex-link $node1 $router 100Mb 2ms RED
$ns duplex-link $node2 $router 10Mb 20ms DropTail

#Configure operating systems
tb-set-node-os $node1 FBSD-STD  #Node 1 uses FreeBSD
tb-set-node-os $node2 RHL-STD   #Node 2 uses RedHat Linux

#Execute experiment on Emulab
$ns run
```

Listing 4.1 Example Tcl script used by Emulab to setup a simple emulation scenario (Adapted from [emu])

the CMU emulator is that it requires physical hardware for each wireless node. Hence, conducting wireless emulation studies with many nodes is costly compared to emulation approaches such as SliceTime/DDWNE that operate using virtualized resources.

4.4.4.2 Emulab

One of the most comprehensive projects in the domain of network emulation is Emulab [WLS+02]. Emulab can be described as a heterogeneous network testbed that aims at flexibly reproducing the topology and the behavior of arbitrary computer networks. For this purpose, it integrates a diverse set of networking hardware and different software emulation tools to mimic the desired target network. The feature set and the infrastructure of Emulab has been steadily extended over the past years [HR12]. Nowadays, Emulab allows one to construct network emulation scenarios using the following building-blocks:

- **Physical Hardware**, precisely PCs equipped with different network interfaces (Ethernet, partially WiFi) and programmable network switches, is the most important type of resource available to EmuLab users. In addition, link emulators such as the earlier described Dummynet (cf. Sec. 4.4.1.1) are used to reproduce the required link behavior. The Emulab deployment at the University of Utah nowadays consists of approximately 600 nodes that are interconnected using 13 Ethernet switches [HR12].
- **Virtual Nodes** based on OpenVZ [ope] containers can be used to multiplex the capacity of Emulab's physical resources. According to the documentation [emu] virtual nodes enable between 10 and 20 times more application instances to be executed on a physical machine of Emulab.
- **Xen-based Virtual Machines** can be used to multiply the capacity of the testbed as well and provide the user with the possibility of incorporating kernel-level applications into the emulation scenario.

- In the past, Emulab also supported **simulated resources** by incorporating the emulation facility of the ns network simulator [Fal99], resulting in the possibility of modeling network links, nodes and traffic sources in the simulation domain. A core motivation was to take advantage of the abstract modeling approach of network simulators for the purpose of gaining a higher scalability. However, the Emulab documentation [emu] states that support for simulated resources has now been abandoned.

Emulab is a publicly shared testbed. It allows its users to specify an emulation experiment with a Tcl[10]-based scripting language whose syntax is very similar to the scripting language used to describe ns and ns-2 simulations. Listing 4.1 displays an example of such a script. It creates a very simple topology of two connected nodes running FreeBSD and Linux, respectively.

One of the most exciting aspects of Emulab is that it automatically maps these Tcl scripts to the available testbed hardware and configures the infrastructure accordingly. The problem of mapping an Emulab description to the physical testbed resources in an optimal way is NP-hard; the authors of Emulab have developed a randomized solver based on simulated annealing [RAL03]. The solver is conservative and prevents mappings that cannot be fulfilled using the available physical resources.

In our example, Emulab would automatically instruct the programmable switches to place the nodes and the router into Virtual Local Area Networks (VLANs). Dummynet would be triggered by the Emulab software in order to model the desired link behavior. In addition, Emulab would also bootstrap two nodes running the desired operating systems before starting the execution of the experiment.

Over the past decade Emulab has successively been enhanced in different ways. For example, Johnson et al. have extended Emulab in order to support controlled node mobility [JSF+06]. A mobile node consists of a moveable robot platform carrying a wireless sensor mote, a Stargate computer and a 802.11 interface. In order to control the nodes' movement, the authors have added corresponding commands to the Tcl-based scripting language of Emulab. Later enhancements of Emulab facilitate the accurate emulation of Internet paths [SDRL09] or a live calibration of the emulated network by incorporating measurements from Planetlab [RDS+07].

Emulab Usage

A recent long-term study [HR12] investigates the properties of 500.000 network topologies that have been submitted to Emulab during 13000 experiments over the last decade. The outcome of this study reveals interesting general patterns about how Emulab is used by researchers and developers.

A very noteworthy result is the finding that most emulation experiments conducted with Emulab require only a small number of nodes. However, a rather significant portion of all topologies is also very large. The authors attribute this observation to many smaller experiments being used for preparative tasks. The larger topologies are put down to fewer but actual emulation trials. This is a very exciting result with

[10]Tcl stands for *Tool command language*. It is a scripting language originally developed at the University of California Berkeley in the late 1980s.

a view to SliceTime. SliceTime can be seen as an extension for the ns-3 emulation facility, as it enables large-scale simulation-based emulation studies. It can be expected that many small-scale emulation studies are able to execute in real-time, thus taking away the need of synchronizing the execution of a VM and ns-3. Before SliceTime was available, crossing the boundary at which a network simulation stopped to be real-time capable inevitably forced users of a simulation-based emulation tool to alter the methodology of performance evaluation, for example by using simulation models of less computational complexity. SliceTime takes away this burden. If issues like simulation overload are observed in a real-time simulation, the entire setup can be easily changed to use SliceTime and no further changes to the chosen performance evaluation method need to be applied.

Comparison of Emulab with our work

Directly comparing Emulab to our work is difficult, as both projects differ very much, both conceptually and regarding their scope. Emulab is a very mature network emulation testbed that has supported numerous scientific studies in the domain of computer networks. In contrast to that, SliceTime is a new emulation technique that broadens the applicability of simulation-based emulation experiments. What unites SliceTime and Emulab, however, is their common goal of providing a flexible emulation environment to their users. SliceTime aims at achieving this goal by enabling users to create emulation scenarios that can make use of almost arbitrary network simulations; this property of SliceTime is also rooted in the high flexibility of ns-3, which offers a wide set of network protocol models and provides emulation features out-of-the-box. Emulab, by contrast, gains its flexibility from the large amount of hardware that is available in the testbed. As the network of Emulab is some order of magnitude larger than a SliceTime setup operating an Emulab-like testbed of course is also magnitudes more costly. On the other hand, the Emulab infrastructure is publicly available, thus giving everybody the possibility of running their experiments on this mature emulation infrastructure. Similarly, SliceTime is publicly available for download to provide anyone with the possibility of conducting large-scale simulation-based emulation studies.

4.4.4.3 ModelNet

A project that resembles Emulab in many ways is ModelNet [VYW+02]. Along the lines of Emulab, ModelNet has developed a network emulation software suite, which is able to reproduce the behavior of different topologies on a server cluster using a network specification. This network specification is automatically translated and mapped to the hardware that is available within the ModelNet testbed.

Another focus of the ModelNet project is increasing the emulation scalability of physical network testbeds. The project has developed and investigated a number of exciting concepts over the past years, for example intelligent topology partitioning strategies [YED+03] or optimized routing schemes [CGV+04], which both are able to boost the scalability of ModelNet. In addition, the previously discussed virtualization-based emulation tools DieCast [GVM+11] and Time-Warped Network Emulation [GYM+06] were also developed by the ModelNet research group.

A comparison between ModelNet and our work on DDWNE and particularly SliceTime is intricate because of their diverging approaches to network emulation. ModelNet defines network emulation as the process of recreating a target network's characteristics on a physical network testbed, while SliceTime and DDWNE model large portions of the emulated network using a network simulator. Mastering the common challenge of enhancing the scalability of the respective emulation technique hence yields to widely differing concepts. However, both projects have developed technologies, for example DieCast and SliceTime, that enable novel kinds of emulation studies that were impossible to conduct before their respective publication.

4.5 Interim Conclusion

In this chapter, we have presented two hybrid emulation frameworks, namely SliceTime and Device Driver-Enabled Wireless Network Emulation (DDWNE).

DDWNE realizes the idea of hybrid system representations (cf. Sec. 3.2.6). It facilitates the evaluation of legacy Linux networking software in a wireless network, whose PHY and MAC layers are fully and deterministically modeled by a network simulator. This enables emulation studies, in which actual networking software can be evaluated in a fully-simulated deterministic and reproducible wireless environment. Our evaluation of DDWNE has shown that our according implementation of 802.11 is accurate enough to support actual evaluations of WiFi software.

SliceTime implements the concept earlier introduced as Synchronized Network Emulation (cf. Sec. 3.4.1). It enables the detailed analysis of protocol implementations and entire instances of operating systems inside simulated networks of arbitrary size. We achieve this goal by matching the execution speed of software prototypes encapsulated in virtual machines to the run-time performance of the event-based simulation. Our evaluation has shown that SliceTime is accurate as it integrates simulations of any size with VM based prototypes regarding timing and network bandwidth in a transparent way.

We regard SliceTime to be resource efficient. We model large parts of the experiment with a simulation and match its overall execution speed to the available hardware resources. This enables large-scale network emulation studies at moderate hardware costs, especially if compared to equally sized physical testbeds.

As shown by our use case scenarios, SliceTime opens up new application areas for network emulation. In the past, only event-based simulations executing in real-time could form a basis for network emulation. This is not true for the vast majority of network simulations. For example, the computation complexity of 802.11 channel models so far hindered the use of network emulation for larger WiFi scenarios. By eliminating this burden of real-time execution, SliceTime allows any simulation to be used for network emulation. We have demonstrated that this extends the applicability of network emulation to large-scale WAN and 802.11 scenarios. Our reproduction of a large scale AODV measurement using DDWNE and SliceTime together has also demonstrated that both emulation concepts are well interoperable with each other.

5

A hybrid evaluation framework for networked embedded systems

Overview

This chapter describes a framework [SWK+10] for synchronized hardware-network co-simulation (cf. Sec. 3.4.2) of embedded systems. Such a hybrid solution enables the co-design of software and hardware, particularly for embedded systems, to be carried out using a repeatable and flexible network model.

The results presented in this chapter are a result of joint research conducted together with the Institute for Communication Technologies and Embedded Systems (ICE) at RWTH Aachen University.

Structure of this Chapter

Following a brief motivation we first describe the conceptual design of our framework for synchronized hardware-network co-simulation. We later discuss our SystemC-based system representation that enables arbitrary SystemC simulations to be included into a SHE composition. The second part of this chapter presents a performance evaluation of our tool chain and discusses relevant related work.

5.1 Motivation

Over the past decade, embedded systems with communication features have become a pervasive reality. For example, cellular phones and Wi-Fi home routers are embedded systems with the core task of providing network access to end users. Many multimedia devices such as set-top boxes or portable media players implement communication features for the purpose of retrieving additional content from the Internet or for sharing media data with other users.

The actual design of embedded systems, especially their core architecture, is often carried out using Virtual Platforms (VPs), which are full system simulators of hardware platforms and the corresponding software. Hence, we will use the terms virtual platform and full system simulators synonymously in the following.

Well-established tools in the area of hardware-software co-design, for example Virtutech Simics [MCE+02] or CoWare Platform Architect [cow], enable the flexible composition of new system designs. These consist of IP (Intellectual Property) blocks, for example processor cores or communication architectures, as well as custom hardware modules. Virtual platforms simulate the hardware of the entire embedded system at its present design state. This enables the execution of actual software within these environments. Therefore it becomes possible to develop system software like device drivers or operating systems concurrently to the hardware design phase. Moreover, the ability to observe and influence the behavior of the simulated system arbitrarily, e.g., by setting breakpoints and possibly changing the state of the system in a reproducible fashion, facilitates the robust implementation of both system software and hardware. However, VPs are difficult to employ for the investigation of large network scenarios that involve many communication peers, as simulating the entire system hardware for every host node is not feasible due to the high computational effort.

Embedded devices with networking functionalities often are resource constrained, for example in terms of available energy, system memory and CPU performance. In order to validate the resource effectiveness of corresponding hardware and software, the early analysis of such in the design cycle is vital. For this purpose, we propose the integration of virtual platforms with network simulations, aiming at the network centric design of embedded system software and corresponding hardware. We use network simulation for the context provisioning to the VP. This way, it becomes possible to analyze the system behavior given stimuli that closely resemble real-world network scenarios. Moreover, the detailed system model of today's VPs makes it possible to overcome the modeling limitations of current network simulations, especially according to the end host behavior and related performance metrics, such as CPU utilization, bus load, energy consumption and memory usage.

5.2 Conceptual Design

Conceptually, our framework (cf. Fig. 5.1) for the network centric design of embedded systems forms an implementation of synchronized hardware/network co-simulation as earlier introduced in Section 3.4.2. For this reason, it contains the synchronization component and two types of System Representations (SRs), a network simulation and a Virtual Platform (VP) that is modeled using a full-system simulator (FSS).

5.2.1 Synchronization Component

In order to align the execution speed of the virtual platform and the network simulation, we resort to the synchronization component of SliceTime (cf. Sec. 4.1). The synchronization process is based on discrete time slices that are provided to the VP

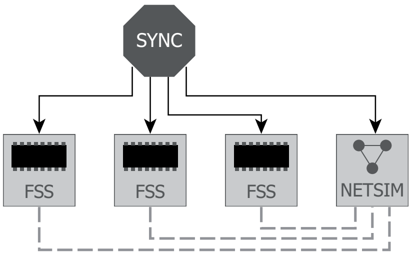


Figure 5.1 Our framework comprises two types of system representations, a network simulation and a virtual platform modeled using a Full-System Simulator (FSS). To enable the incorporation of the FSS into the synchronized hybrid evaluation setup, it needs to be interfaced with the synchronizer via the Time Control Interface (TCI) and with the network simulation using the Packet Exchange Interface (PEI).

and the network simulation. After both have processed the assigned time slice, they report the completion of the slice back to the synchonization component, which then in return subsequently assigns a new time slice. We further discuss this so-called barrier synchronization scheme in Section 3.1.3.

5.2.2 Network Simulation

The role and the functionality of the network simulation is exactly the same as in a synchronized network emulation framework such as SliceTime. The network simulator models an arbitrary communication network that interconnects different virtual platforms. In order to be incorporated into a synchronized hybrid evaluation framework, it also implements the synchronization protocol to follow the barrier synchronization scheme and the packet exchange interface in order to exchange network packets with other system representations. We provide a more elaborate discussion on these matters in Sections 3.2.3 and 4.1.2.4.

5.2.3 Virtual Platform (VP)

A Virtual Platform (VP) [cow] or virtual system prototype [Hel99] defines a behavioral model of a system at different levels of abstraction. Virtual platforms are mostly used as an executable specification in order to support software and hardware development before a hardware prototype is available. One of the core concepts of VPs is to model the entire system hardware in software. This allows for non-intrusive debugging of the system prototype with a high degree of control and observability according to the system software and hardware state.

Finally, the Virtual platforms are typically modeled using SystemC [JA06] and the Transaction Level Modeling standard 2.0 (TLM-2) [Ayn09]. Here a complete hardware platform is a collection of different components that are connected in order to execute a certain function, e.g. an entire wireless handset. Fundamental building

```
void SyncVP_thread() {
  while (true) {
   do {
    msg = recv(SYNCHRONIZER);
   } while (msg.type != RUN_PERMISSION);
   wait(msg.timeslice_size);
   send(SYNCHRONIZER, ACK);
  }
}
```

Listing 5.1 Pseudocode of the main loop of the SyncVP component

blocks are processors (e.g. RISC, DSP), communication architectures (e.g. bus, crossbar) and components for data exchange like packet radios.

In order to integrate a virtual platform into a synchronized hybrid evaluation (SHE) framework as a SR, we first need to integrate it with the packet exchange interface (PEI) in order to make the exchange of network packets between the VP and the network simulation or potentially other system representations possible. Since any form of communication with peers outside the VP domain may only be carried out using communication interfaces, we bridge the SystemC environment with the network simulation using a dedicated network interface. For this purpose we harness the component based design principle of virtual platforms and integrate the VP with the PEI using a dedicated *NetChip* IP component based on SystemC. The NetChip component forms a virtual network adapter that looks to the virtual platform like a standard radio component, but on the other hand bridges the gap to the network simulation over the packet exchange interface.

The second requirement to incorporate a VP into a SHE tool-chain is the integration with the synchronization component and the time control interface. This task is carried out using the *SyncVP Component*. This standard SystemC component forms a virtual device that can be easily embedded into an arbitrary SystemC simulation. It requires no connection or further hardware specific configurations. The component implements the synchronization protocol (cf. Sec. 3.1.4) and puts the barrier synchronization scheme into effect by blocking the SystemC simulation until the next time slice is assigned.

5.3 Implementation

The implementation of our framework for the network centric design of embedded systems reuses the synchronization component and the ns-3 SR of SliceTime, which are described in Chapter 4. For this reason we now focus on the implementation of our virtual-platform based system representation in the following. Specifically, we describe the two SystemC components that cooperatively enable arbitrary SystemC-based VPs to be used as system representations.

5.3.1 The SyncVP Component

The so-called *SyncVP Component* implements a client for the barrier synchronization protocol (cf. Sec. 3.1.4) and thus integrates any SystemC-based VP of choice with the time control interface. The SyncVP component does not provide any SystemC ports nor does it require any connections to other SystemC modules. Existing SystemC designs are not required to be changed in any other way. The SyncVP component carries out its task using only standard SystemC features. As the pseudocode in Listing 5.1 illustrates, it executes the following four subtasks in an endless loop:

1. **Reception of a Run Permission Message**
   The synchronizer sends an UDP packet containing a run permission message whenever it assigns a new time slice to the VP. As SystemC is an extension of C, the SyncVP component can use a standard Berkeley Socket [WPR+04] to receive incoming messages.

2. **Suspending the Simulation**
   Before a time slice has been assigned by the synchronizer, the entire simulation has to be halted. A common SystemC module can accomplish this by not returning control to the SystemC scheduler until the time slice has been assigned. This way, the SystemC kernel does not know if any events will be scheduled before the code of the module returns control, so the kernel cannot execute any events after the current point in time. The SyncVP component executes blocking reads on the communication socket, which halts execution without busy waiting. Execution continues in the code of the SyncVP component when a run permission message is received.

3. **Running the Simulation**
   By putting itself to sleep for the duration of the assigned time slice, the SyncVP component is able to advance the simulation. This is done by calling SystemC's `wait()` function with the duration of the slice as parameter. All other parts of the simulation will continue to run for this time. Afterwards, the `wait()` function returns and control is passed back to SyncVP component.

4. **Acknowledgement of Time Slice Execution**
   After completing a time slice, the SyncVP component notifies the synchronizer using a corresponding message over the time control interface.

5.3.2 Virtual Network Chip

The *Virtual Network Chip* implements the communication with the network simulator on the VP side. It has been modeled to work like a real network chip, except that the network link is modeled using a UDP based tunnel protocol for simulated Ethernet frames.

Figure 5.2 displays the internal structure of the network chip. In the VP environment, it provides a bus target interface and an interrupt initiator port. The bus interface enables the access to a memory buffer of 16kB and to four control

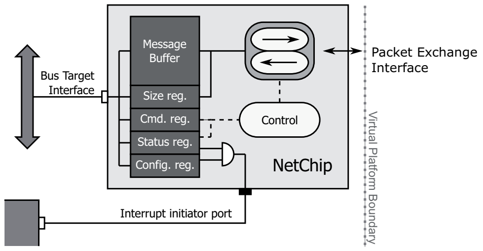


Figure 5.2 Internal structure of the Virtual Network Chip. This SystemC component models a network chip. It connects to the packet exchange interface, to which other SRs such as a ns-3 simulation can be connected. The chip model delivers and obtains network packets from a system bus that is connected to other parts and peripherals of the VP (e.g., a virtual CPU).

registers: a *configuration register*, a *status register*, a *command register* and a *size register*. Incoming messages are not directly written to the message buffer in order to retain full control of the buffer via the bus interface. Instead, a reception is indicated in the status register. The command register can be used to make a received message available in the buffer and to send the buffer contents as a new message. The configuration register allows configuring interrupts for the flags set in the status register.

