VMSimInt: A Network Simulation Tool Supporting Integration of Arbitrary Kernels and Applications

Thomas Werthmann, Matthias Kaschub, Mirja Kühlewind, Sebastian Scholz, David Wagner
Institute of Communication Networks and Computer Engineering (IKR), University of Stuttgart, Germany
simutools@thomaswerthmann.name, simutools@mkaschub.de,
mirja.kuehlewind@ikr.uni-stuttgart.de, sebastian.scholz@ikr.uni-stuttgart.de,
research@davidwagner.de

ABSTRACT

Integrating realistic behavior of end systems into simulations is challenging since the mechanisms used in protocols and applications such as Transmission Control Protocol (TCP) are complex and continuously evolving. In this paper, we present VMSimInt, a new approach which allows the INTegration of arbitrary Operating Systems (OSs) and application code into an event-driven network SIMulator by using Virtual Machines (VMs). In contrast to existing approaches which integrate parts of OS kernels, our approach uses unmodified OS kernels, which eases maintenance and provides additional flexibility. By controlling the time and all I/O of the VMs, our approach guarantees that external factors such as the performance of the host do not influence the simulation outcome, so that simulations are exactly reproducible. We validated our system against the Network Simulation Cradle (NSC) by simulating the same models and comparing the system behavior. In addition, we show that our approach provides sufficient performance for usage in day-to-day research.

Categories and Subject Descriptors

C.4 [**Performance of Systems**]: Modeling Techniques; I.6.8 [**Simulation and Modeling**]: Types of Simulation—*Discrete Event*

General Terms

Performance, Measurement

Keywords

Network Protocols, TCP Simulation, Virtual Machines

1. MOTIVATION

Network research strongly relies on tests, emulations, or simulations in a controlled environment to perform studies on new protocols, network or application mechanisms. In contrast to emulation, simulation has the advantage that it avoids any external influence. Due to the complexity and rapid development of protocol and application behavior, abstract models often do not reflect the characteristics of real systems well enough.

The Transmission Control Protocol (TCP) dominates the traffic behavior in today's Internet and thus is an important factor in the investigation of new proposals. Therefore providing realistic Operating System (OS) behavior such as TCP in packet-level simulations is one of the key challenges for network simulation. In addition, when evaluating TCP mechanisms like congestion control, it is important to use recent TCP implementations, because, for example, using an initial TCP congestion window of 10 and Proportional Rate Reduction (PRR) strongly influences congestion control behavior. Most simulation environments either re-implement protocol standards or re-use code fragments from OS kernels. However, the networking code of modern OSs evolves fast. Thus these approaches demand high maintenance effort to keep the implementation up-to-date. Furthermore, validation of derived models is difficult and often omitted.

Additionally, Internet traffic is strongly influenced by application behavior which evolves even faster. Instead of modeling applications it is often desired to use real application code in a simulation environment (potentially on top of realistic TCP behavior). For example, the loading time of a web page strongly depends on the number of parallel TCP connections and their interaction in the network.

In this paper, we present VMSimInt, a new approach for packet-level simulations which is based on the integration of Virtual Machines (VMs) into the simulation environment. Thus it allows using unmodified kernel code, OS mechanisms or existing applications within the simulation. The main advantage regarding simulation of transport protocol behavior is that our simulation approach can use any OS without any patching, given that it can be run on the virtualized hardware platform. Compared to emulation-based approaches which connect real computer systems or several VMs on one computer to a simulated network, VMSimInt provides full isolation from the host system. The time perceived by a kernel running in a VM is controlled by the simulation framework and completely independent of the host time. This simulated time does not proceed while the VM's processor is operating, so the performance of the host computer or the virtualization tool does not influence the simulation results. This allows us to reproduce exactly the same behavior with multiple simulation runs without any influence, e. g., of other processes running on the host system.

Simutools 2014, March 17-19, Lisbon, Portugal

DOI 10.4108/icst.simutools.2014.254623

We implemented VMSimInt as an add-on for the IKR SimLib [11], a freely available library for event-driven (network) simulation written in Java. However, it could also be used by other event-driven network simulators such as ns-3 [10]. VMSimInt uses the QEMU software [2] for hosting VMs. We patched the QEMU software such that the simulation environment fully controls the virtual hardware clock of the VM as well as all input and output processing. Moreover, a helper program inside each VM allows the simulation environment to access the socket Application Programming Interface (API) of the VM and thus to generate TCP traffic. During simulation, packets sent by any VM are not forwarded to the host's network but are handled and forwarded within the simulated network, which is modeled in the simulation environment. This framework allows defining simulation scenarios consisting of simulation entities, e. g., links with defined delay and bandwidth and switches, as well as functions provided by arbitrary OSs and applications.

The remainder of the paper is organized as follows: Section 2 provides an overview over different approaches to integrate transport protocol behavior into network evaluation tools. Section 3 describes general concepts of our approach and the implementation of VMSimInt. In Section 4 we validate VMSimInt against the NSC and present a performance evaluation regarding memory usage and runtime.

2. APPROACHES FOR TCP RESEARCH

In this section we give an overview of approaches used in existing tools for research on transport layer mechanisms.

2.1 Abstract Model

In literature a wide range of models for TCP exist, e. g., modeling the sending rate of a TCP instance for stable scenarios. Analytical models are very limited as they usually focus on single aspects of TCP, e. g., a certain congestion control algorithm [14]. They often do not reflect real systems well due to complexity and rapid development of TCP. Moreover, these models are too abstract for event-driven, packet-based simulations: they do not provide exact timings and do not reflect the reactive ACK-clocking of TCP.

2.2 Emulation based on Host Clock Time

Network emulation tools connect a modeled network to real computer systems. This approach provides realistic endsystem and transport protocol behavior, but has a strong dependency on the real time clock of the used computer systems. Moreover the emulation runs are possibly influenced by other processes that run in parallel, e. g., background services of the OS. Several approaches for network emulation have been proposed that use real TCP traffic in a simulated network [9, 4, 15].

It has also become common practice to use VMs for emulation as this saves hardware resources, also for other use cases than TCP research. Some of these approaches scale the time basis of the VMs, so that it becomes possible to emulate a high performance network with many VMs on a host system with lower performance. However, there is still a dependency on the real clock time. Mininet [7] and ModelNet [17] are emulators where the time in the VM can run slower, but is still coupled with the wall clock time of the host system by a fixed factor. Time Jails [6] is an approach that manipulates the time basis of VMs to use processing resources for large scale network emulation more efficiently.

