Synchronized Network Emulation: Matching prototypes with complex simulations

Elias Weingärtner, Florian Schmidt, Tobias Heer, and Klaus Wehrle
Distributed Systems Group
RWTH Aachen University

{weingaertner, heer, wehrle}@cs.rwth-aachen.de, florian.schmidt@rwth-aachen.de

ABSTRACT

Network emulation, in which real systems interact with a network simulation, is a common evaluation method in computer networking research. Until now, the simulation in charge of representing the network has been required to be real-time capable, as otherwise a time drift between the simulation and the real network devices may occur and corrupt the results. In this paper, we present our work on synchronized network emulation. By adding a central synchronization entity and by virtualizing real systems for means of control, we can build-up network emulations which contain both unmodified x86 systems and network simulations of any complexity.

1. INTRODUCTION

According to the evaluation in the domain of computer networks, we either rely on network simulation or prototype implementations. Network simulations consist of virtual nodes and virtual channels over which messages are exchanged. Common simulators, such as ns-2 [16], JiST/SWANS [4] and OMNeT++ [18], are long-established in the research community. Their major strength is their flexibility: One can easily simulate a networking protocol and evaluate its performance in large networks with thousands of nodes. Simulators provide a global picture of the network and an unsurpassed degree of interactivity as all simulation parameters can be easily modified. On the other hand, network simulation can not entirely cope with all requirements needed for comprehensive evaluation: While the protocol semantics are usually well modeled, the implementation context (e.g., influence of an operating system and concurrent processes) is mostly disregarded or neglected. This is especially a problem, as modern protocols often show higher resource demands. For example, they may employ heavy use of cryptography or simply require many open connections to be maintained. Hence, in many cases it is difficult to deduct meaningful results about the real performance of a protocol and system related resource requirements from a network simulation alone.

For performance evaluation under realistic conditions, protocols are usually implemented as prototypes and investigated in a testbed of real networked systems. However, large-scale testbeds are normally very costly and often cumbersome to maintain. Public testbeds, most notably PlanetLab [8], facilitate the study at a global scale, but their lack of ability in granting exclusive access complicates the measurement of performance footprints. In addition, due to their shared nature, these testbeds lack flexibility because fundamentally changing the testbeds' nodes or their topology is not possible.

One option to bring together the flexibility of network simulators and the precision of real systems is network emulation. Real systems are connected to a simulation and interact with a simulated network. This concept was first introduced by Kevin Fall [9]. Nine years ago, Fall already stated that this approach is only useful if the simulation executes in real time, and that there was "no simple solution to this issue": If the simulation lags behind in time, it is unable to deliver packets in a timely manner, which leads to artifacts such as unpredictably varying or largely increased network latency. Such simulator overload may arise whenever complex network simulations are used or if large amounts of packets need to be processed by the simulator. Hence, simulator overload has to be prevented by all means because erroneous protocol behavior, such as connection time-outs, unwanted retransmissions, or the assumption of network congestion would be straight consequences, thus rendering measured results unusable or at least questionable. To our knowledge no solution exists which relieves the network simulation from being real-time capable. This is a major constraint which hinders the evaluation of an actual implementation against a simulated network of very high complexity.

In this paper, we propose an approach for network emulation that copes with a network simulation of any complexity without risking simulator overload. As described in Section 2, we tackle this problem with a concept called *synchronized network emulation*. We exchange the real systems with virtualized hosts (VHs) in order to be able to synchronize their execution behavior with the network simulation. Our VH implementation is based on Xen [3] and supports the incorporation of any unmodified x86-operating system running prototype implementations. As the virtualization takes place below the operating system level, the interaction of both user-land applications and system components such as network stacks with event-driven simulations of any complexity can be analyzed. We outline this implementation and our extensions of OMNeT++ in Section 4. Our evaluation in Section 5 shows that we achieve synchronous execution of the simulation and x86 machines with an accuracy up to 10μs and a reasonable amount of synchronization overhead. Section 6 concludes this paper with a brief discussion.

