SliceTime: A platform for scalable and accurate network emulation

Elias Weingärtner, Florian Schmidt, Hendrik vom Lehn, Tobias Heer and Klaus Wehrle
Communication and Distributed Systems
RWTH Aachen University, Germany

Abstract

Network emulation brings together the strengths of network simulation (scalability, modeling flexibility) and real-world software prototypes (realistic analysis). Unfortunately network emulation fails if the simulation is not real-time capable, e.g., due to large scenarios or complex models. So far, this problem has generally been addressed by providing massive computing power to the simulation, which is often too costly or even infeasible.

In this paper we present *SliceTime*, our platform for scalable and accurate network emulation. It enables the precise evaluation of arbitrary networking software with event-based simulations of any complexity by relieving the network simulation from its real-time constraint. We achieve this goal by transparently matching the execution speed of virtual machines hosting the software prototypes with the network simulation. We demonstrate the applicability of *SliceTime* in a large-scale WAN scenario with 15 000 simulated nodes and show how our framework eases the analysis of software for 802.11 networks.

1 Introduction

We are still in need of adequate tools for performance investigations as well as for testing of real-world network protocol implementations and large-scale distributed systems. In this regard, the first major requirement is scalability. For example, in order to facilitate the analysis of contemporary P2P applications, such a tool needs to scale up to potentially thousands of nodes. Second, we need experimentation platforms that isolate the protocol implementation and its communication from real-world communication networks. Such strong isolation is important for the investigation of malware to prevent a potential outbreak. Isolated evaluation environments are also well suited for the analysis of software for wireless networks as unwanted disturbances on the wireless channel can be avoided.

Discrete event-based network simulation is a well-established methodology for the evaluation of network protocols. Network simulators, such as ns-3 [27] or OMNeT++ [37], facilitate the flexible analysis of arbitrary network protocols. Due to their abstract modeling approach, network simulations scale well to network sizes of up to many thousand nodes.

However, abstract simulation models focus only on the most relevant aspects of the communicating nodes. They disregard the system context of a network protocol and its run-time environment, like the influence of an operating system regarding timing, concurrent processes, and resource constraints. This fundamental concept of abstraction limits the applicability of network simulations to network performance metrics. For instance, investigations of run-time performance, resource usage, and the interoperability with other protocol implementations are difficult to obtain by solely using simulations. The strict event-based notion of network simulators also makes it generally impossible to execute arbitrary networking applications inside the simulation environment. These issues complicate performance studies that are very important for the applicability of communication systems.

Performance evaluations under real conditions are mostly carried out within network testbeds of prototype implementations. However, setting up large-scale testbeds is expensive and their maintenance is often cumbersome. Shared testbeds such PlanetLab [7], Emulab [42] and MoteLab [41] partially fill this gap. Yet their flexibility is limited due to a lack of topology controllability, shared testbed usage or insufficient scalability.

Network emulation as introduced by Fall [10] brings together the flexibility of discrete event-based of network simulation with the precision of evaluation using real-world testbeds. An event-based simulation modeling a computer network of choice is connected to real-world software prototype. Traffic from the prototype is fed to the simulation and vice versa. This way, the software prototype can be evaluated in any network that can be mod-

eled by the simulator. One fundamental issue of network emulation are the different time representations of event-based simulations and software prototypes. Event-based simulations consist 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 prototypes observe a continuously progressing wall-clock time.

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 timely. Such *simulator overload* may result from complex network simulations, for example due to a high number of simulated nodes or models of high computational complexity. Simulator overload has to be prevented because deficient protocol behavior such as connection time-outs, unwanted retransmissions, or the assumption of network congestion would be the direct consequence. Moreover, even slight simulator overload may invalidate performance evaluations because the network cannot be simulated within the required timing bounds.

Speeding up the simulation to make it real-time capable is the first obvious option to deal with simulation overload. This speed-up can be achieved by supplying the simulation machine with sufficient computational resources in forms of hardware or by parallelizing the network simulation. 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 research institutes and individuals.

So far, network emulation has merely been an arms race between the complexity of the simulation model and the computational power of the simulation hardware. Hence, traditional approaches result in *variable hardware requirements* and *fixed execution time* (real time). By contrast, we 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). More specifically, the main contributions of this paper are the following:

1. We thoroughly elaborate the design of *SliceTime* and its underlying concept of synchronized network emulation [39, 40] (Section 2). It eliminates the need of the network simulation to execute in real-time. This enables network emulation scenarios using simulations of any complexity. We achieve this goal by synchronizing the software prototypes with the network simulation. Using virtualization, we decouple the software prototypes' perceived progression of time from wall-clock time.
2. Our implementation of *SliceTime* (cf. Section 3) for x86 systems enables the synchronized execution of Xen-based [3] virtualized prototypes and ns-3 simulations with an accuracy down to 0.01 ms.
3. We show that *SliceTime* delivers a high degree of accuracy and transparency, both regarding timing and perceived network bandwidth (Section4). We further demonstrate in our evaluation of *SliceTime* that is run-time efficient and that the synchronization overhead stays below 10% at an accuracy of 0.5 ms.
4. We illustrate how *SliceTime* simplifies testing and performance evaluations of WiFi software for Linux by remodeling a large-scale AODV field test entirely in software. We further demonstrate the scalability of *SliceTime* by applying it to a large-scale wide-area network (WAN) scenario with 15 000 nodes (Section 5).

In Section 6 we discuss the related work before concluding this paper in Section 7.

2 *SliceTime*

We now present the design of *SliceTime*. A *SliceTime* setup incorporates three main components (cf. Figure 1): The central synchronization component (*synchronizer*), at least one virtual machine (VM) carrying a software prototype of choice, and an event-based network simulation. The synchronizer controls the execution of the network simulation and the software prototypes. In order to carry out such a synchronization, the synchronizer must interrupt the execution of the prototype or the simulation at times to achieve precise clock alignment. To enable this suspension, the software prototypes are hosted inside virtual machines for means of control.