5.4 Evaluation

We now evaluate our framework using a straightforward setup that comprises one virtual platform (VP), one ns-3 network simulation and the synchronization component. The simulated network consists of one link, connecting a simulated network host with a gateway node that integrates the VP with the network simulation over the packet exchange interface. For synchronization, the VP utilizes the SyncVP Component and the rest of the system is built based on the Virtual Processing Unit (VPU) technology [cow]. In this scenario, a single VPU instance represents a generic RISC processor core, which is connected via an Advanced High-Performance Bus (AHB) clocked with 10 MHz to a memory subsystem and to the NetChip IP component, whose interrupt line is connected to the VPU.

The software executed on the Virtual Processing Unit (VPU) is a port of the micro IP (uIP) stack [Dun03] extended with timing annotations to model the timing behavior of packet processing on an embedded processor core.

The setup used for evaluation consists of a version of our ns-3 system representation (cf. Sec. 4.1.2.4) and CoWare Platform Architect [cow] version 2009.1.1 to run the SystemC simulation. All measurements have been performed on an AMD Athlon 64 X2 with 3GHz clock frequency and 6GB main memory running Scientific Linux 5 as operating system.

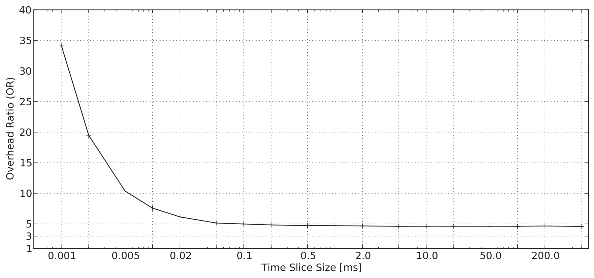


Figure 5.3 Synchronization overhead of the VP given different time slice sizes. The synchronization adds only a small amount of overhead for time slice sizes $\geq$ 0.1ms. For small time slices, the overhead grows due to increasing messaging overhead.

5.4.1 Virtual Platform Performance

We now investigate the performance of our virtual-platform based system representation and its dependency on the choosen time slice size. As discussed previously in Section 3.1.3.2, the time slice size directly corresponds to the synchronization accuracy. While smaller time slices for this reason allow a more precise time integration of the VP-based system representation and the network simulation, the execution is also slowed down due to a higher messaging overhead (cf. Section 3.1.4.1). For measuring the effect of time slice size Δt on the execution performance and the synchronization accuracy, we synchronize the VP and the network simulation with Δt ranging from 1μs to 500ms. During each measurement run, the host modeled by the ns-3 network simulation sends 100 ICMP echo replies (pings) to the VP at a rate of 1 message per second. The ping messages carry a payload of 56 bytes each.

Analog to the overhead evaluation of SliceTime (cf. Sec. 4.1.3.3) we quantify the performance overhead using the overhead ratio (OR) by dividing the wall-clock time consumed by the VP during the measurement run by the total assigned virtual runtime. The overhead ratio for the measured time slice sizes is plotted in Figure 5.3.

For time slice sizes of 0.1 ms and less the overhead ratio converges because the time needed for synchronization is getting smaller compared to the time needed for execution of the simulation for one slice. The execution performance is no more increasing for slices $\geq$1 ms as the synchronization overhead is negligible in relation to the simulation itself.

5.4.2 Influence of Synchronization Accuracy

The accuracy of simulation coupling manifests in the observed time deviations between network simulator and VP. Every simulated network packet exchanged between network simulator and VP is seen once in the network simulator and once in the VP. We denote the time of a network packet entering/leaving the network simulator by T_{NS} and the time of it entering/leaving the Virtual Platform (VP) by T_{VP}. Thus, the time deviation is $\delta := |T_{NS} - T_{VP}|$. Figure 5.4 shows the distribution

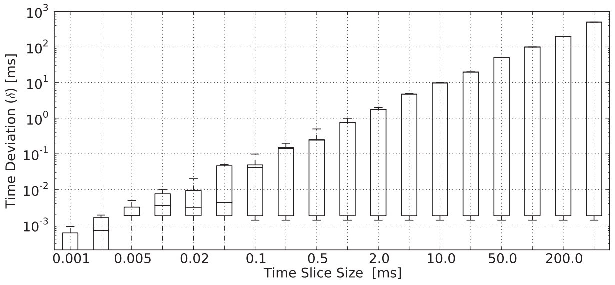


Figure 5.4 The time deviation between the virtual platform and the network simulation is always bounded by the time slice size. This affirms that the size of the time slice directly corresponds to the synchronization accuracy.

of the deviations δ observed for the 200 network packets exchanged in the test case described above given the time slice size. The width of the boxes corresponds to the range between the 25% and the 75% percentiles.

The boundedness of δ is the most important fact observable in the graph: The time deviation δt is always less than the time slice size, which shows that the barrier synchronization scheme limits the time drift to the given time slice size.

For small slice sizes ≤ 0.002 ms, a lot of messages are observed in the network simulator and in the VP at the same time, the 25% pecentile lies at the bottom of the plot. This is caused by the observed deviation being almost zero for a large percentage of packets. Exact timing still occurs occasionally up to a slice size of 50μs as it can be seen from the minimum touching the bottom. However, the 25% percentile is shifted towards the maximum, indicating that most messages are already observed with a time deviation. For time slice sizes greater than 0.1 ms most messages experience almost the full time deviation possible. Additionally, no message is arriving at the exact point in time any more.

Both effects observed in this measurement can be explained by looking at the simulation speed of the network simulator and the VP. As the most dominant timings in the network simulator are in the range of milliseconds, whereas the VP deals mostly with timings below microseconds, the VP has to process considerably more events per simulation time slice than the network simulator. The network simulator being significantly faster than the VP results in the VP having just started the simulation of a time slice when the network simulator has already completed it. Thus all packets sent from the network simulator to the VP will be seen on the VP at the beginning of the current time slice and all packets sent from the VP to the network simulator will not be detected by the network simulator until the beginning of the next slice. However, if the network simulation would be of higher computational complexity and thus be less faster than the VP, this effect would be less distinctive.

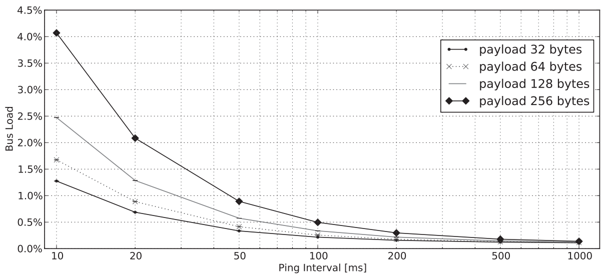


Figure 5.5 Bus Load on Virtual Platform.

5.4.3 Influences of Network Traffic on the VP Bus Load

One main motivation of this work is to support analyzing network induced effects on the VP. To illustrate this using a simple example, we measure the effects of incoming ping requests onto the load observed on the VP.

We first fix the time slice size used for synchronization to 0.1ms, which provides the best accuracy that can be obtained without significantly impairing performance. We now vary the ping payload length from 32 to 256 bytes and the ping interval from 10 ms to 1 s. For every configuration, we measure the bus load of the VP, which is plotted over the length of the ping interval for the four different payload lengths in Figure 5.5.

With shorter intervals between the ping requests, the number of network packets to process increases, which leads to a higher bus load on the VP. For small ping intervals of up to 100ms, the load is almost proportional to the number of packets to process per time unit and thus reciprocal to the length of the ping interval. For longer intervals, the load imposed by the ping is no more significantly larger than the background load on the VP unrelated to processing network packets. Thus, the decrease of the load is getting smaller for large intervals. The effect of the payload length is similar to the effect of the ping interval. With increasing payload size, the load rises. For small payload sizes of 32 and 64 bytes, doubling the size only leads to a growth in load of about one third. The reason is the constant load needed for reacting to a packet reception and for processing the header. In contrast, the bus loads for payload lengths 128 and 256 differ by almost a factor of two, because the header processing overhead is less predominant for those larger packet sizes.

As the VP was not modified or reconfigured during those measurements and all adaptions were only done to the network simulator, this example clearly demonstrates how network effects influencing the status of a platform can be analyzed using the proposed approach.

5.4.4 Influence of Payload Processing on Round Trip Time

To show that the effects on the platform also impact the network, we measured the round trip times (RTTs) of the pings in the network simulator. These measurements

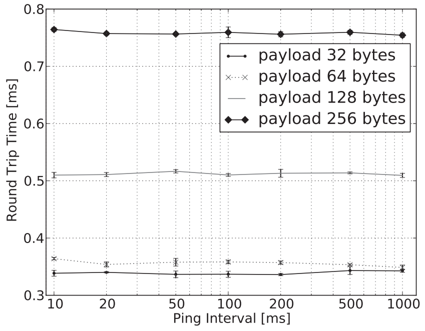


Figure 5.6 RTTs observed by Node inside Network Simulation.

were taken from the simulation runs which were also used for measuring the bus load on the VP. Figure 5.6 depicts the measured round trip times on the Y-axis of the diagram for different ping intervals.

The payload length influences the response time of the VP and thus the round trip time seen in the network simulator. Processing of longer packets takes longer as more data transfer has to happen between network chip and processing core and more computation has to be performed, for example for calculating the checksums of the reply. The round trip times show a dependency on payload length that is composed of a part proportional to payload length, which is dominant for the longer payloads of 128 and 256 bytes, and a constant part, which is dominant for the shorter payloads of 32 and 64 bytes. In contrast, the round trip time is almost independent of the ping interval. The reason is the RTT being shorter than the interval. Thus, a ping request is fully processed and the VP is idle again before the next ping request arrives.

From these measurements we conclude that processing efforts on the VP in fact may influence sensitive network performance metrics. Thus, the integration of the VP provides the network simulation with deeper insight into the timings to expect from the host. The inclusion of a host simulated in detail on a VP can hence help to improve the analysis of the network, as it allows to check if the timing behaviour of the hosts modeled inside the network simulator is realistic.

5.5 Related Work

There are different approaches and tools in the literature that also target the development and the evaluation of embedded networked systems. In the following, we discuss respective approaches and compare them with our framework.

5.5.1 Hardware Simulator-based Approaches

There are different proposals according to the implementation or the extension of existing hardware simulators for the analysis of networked systems.

5.5.1.1 The gem5 Simulator

The gem5 simulation framework [BBB+11], a recent merger of the earlier hardware simulators M5 [BDH+06] and GEMS [MSB+05], allows the simulation of entire computer networks with host models of very high detail. It is a very modular simulation environment specially targeting research on computer architectures. The gem5 framework allows one to compose system models by combining different modules that describe the partial (hardware) behavior of the system. More specifically, the gem5 documentation[1] lists different CPU models, memory system models and different device models (network interfaces, timers, serial terminals and IDE peripherals) as the building blocks that can be used for composing a system. gem5 relies on the C++ programming language for implementing the simulation modules, while Python is used for configuring and controlling the simulation. According to the documentation, the system models of gem5 allow one to boot unmodified versions of Linux 2.4/2.6, FreeBSD and L4Ka::Pistachio[2] on the simulated hardware.

In order to model a computer network of multiple fully-simulated hosts, the first step is to specify the hardware of the host system in a Python simulation script. These hosts can then be interconnected using simulated Ethernet links. To our knowledge, the only way to model network applications and services in gem5 is to execute according networking software and an operating system on the simulated hardware, as the framework does not provide abstract models, e.g., for network protocols or applications.

In direct comparison with our work it is apparent that gem5 is not a hybrid emulation framework. Instead, all hosts, their according hardware and the network links are entirely modeled in the same simulation domain. As gem5 does not implement abstract simulation models, all hosts have to be modeled at the system level. This limits the scalability of a gem5 simulation in comparison to an ordinary network simulation, which only models the functional behavior of network hosts. For this reason, we consider gem5 to be more suitable for studies of low-level networking software and hardware in scenarios with a small to a medium number of hosts. It is noteworthy that the maturity of the gem5/M5 models allow one to run an actual operating system on a fully simulated platform. Conceptually, our virtual platform SR is also able to support such endeavours, however booting a legacy operating system on top of CoWare Virtual Platform requires commercial IP blocks that were not accessible for the author of this dissertation. The gem5 framework, by contrast, is available as open source and provides the respective support out of the box.

[1]Available online at `http://www.m5sim.org/Documentation` (accessed 06/2012).

[2]L4Ka::Pistachio is a variant of the L4 micokernel (`http://www.l4ka.org/`) being developed at the Karlsruhe Institute of Technology.

5.5.1.2 Using Simics for the development of embedded networked systems

Engblom et al. have proposed to apply the Simics full-system simulator (cf. Sec. 2.3.1) for the analysis of networked embedded systems [EKMR05]. It allows one to construct virtual Ethernet topologies that interconnect a set of fully-simulated systems. Each of these network links can be associated with a configurable latency in order to model different network topologies. In their paper, Engblom et al. describe four use cases of applying Simics for the evaluation of networked embedded systems [EKMR05]:

- Large scale network of embedded systems: In order to demonstrate the scalability of Simics for the simulation of networked embedded systems, the authors have modeled a topology of 1000 Internet Relay Chat (IRC) clients that are each executed on a fully simulated PowerPC 440GP[3] system. The authors report that it was smoothly possible to execute the simulation of this rather large topology on system with 11 AMD Opteron processors operating at a clock speed of 1.8 GHz. This comparatively good performance is explained with the capability of Simics to detect an IDLE state at a system it models, which was mostly the case for the given scenario.

- Fully-Simulated Ethernet/ATM Switch: The authors describe the full-system simulation of a telecom switch that features a larger number of Ethernet and ATM interfaces. It is noteworthy that the simulation of these Ethernet switches can be carried out in real-time. As Simics allows one to interconnect the full-system simulator with real-world network devices, real-world workloads can be used to evaluate a fully-simulated switch design.

- Wireless Sensor Networks: Simics supports the full-system simulation of sensor motes based on the TI MSP430 processor, for example the Telos B [PSC05] mote. The authors, however, report that sensor networking support was still under development at the time of publication. To our knowledge, more results or simulation studies of sensor networks using Simics have not been made publicly available since then.

- Self-Configuration of Storage Area Networks: The authors describe that Simics was successfully applied to the simulation of a large-scale fiber-channel storage fabric with more than 200 nodes.

In comparison with our framework it is most apparent that Simics models the computer network, its links, the hosts and their hardware within the same simulation domain. By contrast, our framework integrates two specialized simulation tools, namely ns-3 and the Virtual Platform, for the purpose of modeling the network and the host hardware. This enables our framework to resort to the rich collection of protocol models that are available for the network simulator. However, it is also noteworthy that Simics in contrast to other full-system simulators such as the previously discussed gem5 also supports so-called abstract nodes, which implement only

[3]The PowerPC 440GP is a processor core by IBM that is specially designed for networked embedded systems. It contains an Ethernet controller operating at 10/100 Mbps, a RISC core and different interfaces for peripherals and system components [IBM03].

the functional behavior of a network protocol in the same way as an ordinary network simulator. Simics provides abstract models for several protocols of the IPv4 family, for example DHCP, DNS and ICMP [EKMR05]. Needless to say, the number of protocols supported by ns-3 is much higher, given the fact that it is a dedicated network simulator.

A second interesting capability of Simics is its ability of being interfaced to real-world networking systems, as demonstrated by the second use case. This feature is mostly used to evaluate networked systems that are modeled with Simics using real-world workloads. In addition, this capability implies that Simics can be used to serve as simulation-based network emulation engine along the lines of ns-3. However, the limited availability of protocol models for Simics narrows this possibility to a small set of possible application scenarios.

5.5.1.3 The SystemC Network Simulation Library

SystemC (cf. Sec. 2.3.2), which forms the implementation basis for the VPs in our framework, is one of the contemporary standard languages for modeling and designing embedded systems. As these models inherently can be evaluated using an event-based simulation kernel, Fummi et al. [FPM$^+$04] have developed a network simulation library based on SystemC. It allows one to model the interaction and the communication of arbitrary SystemC-based systems in different network environments, effectively turning it into a discrete event-based network simulator.

The so-called SystemC Network Simulation Library (SCNSL) provides the following five types of components:

1. The SCNSL kernel organizes the execution of the SystemC-based network simulation and mostly implements the packet transmissions and the timing behavior of the simulation, which is for example important for correctly reflecting network characteristics such as propagation delays.

2. The SCNSL node objects implement the active elements of the simulated network, i.e., network devices or network infrastructure. Developers and researchers may also resort to existing SystemC models or IP blocks for implementing the node behavior.

3. Network packet objects are employed to model the data transfer among SCNSL nodes. SCNSL differentiates between the packet formats being internally processed by its simulation engine and the ones used the design domain. The conversion between these packet formats takes place at so-called NodeProxies.

4. SCNSL channels model the medium which is used to exchange data. The architectural abstractions of SCNSL allow one to implement different kinds of channels. In their paper, the authors describe a corresponding wireless channel model.

5. SCNSL enables nodes to receive and to send packets from channels using different ports.

Conceptually, SCNSL offers a flexible way of investigating present and future designs of embedded devices in a networking context. However, the applicability of SCNSL directly depends on the availability of models for the simulation library. In order to judge the adaptability of SCNSL for actual evaluation we performed a quick code inspection of the latest version of the SCNSL source code[4]. The reviewed source code of SCNSL solely contains two MAC layers model for 802.15.4, making it particularly useful for the analysis of WSN hard- and software. IDEA1 [DMNO11], a dedicated simulation tool for sensor networks, is based on SCNSL. However, the evaluation of software/hardware co-designs operating on top of MAC layers other than 802.15.4, for example Ethernet or Bluetooth, require the development of additional MAC layer models for SCNSL.

From an architectural point of view, integrating a network simulation library directly into the SystemC environment is a rather intelligent way to enable hardware/software co-design for embedded systems in a networking context. As SystemC already comprises an event-based simulation engine, SCNSL in contrast to our framework does not need to integrate two different simulators and timing domains. On the other hand the limited availability of network models constrains the scenarios that can be investigated with the off-the-shelf version of SCNSL. Contrary to SCNSL, users of a hybrid toolchain that combines a network simulator with a SystemC environment are able to benefit from the typically richer collection of protocol models available for the simulator; the latter is the case for our emulation framework.

5.5.2 Approaches based on Co-Simulation

On the lines of our framework there are different proposals of integrating a network simulator with a hardware simulation environment for the purpose of facilitating the development of embedded network systems.

5.5.2.1 Modeling Network Embedded Systems with NS-2 and SystemC

Prior to the development of SCNSL, the same research group had already proposed a hybrid co-simulation framework that combines SystemC with ns-2 [DFP02].

Drago et al. treat SystemC and ns-2 as two separate simulation engines and subsequently establish a bi-directional communication between both. For this purpose, their framework relies on a common event queue residing in a memory region that is shared among both processes. This enables ns-2 to push simulation events (for example network packets) to SystemC and vice versa. This queue serves as an abstract communication channel between both simulation engines, whose individual event queues remain untouched. Unfortunately, the authors do not explicitly state if and how the event queue is used to synchronize the execution of ns-2 and SystemC.

Drago et al. also describe two use cases in their paper. In the first one, two ns-2 nodes are interconnected using an IEEE 1355[5] link that is modeled using SystemC.

[4]Our findings are based on the source code of the SCNSL beta version (dated June 18th, 2010). We obtained the source code from `http://sourceforge.net/projects/scnsl/files/Scnsl/` (accessed 06/2012).

[5]IEEE 1355 is a standard for low-cost serial communication between heterogeneous devices.

In the second use case SystemC is used to model the CPUs of different network nodes that are part of an ns-2 simulation.