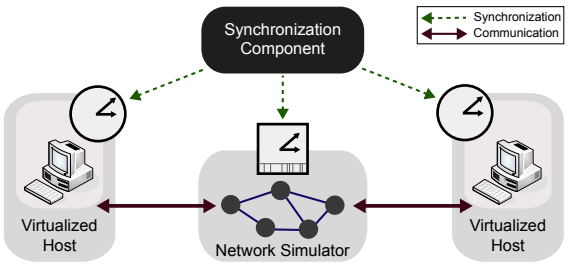


Figure 1: Synchronized Network Emulation

2. SYNCHRONIZED NETWORK EMULATION

As illustrated in Figure 1, a synchronized network emulation set-up contains three different kinds of components: The VHs and the network simulator are connected to a synchronization component, which aligns the local times of the VHs and the simulator. In this Section, we describe important aspects and properties of the individual building blocks.

2.1 Synchronization component

The synchronization component is the central coordinator which manages the synchronous execution of the simulators and the VHs attached. In order to prevent potential time drifts, the coordinator needs to implement a suitable synchronization algorithm. This is closely related to the synchronization problem in the area of parallel discrete event-based simulation [10] (PDES), where an event-based simulation is partitioned and processed simultaneously by multiple machines or processes. However, many of the synchronization algorithms proposed for PDES rely on the ability to predict the future execution behavior of entities within the simulation (look-ahead). Optimistic approaches, such as time warping, use check-pointing and occasional roll-backs in case a time drift over a certain threshold occurs. These concepts are often well suited for event-based simulations, but with the integration of VHs in mind, all concepts involving look-aheads and roll-backs cannot be applied because the prediction of the their future run-time behavior is impossible and the state-space of VHs is too large to allow fine-granular check-pointing. To avoid the dependency on potentially erroneous run-time prediction, we employ conservative time windows [13] for our purpose. The synchronization component assigns small time-slices to VHs and the simulator. The end of each time-slice constitutes a barrier. If the network simulation or a VH has reached this barrier, it stops executing and notifies the synchronization component. Only when all entities have cleared the barrier, the next time-slice is assigned. From this it follows that the synchronization accuracy is directly given by the the size of the time-slices because the drift is bounded by the size of one time-slice.

2.2 Virtualized Hosts

From a conceptual point of view, any real system supported by a suitable virtualization environment may be integrated into a synchronized network emulation set-up. In the following, we restrict ourselves to wired networked computers which are interconnected using Ethernet. Moreover, we consider a real system to be an ordinary x86-based machine running an operating system such as Linux, Open Solaris or Windows.

The virtualization of the real systems is needed to obtain full control over their run-time behavior: First of all, the execution of a VH needs to be stalled until the synchronization component allocates the next time-slice. Another issue is what we refer to as *timekeeping*: As the systems execution is interrupted due to the synchronization process, we need to avoid that the VHs perceive these gaps in time. Hence, we need to manipulate their internal clocks to still provide the VH with a consistent and continuous time.

With the goal of incorporating any unmodified operating system into synchronized network emulation, we need a flexible virtualization environment which provides the desired level of control. Many System emulators and full system simulators (Virtutech Simics, CoWare Virtual Platform) already provide the needed interfaces. However, complete system emulation at the instruction level naturally bears a lot of overhead, and the accuracy delivered is usually not needed, neither for debugging nor for the analysis of network implementations. While we leave them aside for now, we expect them to be easily adoptable for synchronized network emulation.

In this paper, we investigate the utilization of virtual machine monitors, precisely the Xen hypervisor [3], for this purpose. Xen facilitates the parallel execution of multiple operating systems on the same physical machine, and it is implemented as a thin layer between the system hardware and the operating systems. Hence, it is able to control the running behavior of any OS on top of it. As the operating system's code is in fact executed natively, Xen's footprint on performance is rather small. Since its original publication and release, Xen itself has undergone many changes and improvements [19]. Most importantly, Xen nowadays supports the virtualization of arbitrary, unmodified x86 operating systems by means of hardware virtualization (HVM), as supported by modern CPUs (AMD-V, and Intel-VT, respectively). Further details are presented in Section 4.

2.3 Network Simulator

The network simulator's task is to model the network which the VHs are connected to. Following the terminology in [9], we distinguish between opaque and protocol network emulation mode. In opaque network emulation, traffic is simply passed through the simulator. In this case, the simulator merely influences the propagation of network packets, for example by delaying packets or by simply dropping frames. This approach is prevalent in many available toolkits [1, 2, 7, 17]. We focus on the protocol mode where the network simulation implements the communication protocols that are used by the VHs. This is required for the implementation of simulated hosts which communicate with the VHs attached.