2.1 Synchronization Component

The synchronization component centrally coordinates a *SliceTime* setup. Its task is to manage the synchronous execution of the network simulation and the attached virtual machines. It implements a synchronization algorithm to prevent potential time drifts and clock misalignments between the virtual machines and the network simulation. As choice for the synchronization algorithm, we consider solutions known from the research domain of parallel discrete event-based simulation (PDES) [11]. In this regard, two classes of synchronization are distinguished, optimistic synchronization schemes and conservative synchronization schemes.

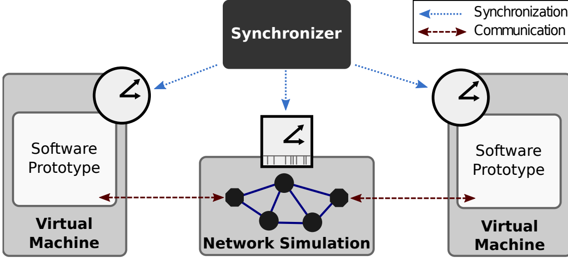


Figure 1: Conceptual Overview of *SliceTime*: By relying on entirely virtualized prototypes, we are able to synchronize the execution speed of the simulation and the prototypes. The simulation is relieved from its real-time constraint, enabling large-scale network emulation scenarios on off-the-shelf hardware.

Optimistic schemes, most notably Time Warp [18], execute the parallel simulation in a speculative fashion. In case of synchronization errors, roll-backs are used to restore a consistent and error-free global state. For the ability to roll back to a consistent state, optimistic schemes often incorporate regular snapshots of the synchronized peers. As the state of a virtual machine includes the memory allocated for the running operating system instance, check-pointing is costly at the desired level of synchronization granularity. Conservative synchronization schemes, by contrast, guarantee a parallel execution without synchronization errors, and hence, do not require a roll-back mechanism. However, most conservative schemes, such as the null-message algorithm by Chandy and Misra [6], require knowledge about the future behavior (look-ahead) of a system. While the look-ahead in event-based simulations can be determined by inspecting their event queue, predicting the future run-time behavior of a virtual machine is generally not possible. In effect, this limits the choice of a synchronization algorithm for *SliceTime* to a scheme which neither makes assumptions about the future behavior nor requires regular snapshots to be taken.

SliceTime uses a scheme similar to conservative time windows (CTW) [23] for synchronizing network simulations and VMs. In the following, we refer to this algorithm as *barrier synchronization*. Figure 2 shows the synchronization of two components, one VM and one network simulation, via the barrier synchronization algorithm. It allows every synchronized peer to run for a certain amount of time, the so-called *time slice*, after which it blocks until all other peers reach the barrier. At this point, the barrier is lifted, and a new future barrier is set up to which the execution of the synchronized components continues again. As the execution of both the network simulation and the virtual machine is always bounded by a barrier, the time drift between them 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.

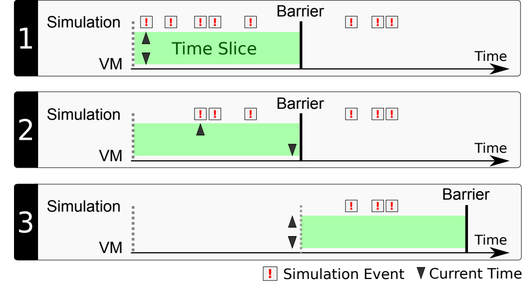


Figure 2: Different steps of the barrier algorithm used for the synchronization of one VM and one event-based simulation. The execution of the simulation and the VM is blocked until both have finished the time slice.

2.2 Virtual Machines

The virtual machines encapsulate the software prototype to be integrated with the network simulation. 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.

Therefore, the execution of the prototype can be suspended until all synchronized components have reached the end of the time slice. This suspension avoids simulator overload by allowing the network simulator to run while the virtual machines are waiting. However, this suspension is typically detectable by the VMs, because they are relayed information from hardware time sources. Under normal circumstances, this behavior is desired to keep the clock synchronized to wall-clock time and to make sure that timers expire at the right point of time. However, since we suspend the VMs in order to synchronize their time against each other and the simulation, we must avoid this behavior. Having full control over the VM's perception of time we instead provide them with a consistent and continuous logical time. This leaves us with the possibility of transparently suspending the execution of a prototype without the implementation noticing the actual gap in real-world time.

2.3 Event-based Network Simulation

The key task of the network simulation is to model the network that connects the virtual machines. Following the terminology of Fall [10], we distinguish between an opaque and a protocol-aware network emulation mode. In the case of opaque network emulation, the simulator merely influences the propagation of network traffic,

for example by delaying or duplicating packets. This approach is prevalent in many available tools [1, 2, 5, 30]. By contrast, we focus on protocol-aware network emulation. In this case, the network simulation implements the communication protocols that are used by the VM prototypes. This enables the provision of simulated hosts that interact with the VMs.

For integrating an event-driven network simulation with a *SliceTime* setup, it needs to be interfaced to both the synchronization component (*timing control interface*) and the virtual machines (*data communication interface*). The timing control interface is tightly coupled with 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. Typically, the simulation simply processes these events sequentially until the event queue is empty. In *SliceTime*, a custom 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 through the timing control interface. The next event is processed after the barrier has been shifted past the execution time of the event.