As stated in [7] by the inventors of Mininet-HiFi, performance isolation is an important issue: The performance of the host system and the computation load of background jobs should not influence the simulation results. Nevertheless, ideal isolation cannot be achieved if the VMs still interface the host OS for timing or network communication. So, results are not reliably and exactly reproducible.

Emulation techniques can provide more realistic results than a simulation model, but have a higher complexity in operation and maintenance and do not provide reproducible results.

2.3 Protocol Implementation

Another approach to get realistic TCP behavior is to embed a full implementation of the protocol into the simulation program. This can either be achieved by an independent implementation of the standards, or by re-using code of existing OSs. Often the latter results in more realistic behavior, since the widely used real system implementations do not always comply with the standards.

2.3.1 Independent Implementation

A simulation tool can be enhanced by just adopting selected function code from the protocol to exactly cover the needed behavior. This can be a new implementation or based on code from existing OSs. Like other simulators, the native TCP stack of ns-2, a commonly used network simulator allowing for TCP packet-level simulation, aims to re-implement relevant functionalities of TCP. Even though ns-2 is used and maintained by a large community, it can hardly be validated as the implemented functionality does not cover the complexity of a real TCP system [8, 20]. Besides, the process of (manually) adopting protocol changes is time consuming. [18] goes one step further and describes an extension to ns-2 that allows to integrate certain complete files of the Linux kernel with a well-defined interface, the congestion control modules, directly into the simulation.

2.3.2 Adaptation of the Network Stack of a Real OS

To address this problem more generally, the Network Simulation Cradle (NSC) [12, 13] provides an automated framework converting the network stack of selected versions of some OSs like Linux or FreeBSD such that it can be used as a library in user space programs. Further, the NSC defines an interface for network simulators to interact with a network stack library. The NSC concept has been fully integrated in ns-3 which is the successor of ns-2. This approach heavily modifies the kernel code and uses internal interfaces between the protocol stack and the remainder of the kernel. Manual adaptation is required to apply the framework to other (new) kernel versions. Currently the latest version of the Linux kernel that can be used by NSC is 2.6.26, which was released in 2008. Besides, the NSC only supports IPv4.

Recently, Direct Code Execution (DCE) Cradle [16] was proposed, which also supports Linux version 3.4. As NSC, DCE Cradle extracts the network stack from the kernel code. By using DCE the authors automated the process of making several instances of kernel code usable in user space. While only minor changes to the kernel code itself where performed, a wrapper is introduced to access kernel internal function calls. As NSC, DCE bases on internal kernel interfaces which might be subject to change. Thus to update to a new kernel version, typically there is some manual work

required to adapt the wrapper to changed kernel structures.

Similar approaches were proposed for other network simulators, e. g., OMNeT++ [3].

2.3.3 Integration of VMs Running Full OS Kernels

Another approach is using virtualization technology to integrate behavior of real systems into simulations. The UML-Simulator [1] modifies UserModeLinux for this purpose. Due to its virtualization choice it only supports integrating Linux kernels and still needs some changes to this kernel. The latest supported kernel is 2.6.11. SliceTime [19] is a more recent approach to integrate VMs into simulation based on the Xen hypervisor. Due to the choosen concept of connecting the VMs with the simulation, this approach has timing and also bandwidth inaccuracies. The NetWarp [21] simulator is based on Xen, too. It aims to support multicore VMs on multicore hosts. Therefore the Xen internal scheduler is modified in a way that the internal times of the VMs are aligned close to the simulation time. Even if the simulation specific scheduler can reduce timing inaccuracies, they can not be eliminated completely.

The approach presented in this paper is based on integrating unmodified kernels and applications into simulations by using a modified virtualization tool. The virtualization tool completely decouples the VMs from the host system by controlling the clock and all I/O. Therefore, in contrast to emulations using VMs, like Mininet and ModelNet, the performance isolation in this approach is ideal by design.

Compared to the integration of network stacks, the runtime overhead of VMSimInt is increased by including a whole operating system. However, our approach, other than NSC and DCE Cradle, does not rely on internal interfaces of the OS kernel. Instead, we interface the kernel via the emulated hardware of the VM and via a userspace program running inside the VM. Both hardware support and user space APIs are known to be stable for a long time (e. g., current OSs still run on old hardware, and old programs still run on current OSs). This significantly eases maintenance, as we can use new kernel versions without any adaptations. These interfaces do also allow our approach to support closed source OSs. Although not realized yet, the behavior of the Microsoft Windows network stack could be included without changing the OS code at all.

In this section we classified existing approaches for packet-level simulation of transport protocols. While emulation-based approaches are not independent of the host system, simulation approaches which re-use existing code instead come with higher maintenance cost. Therefore, we chose an approach which is based on integrating VMs into simulation. In the following section we present a concept that fully decouples the VMs from the host system, and thereby provides reproducible results in simulations. Although this approach has more processing overhead than approaches that only include the relevant part of the code, it is easy to maintain and simple to extend to new kernel versions.

3. CONCEPT AND IMPLEMENTATION

Our framework (Figure 1) is based on an event-driven simulation program, which models the network and is controlled by a simulation calendar. To use OS and application code, this simulation program interacts with multiple VM instances. The framework allows interactions with the user

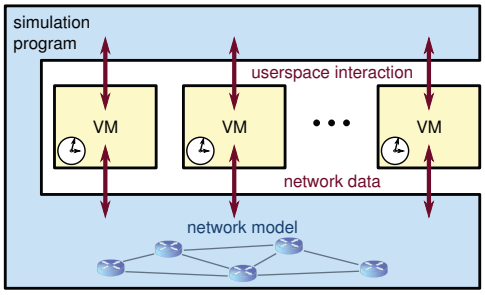

Figure 1: Overview of the architecture.

space of the VMs, it intercepts I/O to forward network data to the simulation program, and it couples the time of the VMs to the simulation calendar.

Our framework consists of three components: an extended QEMU running as a separate process for each VM, a QEMU Adapter class in the simulation program that handles all communication with a QEMU process, and a relay program running in the user space of each VM to provide direct access to the socket API of the OS instance.

