

VITO: Virtual Testbed Orchestration for Automation of Networking Experiments

Andreas Stockmayer, Christian Kindermann, Michael Menth

Chair of Communication Networks, University of Tuebingen, Tuebingen, Germany

{andreas.stockmayer,menth}@uni-tuebingen.de,christian.kindermann@student.uni-tuebingen.de

CCS CONCEPTS

• **Networks** → **Network experimentation**;

ACM Reference Format:

Andreas Stockmayer, Christian Kindermann, Michael Menth. 2017. VITO: Virtual Testbed Orchestration for Automation of Networking Experiments. In *ACM SIGMETRICS*, New York, NY, USA, 7 pages. <https://doi.org/10.1145/3150928.3150954>

1 INTRODUCTION

In computer networking research, simulation, emulation, and hardware testbeds are used to implement new protocols and control mechanisms and evaluate their performance. With simulation, network control is easy and it is convenient to perform large experiments. However, a challenge is to correctly model complex protocols like various TCP variants with sufficient accuracy and to validate them. Some simulators allow the integration of real Linux network stacks [15], but this is bound to certain operating system versions that are currently not up to date which is a problem for testing latest protocol enhancements. Highly complex communication technologies may take long simulation times if modelled on a low level. Network emulators interconnect real devices over a simulated network which can be easily configured. An example is NetSim [26]. Network emulators generally can support only low networking speeds so that only limited experiments can be conducted. Highspeed experiments are not possible.

Experimentation on hardware testbeds allows application of original protocol stacks and real protocol implementations. However, hardware testbeds are heavy-weight solutions. If an experiment is large in terms of nodes, it requires lots of physical hardware, administration overhead. Experimentation on hardware is expensive, requires lots of space, energy, i.e., large overhead is involved, and it lacks configuration flexibility. In addition, separate management tools are needed to efficiently leverage different kinds of hardware.

Another solution are networking experiments in a virtualized environment. The nodes of an experiment are modeled by virtual

machines (VMs) which may run any operating system. This facilitates evaluation of latest protocol stacks. Communication links between the VMs may be modelled with appropriate characteristics such as rate, delay, buffer sizes, packet loss, and possibly jitter.

There are a few frameworks for network experimentation in virtualized environments, e.g., Mininet [14]. They allow virtual interconnection of virtual nodes, but they do not provide detailed control over link characteristics, which is essential for performance evaluation.

In this paper we describe a platform for Virtual Testbed Orchestration (VITO) that supports automation of networking experiments on a single server for the purpose of performance evaluation. A major contribution is the automation framework and the configuration of virtual links using the Linux tool TC. We use netem to add delay and a token bucket filter (tb) to model bandwidth and buffer size. We give recommendations for configuration and application examples.

The paper is structured as follows. In Section 2 we give an overview of related work. We describe VITO in Section 3. In Section 4, we experimentally analyze various configuration options for TC and give recommendations for configuration. Section 5 summarizes this work.

2 RELATED WORK

In this section, we briefly review other testbed orchestration tools. There are several commercial testbed orchestrators for both virtual and physical testbeds. We describe Mininet for its prevalence, Spirent Velocity [24] as an example for high-efficiency, the TRIANGLE project [10] as an example for technology-specific experimentation, Virtual Wall [5] as an example for large scale testing, and VMSimInt as an example for fully virtualized machines in a simulated network.

2.1 Mininet

The most widely known testbed environment is Mininet. It was developed for SDN testing, especially in connection with OpenFlow [16]. Mininet emulates multiple nodes and links on a single, local machine. The nodes run in different name spaces, but share the same kernel. Their virtual interfaces are directly connected to a software switch. The Mininet approach is very lightweight with regard to memory and computation overhead. Configuration flexibility is limited because the configuration of the host applies to all virtual nodes. Due to their weak separation, the nodes can influence each other. Mininet is often used for testing of various SDN protocols and for demonstration purposes. This does not require accurate modelling of link behaviors. Network interfaces are all on the same host and transmission are realized as memory copies. Since version

The authors acknowledge the funding by the Deutsche Forschungsgemeinschaft (DFG) under grant ME2727/1-1. The authors alone are responsible for the content of the paper.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

VALUETOOLS 2017, 11th EAI International Conference on Performance Evaluation Methodologies and Tools

© 2017 Copyright held by the owner/author(s).