For coupling the simulator and the VHs, we implemented time synchronization in the event scheduler of the simulator: 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.

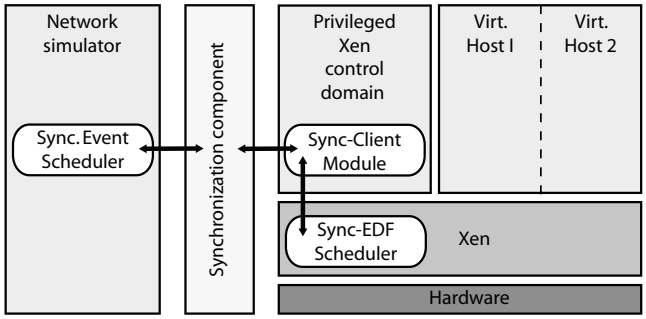


Figure 2: System Architecture

3. RELATED WORK

Many available tools [1, 2, 7, 11, 17] share the term *network emulation* with our work, however, they only provide opaque network emulation. While our work mainly targets simulations in protocol mode, we of course also support an opaque setting, since it is merely a simpler subset of the former. Considering protocol-aware network emulation, ns-2 provides adequate extensions [9, 15]. The commercial OPNet modeler implements similar features [5]. However, none of these tools copes with simulation overhead. The idea of using Xen for network emulation has been addressed in [12], where it is used to simulate the effects of high-speed networks which are faster than any available technology. In addition, Gupta, Vishnawath and Vahdat recently proposed the use of virtualization to reduce the amount of physical machines needed in a testbed [11].

One way to measure the performance footprint is the use of full system simulators like Mambo [6] or Simics [14]. They provide highly sophisticated models for entire networked systems, thus allowing the execution of real implementations within a simulated context. While this approach offers a great level of possible accuracy, the instruction-level based simulation introduces a high simulation overhead which makes it hard to employ for the simulation of very large networks. As an example, Windows XP needs about 40 seconds of boot-up time within our Xen-based VH implementation. However, Simics completes the same task in 335 seconds on average if executed on our evaluation machine.

4. IMPLEMENTATION

In order to investigate the concept of virtualization supported synchronized network emulation, we have implemented a toolkit consisting of three building blocks which are jointly depicted in Figure 2: The network simulator, the synchronization component, and the virtual host infrastructure. While the network simulator and the synchronization component are realized as individual entities, multiple virtual hosts can be run on top of one host machine that executes our Xen-based VH core. In this Section, we describe important issues regarding the implementation of the system's building blocks.

4.1 Synchronization component

The synchronization component applies the conservative time window algorithm in a straightforward manner, synchronizing the components with a packet-based blocking technique. The challenging part of the synchronization of the distinct components is that with increasing precision, and therefore, smaller time-slice size a greater amount of synchronization messages between the simulator and the VHs is needed. Hence, we emphasize the requirement of keeping the messaging complexity as low as possible in order to limit the performance impact caused by message processing. To this end, we implemented a lightweight protocol based on UDP broadcasts for synchronizing the simulator and the VHs.

4.2 Xen-based Virtualized Hosts

The Xen-based VH core in our implementation consists of two major parts. On the one hand, we developed a Linux kernel module that communicates with the synchronization component. On the other hand, we modified the Xen kernel in several ways to achieve two basic goals: To make it possible to start and stop virtualized operating systems (called domains in the Xen context) and let them run for exact amounts of time. Additionally, for presenting a consistent gapless time flow without interruptions to the VHs, we modified the Xen timekeeping architecture that manages the variables that a VH uses to determine the current system time.

Our implementation both works with paravirtualized and unmodified operating systems, enabling the use of closed as well as open source operating systems for the x86 architecture. We successfully tested a Debian-based Linux distribution and a Windows XP operating system as VH. This flexibility gives protocol developers the freedom to test their protocols in the OS environment of their choice. In the following, we present three aspects of our Xen-based implementation: The signaling between the synchronization component, the scheduling component, and the timekeeping component.

4.2.1 Signaling

The sync-client module is remotely controlled by the synchronization component. It runs in the Xen control domain (see Figure 2), where it communicates with the Xen scheduler. This architecture integrates into the concept of a privileged Xen domain from which all administrative tasks concerning the virtualization are performed. For performance reasons, we chose to implement the sync-client module as a kernel module.