Figure 2 shows these components. The QEMU Adapter encapsulates the VMs in the simulation. The Adapter interfaces the virtual kernel on two layers. The lower layer (1) interacts with the extended QEMU to control the clock of the VM and to intercept I/O (e. g., the network interface). The upper layer (2) interacts with the relay program to make the socket API of the kernel accessible from the simulation program. When the simulation program triggers a TCP transmission, it uses the upper layer connection to write data to a TCP socket, which causes the kernel to write Ethernet frames to the network interface. The lower layer of the Adapter forwards these to the simulation environment, where the simulated network topology might forward them to another VM instance.

Subsection 3.1 outlines the QEMU extensions as well as the lower layer of the QEMU Adapter. Subsection 3.2 details the mechanism to relay the socket layer API into the simulation program, consisting of the relay program and the upper layer of the QEMU Adapter.

3.1 Integrating QEMU into Simulation

There are many VM solutions available. We based VM-SimInt on QEMU [2], because it is Open Source software, it supports many host and guest platforms, it is widely used and it supports performance features such as Kernel-based Virtual Machine (KVM) and Kernel SamePage Merging (KSM). Further, the software architecture of QEMU is well suited for our approach: All virtual hardware components are split into a virtual device and a device backend. The virtual device makes the emulated hardware component available to the OS running in the VM. The device backend realizes the functionality of the virtual hardware. In the unmodified QEMU, the backend would connect the VM to a device of the host computer. We modified this to connect the guest OS to the simulation program instead.

3.1.1 Extended QEMU

We extended QEMU in two aspects: Clock control and I/O redirection.

The simulation program controls the system time of the

VMs, i. e., the virtual hardware clocks seen by the guest OS instances. Thus when the guest OS reads the time, the current simulation time is returned, and when it programs a timer, an event is posted to the simulation calendar.

Any network interface of the VMs is connected to a logical link of the simulated network topology. Thus the simulation program handles Ethernet frames written by the guest OS according to the simulation scenario and the simulation program may send frames to the guest OS. In addition, the VMs provide a bidirectional interface to relay data between the simulation program and programs running in the user space of the VMs. This can be used to control applications running in the user space of the VM. It is used by the relay program as explained in Subsection 3.2

Clock Control.

We differentiate two sources of time: wall clock time and simulated time. The wall clock time is the real-time as seen by the host OS. Event-driven simulations do usually not rely on wall clock time, but define a simulated time. The simulated time is independent of the wall clock time. During the execution of an event (an arbitrary task in the simulation model), the simulated time does not proceed. Time spans without events are skipped, so the simulated time jumps from event to event. In average, the simulated time can either run faster by skipping time intervals without events or slower if the computational complexity is high (e. g., solving an optimization problem for a routing decision or computing channel models in radio networks).

Standard VMs are synchronized to the wall clock time, i. e., the timing of the guest OS matches that of the host system. In contrast, in our approach we align the time of the VM to the simulated time to maintain the benefits of an event-driven simulation. Software running inside the VMs only sees the simulated time.

The time inside the VMs is provided by a virtual clock chip. This virtual clock chip is a High Precision Event Timer (HPET) that offers sub-microsecond resolution and also allows the OS to omit fixed millisecond ticks (e. g., Linux dyntick). The OS can use the HPET interface to read the current time and to program timer interrupts. We intercepted the virtual clock chip so that reading the current time returns the simulated time instead of the wall clock time. In addition, whenever the OS programs a timer interrupt, this is converted to an event which is scheduled in the simulation calendar. At the scheduled point in (simulated) time, the simulation executes the event. Thereby, it wakes up the VM and triggers the programmed interrupt.

To provide reproducible simulation results, we emulate the VM with an infinitely fast processor (in terms of simulated time). Each activity of a VM is a result of a simulation event (e. g., network I/O or expiry of a timer, maybe created by the VM's OS itself). The simulation wakes up the respective VM and updates the time of the VM to the current simulated time. Then the simulation program provides the required I/O and runs the virtual CPU until it becomes idle. Subsequently the VM is paused and the simulation proceeds by executing the next event. While the virtual CPU is running, the simulated time does not proceed. Thus neither the simulation program, nor any other QEMU process performs any task during this time. By strictly serializing the operations of the simulation program and the VMs, we trade simulation speed for simulation accuracy. Our approach guarantees that timing and results of the operations inside the VMs can not be influenced by the host's environment (e. g., programs running on the host computer).

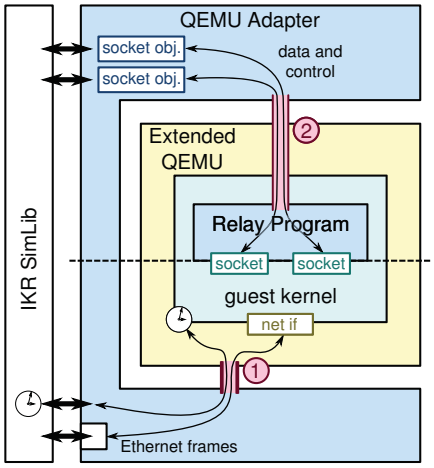

Figure 2: Architecture of the QEMU Adapter.

Input and Output Redirection.

For network simulations, input and output mainly refers to packets sent and received over a network interface. To integrate unmodified applications, it may also include keyboard and mouse events, the graphics output, and serial or parallel connectors. We created special QEMU backends for the most relevant of these devices. These backends forward all output from the VMs to the simulation program, and allow the simulation program to send input to the VMs.

3.1.2 QEMU Adapter

As described, the QEMU Adapter provides an interface to deliver and receive Ethernet frames to and from the simulated network. It also provides a method to send data to a virtual console and a callback mechanism to process any console output. This can be used to log kernel output of the VM (caused by calls to `printk`), which may greatly help when evaluating internal kernel states. (Although not depicted in Figure 2, the upper layer interaction described in Subsection 3.2 is also realized via such a virtual serial interface.) In addition, the Adapter could be extended to provide access to other functions, like requesting screenshots (e. g., to record a video) and typing keys at the virtual keyboard.

In addition, the QEMU Adapter is responsible for the management of the VMs. When the simulation program is started, it spawns a separate QEMU process for each integrated VM. During simulation, it communicates with the QEMU processes through a separate pair of pipes for each instance (not shown in the figures). The control of the VM and the input and output data of the device backends are multiplexed over this connection. When the simulation stops, all QEMU processes are terminated.