It is quite evident that the work of Drago et al. shares different similarities with our framework. Firstly, both rely on SystemC and a network simulator as well-established and respected building blocks for setting up a hybrid evaluation framework. This allows one to benefit from the individual strengths of both simulation environments, namely the detailed level of system modeling offered by SystemC and the strong capabilities of ns-2 in terms of simulating computer networks.

On the other hand, the framework developed by Drago et al. and our work differ significantly in the way they interface the respectively used simulation tools with each other. Drago et al. render the interaction of ns-2 and SystemC possible by installing an abstract shared event queue. Unfortunately, this methodology of integrating a ns-2 and a SystemC requires further customizations and adaptions to be applied to models in both simulation environments. In comparison with SHE, the integration of SystemC with ns-3 in our framework relies on two specific interfaces. The SyncVP component connects the SystemC simulation to the time control interface which is specially designed for synchronizing the execution of ns-2 and SystemC. The NetChip component integrates the SystemC component to ns-3 using the packet exchange interface at the link layer. While this design currently limits the applicability of our framework to the analyses of EtherNet-based embedded systems, this shortcoming can be overcome by amending alternative NetChip designs of the desired behavior. Such extensions are transparent to the used synchronization scheme, as the synchronization and the exchange of data are cleanly separated.

5.5.2.2 A Co-Simulation Framework for a Distributed System of Systems

Another framework that combines SystemC with a network simulator, in this case OMNeT++, was proposed by Müller-Rathgeber and Rauchfuss [MRR08]. Their work is mainly motivated by the design process of embedded systems that are part of computer networks inside vehicles. In contemporary vehicles such computer networks are typically used to connect different sensors (for example radar or laser scanners) as well as entertainment systems. In this scenario, Müller-Rathgeber and Rauchfuss employ SystemC to model processing delays that are caused by embedded systems processing such data.

In a similar way to our work and the integration of SystemC and ns-2 by Fummi et al. the authors also maintain the independence of both SystemC and OMNeT++. They propose a master/slave integration of both, in which SystemC acts as master simulator that drives OMNeT++. For this purpose, the authors have developed a messaging interface that allows SystemC to schedule OMNeT++ events using remote procedure calls.

The strict master/slave architecture enables a rather straightforward synchronization scheme. The SystemC simulator offloads network simulation tasks to OMNeT++ and blocks its own execution until OMNeT++ has completed these tasks. This yields to both simulation engines executing in a strictly alternating fashion. A side effect of declaring SystemC to act as master is that it needs to initiate the execution of the hybrid simulation.

Albeit both the work by Müller-Rathgeber/Rauchfuss and our work share the common goal of integrating SystemC with a network simulator, they differ greatly in their implementation. Despite the fact that both approaches rely on different network simulation tools (OMNeT++ vs. ns-3 in our case), the synchronization schemes are rather diverse. In contrast to the master/slave concept by Müller-Rathgeber and Rauchfuss, our barrier-synchronization scheme organizes the synchronization process in a client/server manner. The synchronizer acts as server and issues time slices of equal size to all system representations, in this case SystemC and ns-3. During these time slices, ns-3 and SystemC are able to process their event queues independently from each other and hence are able to process their time slice in parallel. In contrast to SHE, the master/slave concept forces SystemC and OMNeT++ to mutually wait for each other.

5.5.2.3 Other Related Work

Besides the discussed hardware simulators and the previously explained hybrid frameworks that are able to support hardware and software development efforts in the domain of embedded systems, there are other tools and methodologies aiming for the same goals. First of all, there are specialized hardware simulators that target specific domains of embedded systems. Examples for such tools are Avrora [TLP05] or ATEMU [PBM^{+}04], which constitute sensor network simulators. Many of these sensor network simulators are constricted to a particular hardware platform, for example the Mica2 mote in the case of Avrora. These simulators are well-suited for low-level software development but fall far short if it comes to the design of new hardware platforms because of their monolithic nature. A rare exception is the recent work by Stecklina et al.[SVB^{+}11] who have proposed the integration of the MSP-Sim [EÖF^{+}09] sensor mote simulator with SystemC for the purpose of evaluating new WSN hardware. The integration of both is very reminiscent of the previously discussed work by Fummi et al. and the solution of Müller-Rathgeber and Rauchfuss.

5.6 Limitations

We mostly regard our framework for networking centric development of embedded systems as a proof of concept, which shows that building such a hybrid evaluation framework corresponding to the idea of Synchronized Hybrid Evaluations is possible. Nevertheless, there are two limitations we would briefly like to address in the following.

5.6.1 Non-Optimal Integration of Timing

Although we have shown that the barrier-based synchronization of both the VP and the simulation is operational and working, we regard it not as the ideal scheme for synchronizing this particular set of system representations. In fact, both the VP and its underlying SystemC engine als well as the network simulators are based on the principle of discrete event-based simulation. In contrast to that, our synchronization scheme is essentially continuous, as it provides a stream of subsequent time slices.

This may yield to a non-optimal execution performance especially if both the VP and the network simulations are idle. Instead of simply skipping the idle period, the VP and network simulation have to crawl through the virtual time at a step size that is specified by the time slices. This is a direct consequence of our synchronization scheme being designed for arbitrary execution schedules.

5.6.2 Dependency on Proprietary Software

While ns-3, our synchronization component and the Xen-based VM system representations of SliceTime are all available as open source software, it is unfortunately difficult to make the entire framework available to a bigger community. While the Netchip and the SyncVP component are conceptually compliant with the open source reference implementation of SystemC, the virtual platform used for our evaluation relies on commercial IP blocks that cannot be made publicly available.

5.7 Interim Summary and Conclusion

In this section we have described a framework that allows one to incorporate arbitrary SystemC-based virtual platforms (VPs) with a network simulation, which in our case is based on ns-3. The synchronization of both is required, as their execution performance may differ widely, for example if the network simulator models a very large computer network or if the VP simulates a complex hardware architecture.

The framework allows for the close analysis of embedded systems hardware and software during the design phase inside a large-scale simulated network context. In essence we achieve this goal by extending the SystemC-based VP to act as a system representation (cf. Sec. 3.2) while the remainder of the framework consists of already existing components that have been originally developed for SliceTime (cf. Sec. 4.1).

Our evaluation shows that integrating a network simulator with a VP is feasible with a reasonable accuracy while introducing only a slight overhead. During our evaluations, we modeled an embedded system that is quite resource constrained in terms of CPU performance (10 MHz), which is still a reasonable assumption for low-power network devices. For such a scenario, we have shown that the networking context of an embedded system directly impacts the load on the VP, for example due to a varying network load. In addition, we were able to demonstrate that the accurate modeling of VP timings also influences network performance metrics, for instance round trip times.

All in all, we conclude that our framework especially extends the applicability of virtual platforms for the design of networked systems. Our approach allows the virtual platform to be evaluated and tested in a network environment that can be modeled with all the models and the expressiveness the network simulator provides. Moreover, the network simulation may benefit from the possibility that the VP is able to reproduce accurate timings for packet processing.

6

Monitoring and Debugging Communication Software using Virtual Time

Chapter Overview

We now round out the technical discussions in this thesis with a software framework [WRSW10] that enables distributed monitoring and debugging of communication systems. Being based on a homogeneous SHE configuration (cf. Sec. 3.4.3.2) it inherits the property of synchronously executing virtualized communication systems isolated from wall-clock time. Our framework makes use of this concept for synchronously pausing an entire set of virtualized systems during an inspection process, for example in order to implement distributed breakpoints.

Structure of this Chapter

After a brief motivation, this chapter first focuses on the design and the architecture of our framework. We then discuss a respective proof-of-concept implementation and demonstrate its basic operability with two small functional tests. The chapter concludes with a concise discussion of related work in the domain of distributed debugging and monitoring.

6.1 Motivation

In order to make sure that communication software operates in a valid way, we need software tools that enable their close investigation in a distributed setting. However, analyzing the distributed execution of communication software such as protocol stacks is often difficult, because the global state of a protocol implementation is distributed among all communication peers.

Standard debuggers like gdb [gdb] are mostly not well suited for the run-time analysis of distributed applications and protocol stacks. Such debuggers are restricted to user-space applications which prevents observations of the close interplay of network stacks with other operating system (OS) components and network device drivers. This issue may be overcome by using kernel-level debuggers such as kgdb[1], but these tools in general are limited to one machine and thus can only deliver information about an implementation's local state. A second major problem with the use of classic debuggers consists in the missing synchronization of the execution with the other communication peers. For example, it might happen that the implementation hits a break-point, and thus, the execution is suspended while the execution at the remote communication peers continues. In many cases this may lead to a faulty behavior, for instance due to the expiration of retransmission timers or due to unwanted connection time-outs.

Full system simulators, for example Simics (cf. Sec. 2.3.1), enable the investigation of the run-time behavior of an operating system with global state information at hand. Instead of executing the operating system natively, they simulate the entire system hardware and potentially a network of such in software. As the hardware of every communication peer in the network is entirely simulated, one benefits from an unsurpassed degree of control and detail. However, a major disadvantage of full-system simulators is their restricted scalability which directly results from the simulation complexity. The meticulous level of detail is also not needed for many evaluation purposes.

6.2 A Distributed Analysis Framework

6.2.1 Challenges and Solutions

Scrutinizing the behavior of communication software in a distributed setting is challenging for a couple of reasons. Ideally, such an analysis framework delivers consistent state information for all communication peers at any point in time. Considering the term *state*, we need to distinguish between the local software state and the global state that is constituted by the local state of all peers at the same point in time.

The extraction of local state information is intricate for different reasons. Most importantly, it is vital that the investigation of the run-time behavior is non-intrusive, and hence does not change the way the implementation executes. Otherwise, odd side effects like the occurrence of heisenbugs [Gra86], that only occur during the analysis run, could be direct consequences. Moreover, the collection of local state information is further aggravated as protocol stacks are typically integrated tightly into the operating system's kernel.

We address these challenges by completely abstaining from monitoring the local state on the system that executes the network stack. Instead, we virtualize the entire system and carry out all monitoring operations from an external context governing the VM. This way, the entire extraction of local state information may be performed in a completely transparent fashion.

[1]See `https://kgdb.wiki.kernel.org/` (accessed 10/2012) for further information on kgdb.

A major problem with analyzing the run-time behavior of communication software and protocol stacks is the following: Local state changes are not only dependent on the causal sequence of the network packets being exchanged but also on internal mechanisms, most notably protocol timers. For example, such protocol timers are commonly used to detect packet loss or connection time-outs. In order to maintain global consistency, it is essential to suspend the execution at all communication peers once one peer is paused for the purpose of closer inspection, for instance when it reaches a break point. We tackle this problem by virtualizing the entire progression of time at all communication peers using the SliceTime infrastructure (cf. Chapter 4.1). Thus, if the execution of one peer is interrupted, all other systems are paused as well. Since we execute all systems using a virtual and logical continuous time, no communication peer notices such gaps in his execution flow.

In order to obtain a global view on a distributed software state, we consolidate local state information from the communication peers. The local state of such a peer is quite bulky, as it encompasses the entire memory as well as all other system resources allocated to the VM. Aiming at an on-line analysis of the distributed execution, the size of the state space prevents a frequent collection of the entire local state from the communication peers. However, only a small fraction of the VM state is usually of interest if one is up to analyze a software implementation. Therefore, we only extract selected local state information in order to limit the amount of data retrieved from the communication peers.

A second challenge is the fusion of local state information into a globally consistent view. A well-established method in this context is the use of logical clocks [Lam78]. By associating logical time stamps with every local state, a consistent global state can be consolidated. One example where this concept has been applied to the analysis of distributed applications is D^3S [LGW^+08]. However, the utilization of logical clocks requires every state change to be propagated.

Instead we propose a different approach that utilizes the virtual progression of time at the communication peers for the construction of so-called global soft-states: We assign tiny slices of virtual time to all communication peers. All peers are suspended after they have completed their time slice. The next time slice is assigned after all systems have reported the completion of the time slice. As all communication peers synchronously progress through this series of discrete time slices, we obtain an implicit sequence of global soft-states. Their accuracy is determined by the size of the chosen time slices.

6.2.2 Architecture

Figure 6.1 shows the architecture that puts our concepts into action. It consists of two main building blocks, the Monitoring Front-end that controls the entire investigation process and the progression of time. The execution of the communication software takes place at multiple VMs, from which we collect local state information using the monitoring back-end attached to the VM. The framework belongs to the family of homogeneous SHE configurations (cf. Sec. 3.4.3.2), as it solely employs VMs as system representations.

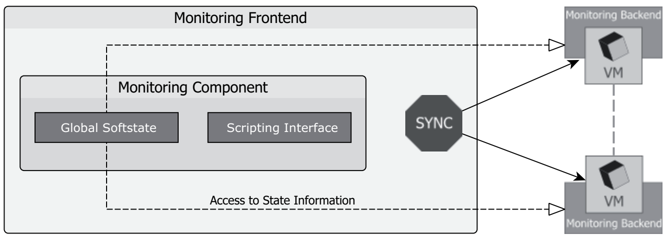


Figure 6.1 Conceptual Architecture of the monitoring framework.

6.2.2.1 Monitoring Front-end

This building block encompasses the synchronizer (cf. Sec. 4.1.2) as well as the monitoring component. All analysis tasks are carried out at the monitoring component. It retrieves the local state from all back-ends attached to a VM using remote procedure calls and consolidates a global soft-state. A scripting interface provides a flexible interface to the global soft-state information. This facilitates both automated and interactive explorations of the collected state information.

In addition, the scripting interface allows the set of investigated state information to be modified at run-time. For example, further breakpoints and inspectors may be added in a conditional way, as in case of a particular behavior observed on one of the VMs. Moreover, the scripting interface not only facilitates the passive inspection of state information, but also allows one to change state descriptors actively. For instance, this allows tuning protocol parameters with “global knowledge” or enables the simulation of software faults.

6.2.2.2 Monitoring Backend

The monitoring back-end implements all required primitives to support the operations provided to the developer through the scripting interface. The primitives are exposed to the front-end using a standard Remote Procedure Call (RPC) library.

The most important primitives implement the access to local state descriptors on a virtual machine. In order to locate the state information within the memory region allocated to the VM, we access the symbol table of the operating system executed in the VM. This way, we are able to directly access the corresponding state descriptors while avoiding a potential external reimplementation of operating system memory management functions such as the traversal of page tables. Besides the bare access to the memory of the VM, the back-end parses the respective memory content in order to provide the developer with a more descriptive representation of the inspected state property.

Another task of the monitoring back-end is to provide further control primitives regarding the VM execution. Basic commands such as `PAUSE` and `UNPAUSE` perform the corresponding actions. More sophisticated primitives such as `SNAPSHOT` allow

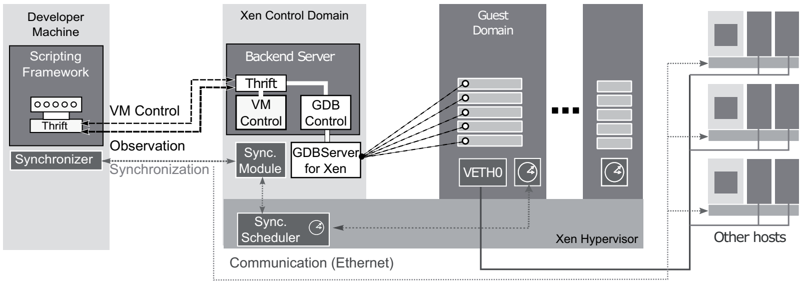


Figure 6.2 The implementation of the monitoring framework is based on the SliceTime infrastructure (cf. Sec. 4.1). It facilitates the analysis and the debugging of multiple VMs using the Python scripting language. The SliceTime synchronization provides a virtual flow of time to the attached VMs.

for storing an entire VM state, which also enables the scripting interface to initiate global distributed snapshots upon the observation of certain behavioral patterns.

Besides the provision of the required access and control primitives, the back-end together with the VM environment needs to support the time-slice based execution required by the synchronization scheme. For this purpose, the VM environment executes a virtual machine for the exact duration of the time slice. In addition the VM environment has to provide the VMs with a virtual progression of time that is aligned to the provisioned time slices.

6.3 Prototype Implementation

Figure 6.2 depicts our proof-of-concept implementation. We distinguish between the developer machine that exposes the global soft-state within a scripting environment and VM hosts executing multiple VM guests on top of the SliceTime-Xen infrastructure (cf. Sec. 4.1). The VM hosts and the developer machine are interconnected using three different flows of communication.

1. A *combined observation and control flow* based on Apache Thrift [SAK07] delivers state information to the developer machine and facilitates controlling the VM execution behavior using the scripting framework.

2. The Time Control Interface (TCI) delivers time slices to the VM hosts using a lightweight UDP messaging scheme, minimizing the complexity due to the potentially high amount of synchronization messages (cf. Sec. 4.1).

3. The Packet Exchange Interface (PEI) (cf. Sec. 4.1.2.3) enables the virtualized communication systems to exchange network packets. Please note that Figure 6.2 displays the packet flow of the PEI in a simplified form for the sake of clarity. In fact, all VM traffic traverses the domain 0 of the respective VM host; at domain 0 a TAP-device based tunnel implementation takes care of forwarding these packets to the other hosts (cf. Sec. 4.1).

```
class hostA_thread(threading.Thread)):
 def run(self):
 while True:
  hostA.wait()
  pp.pprint("host A: *skb:" + hostA.getVariable("*skb",))

class hostB_thread(threading.Thread)):
 def run(self):
 while True:
  hostB.wait()
  pp.pprint("host B: *skb:" + hostB.getVariable("*skb",))

bpA = hostA.setBreakpoint("icmp_rcv",)
bpB = hostB.setBreakpoint("icmp_rcv",)
```

Listing 6.1 Example code fragment that illustrates the application of our framework for monitoring the reception of ICMP frames at two hosts: The contents of the socket buffer is printed out as soon as a system reaches the specified breakpoint.

6.3.1 Scripting Environment

The Python-based scripting framework drives the entire inspection process. At any point in the logical flow of time, the individual local states of all communication peers are accessed using a stub that allows for setting breakpoints or for reading and modifying state descriptors using the common gdb syntax. Listing 6.1 illustrates how our framework can be applied to monitoring the reception of ICMP packets at two Linux hosts. The listing omits the initialization block that establishes the observation and control flow to the sender and the receiver. The example uses the setBreakpoint primitive to interrupt the execution at both hosts upon the reception of an ICMP packet, for instance an echo request. It employs two threads in order to cope with the parallel execution of the sender and the receiver. Using the `wait()` command, each thread waits until one system hits a break-point. In this case, the script outputs the content of the socket buffer. Other commands not used in this example allow for snapshotting a VM and facilitate a manual control of the VM execution.

The entire process of synchronization is transparent to the scripting environment. The provision of small time slices automatically establishes a series of global soft-states, formed by the individual local states for a particular time slice. As a system is not able to signal the completion of its time slice upon hitting a breakpoint, the execution at all other communication peers is automatically suspended at the end of the current time slice.

6.3.2 Back-end Server

The core task of this component is to deliver local state information to the scripting front-end. For the purpose of accessing local state information from the Xen domains, we rely on Gdbserver-xen [KNM06]. Gdbserver-xen provides a gdb interface to the kernel being executed inside the VM. It relies on the available symbol table information in order to locate symbols, e.g. protocol state descriptors, in the memory range of the executed kernel. Gdbserver-xen then internally translates the

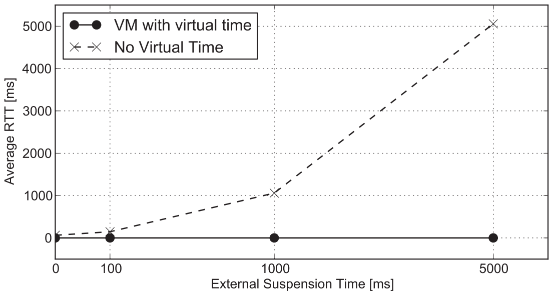


Figure 6.3 Executing a VM in virtual time rather than real-time makes RTTs on the local host invariant to externally imposed execution gaps.