Whenever the sync-client module receives a sync message from the synchronization server, issuing a new time-slice, the module instructs the scheduler to let the synchronized domain run for the specified time. Once the slice is over, the sync-EDF scheduler stops the VH and notifies the synchronization component.

The actual network traffic between the VH and the simulator is tunneled via UDP. A tap device is bridged with the domain's virtual network interface, and the Ethernet frames acquired this way are packed into UDP packets and sent directly to the simulation.

4.2.2 Scheduling

For controlling the time flow for the VHs, we extend the "simple earliest deadline first" (sEDF) scheduler bundled with Xen. The modified scheduler keeps track of the synchronized VH and only allows it to run if it was issued a slice from the sync-client module. If multiple VHs are synchronized, they are scheduled in a strongly sequential manner in order to prevent inaccuracies due to interweaved execution.

Xen was not built for exact time synchronization, and hence, handles the assigned run-time inaccurately. Xen considers the scheduling itself to take no time at all. As an effect, it attributes the time spent in the scheduler to the time-slice of the domain that is chosen for execution, which leads to domains actually running for shorter times than expected. This behavior produces only negligible inaccuracies under normal circumstances, since the time spent in the the scheduler is disproportionally small. However, this is not the case anymore if the execution time of a domain is fragmented into tiny time slices for the means of accurate synchronization: If a typical slice might be 100μs or smaller, the time spent in the scheduler amounts to a sizable portion of the slice. Therefore, the scheduler has to be made aware of its own use of time, so that it can properly schedule the domains for the desired time.

Moreover, VHs are not perfectly stable in their consumption of time. For example, if a domain is scheduled to run for 100μs, it is common to see it return to the scheduler after 99.5μs, or 100.5μs. To make sure that these errors do not accumulate, we modified the scheduler to be self-correcting. That way, the next slices will be set accordingly longer or shorter to keep the average slice length close to the desired value.

4.2.3 Timekeeping

Every operating system has several ways to measure the passing of time that, in the non-virtualized case, are based on hardware counters like a CPU cycle counter, or a programmable interrupt that is periodically executed and signals the passing of a predefined amount of time. In the virtualized case, Xen provides the domains with virtual counterparts to these hardware timing devices. We have to discriminate between the two cases of a counter variable that increases as time passes, and timer interrupts that execute a function on expiration.

In the case of counters, we have to keep track of the difference Δt between the VHs and the Xen counters. To provide the VH with a consistent time, we subtract Δt from the actual counter value whenever the VH accesses it. In the case of periodic timers that on expiration send a "tick" signal to the domain, we keep track of a δt that we can add to the expiration date to ensure that the timer fires at the right point in (synchronized) time. Timers are stopped when the domain is de-scheduled and restarted with δt added to the expiration date when the domain is started again.

4.3 Network Simulator Integration

As network simulator, we use the modular discrete event simulator OMNeT++ and its INET framework. The INET framework is a comprehensive collection of protocol models that span wired and wireless network protocols (e.g. the parts of the IEEE 802 family including IEEE 802.11 wireless networks, IP, TCP and UDP). With the general integration of OMNeT++ simulations in mind, we implemented a custom event scheduler which is controlled by the synchronization component. The scheduler controls the advancement in the event queue and signals passed time-slice borders to the synchronization component.

As many other simulators, OMNeT++ uses its own data structures for packets and packet headers which are not aligned with the network standards used in real networks. Moreover, depending on the level of abstraction, network simulators model protocol functionality and payload to meet their needs, disregarding interoperability with real hosts. Thus, when coupling VHs based on commodity operating systems, a conversion between the formats of the network simulator and the network packets used in real networks becomes necessary. Moreover, missing protocol mechanisms need to be implemented. We extended OMNeTs protocol conversion capabilities to serialize its internal message types to Ethernet frames. These Ethernet frames are tunneled to the VH using UDP. In order to facilitate the communication between a simulated node and VH, a TAP device is used to capture its network packets. The captured network packets are tunneled to the network simulation where a parser converts the received Ethernet frames to OMNeT++ messages. So far, we fully support ARP, ICMP, IP and UDP as transport protocol.