3.2 Relaying Operating System Interfaces

When studying functions and protocols implemented in the kernel, it is convenient to allow the simulation program accessing the user space API of the kernel. This concept allows integrating creation of all events, e. g., to open a TCP

connection, as well as all measurement and logging actions in one scenario. In addition, it eases the use of the system for studies of transport protocol behavior by allowing to implement the whole simulation model, including models of network and applications, in one program. This can be used as an alternative to run real applications inside the VMs.

As shown in Figure 2 (interface (2)), the upper layer of the QEMU Adapter makes the user space API of the kernel accessible from the simulation program. To do so, the Adapter interacts with a small relay program, which is running in the user space of the guest OS. Its focus is on relaying socket I/O, e. g., `listen` or `connect`, and configuring the guest kernel, e. g., setting sysctl values.

We implemented our relay program in C to run on top of the Linux kernel. However, it should be portable to other OSs without much effort.

3.2.1 Relay Program

When a VM boots, a custom init script is started, which performs basic configuration of the kernel. After configuration, the init script starts the relay program.

The relay program reacts on socket events and control messages from the QEMU Adapter by calling `epoll_wait` in a loop in a single thread. Currently, only TCP (over IPv4 and IPv6) is implemented, but the interface can be easily extended, e. g., to cover Stream Control Transmission Protocol (SCTP). The supported functions for socket management and socket I/O are listed in Table 1. Some calls are combined to reduce overhead. We currently do not support ioctls and handling of out-of-band data. In addition to the socket interface, the relay program supports reading and writing files, which can be used to configure the kernel via the sysctl interface in the procfs.

To call a function, the simulation program sends messages over a virtual serial interface. The messages transfer all data which is required to perform the function call (a constant identifying the function and the function arguments). Each function call results in a message sent back to the simulation program carrying the return code, the error number, and the function output. The simulation program can distinguish multiple open sockets by their file descriptors.

The relay program also detects events signaled by the kernel using the socket API and notifies the simulation program about these. Such events are:

- a new connection is established to a listening socket
- a socket becomes ready for reading
- a socket becomes ready for writing

To avoid overhead, the reaction to some events is currently hardcoded in the relay program (i. e., a new connection is accepted or the data is read).

As the VM does only execute the relay program, its resource requirements are small. Nevertheless, the VM must be equipped with enough memory to accommodate the configured TCP buffers (which may be tens of megabytes for large bandwidth delay products).

3.2.2 QEMU Adapter

The QEMU Adapter provides an interface to the simulation environment and abstracts the details of the communication with the relay tool. This consists of requests from the simulation program for functions to be called and a callback mechanism which allows the simulation program being notified about events in the VM's userspace, e. g., data becomes

socket(), bind(), and listen()	create a socket, bind it to a local port and start listening for connections
socket() and connect()	create a socket and start to connect to a remote IP and port
accept()	accept a connection request on a listening socket
close()	close a connection / listening socket
read()	read data from a socket
write()	write data to a socket
getsockopt()	read a socket option
setsocktop()	set a socket option
getsockname()	read local ip and port
getpeername()	read remote IP and port

Table 1: Functions supported by the relay tool

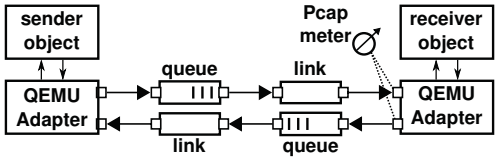

Figure 3: Exemplary simulation model.

available for reading. All functions related to a socket are bundled in a socket object.

When another function calls the Adapter to perform a function call inside the VM, it blocks and the Adapter sends a request message to the relay program. The Adapter gives the VM time to run the virtual CPU until that becomes idle (as described above). During that time, the relay program performs the function call and sends a reply message. When the virtual CPU becomes idle, it is guaranteed that this reply message has arrived at the Adapter. The Adapter uses the reply to construct the return value for the upper layer and returns control to the calling function.

3.3 Embedding the QEMU Adapter into the Simulation Environment

This subsection describes the simulation environment and explains how VMSimInt is integrated into a simulation model.

3.3.1 The Simulation Environment

The used framework for event-driven simulation is the freely available IKR SimLib [11]. It is implemented in Java and provides generic building blocks for arbitrary event-based simulations. The simulation is based on a calendar that advances the time and maintains events to trigger simulation entities. Entities are connected by unidirectional interfaces and exchange information by transmitting messages over these interfaces. The IKR SimLib provides a set of entities for handling and manipulating messages to support network simulation (e. g., queues, traffic generators, link and switch models). Furthermore, it comes with wide support for statistical analysis.

3.3.2 Usage of the QEMU Adapter

Figure 3 shows an exemplary simulation model. A sender transmits to a receiver through a bandwidth-limited link preceded by a queue. The VM and the relay tool are en-

capsulated in the QEMU Adapter and not visible to the simulation program directly.

Sender and receiver objects interact with the socket interface of the QEMU Adapter. They contain abstract models of application behavior. Such a model can, for example, be a generator which creates blocks of data with random size and interarrival time. For each of these blocks, the sender initiates a new TCP connection to the receiver and transmits the data therein.

The QEMU Adapter wraps Ethernet frames sent by the kernel into SimLib messages. It forwards these messages to the simulation model. The assembly of the simulation model defines the further handling and manipulation of each frame (e. g., queues and links in the previous example).

Our framework provides a so-called PcapMeter. This meter allows to capture Ethernet frames coupled with time stamps representing the simulated time. The meter can be attached to arbitrary measurement points in the simulation model. The recorded packet traces can then be analyzed with any tool supporting the pcap format, e. g., wireshark.

3.3.3 *Exemplary Internal Message Flow*

In this subsection, we present the information flow between the components on the sender side in the previously presented scenario. We regard the transmission of one data block from the sender object to the receiver object.

On initiation, all entities are created and connected to define the flow of messages in the simulation. For each QEMU instance a kernel image is started in its VM and configurations are performed (e. g., sysctl values are set). The sender object is parameterized with the receiver's address (IPv4 or IPv6) and port.

At some point in time in the simulation the application model in the sender decides to send a block of data. The flow of messages from now on is shown in Figure 4.