The data communication interface connects the simulation and the virtual machines on the protocol level. The functional integration between the VMs and the network simulation takes place at *gateway nodes* inside the simulation, a concept adapted from [10]. These nodes can be viewed as a simulation's internal representation of the virtual machine they are connected to. Their real functionality is inside the virtual machine and their purpose is to have a communication endpoint inside the simulation at which the packet exchange with the virtual machines takes place.

For performance reasons, many network simulation frameworks use custom data structures to model a network packet, and encapsulation is mostly expressed using pointers to secondary message structures. In contrast, real systems exchange binary information, for example, Ethernet frames. When a binary packet generated by a VM arrives at the simulator, the gateway node takes care of converting it into a network simulation message. Similarly, an outgoing packet must be serialized in an adequate fashion before it leaves the simulation.

3 Implementation

We now discuss our implementation of *SliceTime* comprising three types of main components (see Figure 3): a synchronization component (*synchronizer*), the virtual machine infrastructure and a network simulation. Two different flows of communication are present in our system. The synchronizer delivers the synchronization information over the *timing control interface* using a lightweight signaling protocol. A tunnel that carries Ethernet frames from the VMs to the simulation and vice versa serves as our *data communication interface*. 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.

3.1 Synchronization component

The synchronizer 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. In order to distribute the synchronized components across different physical systems, the synchronization signaling is implemented on top of UDP. 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 the synchronized setup.

3.1.1 Timing Control Interface

One challenge is the large amount of messages that needs to be exchanged between the synchronized VMs and the simulation. For example, if the time slices are configured to a static logical duration of 0.1 ms, the synchronization component needs to issue 10 000 time slices to all attached VMs and the simulation for one second of logical time. An additional massive amount of messages is caused by the synchronized peers to signal the completion of every time slice individually to the synchronizer. Therefore, in order to maintain a good run-time efficiency, it is vital to limit the delays and the overhead caused by synchronization signaling and message parsing. For these reasons, we created a lightweight synchronization protocol based on UDP for *SliceTime*. It provides all communication primitives of the timing control interface. 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. This is necessary to keep both the packet size and the parsing complexity at a very low level.

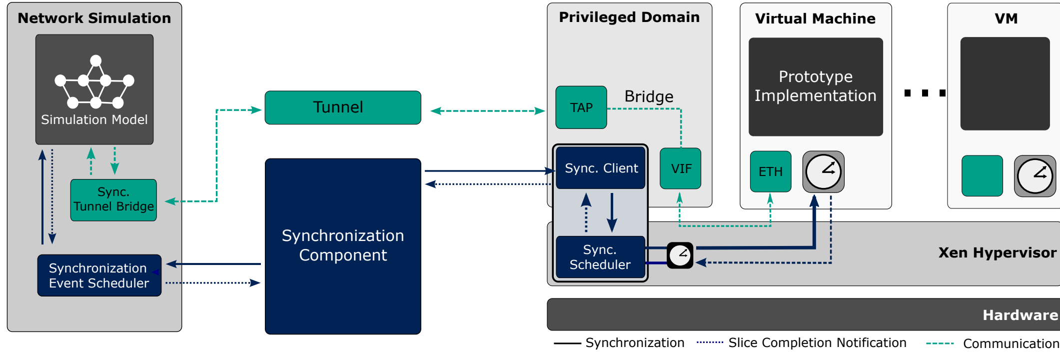


Figure 3: *SliceTime* consists of a central synchronization unit, at least one network simulation based on ns-3 and one or more Xen hypervisor systems serving as the VM infrastructure.

3.2 Virtual Machines

We use the Xen hypervisor 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, it is often referred to as dom0.

Xen supports two modes of operation: para-virtualization mode (PVM) and hardware virtualization mode (HVM). *SliceTime* uses Xen HVM domains for virtualizing operating systems and software prototypes. In contrast to para-virtualization, HVM Xen domains do not require the kernel of the guest system to be modified for virtualization. This allows any x86 OS, also closed source OS such as Windows, to be incorporated into a *SliceTime* set-up.

We now describe the main parts of our work: a) the data communication interface 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.

3.2.1 Data Communication Interface

For the network data communication between virtual machines and simulation, it is first important to note that every virtual machine can have one or several virtual network interfaces that look like real interfaces to the virtual machine, and can be accessed inside dom0. We bridge the virtual interface in the dom0 with a tap device and redirect all Ethernet traffic from the VM to the computer running the simulation. Conversely, all Ethernet frames received from the simulation over the tunnel are fed back to the virtual machine in the same way.

3.2.2 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 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. Therefore, 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 3.1.1. 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 it can inform the synchronizer via UDP. This interrupt-based signaling ensures a prompt processing by the involved entities.

3.2.3 Xen Extensions

The other tasks necessary for our synchronization scheme are carried out within the Xen hypervisor. To reach the goals we set forth, it is necessary to be able to precisely start and stop the VM's operation according to the assigned time slices 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.

To reach the first goal, that is, starting and stopping VMs and running them for precise number of times, 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 these are only 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.

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. Within the modified scheduler, we take 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 RTC and APIC timers. 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.

3.3 Network Simulation

SliceTime relies on ns-3 as network simulator, as opposed to our preliminary work [39,40] in which OMNeT++ was used. In contrast to OMNeT++ and the vast majority of all event-based network simulators, ns-3 internally represents packets as bit vectors in network byte order, resembling real-world packet formats. This removes the need of explicit message translation mechanisms and simplifies the implementation of network emulation features.

The modular design of ns-3 facilitates the integration of the additional components as needed by *SliceTime*. The timing control as well as the communication interface are implemented as completely separate components whose implementation is not intermingled with existing code.