pseudo-physical addresses in the symbol table to the corresponding addresses in the address range of the physical host. For the purpose of providing a convenient and efficient access to kernel-level state descriptors, gdbserver-xen maps these memory ranges to the address space of the privileged control domain (domain 0).

Besides the exposure of state information, the back-end server also implements a set of control primitives. For this purpose, our implementation executes so-called hypercalls to directly control the state and the scheduling behavior of the VMs. In addition, few operations such as snapshotting are implemented by invoking the corresponding calls to the Xen Management daemon.

6.4 Functional Tests

In order to check the operation of our prototype implementation, we conducted two functional tests with our framework. Both experiments were carried out on two PCs. Both machines executed our Xen-based implementation of the monitoring backend. One machine also hosted the monitoring frontend and the synchronizer. For all experiments the synchronization accuracy was set to 0.1ms.

6.4.1 Time Isolation

With the goal of determining if the isolation of time works for the virtualized communication software, we used a straightforward monitoring script to suspend the execution of a VM for a specified amount of time upon the reception of an ICMP frame. In order to trigger this behavior, ICMP echo requests were sent to the local host at a constant rate. On the VM, we measured the average RTT of ICMP replies.

Figure 6.3 displays the result: As expected, for non-synchronized VMs the measured RTT increases for longer execution pauses, as ICMP internally utilizes timestamps to conduct round-trip measurements. The round trip times hence correspond to the artificial external delay. By contrast, if we use our implementation to supply a VM with a virtual progression of time, the measured round trip times show the desired invariance to external interruptions of the VM execution.

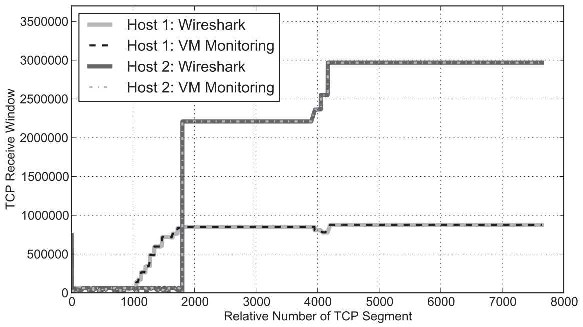


Figure 6.4 Our framework accurately captures state changes of the TCP implementation in the Linux 2.6 kernel. The state information extracted from the VM well matches the reference values obtained from packet traces.

6.4.2 Monitoring State Changes

As internal states of communication software may change with every packet sent or received, it is vital to accurately capture state changes at a very fine granularity. For the purpose of evaluating our implementation against this requirement, we applied our framework to the TCP implementation of Linux 2.6. One of the TCP protocol states that are subject to change with every packet is the TCP receive window. We are aware that our VM-based monitoring approach is not required for tracing the receive window of a TCP connection, as the receive window is also part of the TCP header and thus may also be observed using a packet capturing tool like Wireshark [Wira]. However, the presence of this information within the TCP packet header allows to use it as a reference value and thus allows for investigating the accuracy of the state information gathered with our framework.

Consequently we used our framework to monitor the receive window on two VMs exchanging data over a bi-directional TCP connection. Figure 6.4 compares the window sizes extracted from the VMs using our framework with the reference window sizes as reported by Wireshark. The window sizes extracted from the VMs well match the reference values. From this we conclude that our approach enables observations of protocol descriptor state changes at th e granularity of one packet.

6.5 Related Work

For the discussion of related work we consider contributions from different areas, namely VM-based debugging, distributed debugging and online system monitoring.

6.5.1 Virtual Machine-based Debugging

The idea of applying virtual machines for debugging dates back to the year of 1974, when Goldberg and Galley proposed a set of core principles for VM-based debugging [GG74]. The core finding of this work is the idea of isolating a system from the hardware for debugging purposes. Employing a VM for this task allows one to retain full control over the execution of a communication system and to observe all internal operations of a system closely.

Over the past years a number of virtualization frameworks have implemented according debugging functionalities. For example, VMWare Fusion[2] and qemu [Bel05] provide their users with a debugging stub for gdb. Recent versions of Bochs [Law96] feature an integrated debugger with similar capabilities. All these approaches share many commonalities with gdbserver-xen [KNM06], which forms the basis for the implementation of our monitoring back-end.

King and Dunlap have shown that VM-based debugging can be enhanced with the possibility of navigating arbitrarily backward and forward through the execution flow of an VM [KDC05]; their framework also allows one to deterministically repeat VM executions. This greatly simplifies the debugging of low-level software, as the non-deterministic nature of VMs and operating systems make it often difficult to reproduce software faults.

While all these frameworks facilitate debugging low-level system software with a high degree of flexibility, they are all limited to single host applications. By contrast, our work focuses on the inspection and debugging of distributed network applications.

6.5.2 Distributed Debugging

The challenges associated with debugging distributed systems and applications have been identified a long time ago, and hence a diverse amount of prior work exists in this domain [MH89]. For the sake of brevity we hence discuss only two approaches with strong analogies to our work.

Garcia-Molina [GMGK84] et al. have proposed a distributed debugging architecture, which is reminiscent of our work in different ways. First, the authors propose to decouple the system into one single master debugging node and a set of network nodes, executing and observing the software to be debugged. The master debugging node serves as central control point for the debugging process. All nodes are interconnected using a communication network. This architecture is conceptually equivalent to our approach, where the system is split up into a monitoring back-end and a monitoring front-end. A second similarity is the elevated priority of the debugging process on the network nodes, which enables a close inspections of the processes and their interactions. Our framework implements this elevation of priority for the monitoring back-end by virtualizing the communication software and by accessing the VM internals from a privileged Xen domain. Other distributed debuggers that implement similar concepts are p2d2 [Hoo96] and a distributed debugger for Amoeba [Els88].

A framework very related to ours is PDB [HHH04]. PDB is a distributed debugger based on Xen. It enables distributed debugging of both user-space and kernel-space communications software. The core idea of PDB is to deploy an entire set of communication applications on one physical machine, using a set of para-virtualized

[2]VMWare Fusion is a commercial type 2 hypervisor developed by VMWare, Inc.

domains that operate on top of a modified Xen hypervisor. As the core of PDB is implemented at the hypervisor layer, its higher level of prioritization enables it to fully observe and to intervene in the execution of the virtual machines. Regarding the architecture and the implementation, our work resembles PDB in many ways.

In contrast to PDB, our framework is able to span over a number of physical machines, with each machine hosting a set of VMs. The software under investigation is executed on a synchronized virtual time line that is commonly shared by all VMs. This enables an entire virtualized deployment of communication software to be synchronously paused if one VM is suspended, for example if a breakpoint is reached. In addition, our framework provides a scripting environment that allows for automatically carrying out more complex monitoring and debugging tasks.

6.5.3 Online System Monitoring

The spread of more complex distributed systems has brought up the need of monitoring their execution, for example in order to identify software faults or security problems. Given the high diversity of distributed systems today, according solutions differ widely in their architecture and their implementation.

6.5.3.1 VM Monitoring for Intrusion Detection

Garfinkel and Rosenblum have proposed an architecture for intrusion detection that makes use of virtual machine monitoring [GR03]. The core idea behind this approach is to externally inspect the state of an VM by closely observing its memory, its CPU state and its accesses to I/O devices. By matching a set of policies against the VM state an external privileged monitoring entity can detect if a VM was compromised. The authors prove the feasibility of this concept by implementing an according intrusion detection system for Linux using a modified VMWare hypervisor.

XenAccess is a very similar architecture developed by Payne et al [PL07]. Like the work by Garfinkel and Rosenblum, XenAccess was also developed for security monitoring. XenAccess is able to monitor Linux domains hosted on a Xen hypervisor from domain 0. In this regard, the core task of XenAccess is to establish a mapping between “internal” memory addresses used by the guest OS and the address ranges used by Xen. In order to access specific memory regions, for example kernel variables, XenAccess relies on the availability of a symbol table for the guest operating system and partially reimplements the memory mapping algorithm of the guest operating system for the translation process.

Albeit both of these tools were developed for a different purpose, both XenAccess and the work by Garfinkel and Rosenblum hold many architectural similarities in comparison with our work. In fact, these similarities stem mostly from the plain fact that all rely on hypervisors and thus make use of the same core concept, namely the isolation of the VM and the execution of the monitoring framework in a privileged context. The biggest difference is our strong focus on monitoring distributed systems. Both XenAccess and the framework by Garfinkel and Rosenblum are tailored towards local VM monitoring. A second difference is that our framework executes the VMs decoupled from wall-clock time, which conceptually would make it possible to conduct monitoring tasks that are too complex to carry out in real-time.

6.5.3.2 Distributed Systems Monitoring

Dodd and Ravishankar [DR92] have proposed a monitoring and debugging framework for a custom distributed operating system. The framework allows low level internals, such as context switches and system calls, to be observed at a central monitoring unit. The monitoring of such events is facilitated through a number of hooks that were either already available in the kernel or were added by the developers. By contrast, our framework does not require changes to be applied to the system being monitored or debugged.

Two software tools that clearly outrival the monitoring functionalities of our framework are the P2 monitoring system [SMRD06] and D^{3}S [LGW$^+$08]. P2 is a declarative logic programming language that allows one to implement complex distributed and P2P protocols using high-level statements [LCH$^+$05]. In [SMRD06] this framework is extended with the possibility of formulating and processing complex queries for monitoring using a language based on Datalog [CGT89]. Such queries then can be successively used to validate the operation and the state of a distributed system.

D^{3}S [LGW$^+$08] facilitates the debugging of distributed systems implemented for the Windows operating system. The framework is based on checking a set of predicates that operate on distributed state information. The predicates are implemented in C++ and obtain the required state information from so-called state exposers, which are inserted into the local processes of the distributed system using binary instrumentation. The authors have shown that D^{3}S is capable of identifying software faults in different kinds of distributed applications, for example BitTorrent clients. In addition, D^{3}S has also been applied for assuring the consistency and the availability of P2P systems such as the DHT implementation of i3 [SAZ$^+$02].

The main reason why D^{3}S and the P2 monitoring system outperform our framework in terms of monitoring functionalities are their sophisticated policy checking and query processing engines. However, technically it would be possible to implement such an engine inside of our monitoring back-end, possibly by extending or exchanging the scripting framework.

One distinctive feature of our system that sets it apart from approaches like the P2 monitor and D^{3}S is its non-real-time operation. D^{3}S and the P2 monitor are on-line monitors that are designed for observing the execution of massively distributed systems in real-time. By contrast, our system aims at the close inspection of system software in a lab setting. Here, our execution model can be helpful for the deep investigation of system behavior, which is further aided by support for distributed break-points and globally pausing the execution of a distributed system.

6.6 Limitations and Potential Future Work

Due to the proof-of-concept nature of our framework there are different limitations and shortcomings.

- **Missing Query Checking and Policy Framework:** Frameworks such as D^{3}S [LGW$^+$08] are able to check on-line if a set of policies holds for a deployment of distributed systems. Our framework currently does not employ such a framework; however custom checking methods could be implemented using the scripting environment. To better aid complex checking tasks, enhancing the scripting framework with a predicate-based policy framework along the lines of D^3S [LGW+08] or the P2 monitor [SMRD06] would be a reasonable direction for future work. We believe that knowledge engines such as PyKE [Fre08] could well serve as implementation basis.

- **Preliminary Evaluation:** Our monitoring framework at its present stage was mainly developed to demonstrate the usefulness of homogeneous SHE configurations (cf. Sec. 3.4.3.2). Hence, we have not conducted a thorough evaluation or an application to real-world systems so far. In order to finally judge on its applicability for complex monitoring tasks a further evaluation of the framework would be required.
- **Dependency on Symbol Information:** A noteworthy technical limitation of our monitoring system is its dependency on the availability of symbol information for the code under investigation. The underlying reason is the need of locating state descriptors, which mostly correspond to memory regions in the VM's address range. Only the availability of symbol information makes it easy to locate those by calculating valid memory offsets. In effect, our system can only be applied if a symbol table (e.g. for an OS kernel) is available. Unfortunately, this is often not the case for closed-source software.

6.7 Interim Summary and Conclusion

In summary, we conclude that homogeneous SHE configurations (cf. Sec. 3.4.3.2) are principally able to form effective evaluation environments for computer networks

Combining the concept of VM monitoring with the provision of a consistent and virtual progression of time to a set of virtualized communication peers results in two important properties. First, the succession of time slices yields to a series of global soft-states whose size is given by the size of the time slice. This property principally also holds for all SR setup types discussed in Chapter 3, although only the framework discussed in this chapter makes use of this characteristic. Second, the synchronization relieves the investigation process from any real-time constraints. Hence, it becomes possible to transparently carry out extensive analysis tasks like the deep exploration of a network protocol's state space or complex operations such as snapshotting an entire system.

From the experience with our prototype we conclude that VM monitoring in virtual time is a feasible concept. Although our framework is a mere proof of concept and far from technical maturity, we were able to demonstrate that the core functionality is operational. We hence believe that with additional development effort VM monitoring in virtual time can be turned into a helpful methodology for the future development of distributed systems.

7 Conclusion

Evaluating the performance of communication systems is a challenging task. The huge diversity in the communication eco-system of today leads to a plethora of performance aspects researchers and developers need to investigate. The wide range of possible evaluation tasks and questions makes it impossible for one single methodology to tackle them all. For this reason, the comprehensive analysis of a communication system often relies both on network simulations and analytical methods as well as investigations with actual prototypes to answer specific subquestions.

There are certain cases for which answering a performance subquestion with a single methodology is difficult or even impossible. For example, consider the evaluation of a TCP implementation optimized for wireless environments. In order to investigate how the TCP throughput depends on factors such as node movement or dynamic channel conditions, the ability to vary these factors between different evaluation runs in a controlled fashion is desirable. However, this only is possible in real-world deployments, for which precisely repeating the node mobility is challenging. Even worse, exactly replicating channel conditions is impossible in such an environment. On the other hand, simulations provide such capabilities but fall far short when it comes to the execution of real-world systems. In addition, network simulations mostly abstract from modeling the functional and system behavior in a very detailed way, which may lead to artifacts and might even impair the validity of simulation-based research [FP01, FK03, KCC05].

Hybrid performance evaluation methods, for instance network emulation which integrates a network simulation with real-world prototypes, are a well-known approach to overcome the individual limitations of analytical, simulation- and measurement-based approaches. Unfortunately, the applicability of hybrid tools has been often limited by two main constraints. First, the vast majority of network emulation tools requires the network simulation to execute in real-time. Second, many performance evaluation tools lack common interfaces (e.g., for packet exchange or synchronization) that enable the easy construction of hybrid performance toolkits.

Structure of this Chapter

The remainder of this chapter first summarizes the core results of this thesis. In a second step we point out the limitations associated with our solutions before discussing a set of potential future research directions.

7.1 Key Results

In summary, we have advanced the flexibility and the applicability of different hybrid performance evaluation methods, most notably network emulation. We now briefly recapitulate the three key results of this thesis and relate them to the challenges as stated in Chapter 1.

Generalization of Hybrid Performance Evaluation Methods

With Synchronized Hybrid Evaluation (SHE) (cf. Chapter 3) we have proposed a concept for the generalization of hybrid performance evaluation methods for communication systems. The core idea of this approach is to regard hybrid evaluation frameworks as the integration of different System Representation (SR) modules. In this thesis we have considered network simulations, virtual machines and full-system simulators as possible SR types. A major strength of this approach is that it allows new hybrid evaluation frameworks to be formed simply by recomposing system representations. This results in the possibility of efficiently reusing existing SR implementations and thus aids the development of new hybrid evaluation technologies.

A second cornerstone of SHE is the virtualization of time and the accordant barrier synchronization scheme that aligns the execution of different SRs. This enables the conjoint progression of network simulations, virtual machines and full-system simulators on one virtual time-line. As a result, a SR does not need to model its target system in real-time. Doing so enables SHE setups to overcome the synchronization problem, which so far has limited the applicability of network simulations with a high computational complexity for network emulation.

The SHE concept directly addresses different challenges as stated in Chapter 1. By specifying two general interfaces, namely the *Time Control Interface (TCI)* and the *Packet Exchange Interface (PEI)*, the SHE concept defines an integration baseline for SRs that differ widely in their internal operation. The TCI provides a general common ground for the different time representations internally used by the SR implementations. As it internally relies solely on the provision of virtual run-time allocations it forms a very universal synchronization protocol; we thus believe it can be applied to any imaginable type of SR. In its present stage, the PEI addresses the interoperability of different system representations by declaring the MAC layer as lingua franca among the system representations. For the set of SRs considered in this thesis we have experienced this as a sufficient approach that was easy to implement for the respective components.

Elevation of the Scalability and Applicability of Network Emulation

The work presented in this dissertation has significantly advanced the applicability of network emulation. SliceTime [WSvL$^+$11] (cf. Sec. 4.1) is a synchronized network emulation framework based on ns-3 that follows the design principles of SHE. We have shown that SliceTime is efficient and enables network emulation studies of very high scale and complexity to be carried out on legacy hardware. By amending ns-3 with a BitTorrent simulation model that is interoperable with real-world clients, we have further demonstrated that SliceTime can be used for the investigation of complex application-level protocols. In this regard, we have indirectly addressed the challenge of developing a complex simulation model that is interoperable with real-world communication systems.

We have further improved the applicability of wireless network emulation with our work on Device Driver-enabled Wireless Network Emulation (DDWNE) [WvLW11] (cf. Sec. 4.2). DDWNE also improves the degree of realism and eases testing of unmodified wireless software. This technique employs a device driver that not only integrates the communication channels of a simulation with a real system, but which also provides primitives to access and to sense the simulated environment. We mostly rely on this method to bridge the perception of WiFi transmission characteristics, for instance Received Signal Strength Indicator (RSSI) values, with the OS environment. However, it would be also possible to apply this scheme for enabling software on the real system to access location information or to "virtually" sense the simulated scenario. Hence, we regard DDWNE as a major step towards a more tight integration of the software context and the simulated environment.

Two Synchronized Hybrid Evaluation (SHE) frameworks for advanced system design, monitoring and debugging

The work presented in the last two chapters has shown that the SHE concept is pertinent for evaluation frameworks beyond the scope of network emulation.

In this regard, we first have proposed the integration of a network simulation with a full-system simulator for the purpose of co-designing networked embedded systems (cf. Chapter 5). Our according implementation reuses parts of the SliceTime code base and amends them with an additional system representation. This SR is able to integrate arbitrary SystemC models with a network simulation. We have shown that such a tool-chain can be used to analyze the impact of network communication on low-level hardware characteristics, for instance bus load.

Finally, we have discussed the applicability of SliceTime VMs for debugging and monitoring distributed systems using virtual time. The according proof-of-concept framework reuses the synchronizer and the VM infrastructure of SliceTime and extends it with a Python scripting interface. The scripting interface facilitates the observation and the control of distributed applications deployed on the SliceTime VMs. The synchronization and the according virtualization of time make it possible to set distributed break points while enabling the global execution to resume after the analysis at the breakpoint has been completed. While this framework is at a very early development stage, it demonstrates that the synchronization of virtual machines is helpful for evaluation tasks that originally had not been envisioned at the time of conceptualizing SHE and Synchronized Network Emulation (SNE).

7.2 Conceptual Limitations

The design choices made by our work imply a few limitations that are of interest for actual performance studies carried out with SliceTime and partially other SNE frameworks. In the following we discuss these issues and describe possible solutions and workarounds.