The sender object first creates a socket object as depicted in the QEMU Adapter in Figure 2, passing the configured destination. It registers a callback at the socket object to be notified whenever the socket state changes, i. e., it becomes writable or readable.

The QEMU Adapter translates the socket-and-connect request to a function call message which it sends via QEMU to the relay program. On reception of that message the extended QEMU resumes the VM and forwards the message to the relay program. The relay program creates a non-blocking socket in the VM. It also executes the `connect` system call with appropriate address family, destination and port. The file descriptor of the new socket is sent back to the QEMU Adapter. This file descriptor is used as identifier in all socket related communication between the relay program and QEMU Adapter. After the relay program has completed the processing of the function call message, it calls `epoll_wait` to wait for events on the socket or messages on the control interface.

The `connect` triggers the kernel to set up a TCP connection by sending a SYN message to the destination. This frame is captured by the extended QEMU and forwarded to the simulation framework that handles it according to the setup of the simulated network model.

If everything works fine, after some (simulated) time a response with SYN and ACK set is received by the extended QEMU. Upon reception of this message the kernel switches the state of that connection to `connected` and

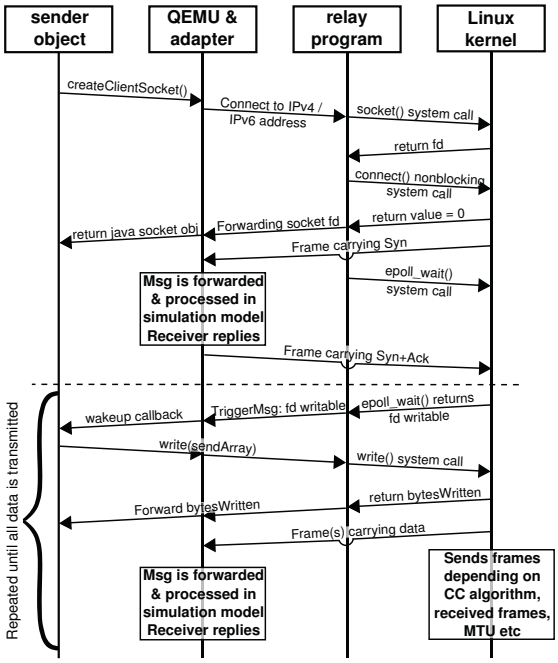

Figure 4: MSC of internal messaging in case of sending data. For simplicity, the QEMU Adapter and the extended QEMU are represented by just one element.

the `epoll_wait` returns signaling that socket being writable now. The relay program forwards this information towards the socket object where the callback is fired.

The sender object then writes data via the QEMU Adapter and the relay program into the socket buffer. For that purpose, the relay program executes `write` to write the data to the socket and signals back the amount of successfully written data. The sender object continues writing until the buffer is full or the message is written completely.

With these system calls, an independent process in the Linux kernel running in the VM is triggered, which will transmit this data according to its algorithms. In case of TCP, these algorithms mostly depend on when the kernel receives frames from the virtual network interface, e. g., acknowledgments from the receiver, and on timeouts. At some time, the kernel is ready to accept more data from the socket, wakes up the relay program and signals the socket being writable again. This process of the relay program writing to the socket and the kernel independently transmitting TCP packets is repeated until there is no more data to send.

4. EVALUATION

The evaluation of VMSimInt is split into three parts. In the first part, we compare our approach to the ns-3 simulator to validate that it is implemented correctly. In the second part, we compare the behavior of different Linux kernel versions. In the last part, we evaluate the runtime performance of VMSimInt.

4.1 Validation against NSC

To validate our implementation, we can either compare it to a real setup or to another simulation framework. Using

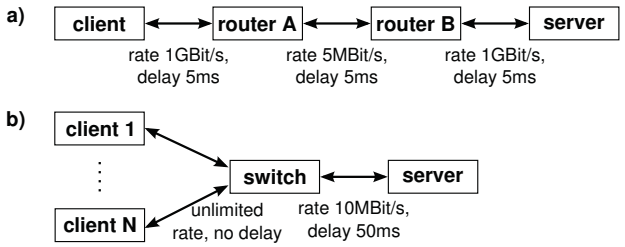

Figure 8: a) Model for the comparison with ns-3 b) Model for the performance evaluation

net.ipv4.tcp_congestion_control	reno
net.ipv4.tcp_no_metrics_save	1
net.core.rmem_max	32768
net.ipv4.tcp_mem (equal values for min, default, max)	10485760
net.ipv4.tcp_rmem (equal values for min, default, max)	10485760
net.ipv4.tcp_wmem (equal values for min, default, max)	10485760
net.ipv4.tcp_moderate_rcvbuf	0

Table 2: Configuration of the Linux kernel used for comparison of our approach with ns-3.

a real lab setup is the soundest approach; however it suffers from inaccuracies (measurement accuracy, influence of OS schedulers, etc.) and would require us to model the processing time in the simulation. As it is not our goal to model these effects, we compare our approach with a simulation setup with NSC in the ns-3 framework. As a third candidate of the comparison, we set up the IKR SimLib to use the NSC. Note that, as we use an unmodified Linux kernel, it is not necessary to validate the behavior of the TCP implementation. Instead, we aim to verify the correctness of the integration of VMs into the simulation environment and the QEMU patches.

We use a simple network model for the comparison, which is configured equally in all simulators. The model is depicted in Figure 8 a). Two hosts are connected via a chain of three bi-directional full duplex links. The middle link is the bottleneck with a data rate of 5 MBit/s, the outer links are configured to 1 GBit/s. Each link introduces a one-way delay of 5 ms, so the resulting Round Trip Time (RTT) is 30 ms. On each side in front of the bottleneck, a drop tail queue with a capacity of 18750 bytes is installed. The Maximum Transmission Unit (MTU) is 1500 bytes. For simplicity of configuration, we used the P2P link model of ns-3. This adds two bytes of header to each IP packet. We configured the IKR SimLib models to resemble the same packet sizes.

Over the chain of links, we transmit data in a single TCP stream from client to server. We start the connection at time 0 and transmit as much data as possible (i. e., the sending rate is not application limited).