5. EVALUATION

In this Section, we provide preliminary evaluation results regarding the precision of the time synchronization and its impact on application-level measurements. An evaluation of the overhead introduced by the synchronization mechanisms concludes this Section. We used an AMD Athlon 64 PC running at 2.4 GHz as base for our VH running our Xen-based implementation. As Xen host system as well as for the VHs in para-virtualized mode, we used Linux 2.6.18.8-xen. The HVM-based VHs ran Linux with the kernel version 2.6.22-14 and Windows XP.

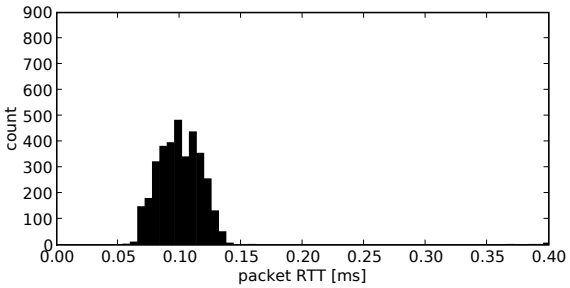
5.1 Timekeeping precision

The main goal of our approach is to provide a consistent and synchronized time to all coupled components. For evaluating the precision of the synchronization component, we ran the coupled simulation for 10.000 seconds with a synchronization granularity of 100μs per time-slice. As expected, even after this prolonged period, the cumulative drift between the synchronized components is bounded by the size of a time-slice (i.e. 100μs). Consequently, no cumulative drift was measurable, resulting in precisely synchronized systems.

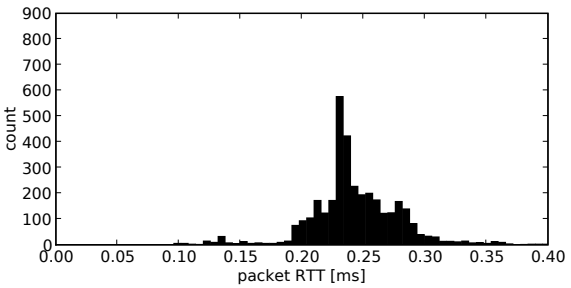
5.2 Application-level measurements

In this Section, we examine how synchronization and lack of synchronization affect the measurements taken in an exemplary test network. In the test setup, a network simulation contains only one simulated host, and is connected to one VH. We artificially slowed down the execution speed of the simulation for generating overload conditions. We first measured the impact of the synchronization on the network delay perceived by the VHs. For all delay measurements we measured 3500 ICMP Echo Request round-trip times (RTTs), more often referred to as "pings", between the VH and the network simulation. In the synchronized case, the duration of each time-slice was set to 100μs. As a basis for comparison, we depict measurements between two directly connected Linux hosts in Figure 3(a).

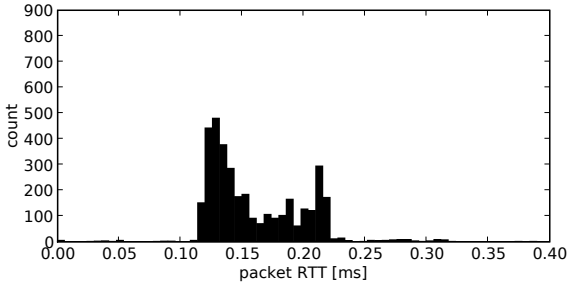
Figure 3(d) depicts a histogram of the RTT without synchronization. In this case, the simulation overhead leads to unusable results because the perceived RTTs exhibit an unrealistic delay and distribution due to the slow processing of the simulator. If we synchronize the VH and the network simulation, we obtain RTTs in the right order of magnitude, as depicted in Figure 3(c). In the case of paravirtualization, it is noteworthy that the bulk of the round-trip times fall



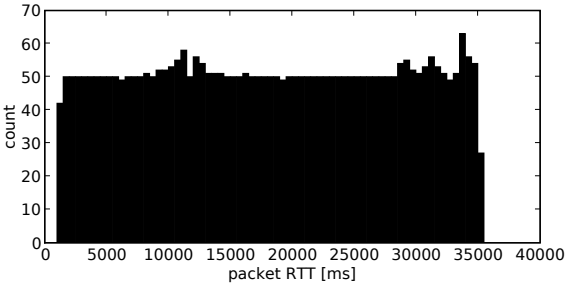
(a) RTTs for two Linux PCs



(b) RTTs between a hardware virtualized Linux VH and a simulation



(c) RTTs between a paravirtualized Linux VH and a simulation



(d) RTTs corrupted by unsynchronized and overloaded simulation

Figure 3: RTT histograms of 3500 echo replies.

into an interval between 120μs and 220μs. While this is on average slightly higher than in the comparison case shown in 3(a), it corresponds closely to the chosen time slice of 100μs. A small amount of RTTs takes longer, up to one more time slice, i.e. until 320μs. Interestingly, a few RTTs are lower than the lowest ones in the comparison case. This can be explained by the fact that on occasion, an echo request will be sent out just before the barrier time is reached. The packet then travels on the wire while the VH is waiting for the next time slice assignment, and will appear to the VH to have traveled instantly.