There are some similarities between the *SliceTime* simulation components and the emulation features already provided by ns-3. 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 timing control interface. 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 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 tap bridge attaches to an existing ns-3 network device and creates a virtual tap device in 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 data communication 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 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 System Evaluation

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 3GHz 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 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 Timing Accuracy

In our first experiment, we captured 1 500 ICMP Echo replies (Pings) between a VM and a simulated host for different simulated link delays and time slice sizes. Figure 4 shows the measured RTT distributions for a fixed time slice size of 0.1 ms. We visualize the RTT distributions using standard box plots. The boxes are bounded by the upper and lower quartile of the corresponding RTT distribution. The box represents the middle 50% of

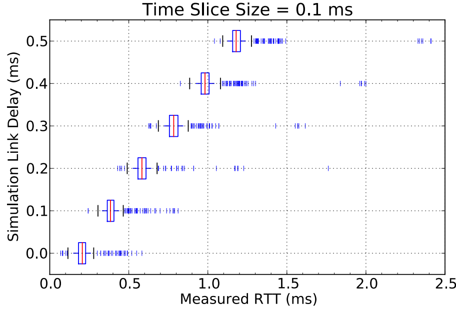


Figure 4: RTTs for different simulated link delays: the simulated delays are correctly perceived by the VM

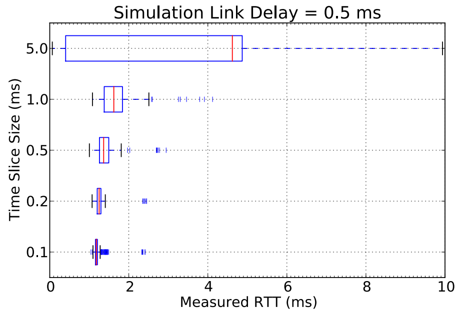


Figure 5: RTT distributions for different time slice sizes: smaller time slices lead to a higher synchronization accuracy and less variance in the measured RTTs.

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. For every series, few outliers are well above the expected range. We explain these deviations with the non-deterministic processing delay of ICMP frames inside the VM's protocol stack. Figure 5 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.

As expected, the variation decreases for smaller time slices and converges towards the expected value of twice the simulated link delay plus the base delay. 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 WAN scenarios (e.g., Internet services) time slices in the range between 0.1 ms and 2 ms are sufficient, as RTTs are mostly in the range of several milliseconds.

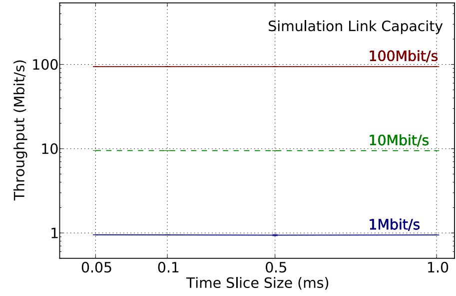


Figure 6: Network Throughput at different time slice sizes: the synchronization does not affect the throughput perceived by the VMs. The measured throughput on the VMs corresponds to the simulated link capacity.

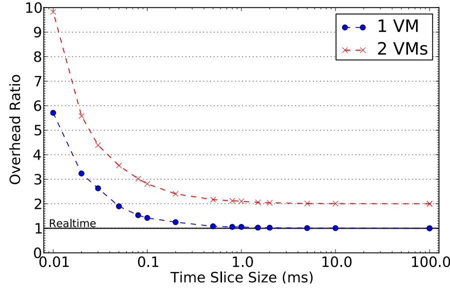
4.2 Throughput Accuracy

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 full-duplex CSMA/CD channels. To each of those two gateway nodes, one VM is connected. Using the netperf [19] `TCP_STREAM` benchmark, we measured the throughput between both VMs. Figure 6 shows the results for different simulated channel bandwidths and varied time slice sizes. The data points are averages over 10 netperf runs, with every run lasting 20 seconds.

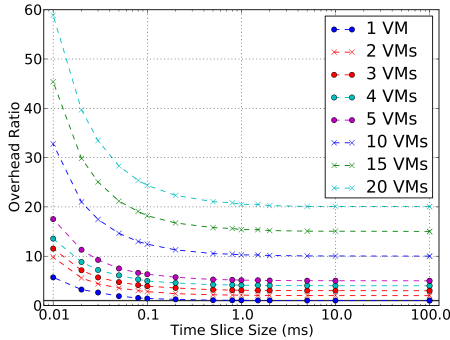
Most notably, the synchronization is transparent to the VMs in terms of perceived TCP bandwidth, as the time slice size has practically no influence on the measured TCP throughput. In addition, the throughput measured on the VMs very well reflects the simulated channel bandwidth. On average, the measured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

4.3 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 7(a) displays the OR of one and two VMs (HVM mode) 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



(a) 1 to 2 synchronized virtual machines



(b) 1 to 20 synchronized virtual machines

Figure 7: 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.

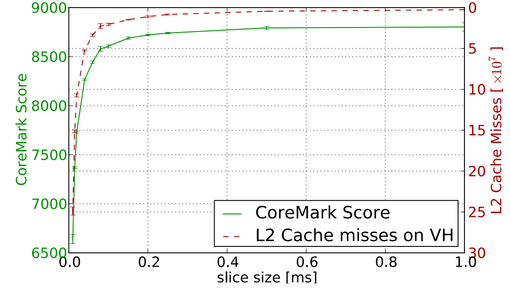


Figure 8: CoreMark CPU Benchmark score at different time slice sizes: For smaller time slices, the CPU performance of a VM decreases due to an increased amount of L2 cache misses. Please note the inverted y-axis on the right.

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 nevertheless facilitates such configurations. Figure 7(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 straight 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. This is a limitation of our current implementation and we regard the parallel execution of multiple synchronized VMs as future work.

4.4 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 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 [34] inside the synchronized VM. CoreMark is a synthetic benchmark for CPUs and microprocessors recently made available by the Embedded Microprocessor Benchmark Consortium (EEMBC). It performs different standard operations, such as CRC calculations and matrix manipulations, and outputs a single CPU performance score. Figure 8 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 then 5%. However, for small 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.