Increased Experiment Run-Time

A key concept of our work is decoupling the wall-clock time from the time progression at the system representations. This enables SRs to execute faster or slower than real-time depending on their computational needs. Network simulations that model large topologies or that employ complex simulation models may easily fully load a computer. Such heavily loaded simulations often are only able to operate at a fraction of real-time. For a Synchronized Network Emulation (SNE) setup, which contains such a “slow” simulation, this means that the execution speed of all VMs is automatically throttled down to match the simulator’s performance.

While conducting our different application studies with SliceTime we observed simulation slowdowns between 1 and 200. Hence, one virtual second in the time domain of the SRs took up to 200 real seconds to complete. Such dramatic slowdowns are problematic for two reasons. First, this immensely increases the time needed for evaluation runs. This is a problem if somebody wants to use a SNE setup for rapid prototyping of communication software. A second problem are studies with communication software that require user interaction. While we have observed that VMs with slowdowns up to 5 may well manually be controlled using a shell or a mouse, it its difficult to interactively work with virtualized software at higher slowdowns. To this end, we have mostly scripted the required behavior if manually interacting with an VM was too cumbersome.

Speeding up the simulation performance is the only possibility to bypass such issues. This can either be achieved by parallelizing the network simulation or by reducing its computational complexity. Although not further investigated in this thesis, our framework can be used for the parallelization of network simulations: a network simulation SR may easily be dissected into multiple simulations with less complexity by partitioning the topology. Such split simulation SRs could then be connected with each other and SRs of different types using the Packet Exchange Interface (PEI). Reducing a simulation’s complexity is also generally possible by increasing its level of abstraction. For example, our BitTorrent model uses real-world MAC frames on all parts of the simulated topology in our case study. By replacing these links with a less detailed model, we expect the simulation performance to improve.

Limited Determinism

In general, a SHE environment does not execute in a deterministic fashion if one of the SRs is non-deterministic. Hence, multiple evaluation runs will yield different execution paths and measurement results.

This is particularly true for Synchronized Network Emulation. Virtual Machines are non-deterministic. Therefore a VM may start to interact with a network simulation at any point in time by communicating with a simulated host. This yields to previously unscheduled activities, for instance to handle the connection or for sending a response, inside the simulation domain. Hence, the non-deterministic execution of the VM(s) directly propagates to the network simulation and hence disrupts its deterministic execution.

For achieving a fully deterministic execution of a SHE deployment a deterministic execution of all involved SRs is needed. In the case of SNE this would require the execution of the VMs to be made deterministic. In the context of the SliceTime implementation, this would require non-trivial changes to the Xen infrastructure to make the execution of a Xen VM deterministic. We did not address this challenge in our work. However we have recently learned about XenTT [BJK+12], a framework that enables the deterministic execution of Xen VM guests. We believe that XenTT could well serve as potential basis for future research efforts in this direction.

Static Interfaces

We have made the design decision to integrate the different system representations using two interfaces, namely the Time Control Interface (TCI) and the Packet Exchange Interface (PEI). Both of them reduce the time synchronization and the exchange of network packets to a common denominator. The TCI is based on the delivery on time slices, while the PEI expects all system representations to exchange network packets at the MAC layer. The motivation behind this design decision has been to impose as few as possible implementation constraints on the different SRs.

However, in certain cases our static interfaces lead to non-optimal integrations of system representations. One example is our work on the integration of ns-3 with a SystemC-based hardware simulation. As both SRs are based on the concept of discrete event-based simulation, here it would be possible to apply an event-aware synchronization scheme, which for instance would enable the setup to skip idle periods. However, our simple TCI scheme does not make use of discrete event timings and hence results in a comparatively inefficient time integration of both SRs. Also, according to the PEI our framework at present requires a full simulation or implementation of the MAC layer, even if all system representations operate at higher levels at the ISO/OSI stack. For a more flexible SR integration it is imaginable to replace these simple interfaces with a capability-based interface that enables SRs to automatically pick the best fitting synchronization and packet exchange scheme.

7.3 Future Research and Possible Extensions

We now briefly point out future research directions for SNE and SHE.

Support for Further System Representations

The flexibility of Synchronized Hybrid Evaluation would naturally benefit from additional system representations. The research group of Sasu Tarkoma at the University of Helsinki has recently implemented an SR for SliceTime based on QEMU [Bel05]

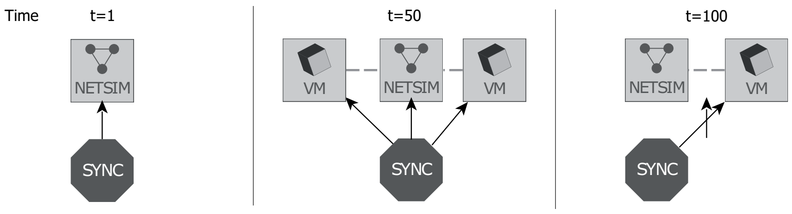


Figure 7.1 Proposed Extension of SHE: Schedulable dynamics in SHE configurations would enable better and flexible modeling of churn in P2P networks. In this illustration, a network simulation is amended with two virtual machines at t=10; one VM would leave the composition at a later time.

in order to enable emulation studies with the Android [But11] mobile platform. As possible future SRs we specifically consider the integration of special purpose simulation tools, for example specialized P2P simulators [SGR+11, MJ09, BHK09] or simulators for wireless sensor networks, into our platform.

Dynamic Synchronized Hybrid Evaluation (SHE) compositions

So far, we have assumed SHE compositions to be of static nature. We strongly believe that the flexibility of SHE setups can be greatly improved by making the respective compositions dynamic over time. In this regard, we envision that our methodology can be advanced in two different ways.

1. **Schedulable SHE dynamics**: The core idea of this possible enhancement (cf. Fig. 7.1) is to schedule the dynamics of a SHE setup over time. This would enable compositions, in which SRs are joining and leaving in a controlled fashion. We believe this would be helpful for modeling churn in a P2P emulation scenario or for investigating dynamics in mobile networking environments.

   In fact, all available SR implementations, both the PEI and TCI interfaces as well as the synchronizer already support the composition to be changed during an evaluation run. The framework is currently solely lacking a global coordination framework that would take care of scheduling the composition dynamics. Aktas et al. have lately designed a control framework [AvLH+12] for SliceTime [WSvL+11] and DDWNE [WvLW11]. We believe that this framework could well serve as starting point to implement support for schedulable and dynamic compositions.

2. **Live Migration of System Representations**: At present, we assume SRs to always operate on the same physical computer. Decoupling this strong bond would enable system representations to be adaptively remigrated to a computer with higher computational complexity if the present host, for instance a simulation server or a VM host, is overloaded. It is also imaginable that the execution of system representations is dynamically offloaded to a cloud service, for instance if not enough computational resources are available.

3. **Dynamic Replacements of System Representations:** Finally, we believe that an exciting area of research would be the dynamic replacement of a system representation with a different type. For example, consider a BitTorrent (BT) simulation that models a set of BT nodes. With SR replacement we would be able to replace a simulated BitTorrent client with a different representation, for instance a VM running a real BitTorrent client, at any point during the evaluation. In order to facilitate this replacement, it would be required to transform and to transfer the local state from one SR to another. We believe that this is a very difficult problem to tackle. However, if we assume that the simulation model and the BT client share an equivalent basis, for instance the same finite state machine, implementing such a scheme seems to be feasible if the according development resources are available.

Graphical User Interface for Live Visualization and Control

At present, controlling a SHE environment involves concurrently handling a number of command-line OS shells; each shell controls a machine executing SRs or the synchronizer. The visualization of measurement data is usually carried out as a post-processing task using tools like GNUPLOT [WKC+90] or Matplotlib [Hun07].

While this mode of operation is certainly viable for network researchers with the required technical skills, we think that a Graphical User Interface (GUI) will greatly simplify the usability of a SHE environment. Frameworks like OMNeT++ [VH08] have demonstrated that a GUI strongly eases the control and the understandability of complex network simulations. For this reason, OMNeT++ is well suited for education purposes [Var99].

We have different ideas and visions how SHE could benefit from an adequate user interface. For instance, we believe that composing SRs using a visual editor would improve the clarity and the maintainability of more complex SR configurations, especially if they should be of time-dynamic nature. In addition, controlling and visualizing a SHE setup in a central user interface might make it more accessible for less experienced users being interested in evaluating systems in a hybrid environment. Finally, we think that integrating a distributed debugging framework, such as our work discussed in Chapter 6, into an Integrated Development Environment (IDE), would enable a wider audience to apply our techniques for software testing.

Semi-automatic Model Calibration

A novel research direction beyond the scope of this dissertation is *semi-automatic model calibration*. As discussed earlier in this document, contemporary network simulation models typically well remodel the functional aspects of a target system or protocol. However, they mostly neglect important system characteristics, particularly the resource usage. This makes it difficult to employ them for the evaluation of system-related performance phenomena. Alizai and Landsiedel have earlier shown that simulators can be manually retrofitted with models for approximating the power and the timing needs of sensor network applications. [LWG05, LAW08]. This was mainly achieved by manually measuring the target system and by adding respective statements in the corresponding simulation model code.

By contrast, we envision the resource estimators to be automatically derived or calibrated using an adequate measurement infrastructure. We believe this will enable general simulation models, for example of network protocols, to be tuned for remodeling the behavior of a specific target system. In this respect, Sebastian Schöppel, whose Diploma thesis was co-advised by the author of this dissertation, has recently demonstrated that a general HTTP model can be automatically calibrated to closely resemble the resource usage of web applications [Sch12]. This opens up new application domains for network simulation, for example in the domain of capacity planning.

With regard to this thesis we believe that this fresh methodology and SHE can mutually benefit from each other. The integration of the tool-chain developed by Schöppel into SHE would enable emulation experiments with a higher level of precision at the simulation side. The calibration-measurement infrastructure itself might benefit from the virtualization of time, as it would enable closer investigations of the target systems.

7.4 Final Remarks

Is Synchronized Hybrid Evaluation (SHE) the swiss army knife for any performance evaluation study of a communication system? Certainly not! Instead we emphasize that most performance evaluation questions can be fully answered with a single analytical, simulative or measurement-based technique. If one of these methodologies is sufficient, applying a hybrid evaluation method mostly does not provide any additional benefit.

However, hybrid performance evaluation methods are well suited for situations where one sole evaluation technique cannot be used to answer a performance question. One example of such a problem is the frequent need of evaluating the real-world performance of software in a fully controllable environment; and as we have discussed, network emulation is a first-rate methodology to tackle such evaluation problems.

In this thesis we have shown that hybrid evaluation methods such as network emulation greatly benefit from the synchronization and virtualization of time. In summary, we believe that our work has advanced the development of hybrid evaluation methodologies. We are both excited and curious how the further development and research on synchronized hybrid evaluation will progress.

Bibliography

[AA06] Keith Adams and Ole Agesen, *A comparison of software and hardware techniques for x86 virtualization*, Proceedings of the 12th international conference on Architectural support for programming languages and operating systems (New York, NY, USA), ASPLOS-XII, ACM, 2006, pp. 2–13.

[ADH$^+$08] J. Ahrenholz, C. Danilov, T.R. Henderson, J.H. Kim, and B.P. Works, *Core: A real-time network emulator*, Proceedings of the IEEE Military Communications Conference (MILCOM), 2008, pp. 1–7.

[Alm99] Werner Almesberger, *Linux network traffic control - implementation overview*, Proceedings of the 5th Annual Linux Expo (Raleigh, NC), 05 1999.

[amd08] *AMD-V^{TM} Nested Paging (white paper)*, `http://developer.amd.com/assets/NPT-WP-1%201-final-TM.pdf` (accessed 10/2012), 07 2008.

[amd12] *AMD64 Architecture Programmer's Manual Volume 2: System Programming, revision 3.22*, `http://support.amd.com/us/Processor_TechDocs/24593_APM_v2.pdf` (accessed 10/2012), 09 2012.

[AO97] Mark Allman and Shawn Ostermann, *ONE: The Ohio Network Emulator*, Technical Report TR-19972, Ohio University, August 1997.

[Ath03] Atheros, *White paper – 802.11 wireless LAN performance*, `http://www.atheros.com/whitepapers/atheros_range_whitepaper.pdf`, 4 2003, (accessed May 23, 2010).

[AV06] Marco Avvenuti and Alessio Vecchio, *Application-level network emulation: the emusocket toolkit*, Journal of Network and Computer Applications **29** (2006), no. 4, 343–360.

[AvLH$^+$12] Ismet Aktas, Hendrik vom Lehn, Christoph Habets, Florian Schmidt, and Klaus Wehrle, *Fantasy: Fully automatic network emulation architecture with cross-layer support*, Proceedings of the 5th ACM International ICST Conference on Simulation Tools and Techniques (SIMUTools '12), ICST, Brussels, Belgium, 6 2012.

[Ayn09] John Aynsley, *OSCI TLM-2.0 language reference manual*, July 2009.

[Bac87] K. Baclawski, *A network emulation tool*, Symposium on the simulation of computer networks (Piscataway, NJ, USA), IEEE Press, 1987, pp. 198–206.

[Bae05] J. C. M. Baeten, *A brief history of process algebra*, Theoretical Computer Science **335** (2005), no. 2–3, 131–146.

[Bar04] Rimon Barr, *Swans – scalable wireless ad hoc network simulator user guide*, Online Ressource http://jist.ece.cornell.edu/swans-user/ (accessed 11/2012), 2004.

[Bau10] Mick Bauer, *Paranoid penguin: Linux VPNs with OpenVPN*, Linux Journal **2010** (2010), no. 190.

[BBB$^+$11] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood, *The gem5 simulator*, SIGARCH Comput. Archit. News **39** (2011), no. 2, 1–7.

[BBD01] H. Bowman, J. W. Bryans, and J. Derrick, *Analysis of a multimedia stream using stochastic process algebra*, The Computer Journal **44** (2001), no. 4, 230–245.

[BD91] B. Berthomieu and M. Diaz, *Modeling and verification of time dependent systems using time petri nets*, IEEE Transactions on Software Engineering **17** (1991), no. 3, 259 –273.

[BDF$^+$03] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Adrew Warfield, *Xen and the art of virtualization*, Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP'03) (Bolton Landing, NY, USA), ACM, October 2003, pp. 164–177.

[BDF$^+$09] Pierre Borgnat, Guillaume Dewaele, Kensuke Fukuda, Patrice Abry, and Kenjiro Cho, *Seven years and one day: Sketching the evolution of internet traffic*, Proc. INFOCOM, IEEE, 2009, pp. 711–719.

[BDH$^+$06] Nathan L. Binkert, Ronald G. Dreslinski, Lisa R. Hsu, Kevin T. Lim, Ali G. Saidi, and Steven K. Reinhardt, *The M5 simulator: Modeling networked systems*, IEEE Micro **26** (2006), no. 4, 52–60.

[BEG$^+$04] Patrick Bohrer, Mootaz Elnozahy, Achmed Geith, Charles Lefurgy, Tarun Nakra, James Peterson, Ram Rajamony, Ron Rockhold, Hazim Shafi, Rick Simpson, Evan Speight, Kartik Sudeep, Eric van Hensbergen, and Lixin Zhang, *Mambo: a full system simulator for the PowerPC architecture*, ACM SIGMETRICS Performance Evaluation Review **31** (2004), no. 4, 8–12.

[Bel05] Fabrice Bellard, *QEMU, a fast and portable dynamic translator*, Proceedings of the USENIX Annual Technical Conference, USENIX, 2005, pp. 41–46.

[Ber06] Craig Casey Bergstrom, *The distributed open network emulator: Applying relativistic time*, Master's thesis, Virginia Tech, 09 2006.

[BGP04] Suman Banerjee, Timothy Griffin, and Marcelo Pias, *The Interdomain connectivity of PlanetLab nodes*, Passive and Active Network Measurement (Chadi Barakat and Ian Pratt, eds.), Lecture Notes in Computer Science, vol. 3015, Springer Berlin / Heidelberg, 2004, pp. 73–82.

[BHK09] Ingmar Baumgart, Bernhard Heep, and Stephan Krause, *Oversim: A scalable and flexible overlay framework for simulation and real network applications*, Proceedings P2P 2009, Ninth International Conference on Peer-to-Peer Computing, 9-11 September 2009, Seattle, Washington, USA (Henning

Schulzrinne, Karl Aberer, and Anwitaman Datta, eds.), IEEE, 2009, pp. 87–88.

[BHvR05] Rimon Barr, Zygmunt J. Haas, and Robbert van Renesse, *JiST: an efficient approach to simulation using virtual machines*, Software: Practice and Experience **35** (2005), no. 6, 539–576.

[BJK$^+$12] Anton Burtsev, David Johnson, Chung Hwan Kim, Mike Hibler, Eric Eide, and John Regehr, *XenTT: Deterministic systems analysis in Xen*, Talk given at the XenSummit, San Diego, 08 2012.

[BK86] J. A. Bergstra and J. W. Klop, *Algebra of communicating processes*, Mathematics and Computer Science (J. W. de Bakker, M. Hazewinkel, and J. K. Lenstra, eds.), CWI Monograph 1, North-Holland, Amsterdam, 1986, pp. 89–138.

[Blo04] Eric Blossom, *Gnu radio: tools for exploring the radio frequency spectrum*, Linux J. **2004** (2004), no. 122, 4–.

[BLPK07] Pere Bonet, Catalina M Lladó, Ramon Puigjaner, and William J. Knottenbelt, *PIPE 2.5: a petri net tool for performance modeling*, Proc. 23rd Latin American Conference on Informatics (CLEI 2007), October 2007.

[BNM$^+$09] R. Beuran, Lan Tien Nguyen, T. Miyachi, J. Nakata, K.-I. Chinen, Y. Tan, and Y. Shinoda, *Qomb: A wireless network emulation testbed*, Proceedings of the IEEE Global Telecommunications Conference (GLOBECOM), 30 2009-dec. 4 2009, pp. 1 –6.

[Bou07] Athanassios Boulis, *Castalia: revealing pitfalls in designing distributed algorithms in wsn*, Proceedings of the 5th international conference on Embedded networked sensor systems (New York, NY, USA), SenSys '07, ACM, 2007, pp. 407–408.

[BSUU00] Russell Bradford, Rob Simmonds, Brian Unger, and Bath Uk, *A parallel discrete event ip network emulator*, In Proceedings of the Ninth International Symposium on the Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS'00), 2000, pp. 315–322.

[But11] M. Butler, *Android: Changing the mobile landscape*, Pervasive Computing, IEEE **10** (2011), no. 1, 4 –7.

[BVB06] Craig Bergstrom, Srinidhi Varadarajan, and Godmar Back, *The distributed open network emulator: Using relativistic time for distributed scalable simulation*, 20th Workshop on Principles of Advanced and Distributed Simulation (PADS'06) (2006).

[BY88] Jed Schwartz David F. Bacon and Yechiam Yemini, *Nest: a network simulation and prototyping tool*, Proceedings of the USENIX Winter Conference (Berkeley, CA, USA), USENIX Association, January 1988, pp. 71–78.

[Car09] Gustavo Carneiro, *Ns-3: Wifi scanning patch*, `http://www.nsnam.org/contributed/ns-3-wifi-scanning.tar.bz2`, 2009, accessed Oct 27, 2010.

[CCR$^+$03] Brent Chun, David Culler, Timothy Roscoe, Andy Bavier, Larry Peterson, Mike Wawrzoniak, and Mic Bowman, *PlanetLab: An Overlay Testbed for Broad-Coverage Services*, ACM SIGCOMM Computer Communication Review **33** (2003), no. 3, 3–12.

[CGHT07] Allan Clark, Stephen Gilmore, Jane Hillston, and Mirco Tribastone, *Stochastic process algebras*, Formal Methods for Performance Evaluation (Marco Bernardo and Jane Hillston, eds.), Lecture Notes in Computer Science, vol. 4486, Springer Berlin / Heidelberg, 2007, pp. 132–179.