To achieve the same behavior of the TCP stacks, we used the Linux kernel version 2.6.26 in all three setups, as that is the latest one supported by the NSC. We configured the kernels in both simulators with the sysctl values listed in Table 2. We configured buffer sizes statically, because by default they are configured dynamically depending on the available memory. We found that enabling moderate_rcvbuf resulted in (virtual-)machine-dependent behavior of the three TCP stacks, so we turned it off. To allow an exact comparison, we switched off the support for dynamic timers in our QEMU Adapter, as those are also not supported by the NSC.

We record the transmitted packets at the sender side in a trace and record the congestion window of the sender using a `getsockopt` call. Figure 5 shows the congestion window over time at the beginning of the TCP transmission. On the left, it can be seen that the congestion window increases exponentially during the slow start phase. After packets have been dropped, the sender decreases the window size and continues in congestion avoidance mode. The lines in the plot match well, meaning that the packets are sent at the same time in all three simulators. Minor deviations exist, however they are not visible in this time scale.

Therefore, we also analyzed the relative TCP sequence number from the recorded trace. Figure 6 shows the sequence number over time at the point where 10 minutes have been simulated. Each point in the plot represents a packet sent by the sender. The spike in the sequence numbers results from a single packet being retransmitted due to an earlier drop. The deviations have added up to a difference of about 240 μs. Closer investigation has shown that the difference increases approximately linearly with the simulated time. As the values of the simulations with NSC + IKR SimLib and VMSimInt are still the same, and only NSC + ns-3 differs, we assume that these deviations result from differences in the simulation model or simulation framework (e. g., the precision of the calendar).

Our validation simulations have shown that VMSimInt provides results which aggree to those of NSC + ns-3 with sufficient accuracy.

4.2 Comparison of Different Kernel Versions

To demonstrate the differences between the TCP implementations in different Linux kernel versions, we performed simulations with VMSimInt and two different kernels. The simulation scenario is the same as described above, except that the queue size is set to 40 kBytes. The compared kernel versions are version 2.6.26 from July 2008 and version 3.9 from April 2013. During that time, multiple modifications have been made to the TCP code. One of those is the increase of the initial congestion window from 3 packets to 10 packets (proposed in 2010 by [5]). To separate this effect from other changes, we patched the older kernel to allow configuring the initial congestion window via a sysctl.

Figure 7 shows the congestion window over time. During the slow start phase, the kernels with an initial congestion window of 10 packets behave identically and start more quickly than the unmodified 2.6.26 kernel. However, after 0.3 s of simulated time, the behavior of those two versions also differs. This is caused by PRR, which is only implemented in the newer kernel and prohibits that the congestion window is more than halved. During the further simulation, the plot shows that the congestion control behaves differently in the latest kernel. Although we did not further investigate the differences of the kernels, it is clear that it is important to use TCP behavior of current kernels for simulations of small-scale effects.

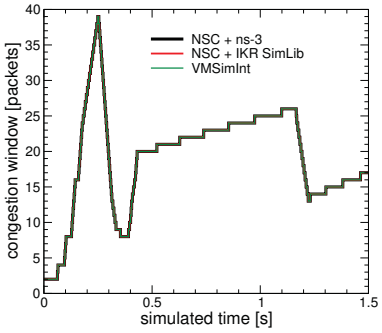

Figure 5: Congestion window over time with three simulators.

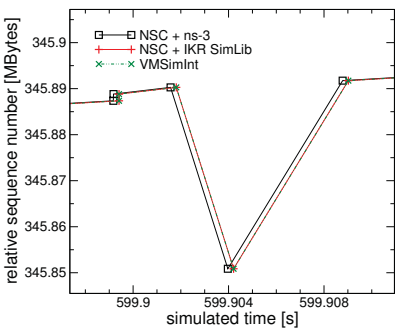

Figure 6: Sequence number over time with three simulators.

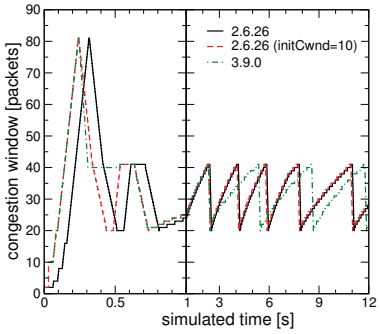

Figure 7: Congestion window over time with three kernels.

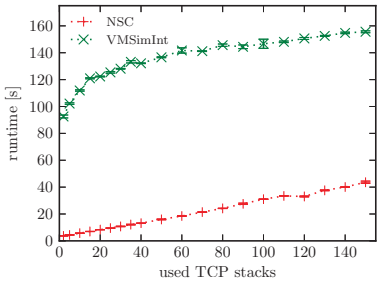

Figure 9: Simulation runtime for greedy traffic

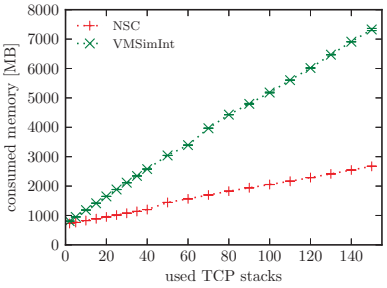

Figure 10: Memory consumption for greedy traffic

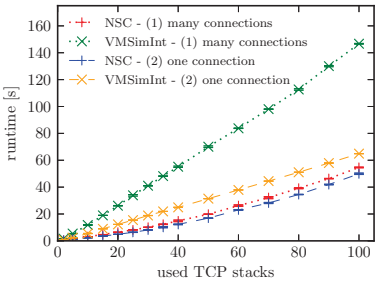

Figure 11: Simulation runtime for light traffic

4.3 Evaluation of Runtime Performance

This section compares the runtime performance of NSC + IKR SimLib and VMSimInt. The ns-3 simulator is not included in this evaluation, because we want to show the differences between the approach of running only adapted extracts of kernel code as a library (as in NSC + IKR SimLib) and running full kernels in VMs in external processes (as in VMSimInt).

The evaluation was done with a simple model (see Figure 8 b)). In this model one TCP stack, which is acting as a server, is connected to a switch. Between server and switch is a link with a rate of 10 MBit/s and a constant one-way delay of 50 ms. On each input of the bidirectional link a queue with a capacity of 125 kB has been installed. The model has a configurable number of TCP stacks acting as clients, which are directly connected to the switch. The links between clients and switch have no delay and unlimited bandwidth. The clients are sending data towards the server. In the case of VMSimInt dynamic ticks are enabled and the VMs are running kernel version 3.10.9. The KVM feature in QEMU is disabled.