Figure 3(b) shows the RTTs for a hardware virtualized, synchronized Linux VH. While the graph is similar to Figure 3(c), the measured round trip times are generally higher. However, hardware virtualized VHs in Xen always suffer from some performance degradation. In the HVM mode, all I/O operations are fully emulated. Hence, additional overhead is introduced compared to paravirtualized VHs which directly access I/O data using shared memory pages. The slightly inferior performance is therefore to be expected. Nonetheless, the results are still close to the ones in set-up in Figure 3(a), and in any case much more meaningful than Figure 3(d). From these results we conclude that synchronized network emulation is able to produce realistic results related to the timing behavior of protocols, even if the simulation suffers from high overload.

5.3 Synchronization Overhead

In order to quantify the overhead introduced by the synchronization at the VH, we assigned 600 seconds of running time using our synchronization component. We ran this experiment using time-slices of different sizes and calculated the ratio between assigned and actually consumed time. Figure 4 shows these ratios for time-slices at different sizes, both

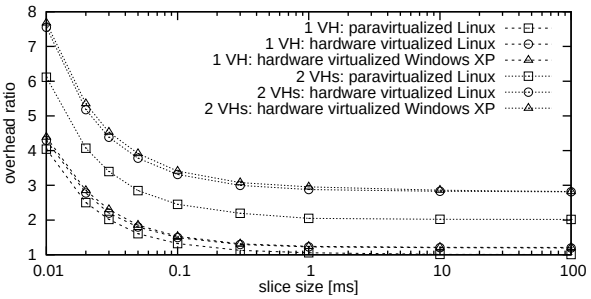


Figure 4: Synchronization overhead vs. accuracy

for VHs using para-virtualization and HVM mode. Recall that the synchronization accuracy corresponds to the size of a time-slice. Most notably, the synchronization overhead remains stable for time-slices greater than $0.1ms$.

Even at an accuracy level of 10μs, the overhead ratio for one VH instance remains below 5 for all types. Hence, the additional overhead introduced is below the factor of 4 for a time-slice of 10μs in this case. Furthermore we emphasize that the ratios of the para-virtualized and the HVM-based VH instances differ slightly, and that the choice of operating system has practically no influence. If the Xen implementation is used to execute two VHs at the same time, the overhead increases in a proportional manner. As HVM-based VHs are able to put any unmodified x86 operating system into execution, these performance measures clearly demonstrate the feasibility of such an incorporation even at higher levels of accuracy. All in all, the results suggest that our Xen-based VHs will not become the performance bottleneck in synchronized network emulation set-ups if the simulation is reasonably complex.

6. CONCLUSION

In this paper, we have introduced the concept of efficient *synchronized network emulation* by means of virtualization. Using virtualized hosts and a central synchronization component, we are able to cope with network simulations of arbitrary complexity.

Even when synchronizing one VH at an accuracy of 10μs, the additional overhead introduced remains below a factor of 4 compared to a non-synchronized emulation scenario, and drops to about 24% for slices of 1ms. Hence, coupling complex simulation with VHs facilitates detailed investigations of protocols in their genuine context (real operating systems, practically deployed protocol stacks etc.) with a comparatively low overhead, reducing hardware requirements and increasing run-time performance. With time-slices sufficient for simulating end-to-end Internet traffic, even applications that require user-interaction can be subject to synchronized network emulation, enabling the evaluation of a protocol's impact on the perceived system performance.

In conclusion, synchronizing network emulation is a feasible approach, and it opens up new ways of network analysis. On the one hand, even closed-source binaries (e.g., operating systems, proprietary protocols, and applications) can be analyzed with the versatility of a simulator without being limited to real-time capable simulations. On the other hand, synchronized networks can be evaluated based on accurate end-host behavior without using static traces, statistical traffic patterns, or simplified models. We therefore anticipate synchronized network emulation becoming an important tool in network analysis.