We further investigated this effect using OProfile [28] and its XenoProf [25] 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 8, 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

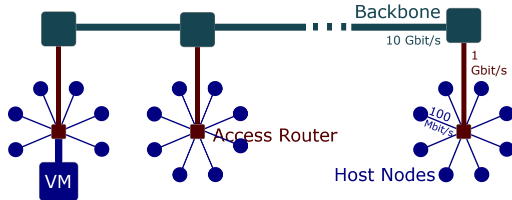


Figure 9: Simple P2P Network: the simulation consisted of one VM and 15 000 simulated nodes (60 backbone nodes with 250 host nodes each)

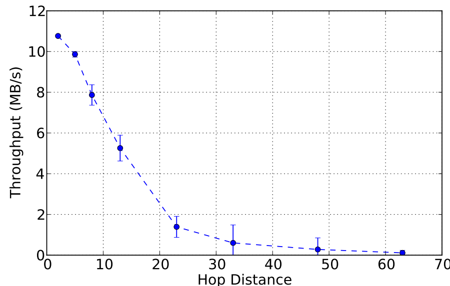


Figure 10: Throughput between VM and simulated hosts at different hopcounts

slices, this minimal performance reduction will have no negative influence on the produced results.

5 Applications

We now describe two typical use cases for *SliceTime*.

5.1 Simple P2P Network

A core motivation of our work is to enable large-scale network emulation setups on customary hardware. In order to stress our framework in this direction we first applied our framework to a large-scale WAN scenario in which 15 000 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 machines (cf. Section 4). One machine executed the VM infrastructure and the synchronizer while the simulation was running on the other one. Figure 9 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 0.1ms. Using the standard `curl` command-line tool we measured the HTTP throughput between the virtual machine and simulated hosts at different hop distances (see Figure 10). 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 shows that *SliceTime* enables one to evaluate real-world networking software in a large-scale simulated context at low hardware and minor setup costs, especially if compared with equally sized physical testbeds or simulation hardware capable of executing the same simulation in real-time.

5.2 WiFi Software

SliceTime enables investigations of WiFi software for Linux in a fully isolated, deterministic and reproducible context. The 802.11 software is deployed on a set of VMs, while the network simulation models the wireless channel, the medium access control as well as potential node movement. In addition, the network simulation can optionally be used to also model other parts of the network, such as 802.11 access points, other mobile hosts or an arbitrary wide-area network connecting the 802.11 infrastructure. In the following, we briefly describe the 802.11 extensions of *SliceTime* before we use our framework to remodel a real-world field test of an AODV routing daemon for Linux.

5.2.1 *SliceTime* 802.11 extensions

To enable WiFi support in *SliceTime* we designed a second data communication interface (cf. Section 3.2.1). Figure 11 illustrates its core components and layers. On the VM a loadable kernel module forms the *SliceTime* device driver that provides a virtual WiFi interface. The device driver implements the 802.11 wireless extensions for Linux network devices. This makes the virtual WiFi interface look like a real wireless networking card. For example, commands such as `iwconfig` may be used to put the virtual WiFi device into monitor mode. The actual WiFi software may directly access this interface or rely on the Linux TCP/IP stack for its communication purposes. So-called WiFi gateway nodes represent the VMs inside the simulation. The WiFi gateway nodes perform all 802.11 MAC layer operations, for instance sending ACKs, that are normally carried out by WiFi hardware. A major benefit of this approach is that all communication events being sensitive to strict timing constraints remain in the simulation domain. Typically a relatively loose VM-simulation synchronization accuracy of 0.5ms and hence low overhead is sufficient for most *SliceTime* WiFi set-ups. By contrast, implement-

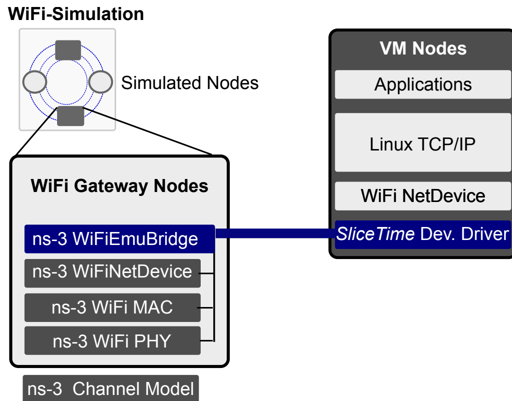


Figure 11: *SliceTime* provides a virtual network device to the VMs that integrates with ns-3 at the MAC layer. This facilitates testing arbitrary WiFi and networking software with, for example, reproducible channel conditions and node movement.

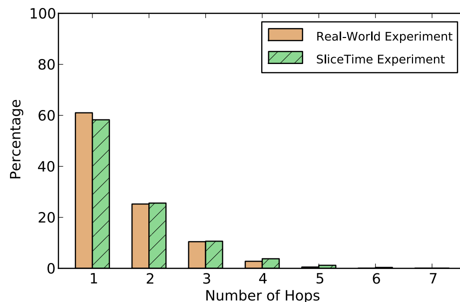


Figure 12: Real-World AODV experiment vs. remodeled *SliceTime* scenario: the hopcount distribution of received packets obtained from the scenario remodeled with *SliceTime* well matches the hopcounts measured in the real-world scenario.

ing the MAC behavior in the driver would require a synchronization accuracy lower than the 802.11 inter-frame spaces (IFS). Despite the IFS being smaller than the maximum synchronization accuracy of *SliceTime*, the high messaging overhead for such tight intervals would also render such a design impractical.