[CGT89] Stefano Ceri, Georg Gottlob, and Letizia Tanca, *What you always wanted to know about Datalog (and never dared to ask)*, IEEE Transactions on Knowledge and Data Engineering **1** (1989), no. 1, 146–166.

[CGV+04] Jay Chen, Diwaker Gupta, Kashi V. Vishwanath, Alex C. Snoeren, and Amin Vahdat, *Routing in an internet-scale network emulator*, Proceedings of the The IEEE Computer Society's 12th Annual International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems (Washington, DC, USA), MASCOTS '04, IEEE Computer Society, 2004, pp. 275–283.

[Chi08] David Chisnall, *The definitive guide to the Xen hypervisor*, Prentice Hall open source software development series, Prentice-Hall, pub-PH:adr, 2008.

[CLM+08] Brendan Cully, Geoffrey Lefebvre, Dutch Meyer, Mike Feeley, Norm Hutchinson, and Andrew Warfield, *Remus: high availability via asynchronous virtual machine replication*, Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation (Berkeley, CA, USA), NSDI'08, USENIX Association, 2008, pp. 161–174.

[CM79] K. M. Chandy and J. Misra, *Distributed simulation: A case study in design and verification of distributed programs*, IEEE Trans. on Software Engineering **SE-5** (1979), no. 5, 440–452.

[CNO99] J.H. Cowie, D.M. Nicol, and A.T. Ogielski, *Modeling the global internet*, Computing in Science & Engineering **1** (1999), no. 1, 42–50.

[Cob11] Stephen Cobb, *Satellite Internet Connection for Rural Broadband (Whitepaper)*, Rural Mobile & Broadband Alliance (RuMBA), 2011.

[Coh03] B. Cohen, *Incentives build robustness in bittorrent*, Proceedings of the Workshop on Economics of Peer-to-Peer Systems (Berkeley, CA, USA), 2003 (English).

[cow] *CoWare Virtual Platform*, [Online] Available `http://www.coware.com/products/virtualplatform.php` (accessed 08/2009).

[CPF99] Christopher D. Carothers, Kalyan S. Perumalla, and Richard M. Fujimoto, *Efficient optimistic parallel simulations using reverse computation*, ACM Trans. Model. Comput. Simul. **9** (1999), 224–253.

[CR10] Marta Carbone and Luigi Rizzo, *Dummynet revisited*, Computer Communication Review **40** (2010), no. 2, 12–20.

[CRM] *CoreMark Benchmarking Tool*, `http://www.coremark.org/` (accessed 10/2012).

[CS03] Mark Carson and Darrin Santay, *NIST Net: a linux-based network emulation tool*, ACM SIGCOMM Computer Communication Review **33** (2003), no. 3, 111–126.

[DD08] Amogh Dhamdhere and Constantine Dovrolis, *Ten years in the evolution of the Internet ecosystem*, Proceedings of the 8th ACM SIGCOMM conference on Internet measurement (New York, NY, USA), IMC '08, ACM, 2008, pp. 183–196.

[del] *Deluge BitTorrent client*, `http://deluge-torrent.org/` (accessed 08/2012).

[DFP02] N. Drago, F. Fummi, and M. Poncino, *Modeling network embedded systems with ns-2 and SystemC*, Circuits and Systems for Communications, 2002. Proceedings. ICCSC '02. 1st IEEE International Conference on, 2002, pp. 240 – 245.

[Dik01] Jeff Dike, *User-mode Linux*, Proceedings of the 5th Annual Linux Showcase and Conference, November 5–10, 2001, Oakland, CA, 2001.

[DMNO11] Wan Du, Fabien Mieyeville, David Navarro, and Ian O'Connor, *IDEA1: A validated systemC-based system-level design and simulation environment for wireless sensor networks*, EURASIP J. Wireless Comm. and Networking **2011** (2011), 143.

[DR92] Paul S. Dodd and Chinya V. Ravishankar, *Monitoring and debugging distributed real-time programs*, Software—Practice and Experience **22** (1992), no. 10, 863–877.

[DSYB90] A. Dupuy, J. Schwartz, Y. Yemini, and D. Bacon, *NEST: A network simulation and prototyping testbed*, Communications of the ACM, CACM **33** (1990), no. 10, 63–74.

[Dun03] Adam Dunkels, *Full tcp/ip for 8-bit architectures*, Proceedings of the 1st international conference on Mobile systems, applications and services (New York, NY, USA), MobiSys '03, ACM, 2003, pp. 85–98.

[EAHC05] Christian J. Eibl, Carsten Albrecht, Rainer Hagenau, and Simulation Controller, *gsysc: A graphical front end for systemc*, In European Conference on Modelling and Simulation, 2005, pp. 257–262.

[EAW10] Jakob Engblom, Daniel Aarno, and Bengt Werner, *Full-system simulation from embedded to high-performance systems*, Processor and System-on-Chip Simulation, Springer US, 2010, pp. 25–45.

[EAWJ02] E. N. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson, *A survey of rollback-recovery protocols in message-passing systems*, ACM Computing Surveys **34** (2002), no. 3, 375–408.

[EE06] Jakob Engblom and Dan Ekblom, *Simics: a commercially proven full-system simulation framework*, Workshop on Simulation in European Space Programmes (SESP 2006) (Noordwijk, Netherlands), 11 2006.

[EG11] Nathan Evans and Christian Grothoff, *Beyond simulation: Large-scale distributed emulation of P2P protocols*, 4th Workshop on Cyber Security Experimentation and Test (CSET 2011), USENIX Association, 2011.

[EKMR05] J. Engblom, D. Kagedal, A. Moestedt, and J. Runeson, *Developing embedded networked products using the simics full-system simulator*, Personal, Indoor and Mobile Radio Communications, 2005. PIMRC 2005. IEEE 16th International Symposium on, vol. 2, sept. 2005, pp. 785 –789 Vol. 2.

[ELL09] Miguel A. Erazo, Yue Li, and Jason Liu, *SVEET! a scalable virtualized evaluation environment for TCP*, Proc. TRIDENTCOM'09, IEEE Computer Society, 2009, pp. 1–10.

[Els88] I. J. P. Elshoff, *A distributed debugger for Amoeba*, Proceedings of the 1988 ACM SIGPLAN and SIGOPS workshop on Parallel and distributed debugging (New York, NY, USA), PADD '88, ACM, 1988, pp. 1–10.

[emu] *Emulab documentation*, Online available at `https://users.emulab.net/trac/emulab/wiki/` (accessed 06/2012).

[Eng09] Jakob Engblom, *Simics Network Simulation*, White Paper (Virtutech), available at `http://www.virtutech.com/files/whitepapers/wp-networking-2009-03-30-letter.pdf` (accessed 11/2012), 03 2009.

[EÖF$^+$09] Joakim Eriksson, Fredrik Österlind, Niclas Finne, Nicolas Tsiftes, Adam Dunkels, Thiemo Voigt, Robert Sauter, and Pedro José Marrón, *COOJA/MSPSim: interoperability testing for wireless sensor networks*, Proceedings of the 2nd International Conference on Simulation Tools and Techniques for Communications, Networks and Systems (SIMUTools) 2009, ICST, 03 2009, p. 27.

[Fal99] Kevin R. Fall, *Network emulation in the Vint/NS simulator*, Proceedings of the 4th IEEE Symposium on Computers and Communication, IEEE Computer Society, 1999, pp. 244–250.

[FBB$^+$05] W. Feng, P. Balaji, C. Baron, L.N. Bhuyan, and D.K. Panda, *Performance characterization of a 10-Gigabit Ethernet TOE*, Proceedings of the 13th Symposium on High Performance Interconnects, aug. 2005, pp. 58 – 63.

[FK03] Sally Floyd and Eddie Kohler, *Internet research needs better models*, Computer Communication Review **33** (2003), no. 1, 29–34.

[FP01] Sally Floyd and Vern Paxson, *Difficulties in simulating the Internet*, IEEE/ACM Transactions on Networking **9** (2001), no. 4, 392–403.

[FPM$^+$04] F. Fummi, M. Poncino, S. Martini, F. Ricciato, G. Perbellini, and M. Turolla, *Heterogeneous co-simulation of networked embedded systems*, Design, Automation and Test in Europe Conference and Exhibition, 2004. Proceedings, vol. 3, feb. 2004, pp. 168 – 173 Vol.3.

[FPS$^+$11] Chien-Liang Fok, Agoston Petz, Drew Stovall, Nicholas Paine, Christine Julien, and Sriram Vishwanath, *Pharos: A testbed for mobile cyber-physical systems*, Tech. report, University of Texas at Austin, 2011.

[Fre08] Bruce Frederiksen, *Applying expert system technology to code reuse with Pyke*, Proceedings of the PyCon conference (Chicago, Illinois), 03 2008.

[Fuj88] Richard M. Fujimoto, *Lookahead in parallel discrete event simulation*, Proceedings of the 1988 International Conference on Parallel Processing. Vol. III, Algorithms and Applications, ICPP'88 (St. Charles, IL, August 15-19, 1988) (University Park-London) (David H. Bailey, ed.), Pennsylvania State University, Pennsylvania State University Press, 1988, pp. 34–41.

[Fuj90] Richard M. Fujimoto, *Parallel discrete event simulation*, Commun. ACM **33** (1990), 30–53.

[Gar06] Manu Garg, *Sysenter based system call mechanism in linux 2.6*, Online Article, `http://articles.manugarg.com/systemcallinlinux2_6.html` (accessed 10/2012), 2006.

[GBST08] D. Giustiniano, G. Bianchi, L. Scalia, and I. Tinnirello, *An explanation for unexpected 802.11 outdoor link-level measurement results*, Proceedings of the 27th IEEE Conference on Computer Communications (INFOCOM 2008), 04 2008, pp. 2432 –2440.

[gdb] *GDB documentation: (the GNU source-level debugger)*, `http://www.gnu.org/software/gdb/documentation/` (accessed 10 2012).

[GG74] S. W. Galley and Robert P. Goldberg, *Software debugging: the virtual machine approach*, Proceedings of the 1974 annual ACM conference - Volume 2 (New York, NY, USA), ACM '74, ACM, 1974, pp. 395–401.

[GKN$^+$04] Robert S. Gray, David Kotz, Calvin Newport, Nikita Dubrovsky, Aaron Fiske, Jason Liu, Christopher Masone, Susan Mcgrath, and Yougu Yuan, *Outdoor experimental comparison of four ad hoc routing algorithms*, Proceedings of the ACM/IEEE International Symposium on Modeling, Analysis and Simulation of Wireless and Mobile Systems (MSWiM), 2004, pp. 220–229.

[GKN$^+$06] Robert S. Gray, David Kotz, Calvin Newport, Nikita Dubrovsky, Aaron Fiske, Jason Liu, Christopher Masone, Susan McGrath, and Yougu Yuan, *CRAWDAD data set dartmouth/outdoor (v. 2006-11-06)*, Downloaded from `http://crawdad.cs.dartmouth.edu/dartmouth/outdoor`, November 2006.

[Gle11] René Glebke, *A Benchmarking Platform for BitTorrent-based Video-on-Demand Protocols*, Bachelor Thesis,RWTH Aachen University, 05 2011.

[GLMS02] Thorsten Grötker, Stan Liao, Grant Martin, and Stuart Swan, *System design with SystemC*, Springer US, 2002.

[GLT04] Yu Gu, Yong Liu, and D. Towsley, *On integrating fluid models with packet simulation*, Proceedings of the Twenty-third AnnualJoint Conference of the IEEE Computer and Communications Societies(INFOCOM), vol. 4, march 2004, pp. 2856 – 2866 vol.4.

[GMGK84] Hector Garcia-Molina, Frank Germano, and Walter H. Kohler, *Debugging a distributed computing system*, IEEE Transactions on Software Engineering **SE-10** (1984), no. 2, 210 –219.

[GMHR08] A. Grau, S. Maier, K. Herrmann, and K. Rothermel, *Time jails: A hybrid approach to scalable network emulation*, Proceedings of the 22nd Workshop on Principles of Advanced and Distributed Simulation (PADS) (2008).

[Gol73] Robert P. Goldberg, *Architectural principles for virtual computer systems*, Master's thesis, Harvard University, 2 1973.

[GR03] Tal Garfinkel and Mendel Rosenblum, *A virtual machine introspection based architecture for intrusion detection*, Proceedings of the Network and Distributed Systems Security Symposium, 2003, pp. 191–206.

[Gra86] Jim Gray, *Why do computers stop and what can be done about it?*, Proceedings of the Fifth Symposium on Reliability in Distributed Software and Database Systems (5th SRDS'86) (Los Angeles, CA, USA), IEEE Computer Society Press, January 1986 July 1986, pp. 3–12.

[Gra95] B. Grahlmann, *PEP: A programming environment based on petri nets.*, ATPN'95, Tool Presentation, Torino, Italy (1995), 1–6 pp., Internal-Note: Submitted by: bernd@informatik.uni-hildesheim.de available at http://www.informatik.uni-hildesheim.de/ pep.

[GVM$^+$11] Diwaker Gupta, Kashi Venkatesh Vishwanath, Marvin McNett, Amin Vahdat, Ken Yocum, Alex Snoeren, and Geoffrey M. Voelker, *Diecast: Testing distributed systems with an accurate scale model*, ACM Transactions on Computer Systems **29** (2011), no. 2, 4:1–4:48.

[GVV08] Diwaker Gupta, Kashi V. Vishwanath, and Amin Vahdat, *DieCast: Testing distributed systems with an accurate scale model.*, Proceedings of the 5th USENIX Symposium on Networked System Design and Implementation (NSDI'08) (San Francisco, CA, USA), USENIX Association, 2008.

[GYM$^+$06] Diwaker Gupta, Kenneth Yocum, Marvin McNett, Alex C. Snoeren, Amin Vahdat, and Geoffrey M. Voelker, *To infinity and beyond: time-warped network emulation*, Proceedings of the 3rd USENIX Symposium on Networked Systems Design and Implementation (NSDI'06) (Berkeley, CA, USA), USENIX Association, 2006, pp. 87–100.

[Hel99] G.R. Hellestrand, *The revolution in systems engineering*, IEEE Spectrum **36** (1999), no. 9, 43–51.

[Hem05] Stephen Hemminger, *Network emulation with NetEm*, Proceedings of Australia's 6th national Linux conference (LCA) (Sydney NSW, Australia) (Martin Pool, ed.), Linux Australia, Linux Australia, April 2005.

[HHH04] Alex Ho, Steven Hand, and Tim Harris, *Pdb: Pervasive debugging with xen*, GRID '04: Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing (Washington, DC, USA), IEEE Computer Society, 2004, pp. 260–265.

[HL01] Y. Hu and V. Li, *Satellite-based internet: A tutorial*, IEEE Communications Magazine **39** (2001), no. 3, 154–162.

[Hoa78] C. A. R. Hoare, *Communicating sequential processes*, Communications of the ACM **21** (1978), no. 8, 666–677.

[Hoc12] Alexander Jacob Hocks, *Emulating BitTorrent Swarms using a Discrete-Event Based Simulator*, Bachelor Thesis,RWTH Aachen University, 03 2012.

[Hof08] John Hoffman, *BEP 16 - Superseeding (draft)*, `http://bittorrent.org/beps/bep_0016.html`, 02 2008.

[Hoo96] Robert Hood, *The project: building a portable distributed debugger*, Proceedings of the SIGMETRICS symposium on Parallel and distributed tools (New York, NY, USA), SPDT '96, ACM, 1996, pp. 127–136.

[HR12] Fabien Hermenier and Robert Ricci, *How to build a better testbed: Lessons from a decade of network experiments on emulab*, Proceedings of the 8th International ICST Conference on Testbeds and Research Infrastructures for the Development of Networks and Communities (Tridentcom), 06 2012.

[HRFR06] Thomas R. Henderson, Sumit Roy, Sally Floyd, and George F. Riley, *ns-3 project goals*, WNS2 '06: Proceeding from the 2006 workshop on ns-2: the IP network simulator (New York, NY, USA), ACM, 2006, p. 13.

[HSK99] X.W. Huang, R. Sharma, and S. Keshav, *The ENTRAPID protocol development environment*, Proceedings of the Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM), vol. 3, mar 1999, pp. 1107 –1115 vol.3.

[Hun07] John D. Hunter, *Matplotlib: A 2D graphics environment*, Computing in Science and Engineering **9** (2007), no. 3, 90–95.

[IBM03] IBM Cooperation, *PowerPC 440GP Embedded processor data sheet*, 04 2003.

[IP94] David B. Ingham and Graham D. Parrington, *Delayline: a wide-area network emulation tool*, Computing Systems **7** (1994), no. 3, 313–332.

[JA06] David Long John Aynsley, *IEEE Standard SystemC language reference manual*, March 2006.

[Jai91] R. Jain, *The art of computer systems performance analysis*, John Wiley & Sons, Inc. (1991).

[JCJ00] C. Jin, Q. Chen, and S. Jamin, *Inet: Internet Topology Generator, Technical Report CSE-TR443-00*, Tech. report, Department of EECS, University of Michigan, 2000.

[JCS] R. Jones, K. Choy, and D. Shield, *Netperf*, [Online] Available `http://www.netperf.org` (accessed 11/2012).

[JS85] D. R. Jefferson and H. Sowizral, *Fast concurrent simulation using the time warp mechanism*, Simulation Series, Society for Computer Simulation 24-26 Jan 1985 **15** (1985), 63–69.

[JS04] G. Judd and P. Steenkiste, *Repeatable and realistic wireless experimentation through physical emulation*, ACM SIGCOMM Computer Communication Review **34** (2004), no. 1, 63–68.

[JS05] Glenn Judd and Peter Steenkiste, *Using emulation to understand and improve wireless networks and applications*, Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation - Volume 2 (Berkeley, CA, USA), NSDI'05, USENIX Association, 2005, pp. 203–216.

[JSF$^+$06] David Johnson, Tim Stack, Russ Fish, Daniel Montrallo Flickinger, Leigh Stoller, Robert Ricci, and Jay Lepreau, *Mobile Emulab: A robotic wireless and sensor network testbed*, Proc. INFOCOM, IEEE, 2006.

[KBHS07] Tronje Krop, Michael Bredel, Matthias Hollick, and Ralf Steinmetz, *JiST/MobNet: combined simulation, emulation, and real-world testbed for ad hoc networks*, Proc. WinTECH'07 (New York, NY, USA), ACM, 2007, pp. 27–34.

[KCC05] Stuart Kurkowski, Tracy Camp, and Michael Colagrosso, *Manet simulation studies: the incredibles*, SIGMOBILE Mob. Comput. Commun. Rev. **9** (2005), 50–61.

[KDC05] Samuel T. King, George W. Dunlap, and Peter M. Chen, *Debugging operating systems with time-traveling virtual machines*, Proceedings of the annual conference on USENIX Annual Technical Conference (Berkeley, CA, USA), USENIX Association, 2005, pp. 1–1.

[Ker] Mike Kershaw, *Kismet wireless network detector and sniffer*, `http://www.kismetwireless.net/` (accessed Oct.2012).

[KGM$^+$01] Markku Kojo, Andrei Gurtov, Jukka Manner, Pasi Sarolahti, Timo Alanko, and Kimmo Raatikainen, *Seawind: a wireless network emulator*, In Proceedings of 11th GI/ITG Conference on Measuring, Modelling and Evaluation of Computer and Communication Systems, 2001.

[Kiv07] Avi Kivity, *KVM: the linux virtual machine monitor*, OLS '07: The 2007 Ottawa Linux Symposium, July 2007, pp. 225–230.

[KNM06] N.A. Kamble, J. Nakajima, and A.K. Mallick, *Evolution in Kernel Debugging using Hardware Virtualization With Xen*, Proceedings of the Ottawa Linux Symposium, 2006.