We compare runtime and memory consumption of both simulators. The comparison was performed on a machine with an Intel Core i5-2405S CPU and 8 GB of memory, running a recent Linux distribution (Ubuntu 13.04). The plots show the mean values of ten runs as well as the standard deviation in the error bars.

4.3.1 *Greedy Traffic*

First, we consider a scenario where the clients generate as much traffic as possible. The simulation was configured to stop after a simulated time of 180 s.

Figure 9 shows the comparison of the simulation runtime for VMSimInt and NSC approaches. The results show only the time to run the simulation, the time for the setup of the VMs is not considered. (It takes approximately 5 s to start a VM in our environment.)

The performance of VMSimInt is dominated by communication and context switches between the simulation program and the QEMU processes. Thereby, a high cost is introduced for transmitting packets, executing timer ticks and sending control messages to or from the relay program. By using dynamic ticks in VMSimInt, the time spent with processing timer ticks was reduced. The transmissions of packets and control messages (signaling that the receiving socket is writable) have the highest influence on the total runtime.

The non linear increase of the procesing time between 2 and 40 stacks can be explained by an increasing number of ACK packets we observed in this range. As in our scenario all transmissions share a single bottleneck, the number of data packets is (nearly) independent of the number of clients. For higher numbers of stacks, the server sends an ACK packet for each data packet, so the number of generated packets is constant.

Figure 9 shows that the runtime increases linearly for the NSC simulator with increasing number of stacks. This means that the costs for the ticks are dominant, whereas the costs for the exchange of data packets have less influence. We plan to investigate the reason for the high performance difference and further optimize VMSimInt.

The second measurement compares the memory consumption. We used the Resident Set Size (RSS), which can be obtained from the procfs on Linux machines, as an indicator for the memory consumption. The RSS gives the number of

pages occupied by a process, including shared libraries. The result is that the RSS overestimates the memory consumption in most cases.

Figure 10 shows the total consumed memory of NSC and VMSimInt. As can be seen, the consumed memory for VMSimInt increases almost linearly with the number of used TCP stacks. The offset is introduced by the memory needed for the Java based simulation tool. The reason for the linear increase is that every VM needs approximately 45 MB of memory, whereas the Java memory is almost constant. The amount of 45 MB memory for each VM is a result of the minimalistic OS we use in the simulation. It contains only the Linux kernel and our relay program. The memory consumption of the NSC setup increases slower with increasing number of used TCP stacks. For small numbers of stacks the memory requirements of VMSimInt are similar to the NSC simulator. For higher number of stacks the difference is significant. However, it is still possible to run those simulations on current desktop computers. We plan to investigate to which extend KSM can reduce this memory footprint.

4.3.2 Light Traffic

The second performance evaluation is performed with the same network topology, but different traffic. Each client transmits 1000 bytes of data on discrete events, with a negative exponentially distributed Inter Arrival Time (IAT) with a mean value of 5 s. We modeled two variants of transmissions: (1) For each transfer a new TCP connection is created and afterwards closed again. (2) Each client maintains one TCP connection which is opened at the beginning of the simulation. This connection is used for all transfers.

The simulated time was set to 600 s. As the generated traffic per client is constant and not limited by the bottleneck link, the amount of transferred data increases with increasing number of TCP stacks.

As can be seen from Figure 11 for variant (1) the runtime of VMSimInt increases about proportionally with the number of stacks. The same holds for the NSC variant, but the gradient is lower. For variant (2) the difference between VMSimInt and the NSC approach is much smaller, but still the NSC approach offers a faster execution.

From the runtime analysis we can conclude that NSC performs better, especially if the network is occupied. This was expected, as the context switches between the simulation program and the VMs have a high cost. The runtime performance of the NSC is dominated by the constant tick rate. In VMSimInt ticks are only needed when they are scheduled by the guest OS. However, this does not compensate the higher overhead of the interprocess communication.

As a conclusion from the performance evaluation we can remark that although VMSimInt has a lower performance than NSC, it still scales well enough to perform simulations on typical desktop computers.

4.4 Discussion

4.4.1 Maintenance and Extensibility

With our approach it is possible to use arbitrary OSs, in particular the latest versions and even closed source OSs. The extensibility to new kernels is one of the main advantages of our approach. It does not require any modification and so it is achieved without further maintenance costs. This means new concepts regarding protocol design or kernel features can easily be applied in the simulation without modifying the kernel code.

Nevertheless, we introduced changes in the QEMU software. These need to be maintained if a new version of QEMU should be supported to allow taking advantage of improved efficiency in virtualization. However, the used QEMU version does not influence the transport protocol behavior. As long as future kernels will support the virtualized hardware of today's QEMU, we will be able to run simulations with these kernels without any maintenance effort.

4.4.2 Reproducibility

Simulations must be deterministic and reproducible. VMSimInt achieves this by completely decoupling the VMs from the host system. We neglect processing delays by emulating an infinitely fast processor, which performs all processing while the simulated time does not proceed. This makes the simulation result independent of the available processing resources of the host system.

Our framework does not control the random number generator of the VM kernel. However, TCP processing does only depend on the clocks and computation delays which are controlled by the simulation. Other protocol mechanisms, e. g., the stateless address auto-configuration of IPv6, are not fully reproducible with our approach. Nevertheless for simulations, this approach provides a controlled and reproducible environment, allowing for deterministic results and thus allowing an in-depth analysis of protocol mechanisms.

4.4.3 Validation

A model has to be validated against real systems. By using an unmodified OS implementation, we do not need to validate the transport protocol behavior. In our approach, validation is only required for the virtualization tool. Our evaluation results show minor timing differences compared to the NSC + ns-3 simulation approach and no differences compared to the NSC + IKR SimLib setup. Compared to a real system, e. g., in a testbed set-up, we expect larger timing differences as we account zero time for any processing within the VMs and do not model external effects like interrupts.

4.4.4 Scalability

To simulate large-scale networks and perform simulations on different time-scales, computation time and memory usage has to be reasonable. As the comparative results in Section 4 show, the execution of full OSs and the more generic interface to those OSs costs performance. However, we still deem performance and resource consumption acceptable, as it does not hinder in typical research workflows.