Besides implementing the data exchange between the VM device driver and the ns-3 simulation model, the WifiEmuBridge also maps configuration actions such as triggered by `iwconfig` to corresponding operations in ns-3. In addition it is able to export packet-level statistics such as RSSI values to the software running on the VM using Radiotap packet headers. A more elaborate discussion of our ns-3 WiFi emulation extensions can be found in [38].

5.2.2 AODV routing daemon study

We used *SliceTime* to remodel the AODV part of a real-world field test [13] 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. Corresponding trace files are publicly available at the CRAWDAD repository [14]. 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 *SliceTime*. 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. Figure 12 compares the AODV hopcount distributions of received packets for the real-world data and the corresponding remodeled scenario. The hopcounts measured using *SliceTime* well match the observations from the real-world field test. We also determined the average packet delivery ratio (PDR) for the real-world experiment and the emulated scenario. From the CRAWDAD traces we calculated the avg. PDR to be 42.10% for the real-world AODV experiment. In our remodeled scenario the avg. PDR amounts to 46.39%.

There will always be differences between real-world measurements and observations taken with systems such as *SliceTime*. 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 that could have influenced the real-world measurements.

Nevertheless, this use case shows that *SliceTime* is well able to provide a testing environment for 802.11 software that delivers results being close to reality. Repeating real-world experiments like the one conducted by Gray [13] is costly and often challenging due to continually changing conditions, for example, regarding the wireless channel. By contrast, *SliceTime* allows one to ar-

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

6 Related Work

Early contributions [1, 17, 30, 36] in the field of network emulation focus on opaque network emulation in which physical network systems are connected to an emulation engine that models the network propagation. The model affects the packet flow, either by introducing delay, jitter, bandwidth limitations, or packet errors. Later contributions extend this methodology for the emulation of Internet paths [31] or use real-world measurements [5] for accurately reproducing the behavior of large-scale networks. Opaque network emulation is an effective method to investigate the impact of network propagation characteristics on protocol performance. However, because all communicating peers are physical systems, the analysis of large-scale scenarios (e.g., P2P and overlay networks) with many hosts is difficult.

Protocol-aware network emulation was introduced by Fall [10], proposing the combination of real network systems and discrete event-based simulations. This implementation has been improved later in terms of timing accuracy [24]. Protocol-aware emulation features also exist for other event-based network simulators [35]. All of these implementations are subject to potential simulation overload. Kiddle [21] used massive computing power in form of hardware to increase the execution speed of the simulation to circumvent this problem. While this works up to a certain point, our aim is in the opposite direction of slowing down the real system, saving on hardware expenses and setup complexity.

Erazo et al. recently proposed SVEET! [9], a hybrid TCP evaluation environment that integrates Xen-based VMs with an SSFNET [8]-based emulation engine. Although SVEET! involves a mechanism to cope with simulation overload, it differs significantly from our work. In order to match the execution speed of both the VMs and the emulation engine, SVEET! utilizes a static time dilation factor (TDF). The TDF is used to throttle down the speed of both the simulator and the VMs to the worst-case run-time performance of the emulation engine. The main drawback here is the need to correctly choose the TDF beforehand. If the chosen TDF is too large, the run-time is increased without any benefit due to underutilization of system resources. If the chosen TDF is too small, simulation overload and time drifts can occur, leading to flawed results. In contrast, our approach does not statically throttle the execution speed of any component by a constant factor. Moreover, the conservative barrier algorithm used in our work limits the drift of all components to the duration of one time slice.

Different virtualization-based opaque network emulation approaches have been discussed over the past years. ENTRAPID [16] executes multiple instances of the FreeBSD network stack in the user space. These virtual network kernels (VNKs) are wired together and form a network emulation environment. As the VNKs are executed simultaneously and operate in wall-clock time, this limits the scalability of this approach. dONE [4] proposes the virtualization of time to address this problem. Despite this similarity *SliceTime* differs significantly from both dONE and ENTRAPID: first, neither dONE nor ENTRAPID integrate software prototypes with an event-based network simulation at all. By contrast, *SliceTime* relies on ns-3 as emulation backend. This enables the set-up of emulation scenarios that access all models and features of the network simulator. Second, in opposition to *SliceTime*, neither dONE nor ENTRAPID allow the investigation of entire network protocol stacks, as both draw the line between the emulation environment and software prototypes right at the socket layer. Diecast [15] and Time Jails [12] facilitate the setup of a network emulation testbeds solely based on virtual machines. The main advantage compared to the aforementioned systems is that they allow one to execute unmodified software and protocol stacks. Both are an attractive option for real-world experiments in which the number of nodes exceeds the quantity of physical hosts of a testbed. In addition, Diecast not only scales time, but also the performance of system components to accurately model a realistic hardware behavior profile. *SliceTime*, by contrast, follows a different goal. Instead of virtualizing time to increase the capacity of a physical testbed, we employ it for synchronizing a VM with a network simulation that forms the emulation engine. This has two advantages. First, using a network simulator as backend allows us to put concepts such as virtual node mobility into action, which is not possible with neither DieCast nor Time Jails. Second, the scalability of the simulator opens up the possibility of implementing large-scale emulation scenarios that could not be realized using VMs alone without taking up much higher hardware resources.

Emulab [42] is a well-established large network testbed allowing for the evaluation of networked software in different communication environments. Its main strength is the ability to specify network scenarios using a configuration file which Emulab maps to the testbed hardware. In order to reproduce the characteristics of networks of many kinds, Emulab also employs opaque network emulation between the testbed nodes. In direct comparison with *SliceTime*, Emulab achieves its flexibil-

ity by incorporating a huge amount of networked computers, network infrastructure as well as auxiliary components. We admire the efforts and achievements of its creators in this regard. *SliceTime* instead aims at providing a flexible and scalable network experimentation and evaluation platform with very modest hardware requirements. This is reflected in our evaluation which at most required two Desktop PCs to carry out the large-scale WAN experiment. We achieve this goal by scaling execution time and by modeling large parts of the scenario using the ns-3 simulator. On one hand the use of a simulator limits the possible degree of realism due to discrepancies between the real world and the corresponding simulation models. On the other hand relying on a simulation allows the construction of "virtual network testbeds" that are not dependant on the availability of physical hardware or real network infrastructure.