[KSU05] Cameron Kiddle, Rob Simmonds, and Brian Unger, *Improving scalability of network emulation through parallelism and abstraction*, Proceedings of the 38th annual Symposium on Simulation (Washington, DC, USA), ANSS '05, IEEE Computer Society, 2005, pp. 119–129.

[Lac10] Mathieu Lacage, *Direct code execution with ns-3*, Talk at the Workshop on NS-3, Slides available at `http://www.nsnam.org/workshops/wns3-2010/code-execution.pdf` (accessed 08/2012), 3 2010.

[Lam78] Leslie Lamport, *Time, clocks, and the ordering of events in a distributed system*, Communications of the ACM **21** (1978), no. 7, 558–565.

[Law96] Kevin P. Lawton, *Bochs: A portable pc emulator for unix/x*, Linux Journal **1996** (1996), no. 29es.

[LAW08] Olaf Landsiedel, Hamad Alizai, and Klaus Wehrle, *When timing matters: Enabling time accurate and scalable simulation of sensor network applications*, Proceedings of the 7th international conference on Information processing in sensor networks (Washington, DC, USA), IPSN '08, IEEE Computer Society, 2008, pp. 344–355.

[LCH$^+$05] Boon Thau Loo, Tyson Condie, Joseph M. Hellerstein, Petros Maniatis, Timothy Roscoe, and Ion Stoica, *Implementing declarative overlays*, SIGOPS Oper. Syst. Rev. **39** (2005), no. 5, 75–90.

[LGW$^+$08] Xuezheng Liu, Zhenyu Guo, Xi Wang, Feibo Chen, Xiaochen Lian, Jian Tang, Ming Wu, M. Frans Kaashoek, and Zheng Zhang, *D^3S: Debugging deployed distributed systems*, NSDI, USENIX Association, 2008, pp. 423–437.

[LH06] M. Lacage and T.R. Henderson, *Yet another network simulator*, Proceeding from the 2006 workshop on ns-2: the IP network simulator, ACM, 2006, p. 12.

[Lin95] Christoph Lindemann, *DSPNexpress: A software package for the efficient solution of deterministic and stochastic petri nets*, Perform. Eval **22** (1995), no. 1, 3–21.

[LLPM$^+$03] Yong Liu, Francesco Lo Presti, Vishal Misra, Don Towsley, and Yu Gu, *Fluid models and solutions for large-scale IP networks*, Proceedings of the 2003 ACM SIGMETRICS international conference on Measurement and modeling of computer systems (New York, NY, USA), SIGMETRICS '03, ACM, 2003, pp. 91–101.

[LLWC03] Philip Levis, Nelson Lee, Matt Welsh, and David Culler, *TOSSIM: accurate and scalable simulation of entire tinyOS applications*, Proceedings of the first international conference on Embedded networked sensor systems (SenSys-03) (New York), ACM Press, November 5–7 2003, pp. 126–137.

[Loe08] Andrew Loewenstern, *BEP 05 - DHT Protocol (draft)*, `http://bittorrent.org/beps/bep_0005.html`, 02 2008.

[Lub89] B. D. Lubachevsky, *Efficient distributed event-driven simulations of multiple-loop networks*, Communications of the ACM **32** (1989), no. 1, 111–123,131.

[LW06] Christoph Lüders and Martin Winkler, *Pingpong: Wie die TCP/IP-Flusskontrolle das Surf-Tempo bestimmt*, c't Magazin für Computertechnik **23** (2006).

[LWG05] Olaf Landsiedel, Klaus Wehrle, and Stefan Götz, *Accurate prediction of power consumption in sensor networks*, Proceedings of the 2nd IEEE Workshop on Embedded Networked Sensors (EmNetS-II) (Sydney, Australia), IEEE, 2005, pp. 37–44.

[LZGS84] Edward D. Lazowska, John Zahorjan, G. Scott Graham, and Kenneth C. Sevcik, *Quantitative system performance: computer system analysis using queueing network models*, Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1984.

[MCE$^+$02] Peter S. Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hållberg, Johan Högberg, Fredrik Larsson, Andreas Moestedt, and Bengt Werner, *Simics: A full system simulation platform*, Computer **35** (2002), 50–58.

[Men07] Paul B. Menage, *Adding Generic Process Containers to the Linux Kernel*, Linux Symposium, Google Inc., June 2007.

[MFF$^+$97] S. McCanne, S. Floyd, K. Fall, K. Varadhan, et al., *Network simulator ns-2*, 1997.

[MH89] Charles E. McDowell and David P. Helmbold, *Debugging concurrent programs*, ACM Comput. Surv. **21** (1989), no. 4, 593–622.

[MHW02] Carl J. Mauer, Mark D. Hill, and David A. Wood, *Full-system timing-first simulation*, July 10 2002, p. 108.

[MI04] Daniel Mahrenholz and Svilen Ivanov, *Real-time network emulation with ns-2*, Proceedings of the 8th IEEE International Symposium on Distributed Simulation and Real-Time Applications (DS-RT), IEEE Computer Society, 2004, pp. 29–36.

[mic] *Microsoft hyper-V*, `http://www.microsoft.com/en-us/server-cloud/windows-server/server-virtualization.aspx` (accessed 10/2012).

[Mil99] Robin Milner, *Communicating and mobile systems: The π-calculus*, Cambridge University Press, 1999.

[MJ09] Alberto Montresor and Márk Jelasity, *Peersim: A scalable P2P simulator*, Proceedings P2P 2009, Ninth International Conference on Peer-to-Peer Computing, 9-11 September 2009, Seattle, Washington, USA (Henning Schulzrinne, Karl Aberer, and Anwitaman Datta, eds.), IEEE, 2009, pp. 99–100.

[MLMB01] Alberto Medina, Anukool Lakhina, Ibrahim Matta, and John W. Byers, *BRITE: An approach to universal topology generation*, Proceedings of the 9thth IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer Telecommunications Systems (MASCOTS), 2001.

[MM02] Petar Maymounkov and David Mazières, *Kademlia: A peer-to-peer information system based on the xor metric*, Peer-to-Peer Systems (Peter Druschel, Frans Kaashoek, and Antony Rowstron, eds.), Lecture Notes in Computer Science, vol. 2429, Springer Berlin / Heidelberg, 2002, pp. 53–65.

[MPM$^+$08] J. J. D. Mol, J. A. Pouwelse, M. Meulpolder, D. H. J. Epema, and H. J. Sips, *Give-to-Get: Free-riding-resilient Video-on-Demand in P2P Systems*, Multimedia Computing and Networking 2008, vol. 6818, SPIE Vol. 6818, January 2008.

[MRR08] B. Müller-Rathgeber and H. Rauchfuss, *A cosimulation framework for a distributed system of systems*, Proceedings of the 68th IEEE Vehicular Technology Conference, 2008 (VTC 2008-Fall), sept. 2008, pp. 1 –5.

[MSB$^+$05] Milo M. K. Martin, Daniel J. Sorin, Bradford M. Beckmann, Michael R. Marty, Min Xu, Alaa R. Alameldeen, Kevin E. Moore, Mark D. Hill, and David A. Wood, *Multifacet's general execution-driven multiprocessor simulator (gems) toolset*, SIGARCH Comput. Archit. News **33** (2005), no. 4, 92–99.

[MST$^+$05] Aravind Menon, Jose Renato Santos, Yoshio Turner, G. John Janakiraman, and Willy Zwaenepoel, *Diagnosing performance overheads in the Xen virtual machine environment*, Proceedings of the 1st International Conference on Virtual Execution Environments, VEE 2005, Chicago, IL, USA, June 11-12, 2005 (Michael Hind and Jan Vitek, eds.), ACM, 2005, pp. 13–23.

[Mue] Klaus Mueller, *SimPy documentation*, `http://simpy.sourceforge.net/documentation.htm` (accessed 11/2012).

[NR08] Lucas Nussbaum and Olivier Richard, *Lightweight emulation to study peer-to-peer systems*, Concurrency and Computation: Practice and Experience **20** (2008), no. 6, 735–749.

[ns] *The network simulator ns-2*, `http://www.isi.edu/nsnam/ns/` (accessed 11/2011).

[ns311] *ns-3 Website*, `http://www.nsnam.org/` (accessed May 2011), 2011.

[NSL$^+$06] Gil Neiger, Amy Santoni, Felix Leung, Dion Rodgers, and Rich Uhlig, *Intel Virtualization Technology: Hardware support for efficient processor virtualization*, Intel Technology Journal **10** (2006), no. 3, 167–177.

[ope] *OpenVZ website*, `http://openvz.org` (accessed 05/2012).

[opr] *OProfile: a system profiler for linux*, `http://oprofile.sourceforge.net` (accessed 10/2012).

[Pai10] Nicholas Arden Paine, *Design and development of a modular robot for research use*, Master's thesis, University of Texas at Austin, 2010.

[Pav05] Vojtech Pavlik, *Brief summary on time sources in present day x86 systems*, E-Mail on the Linux Kernel Mailing List; online available at `https://lkml.org/lkml/2005/11/18/261` (accessed August, 2011), 11 2005.

[PBM$^+$04] Jonathan Polley, Dionysys Blazakis, Jonathan Mcgee, Dan Rusk, and John S. Baras, *Atemu: A fine-grained sensor network simulator*, Proceedings of the IEEE Conference on Sensor and Ad Hoc Communications and Networks (SECON), 2004.

[PBRD03] Charles Perkins, Elizabeth M. Belding-Royer, and Suman Das, *RFC 3561 - Ad hoc OnDemand Distance Vector (AODV) Routing*, 2003.

[PD03] Larry Peterson and Bruce S. Davie, *Computer networks: A systems approach*, third ed., Morgan Kaufmann Publishers, pub-MORGAN-KAUFMANN:adr, 2003.

[Pet66] Carl Adam Petri, *Communication with automata*, Ph.D. thesis, Universität Hamburg, 1966.

[PIA$^+$07a] Michael Piatek, Tomas Isdal, Thomas Anderson, Arvind Krishnamurthy, and Arun Venkataramani, *Do incentives build robustness in bit torrent*, Proceedings of the 4th USENIX conference on Networked systems design & implementation (Cambridge, MA, USA), 2007.

[PIA$^+$07b] Michael Piatek, Tomas Isdal, Thomas E. Anderson, Arvind Krishnamurthy, and Arun Venkataramani, *Do incentives build robustness in bittorrent? (awarded best student paper)*, NSDI, USENIX, 2007.

[PL07] Bryan D. Payne and Wenke Lee, *Secure and flexible monitoring of virtual machines*, ACSAC, IEEE Computer Society, 2007, pp. 385–397.

[PP05] M. Pužar and T. Plagemann, *NEMAN: A network emulator for mobile ad-hoc networks*, Tech. Report 321, Department of Informatics, University of Oslo, 3 2005.

[PR08] Maurizio Pizzonia and Massimo Rimondini, *Netkit: easy emulation of complex networks on inexpensive hardware*, Proceedings of the 4th International Conference on Testbeds and research infrastructures for the development of networks & communities (ICST, Brussels, Belgium, Belgium), TridentCom '08, ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2008, pp. 7:1–7:10.

[PSC05] Joseph Polastre, Robert Szewczyk, and David E. Culler, *Telos: enabling ultra-low power wireless research*, IPSN, IEEE, 2005, pp. 364–369.

[Pui03] Ramon Puigjaner, *Performance modelling of computer networks*, Proceedings of the 2003 IFIP/ACM Latin America conference on Towards a Latin American agenda for network research (New York, NY, USA), LANC '03, ACM, 2003, pp. 106–123.

[RAL03] Robert Ricci, Chris Alfeld, and Jay Lepreau, *A solver for the network testbed mapping problem*, SIGCOMM Comput. Commun. Rev. **33** (2003), no. 2, 65–81.

[RDS$^+$07] Robert Ricci, Jonathon Duerig, Pramod Sanaga, Daniel Gebhardt, Mike Hibler, Kevin Atkinson, Junxing Zhang, Sneha Kasera, and Jay Lepreau, *The Flexlab approach to realistic evaluation of networked systems*, Proc. of the Fourth Symposium on Networked Systems Design and Implementation (NSDI 2007) (Cambridge, MA), April 2007.

[RG05] M. Rosenblum and T. Garfinkel, *Virtual machine monitors: current technology and future trends*, Computer **38** (2005), no. 5, 39 – 47.

[RH80] C.V. Ramamoorthy and G.S. Ho, *Performance evaluation of asynchronous concurrent systems using petri nets*, Software Engineering, IEEE Transactions on **SE-6** (1980), no. 5, 440 – 449.

[RHWG95] Mendel Rosenblum, Stephen A. Herrod, Emmett Witchel, and Anoop Gupta, *Complete computer system simulation: The SimOS approach*, IEEE Parallel Distrib. Technol. **3** (1995), 34–43.

[Ril03] George F. Riley, *The Georgia Tech Network Simulator*, Proceedings of the ACM SIGCOMM workshop on Models, methods and tools for reproducible network research (New York, NY, USA), MoMeTools ’03, ACM, 2003, pp. 5–12.

[Rit09] Marko Ritter, *External monitoring of protocol stacks*, Diploma Thesis,RWTH Aachen University, 09 2009.

[Riz97] Luigi Rizzo, *Dummynet: a simple approach to the evaluation of network protocols*, SIGCOMM Comput. Commun. Rev. **27** (1997), no. 1, 31–41.

[RSW03] Thomas Reicher, Christian Schwingenschlögl, and Wolfgang Wein, *Network emulation for application development in heterogeneous networks*, 8th International Workshop on Mobile Multimedia Communications (MoMuC 2003) (Munich, Germany), 2003.

[SAK07] M. Slee, A. Agarwal, and M. Kwiatkowski, *Thrift: Scalable cross-language services implementation*, Tech. report, Facebook Inc., 2007.

[SAZ$^+$02] Ion Stoica, Daniel Adkins, Shelley Zhuang, Scott Shenker, and Sonesh Surana, *Internet indirection infrastructure*, SIGCOMM Comput. Commun. Rev. **32** (2002), no. 4, 73–86.

[SB03] Osman Salem and Abdelmalek Benzekri, *Functional modeling and performance evaluation for two class diffserv router using stochastic process algebra*, 17th European Simulation Multiconference , Nottingham - UK, 09/06/2003-11/06/2003, June 2003, pp. 257–262.

[Sch08] Florian Schmidt, *Synchronization of Operating Systems in Heterogeneous Testing Environments*, Diploma Thesis,RWTH Aachen University, 06 2008.

[Sch12] Sebastian Schöppel, *Predicting the resource usage of web applications*, Diploma Thesis, RWTH Aachen University, 7 2012.

[SDRL09] Pramod Sanaga, Jonathon Duerig, Robert Ricci, and Jay Lepreau, *Modeling and emulation of internet paths*, Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2009, April 22-24, 2009, Boston, MA, USA, 2009, pp. 199–212.

[Sei07] Tim Seipold, *Improving emulation of wireless access networks in ns-2*, Proceedings of the Third International Conference on Wireless and Mobile Communications (Washington, DC, USA), ICWMC ’07, IEEE Computer Society, 2007, pp. 41–.

[Sei08] Tim Seipold, *Emulation of radio access networks to facilitate the development of distributed applications*, Journal of Communications **3** (2008), no. 1, 1.

[SFR04] W. Richard Stevens, Bill Fenner, and Andrew M. Rudoff, *UNIX network programming: The sockets networking API*, vol. 1, Prentice-Hall, 2004.

[SGB09] Thomas Staub, Reto Gantenbein, and Torsten Braun, *VirtualMesh: an emulation framework for wireless mesh networks in OMNeT++*, Proc. SIMUTools’09 (Brussels, Belgium), 2009, pp. 1–8.

[SGH$^+$10] Dennis Schwerdel, Daniel Günther, Robert Henjes, Bernd Reuther, and Paul Müller, *German-lab experimental facility*, Future Internet - FIS 2010 (Arne Berre, Asunción Gómez-Pérez, Kurt Tutschku, and Dieter Fensel, eds.), Lecture Notes in Computer Science, vol. 6369, Springer Berlin / Heidelberg, 2010, pp. 1–10.

[SGR$^+$11] Dominik Stingl, Christian Groß, Julius Rückert, Leonhard Nobach, Aleksandra Kovacevic, and Ralf Steinmetz, *Peerfactsim.kom: A simulation framework for peer-to-peer systems*, Proceedings of the 2011 International Conference on High Performance Computing & Simulation (HPCS 2011), IEEE, IEEE, Jul 2011, pp. 577–584.

[Sly07] Richard A. Slywczak, *Development of network-based communications architectures for future NASA missions*, Tech. report, NASA Glenn Research Center, 2007.

[SMRD06] Atul Singh, Petros Maniatis, Timothy Roscoe, and Peter Druschel, *Using queries for distributed monitoring and forensics*, Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006 (New York, NY, USA), EuroSys '06, ACM, 2006, pp. 389–402.

[SOHL$^+$96] M. Snir, S. W. Otto, S. Huss-Lederman, D. W. Walker, and J. Dongarra, *MPI: The complete reference*, MIT Press, Cambridge, Massachussetts, 1996.

[SPBP06] Neil Spring, Larry L. Peterson, Andy C. Bavier, and Vivek S. Pai, *Using PlanetLab for network research: myths, realities, and best practices*, Operating Systems Review **40** (2006), no. 1, 17–24.

[SPL$^+$12] Florin Sultan, Alex Poylisher, John Lee, Constantin Serban, C. Jason Chiang, Ritu Chadha, Keith Whittaker, Chris Scilla, and Syeed Ali, *Timesync: enabling scalable, high-fidelity hybrid network emulation*, Proceedings of the 15th ACM international conference on Modeling, analysis and simulation of wireless and mobile systems (New York, NY, USA), MSWiM '12, ACM, 2012, pp. 185–194.

[Sta01] William Stallings, *Operating systems - internals and design principles (4. ed.)*, Prentice Hall, 2001.

[SVB$^+$11] Oliver Stecklina, Frank Vater, Thomas Basmer, Erik Bergmann, and Hannes Menzel, *Hybrid simulation environment for rapid MSP430 system design test and validation using MSPsim and systemC*, Proceedings of the 14th IEEE International Symposium on Design and Diagnostics of Electronic Circuits & Systems, DDECS, IEEE, 2011, pp. 167–170.

[SW06] Arne Schmitz and Martin Wenig, *The effect of the radio wave propagation model in mobile ad hoc networks*, Proceedings of the 9th ACM international symposium on Modeling analysis and simulation of wireless and mobile systems (New York, NY, USA), MSWiM '06, ACM, 2006, pp. 61–67.

[SWK$^+$10] Stefan Schürmans, Elias Weingärtner, Torsten Kempf, Gerd Ascheid, Klaus Wehrle, and Rainer Leupers, *Towards network centric development of embedded systems*, Proceedings of the International Communications Conference (ICC) (Cape Town, South Africa), May 2010.

[Sys06] *IEEE Standard SystemC ® Language Reference Manual*, publicly available at `http://standards.ieee.org/about/get/` (accessed August, 2011), 2006, p. 1666–2005.

[TLP05] B.L. Titzer, D.K. Lee, and J. Palsberg, *Avrora: scalable sensor network simulation with precise timing*, Proceedings of the Fourth International Symposium on Information Processing in Sensor Networks (IPSN), april 2005, pp. 477 – 482.

[TRR08] Michael Tuexen, Irene Ruengeler, and Erwin P. Rathgeb, *Interface connecting the inet simulation framework with the real world*, Proceedings of the First International Workshop on OMNet++, 2008.

[Var99] A. Varga, *Using the OMNeT++ discrete event simulation system in education*, IEEE Transactions on Education **42** (1999), no. 4, 11 pp.

[Var01] Andras Varga, *The OMNeT++ discrete event simulation system*, Proceedings of the European Simulation Multiconference (ESM'2001) (Prague, Czech Republic), June 2001.