Acknowledgments

We thank our anonymous reviewers whose comments helped to improve this paper. We also express our gratitude to Simon Rieche, Olaf Landsiedel, Stefan Götz and Hamad Alizai for supportive comments and discussions.

7. REFERENCES

[1] M. Allman and S. Ostermann. ONE: The Ohio Network Emulator. Technical Report TR-19972, Ohio University, Aug. 1997.

[2] M. Avvenuti and A. Vecchio. Application-level network emulation: the emusocket toolkit. *Journal of Network and Computer Applications*, 29(4):343–360, 2006.

[3] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP'03)*, pages 164–177, Bolton Landing, NY, USA, Oct. 2003. ACM.

[4] R. Barr, Z. J. Haas, and R. van Renesse. JiST: an efficient approach to simulation using virtual machines. *Softw, Pract. Exper*, 35(6):539–576, 2005.

[5] E. Biegeleisen, M. Eason, C. Michelson, and R. Reddy. Network in the loop using HLA, distributed OPNET simulations, and 3D visualizations. In *Military Communications Conference, 2005. MILCOM 2005. IEEE*, volume 3, pages 1667–1671, Oct. 2005.

[6] P. Bohrer, M. Elnozahy, A. Geith, C. Lefurgy, T. Nakra, J. Peterson, R. Rajamony, R. Rockhold, H. Shafi, R. Simpson, E. Speight, K. Sudeep, E. van Hensbergen, and L. Zhang. Mambo: a full system simulator for the PowerPC architecture. *ACM SIGMETRICS Performance Evaluation Review*, 31(4):8–12, 2004.

[7] M. Carson and D. Santay. NIST Net: a linux-based network emulation tool. *ACM SIGCOMM Computer Communication Review*, 33(3):111–126, 2003.

[8] B. Chun, D. Culler, T. Roscoe, A. Bavier, L. Peterson, M. Wawrzoniak, and M. Bowman. PlanetLab: An Overlay Testbed for Broad-Coverage Services. *ACM SIGCOMM Computer Communication Review*, 33(3):3–12, 2003.

[9] K. R. Fall. Network emulation in the Vint/NS simulator. In *Proceedings of the 4th IEEE Symposium on Computers and Communication*, pages 244–250. IEEE Computer Society, 1999.

[10] R. M. Fujimoto. Parallel discrete event simulation. *Communcations of the ACM*, 33(10):30–53, 1990.

[11] D. Gupta, K. V. Vishwanath, and A. Vahdat. DieCast: Testing distributed systems with an accurate scale model. In *Proceedings of the 5th USENIX Symposium on Networked System Design and Implementation (NSDI'08)*, San Francisco, CA, USA, 2008. USENIX Association.

[12] D. Gupta, K. Yocum, M. McNett, A. C. Snoeren, A. Vahdat, and G. M. Voelker. To infinity and beyond: time-warped network emulation. In *Proceedings of the 3rd USENIX Symposium on Networked Systems Design and Implementation (NSDI'06)*, pages 87–100, Berkeley, CA, USA, 2006. USENIX Association.

[13] B. D. Lubachevsky. Efficient distributed event-driven simulations of multiple-loop networks. *Communications of the ACM*, 32(1):111–123,131, 1989.

[14] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hållberg, J. Högberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A full system simulation platform. *IEEE Computer*, 35(2):50–58, 2002.

[15] D. Mahrenholz and S. Ivanov. Real-time network emulation with ns-2. In *Proceedings of the 8th IEEE International Symposium on Distributed Simulation and Real-Time Applications (DS-RT)*, pages 29–36. IEEE Computer Society, 2004.

[16] The network simulator ns-2. `http://www.isi.edu/nsnam/ns/`.

[17] L. Rizzo. Dummynet: A simple approach to the evaluation of network protocols. *ACM Computer Communication Review*, 27(1):31–41, 1997.

[18] A. Varga and R. Hornig. An overview of the OMNeT++ simulation environment. In *Proceedings of the First International Conference on Simulation Tools and Techniques for Communications, Networks and Systems (SIMUTools 2008')*, Marseille, France, March 2008.

[19] Xen hypervisor project. `http://www.xen.org/`.