Wireless network emulation tools, such as the CMU Emulator [20], interconnect antenna connectors of standard wireless network hardware via cables. Complex hardware, mostly based on FPGAs and DSPs, is used to model the wireless channel. While this enables a quite realistic emulation, it requires complete physical hardware for each station. There is also number of pure software-based wireless network emulation tools. Most of them, such as [26, 29, 43], only mimic the propagation of packets on the wireless link and do not support simulated wireless stations. A few wireless network emulation systems [22, 32, 33] are based on event-based network simulators. They share some similarities with the WiFi extensions of *SliceTime*, but differ significantly in the way they interface the software prototypes with the 802.11 simulation. In [22,32] the 802.11 simulation model is integrated with the software at the IP layer, which prevents investigations of 802.11 software using a different routing protocol than IP. VirtualMesh [33] bridges the gap between the simulation and the WiFi software at the MAC layer, but requires the modification of all applications making use of the wireless extensions. By contrast, the 802.11 add-ons of *SliceTime* introduce a clean cut between the simulation and the prototypes at the MAC layer. This enables arbitrary WiFi software for Linux to be evaluated without any changes to the software.

7 Conclusion

In this paper we presented *SliceTime*, a platform for scalable and accurate network emulation. *SliceTime* 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 network simulations of any size with VM based prototypes regarding timing and network bandwidth in a transparent way.

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

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. As we believe that *SliceTime* will be useful for a number of researchers and developers, we have made the source code available to the public. It can be downloaded at `http://www.comsys.rwth-aachen.de/research/projects/slicetime`.

Acknowledgements

We express our gratitude to our shepherd Remzi Arpaci-Dusseau and our anonymous NSDI reviewers for their valuable and helpful comments. We also greatly thank Martin Lindner and Suraj Prabhakaran for conducting additional measurements and Simon Rieche and Stefan Götz for many fruitful discussions. This research was partially funded by different DFG grants and the UMIC excellence cluster, DFG EXC 89.

References

[1] ALLMAN, M., AND OSTERMANN, S. ONE: The Ohio Network Emulator. Technical Report TR-19972, Ohio University, 1997.

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

[3] BARHAM, P., DRAGOVIC, B., FRASER, K., HAND, S., HARRIS, T., HO, A., NEUGEBAUER, R., PRATT, I., AND WARFIELD, A. Xen and the art of virtualization. In *Proc. SOSP'03* (Bolton Landing, NY, USA, Oct. 2003), ACM.

[4] BERGSTROM, C., VARADARAJAN, S., AND BACK, G. The distributed open network emulator: Using relativistic time for distributed scalable simulation. In *Proc. PADS'06* (May 2006), pp. 19–28.

[5] CARSON, M., AND SANTAY, D. NIST Net: A Linux-based network emulation tool. *ACM Comp. Commun. Rev. 33*, 3 (2003), 111–126.

[6] CHANDY, K. M., AND MISRA, J. Distributed simulation: A case study in design and verification of distributed programs. *IEEE Trans. on Software Engineering SE-5*, 5 (Sept. 1979), 440–452.

[7] CHUN, B., CULLER, D., ROSCOE, T., BAVIER, A., PETERSON, L., WAWRZONIAK, M., AND BOWMAN, M. PlanetLab: An Overlay Testbed for Broad-Coverage Services. *ACM Comp. Commun. Rev. 33*, 3 (2003), 3–12.

[8] COWIE, J., NICOL, D., AND OGIELSKI, A. Modeling the global internet. *Computing in Science & Engineering 1*, 1 (Jan/Feb 1999), 42–50.

[9] ERAZO, M. A., LI, Y., AND LIU, J. SVEET! a scalable virtualized evaluation environment for TCP. In *Proc. TRIDENTCOM'09* (2009), IEEE Computer Society, pp. 1–10.

[10] FALL, K. R. Network emulation in the Vint/NS simulator. In *4th IEEE Symposium on Computers and Communication* (1999).

[11] FUJIMOTO, R. M. Parallel discrete event simulation. *Commun. ACM 33*, 10 (1990), 30–53.

[12] GRAU, A., MAIER, S., HERRMANN, K., AND ROTHERMEL, K. Time Jails: A hybrid approach to scalable network emulation. In *Proc. PADS'08* (June 2008), pp. 7–14.

[13] GRAY, R. S., KOTZ, D., NEWPORT, C., DUBROVSKY, N., FISKE, A., LIU, J., MASONE, C., MCGRATH, S., AND YUAN, Y. Outdoor experimental comparison of four ad hoc routing algorithms. In *Proc. MSWiM'04* (2004).

[14] GRAY, R. S., KOTZ, D., NEWPORT, C., DUBROVSKY, N., FISKE, A., LIU, J., MASONE, C., MCGRATH, S., AND YUAN, Y. CRAWDAD data set dartmouth/outdoor (v. 2006-11-06). Downloaded from `http://crawdad.cs.dartmouth.edu/dartmouth/outdoor`, Nov. 2006.

[15] GUPTA, D., VISHWANATH, K. V., AND VAHDAT, A. DieCast: Testing distributed systems with an accurate scale model. In *NSDI'08* (San Francisco, CA, USA, 2008), USENIX.