[VH08] András Varga and Rudolf Hornig, *An overview of the OMNeT++ simulation environment*, SimuTools (Sándor Molnár, John Heath, Olivier Dalle, and Gabriel A. Wainer, eds.), ICST, 2008, p. 60.

[VIF06] Aggelos Vlavianos, Marios Iliofotou, and Michalis Faloutsos, *BitoS: Enhancing bittorrent for supporting streaming applications*, INFOCOM, IEEE, 2006.

[vL10] Hendrik vom Lehn, *A Hybrid Evaluation Environment for Wireless Networks*, Diploma Thesis,RWTH Aachen University, 05 2010.

[VMW] VMWare Inc., *VMWare ESX Server: Platform for virtualizing servers, storage and networking (whitepaper)*, `http://www.vmware.com/pdf/esx_datasheet.pdf` (accessed 10/2012).

[vuz] *Vuze open source version*, `http://azureus.sourceforge.net/` (accessed 09/2012).

[VvRBM96] Diederik Verkest, Karl van Rompaey, Ivo Bolsens, and Hugo De Man, *Coware - A design environment for heterogeneous hardware/software systems*, Design Automation for Embedded Systems **1** (1996), no. 4, 357–386.

[VYW$^+$02] Amin Vahdat, Ken Yocum, Kevin Walsh, Priya Mahadevan, Dejan Kostic, Jeffrey S. Chase, and David Becker, *Scalability and accuracy in a large-scale network emulator*, ACM Symposium on Operating System Design and Implementation (OSDI-02), 2002.

[WASW05] Geoffrey Werner-Allen, Patrick Swieskowski, and Matt Welsh, *Motelab: a wireless sensor network testbed*, Proc. IPSN'05 (Piscataway, NJ, USA), IEEE Press, 2005, p. 68.

[Wat08] Jon Watson, *Virtualbox: bits and bytes masquerading as machines*, Linux J. **2008** (2008).

[WGLW12] Elias Weingärtner, René Glebke, Martin Lang, and Klaus Wehrle, *Building a modular bittorrent model for ns-3*, Proceedings of the 2012 workshop on ns-3 (WNS3 2012), 3 2012, Awarded with Best Paper Award and Best Student Paper Award.

[Wil01] C. Williamson, *Internet traffic measurement*, IEEE Internet Computing **5** (2001), no. 6, 70 –74.

[Wira] *Wireshark network protocol analyzer*, `http://www.wireshark.org` (accessed 02/2012).

[Wirb] *Wireless Tools for Linux*, `http://www.hpl.hp.com/personal/Jean_Tourrilhes/Linux/Tools.html#wext`, accessed May 10, 2010.

[WKC$^+$90] Thomas Williams, Colin Kelley, John Campbell, David Kotz, and Russell Lang, *GNUPLOT- an interactive plotting program*, 31 August 1990.

[WLS$^+$02] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar, *An integrated experimental environment for distributed systems and networks*, Proc. OSDI'02, 2002, pp. 255–270.

[WPR$^+$04] Klaus Wehrle, Frank Pählke, Hartmut Ritter, Daniel Müller, and Marc Bechler, *Linux networking architecture – design and implementation of networking protocols in the Linux kernel*, Prentice-Hall, 5 2004.

[WRSW10] Elias Weingärtner, Marko Ritter, Raimondas Sasnauskas, and Klaus Wehrle, *Flexible analysis of distributed protocol implementations using virtual time*, Proceedings of the 18th International Conference on Software, Telecommunications and Computer Networks (SoftCOM 2010) (Split, Croatia), 9 2010, pp. 225 – 229 (en).

[WSHW08] Elias Weingärtner, Florian Schmidt, Tobias Heer, and Klaus Wehrle, *Synchronized Network Emulation: Matching Prototypes with Complex Simulations*, SIGMETRICS Perform. Eval. Rev. **36** (2008), no. 2, 58–63.

[WSHW09] Elias Weingärtner, Florian Schmidt, Tobias Heer, and Klaus Wehrle, *Time accurate integration of software prototypes with event-based network simulations*, Proc. of the Poster session at SIGMETRICS 2009 (Seattle, USA), 2009.

[WSvL$^+$11] Elias Weingärtner, Florian Schmidt, Hendrik vom Lehn, Tobias Heer, and Klaus Wehrle, *Slicetime: A platform for scalable and accurate network emulation*, Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation (NSDI '11), USENIX, 3 2011.

[WvLW09] Elias Weingärtner, Hendrik vom Lehn, and Klaus Wehrle, *A performance comparison of recent network simulators*, In Proc. of IEEE International Conference on Communications (ICC 2009) (2009).

[WvLW11] Elias Weingärtner, Hendrik vom Lehn, and Klaus Wehrle, *Device-driver enabled wireless network emulation*, Proc. SIMUTools 2011 (Barcelona, Spain), March 2011.

[XUSC99] Z. Xiao, B. Unger, R. Simmonds, and J. Cleary, *Scheduling critical channels in conservative parallel discrete event simulation*, Proceedings of the 13th Workshop on Parallel and Distributed Simulation, 1999, pp. 20 –28.

[YED$^+$03] K. Yocum, E. Eade, J. Degesys, D. Becker, J. Chase, and A. Vahdat, *Toward scaling network emulation using topology partitioning*, Proceedings of the 11th IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer Telecommunications Systems (MASCOTS), oct. 2003, pp. 242 – 245.

[ZDJ$^+$10] Jun Zhu, Wei Dong, Zhefu Jiang, Xiaogang Shi, Zhen Xiao, and Xiaoming Li, *Improving the performance of hypervisor-based fault tolerance*, Proceedings of the 2010 IEEE International Symposium onParallel Distributed Processing (IPDPS), april 2010, pp. 1 –10.

[Zec03] Marko Zec, *Implementing a clonable network stack in the freeBSD kernel*, Proceedings of the USENIX Annual Technical Conference (USENIX ATC), USENIX, 2003, pp. 137–150.

[ZGW$^+$06] A. Zimmermann, M. Gunes, M. Wenig, J. Ritzerfeld, and U. Meis, *A hybrid testbed for wireless mesh networks*, Proceedings of the 1st Workshop on Operator-Assisted (Wireless Mesh) Community Networks, sept. 2006, pp. 1 –9.

[ZJTB04] Junlan Zhou, Zhengrong Ji, Mineo Takai, and Rajive Bagrodia, *MAYA: Integrating hybrid network modeling to the physical world*, ACM Transactions on Modeling and Computer Simulation **14** (2004), no. 2, 149–169.

[ZM04] Marko Zec and Miljenko Mikuc, *Operating system support for integrated network emulation in IMUNES*, Proceedings of the 1st Workshop on Operating System and Architectural Support for the on demand IT InfraStructure, 10 2004.

A

Appendix

A.1 SliceTime: Additional Timing Measurements

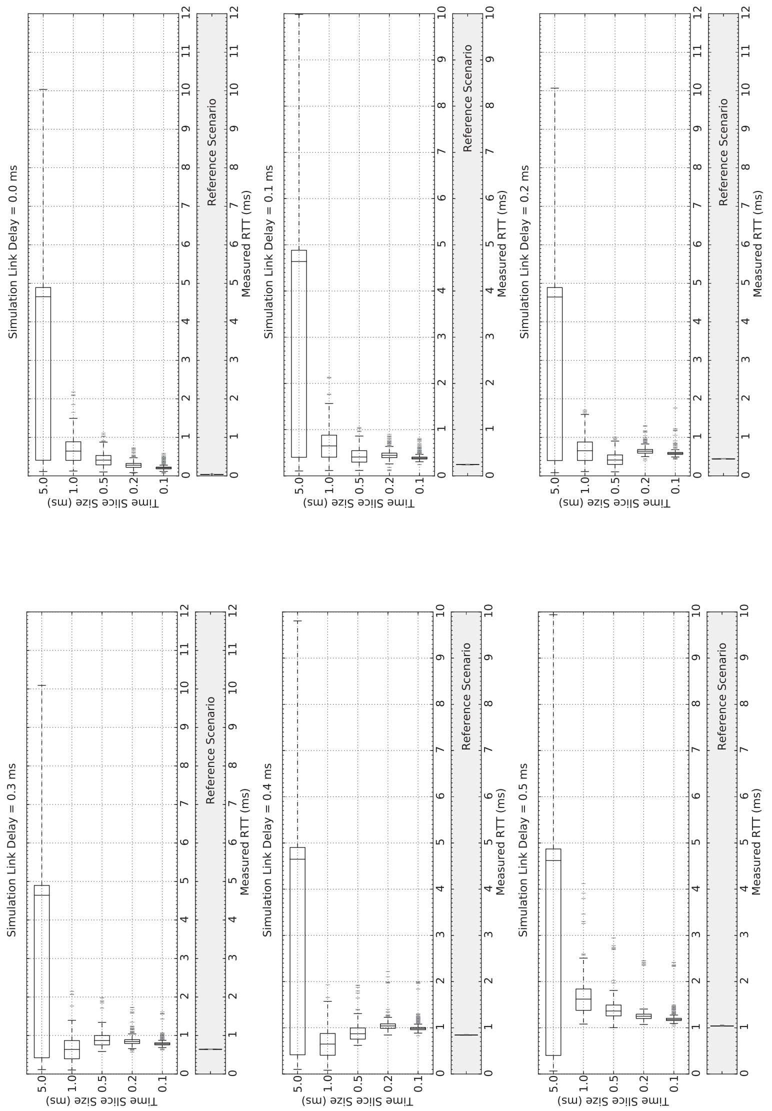


Figure A.1 Influence of time slice size on RTT distributions for different simulated link delays.

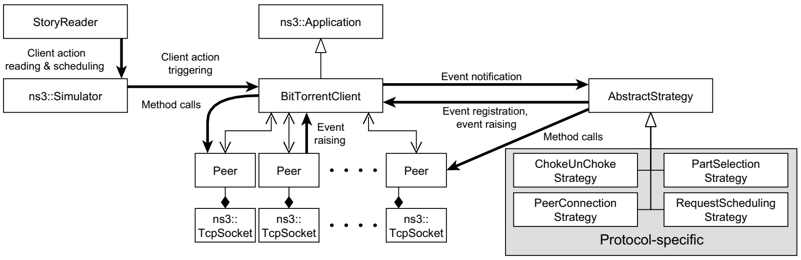


Figure A.2 Class collaboration in our BitTorrent client model: The `BitTorrentClient` class owns a set of `Peer` class instances of which each provides an interface to one remote peer via a TCP connection. A set of derivatives of the `AbstractStrategy` class forms the BitTorrent application's logic. The `Peer` class announces any Peer Wire protocol events to the strategies via an event notification mechanism provided by the `BitTorrentClient` class, whilst communication with remote peers can be initiated directly through method calls with the `Peer` class. For scripting scenarios the `StoryReader` interprets a DSL and schedules actions of selected clients.

A.2 BitTorrent Client Model Implementation

Figure A.2 gives an overview of the implementation structure and the component interaction of our BitTorrent client model. Its core is formed by the `BitTorrentClient` class, which is derived from the `Application` class of ns-3. A BitTorrent client in the simulation is represented by one instance of this class. It stores information about the current client state and acts as a message dispatcher among the different components of the client. First, there is the `Peer` class. This class represents an interface to a remote BitTorrent client. It implements the real-world network representation of the BitTorrent protocol and encapsulates a ns-3 TCP socket for communication with a remote client. All data sent out via the `Peer` class is binary compatible with the BitTorrent protocol specification. This allows hybrid evaluation in emulated networks with existing BitTorrent software. The control logic of a (BitTorrent-based) protocol is captured in several strategies. The specific strategies are implemented in classes which are derived from the class `AbstractStrategy`. A protocol implementation usually consists of several strategies each of them covering a dedicated aspect of the protocol logic. This approach allows us to easily adapt the simulation to the variety of possible strategies for choking, piece selection and neighbor discovery (tracker- and DHT-based).

A.2.1 Component Interaction

A `BitTorrentClient` object represents one instance of a BitTorrent client in the simulation. It stores the status information of the client, for example connections to remote clients, and information regarding the shared file and the swarm. Furthermore, the client object coordinates the communication between the network implementation of the protocol in the `Peer` objects and the protocol logic implementation in the strategies.

The structure of the program flow and component interaction was strongly influenced by the event-based paradigm of the ns-3 simulator. We introduced client-internal events for all network interactions of the BitTorrent protocol, such as reception of a choking message

```
GlobalValue::Bind ("ChecksumEnabled",
 BooleanValue (true));

WifiEmuBridgeHelper wbridge;
wbridge.SetAttribute ("ClientId", IntegerValue(42));
wbridge.Install (c.Get(0), staDevice.Get(0));
```

Listing A.1 Any ns-3 Wi-Fi simulation can be easily turned into an 802.11 emulation scenario using few lines of code.

or the completion of the upload of a requested piece. The events are generated by `Peer` objects and passed on to the `BitTorrentClient` object. The `BitTorrentClient` class implements an event dispatcher infrastructure based on ns-3 `Callbacks`. Strategies can register for certain events and are notified by the client class upon their occurrence. This allows strategies to monitor the subset of the current state of the client they need for operation without having to implement a polling policy for state changes.

The communication from the strategies to the client and the `Peer` objects is realized by direct method invocation. We intended the peer and client classes to provide the full set of communication subroutines needed and the protocol logic to fully reside within the strategy classes. Hence, we saw no benefit in using an event-based approach here. The less number of indirections increases the readability and speed of the code. Following the spirit of ns-3, all requests are handled internally in an asynchronous first-come-first-served fashion, while simulation time is guaranteed to be advanced only after all event listeners have been given time to react. Our model hence regards a client node's computational effort as zero for all operations.

A.3 Setup and Application of DDWNE

A.3.1 Setting up DDWNE for ns-3

In order to use an existing ns-3 Wi-Fi simulation scenario for network emulation only a few lines have to be added to the simulation program (cf. Listing A.1): As in any standard network emulation setup, we first instruct ns-3 to use its real-time scheduler to pin the execution of events to wall clock time. We also need to switch on the calculation of checksums for all packets to enable the communication with real-world hosts. In order to prepare the interaction with the device driver, one simply needs to instantiate a `WifiEmuBridge` and install it onto a simulated node (node 0 in this example), which forms the gateway node. Anything else in the simulation stays untouched, and of course, any feature or model of ns-3 may be used in conjunction with our 802.11 Wi-Fi extensions.

A.3.2 Setup of the DDWNE Driver

Figure A.3 illustrates how the Linux device driver provides a virtual 802.11 networking device serving as entry point to the simulated network. First, two `insmod` commands are used to load and to initialize the emulation Wi-Fi device driver. The only parameters needed to instantiate the driver are the remote location of the network simulation and the ID of the gateway node to which the driver is associated. If the device driver is running, the output of `iwconfig` shows that the network device and its properties resemble a wireless

```
root@wifi-test2:~/wifi-emu-kern# insmod ./wifi-emu.ko client_id=42
&& insmod ./wifi-emu-udp.ko peer_addr=192.168.1.2

root@wifi-test2:~/wifi-emu-kern# iwconfig wemu0
wemu0 IEEE 802.11b ESSID:"wifi-b" Nickname:"wifi-emu"
  Mode:Master Frequency:2.447 MHz
  Access Point: 00:00:00:00:00:02 Bit Rate:11 Mb/s
  Link Quality=45/100 Signal level=-56 dBm
  Noise level=-101 dBm
  Rx invalid nwid:0 Rx invalid crypt:0 Rx invalid frag:0
  Tx excessive retries:0 Invalid misc:0 Missed beacon:0

root@wifi-test2:~/wifi-emu-kern# iwconfig wemu0 mode Monitor
root@wifi-test2:~/wifi-emu-kern# ifconfig wemu0 up
```

Figure A.3 Terminal output showing how to load and configure the wireless emulation driver for use in monitor mode.

networking card. Finally we configure the interface to operate in the 802.11 monitor mode, which instructs the gateway node to relay any frame received on the MAC layer to the `wemu0` interface. The interface acts like a real 802.11 network card and supports the Linux wireless extensions.

A.4 List of Abbreviations

ACP	Algebra of Communicating Processes
AHB	Advanced High-Performance Bus
AODV	Ad-hoc On-demand Distance Vector
AP	Access Point
API	Application Programming Interface
APIC	Advanced Programmable Interrupt Controller
BSSID	Basic Service Set Identification
BT	BitTorrent
CMU	Carnegie Mellon University
CPU	Central Processing Unit
CRC	Cyclic Redundancy Check
CSP	Communicating Sequential Processes
CTW	Conservative Time Windows
CSMA/CD	Carrier Sense Multiple Access/Collision Detection
DCC	Distributed Coordinated Check-Pointing
DCE	Direct Code Execution
DDWNE	Device Driver-enabled Wireless Network Emulation
DHCP	Dynamic Host Configuration Protocol
DHT	Distributed Hash Table
DMA	Direct Memory Access
DNS	Domain Name System
DSL	Domain-specific Language
DSR	Dynamic Source Routing
DSDV	Destination-Sequenced Distance Vector routing
FSS	Full-System Simulator
GN	Gateway Node
GUI	Graphical User Interface
HPET	High Precision Event Timer
HTTP	Hypertext Transfer Protocol

HVM	Hardware-assisted Virtual Machine
ICMP	Internet Control Message Protocol
IDE	Integrated Development Environment
I/O	Input/Output
IP	Internet Protocol
IQR	Interquartile Range
IRC	Internet Relay Chat
ISDN	Integrated Services Digital Network
KVM	Kernel-based Virtual Machine
LEE	Link Emulation Engine
LF	Low Frequency
LTE	Long Term Evolution
MAC	Medium Access Control
MANET	Mobile Ad-Hoc Network
MPI	Message Passing Interface
MTTF	Mean Time to Failure
NED	Network Description Language
NETSIM	Discrete Event-based Network Simulation
OR	Overhead Ratio
OS	Operating System
PHY	Physical (layer)
PDES	Parallel Discrete Event-based Simulation
PDR	Packet Delivery Ratio
PEI	Packet Exchange Interface
PIT	Programmable Interval Timer
RTC	Real-time Clock
RTT	Round-Trip Time
RF	Radio Frequency
RISC	Reduced Instruction Set Computer
RPC	Remote Procedure Call
RSSI	Received Signal Strength Indicator
RWTH	Rheinisch-Westfälische Technische Hochschule
SCNSL	SystemC Network Simulation Library
SEDF	Simple Earliest Deadline First
SHE	Synchronized Hybrid Evaluation
SNE	Synchronized Network Emulation
SR	System Representation
SSID	Service Set Identifier
SYNC	Synchronization Component
TCI	Time Control Interface
TCP	Transmission Control Protocol
TDF	Time Dilation Factor
TSC	Time Stamp Counter
TVEE	Time Virtualized Emulation Environment
UMIC	Ultra High Speed Information and Communication
VCPU	Virtual Central Processing Unit
VLAN	Virtual Local Area Network
VNK	Virtualized Network Kernel
VM	Virtual Machine
VP	Virtual Platform
VPU	Virtual Processing Unit

Curriculum Vitae

Personal

Last Name:	Weingärtner
First Name:	Elias David
Date of Birth:	November 28th, 1979
Place of Birth:	Emmendingen, Baden-Württemberg, Germany
Nationality:	German

Education

High School 1991 - 2000	Goethe-Gymnasium, Emmendingen Abitur: July 2000
University 2001 - 2007	Ulm University, Computer Science (Medieninformatik) Degree: Dipl.-Inf.
Exchange Visit 2004 - 2005	University of Massachusetts Amherst Graduate School of Computer Science
PhD Student 2007 - 2013	RWTH Aachen University Chair of Communication and Distributed Systems Advisor: Prof. Dr.-Ing. Klaus Wehrle