4.4.5 Protocol & Application Support

In general our approach offers the same protocol support as provided by the guest kernel. Currently, VMSimInt supports IPv4 and IPv6 for TCP. We can use the kernels interface to configure protocol behavior, e. g., setting sysctl values of the Linux kernel. An extension to support SCTP or User Datagram Protocol (UDP) is straightforward. Furthermore, we can even provide support for real applications. This allows to follow recent application behavior without a detailed analysis of the implementation and modeling of each single application. Our approach allows deploying any real application in the VM, generating input triggers and capturing the resulting behavior on the network.

4.4.6 Network Interface

Our framework supports Ethernet as Layer 2 protocol including address resolution (Address Resolution Protocol (ARP)). Network simulators often only provide a simplified message handling (storing, dropping). However, real host systems do also provide back pressure from the buffer on the local network interface. We lack support for back pressure of the network interface on the TCP stack. Our guest kernels can always send a packet to the network. If the speed of the link connected directly to the sender is the bottleneck, back pressure can influence the behavior of TCP. This is a border case, but it's in focus for our future work.

5. CONCLUSIONS

In this paper we presented VMSimInt, a framework which integrates VMs into a network simulation tool to provide realistic OS behavior. Although our focus lies on providing a realistic TCP implementation, this approach can also be used to integrate arbitrary software into the simulation.

We extended the QEMU software by adapting the clock to the simulated time and redirecting any I/O to the simulation program. With this approach it is possible to guarantee full isolation of the VMs from the host system, which provides exact reproducibility independent of the host system. To simplify the traffic generation we execute a relay program in the user space of the VM and provide respective interfaces in the simulation program to perform socket operations.

The interface to the OSs running in the VMs consists of the hardware emulated by the virtualization tool and the relay program accessing the OS's user space API. This interface is generic, i. e., it can support different operating systems, and it is stable, i. e., it does not have to be changed when the OS is updated.

We have shown that the simulations performed with VMSimInt provide results comparable to the NSC + ns-3 approach, which is widely used and the basis of recent research in the TCP area. We have also shown that computation effort and memory consumption are higher than with existing tools, but are still viable for typial TCP simulations.

Currently we aim to further optimize our approach with respect to computation effort and memory. Moreover, we will extend the simulation by implementing a realistic back pressure behavior. We also plan to support additional OSs (like Microsoft Windows) in the VMs.

All source code of VMSimInt is available. It can be requested by e-Mail.

6. REFERENCES

[1] W. Almesberger. UML Simulator. In *Linux Symposium*, 2003.

[2] F. Bellard. QEMU open source processor emulator. www.qemu.org/, October 2013.

[3] R. Bless and M. Doll. Integration of the FreeBSD TCP/IP-stack into the discrete event simulator OMNET++. In *Proceedings of the 2004 Winter Simulation Conference*, Dec. 2004.

[4] M. Carson and D. Santay. NIST Net: a Linux-based network emulation tool. *ACM SIGCOMM Computer Communication Review*, 33(3), July 2003.

[5] J. Chu, N. Dukkipati, and Y. Cheng. Increasing TCP's Initial Window (draft-hkchu-tcpm-initcwnd-00), February 2010.

[6] A. Grau, S. Maier, K. Herrmann, and K. Rothermel. Time Jails: A Hybrid Approach to Scalable Network Emulation. In *IEEE Workshop on Principles of Advanced and Distributed Simulation (PADS)*, 2008.

[7] N. Handigol, B. Heller, V. Jeyakumar, B. Lantz, and N. McKeown. Reproducible network experiments using container-based emulation. In *Proceedings of the 8th international conference on Emerging networking experiments and technologies (CoNEXT)*, 2012.

[8] J. Heidemann, K. Mills, and S. Kumar. Expanding confidence in network simulations. *IEEE Network: The Magazine of Global Internetworking*, 15(5), Sept. 2001.

[9] S. Hemminger. Network Emulation with NetEm. In *Linux Conference Australia (LCA)*, April 2005.

[10] T. R. Henderson, M. Lacage, and G. F. Riley. Network Simulations with the ns-3 Simulator. Demo paper at ACM SIGCOMM, Aug. 2008.

[11] IKR Simulation and Emulation Library. http://www.ikr.uni-stuttgart.de/INDSimLib/, Oct. 2013.

[12] S. Jansen and A. McGregor. Simulation with real world network stacks. In *Proceedings of the 37th conference on Winter simulation*, 2005.

[13] S. Jansen and A. McGregor. Performance, Validation and Testing with the Network Simulation Cradle. *IEEE 14th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, 2006.

[14] J. Padhye, V. Firoiu, D. F. Towsley, and J. F. Kurose. Modeling TCP Reno performance: a simple model and its empirical validation. *IEEE/ACM Transactions on Networking*, 8(2), Apr. 2000.

[15] L. Rizzo. Dummynet: A Simple Approach to the Evaluation of Network Protocols. *ACM Computer Communication Review*, 27, 1997.

[16] H. Tazaki, F. Urbani, and T. Turletti. DCE Cradle: Simulate Network Protocols with Real Stacks for Better Realism. In *Workshop on ns-3 (WNS3)*, 2013.

[17] A. Vahdat, K. Yocum, K. Walsh, P. Mahadevan, D. Kostić, J. Chase, and D. Becker. Scalability and accuracy in a large-scale network emulator. *ACM SIGOPS Operating Systems Review*, 36(SI), Dec. 2002.

[18] D. X. Wei and P. Cao. NS-2 TCP-Linux: an NS-2 TCP implementation with congestion control algorithms from Linux. In *Proceeding from the 2006 workshop on ns-2: the IP network simulator (WNS2)*, 2006.

[19] E. Weingärtner, F. Schmidt, H. V. Lehn, T. Heer, and K. Wehrle. Slicetime: A platform for scalable and accurate network emulation. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation (NSDI)*, 2011.

[20] E. Weingärtner, H. Vom Lehn, and K. Wehrle. A performance comparison of recent network simulators. In *Proceedings of the 2009 IEEE international conference on Communications*, ICC, 2009.

[21] S. B. Yoginath and K. S. Perumalla. Efficiently scheduling multi-core guest virtual machines on multi-core hosts in network simulation. In *IEEE Workshop on Principles of Advanced and Distributed Simulation (PADS)*, 2011.