[16] HUANG, X., SHARMA, R., AND KESHAV, S. The ENTRAPID protocol development environment. *INFOCOM '99* (Mar. 1999).

[17] INGHAM, D. B., AND PARRINGTON, G. D. Delayline: A wide-area network emulation tool. *Comput. Syst. 7*, 3 (1994), 313–332.

[18] JEFFERSON, D. R., AND SOWIZRAL, H. Fast concurrent simulation using the time warp mechanism. *Simulation Series, Soc. for Computer Simulation 24-26 Jan 1985 15* (1985), 63–69.

[19] JONES, R., CHOY, K., AND SHIELD, D. Netperf. [Online] Available `http://www.netperf.org` December 21, 2009.

[20] JUDD, G., AND STEENKISTE, P. Repeatable and realistic wireless experimentation through physical emulation. *ACM SIGCOMM Computer Communication Review 34*, 1 (2004), 63–68.

[21] KIDDLE, C. *Scalable Network Emulation*. PhD thesis, Department of Computer Science, University of Calgary, 2004.

[22] KROP, T., BREDEL, M., HOLLICK, M., AND STEINMETZ, R. JiST/MobNet: combined simulation, emulation, and real-world testbed for ad hoc networks. In *Proc. WinTECH'07* (New York, NY, USA, 2007), ACM, pp. 27–34.

[23] LUBACHEVSKY, B. D. Efficient distributed event-driven simulations of multiple-loop networks. *Comm. ACM 32* (1989), 111–123.

[24] MAHRENHOLZ, D., AND IVANOV, S. Real-time network emulation with ns-2. *8th IEEE International Symposium on Distributed Simulation and Real-Time Applications (DS-RT)* (2004).

[25] MENON, A., SANTOS, J. R., TURNER, Y., JANAKIRAMAN, G. J., AND ZWAENEPOEL, W. Diagnosing performance overheads in the Xen virtual machine environment. In *Proc. VEE 2005, Chicago,USA* (2005).

[26] NOBLE, B., SATYANARAYANAN, M., NGUYEN, G., AND KATZ, R. Trace-based mobile network emulation. In *Proc. SIGCOMM'97* (1997), ACM New York, NY, USA, pp. 51–61.

[27] ns-3 Website. `http://www.nsnam.org/` (accessed Oct. 2010.

[28] OProfile: a system profiler for linux. `http://oprofile.sourceforge.net` (accessed Oct. 2010).

[29] PUŽAR, M., AND PLAGEMANN, T. NEMAN: A network emulator for mobile ad-hoc networks. Tech. Rep. 321, Department of Informatics, University of Oslo, 3 2005.

[30] RIZZO, L. Dummynet: A simple approach to the evaluation of network protocols. *ACM Comp. Commun. Rev. 27*, 1 (1997), 31–41.

[31] SANAGA, P., DUERIG, J., RICCI, R., AND LEPREAU, J. Modeling and emulation of internet paths. In *Proceedings NSDI'09* (2009), USENIX Association, pp. 199–212.

[32] SEIPOLD, T. Emulation of radio access networks to facilitate the development of distributed applications. *JOURNAL OF COMMUNICATIONS 3*, 1 (2008), 1.

[33] STAUB, T., GANTENBEIN, R., AND BRAUN, T. VirtualMesh: an emulation framework for wireless mesh networks in OMNeT++. In *Proc. SIMUTools'09* (Brussels, Belgium, 2009), pp. 1–8.

[34] THE EMBEDDED MICROPROCESSOR BENCHMARK CONSORTIUM. CoreMark. [Online] Available `http://www.coremark.org` December 27, 2009.

[35] TUEXEN, M., RUENGELER, I., AND RATHGEB, E. P. Interface connecting the INET simulation framework with the real world. In *Proc. 1st International Workshop on OMNet++* (2008).

[36] VAHDAT, A., YOCUM, K., WALSH, K., MAHADEVAN, P., KOSTIC, D., CHASE, J. S., AND BECKER, D. Scalability and accuracy in a large-scale network emulator. In *Proc. OSDI'02* (2002).

[37] VARGA, A., AND HORNIG, R. An overview of the OMNeT++ simulation environment. In *SIMUTools 2008* (Marseille, France, March 2008).

[38] WEINGAERTNER, E., VOM LEHN, H., AND WEHRLE, K. Device-driver enabled wireless network emulation. In *Proc. SIMUTools 2011* (Barcelona, Spain, March 2011).

[39] WEINGÄRTNER, E., SCHMIDT, F., HEER, T., AND WEHRLE, K. Synchronized network emulation: matching prototypes with complex simulations. *SIGMETRICS Perform. Eval. Rev. 36*, 2 (2008), 58–63.

[40] WEINGÄRTNER, E., SCHMIDT, F., HEER, T., AND WEHRLE, K. Time accurate integration of software prototypes with event-based network simulations. In *Proc. of the Poster session at SIGMETRICS 2009* (Seattle, USA, 2009).

[41] WERNER-ALLEN, G., SWIESKOWSKI, P., AND WELSH, M. Motelab: a wireless sensor network testbed. In *Proc. IPSN'05* (Piscataway, NJ, USA, 2005), IEEE Press, p. 68.

[42] WHITE, B., LEPREAU, J., STOLLER, L., RICCI, R., GURUPRASAD, S., NEWBOLD, M., HIBLER, M., BARB, C., AND JOGLEKAR, A. An integrated experimental environment for distributed systems and networks. *SIGOPS Oper. Syst. Rev. 36*, SI (2002), 255–270.

[43] ZHANG, Y., AND LI, W. An integrated environment for testing mobile ad-hoc networks. In *Proc. MobiHoc'02* (New York, NY, USA, 2002), ACM, pp. 104–111.