ACM ISBN 978-1-4503-6346-4/17/12.

<https://doi.org/10.1145/3150928.3150954>

2.0, Mininet also supports bandwidth and delay options. Nevertheless, it does not implement real separate networking stacks so that we cannot rely on Mininet to reveal the same quantitative behaviour as real hardware and software.

2.2 Spirent Velocity

Spirent Velocity is a testbed orchestrator aiming at high utilization of available resources. The concept comes with a virtualization environment and a software manager schedules tests on physical or virtual testbeds. As the experimentation platform is proprietary, validation of experimental results is difficult. The objective of VITO is not high utilization of involved hardware, but separation of virtualized nodes to avoid performance impacts.

2.3 TRIANGLE Project

The TRIANGLE project is funded by the European Commission under the Fire+ initiative and runs over three years from 2016 until 2018. It offers a 5G testbed for app developers, mobile operators, and device makers. The testbed provides special hardware for the emulation of mobile links because it focuses on 5G operation. The resources can be leveraged from remote for experimental purposes. The objective of VITO is to use the resources of a local server for testing protocols on layer 2 and higher.

2.4 Virtual Wall

Virtual Wall is a large scale testing environment consisting of 300 physical nodes which can be used with any operating system in any topology. The link characteristics are modeled in software using special FreeBSD [27] nodes. This approach offers the same flexibility as VITO but at a higher hardware cost since every node is a physical server. Since the links are also modeled in software this approach is not more accurate than VITO.

2.5 VMSimInt

VMSimInt [32] is a tool developed to integrate arbitrary kernels in a discrete event simulator. The approach includes a conventional discrete event simulator to simulate the underlying network and a modified version of QEMU [18] for virtualization of guest machines. The source code of QEMU has to be altered to forward all interrupts from the guest system to the network simulator which then adds them as simulation events. The advantage of this system is the full integration of VMs in existing simulation environments with all benefits of a real system. The drawback is that the simulation speed decreases significantly. All VMs have to be emulated in software since the clock is altered and the VMs only work as simulation events in the simulator. The drawback of this solution is the poor performance of emulated machines which is significantly lower than hardware assisted VMs. The performance of this solution degrades with the number of VMs included, therefore it is only suitable for small testbeds.

3 VIRTUALIZED TESTBED ARCHITECTURE AND EXPERIMENTATION METHODOLOGY

We first give an introduction into VITO’s virtualized testbed architecture and experimentation methodology. We report the virtualization environment and explain how experiments are orchestrated and executed with VITO. We explain some optimizations for faster generation of virtualized testbeds. We describe the integration of physical network interfaces and their virtualization, which may be useful for some experiments. Finally, we show how link characteristics can be modelled using the Linux tool TC.

3.1 Introduction and Overview

Figure 1 gives an overview of the experimentation methodology. Physical nodes like end systems or routers are modelled by VMs and connected via one or more IEEE 802.1d [4] software bridges through virtual interfaces (vIFs). We call those VMs node VMs (NVMs) and the bridges experiment bridges (EBs). Possibly, additional auxiliary VMs (AVMs) may be used in the experiment to act as routers or switches. NVMs and AVMs are jointly denoted as experiment VMs (EVMs). A single managing VM (MVM) is connected to all EVMs in the testbed via a managing bridge (MB). The MVM orchestrates the virtual testbed consisting of EBs and EVMs, controls experiments, records the results and provides them for download, and finally removes the virtual testbed.

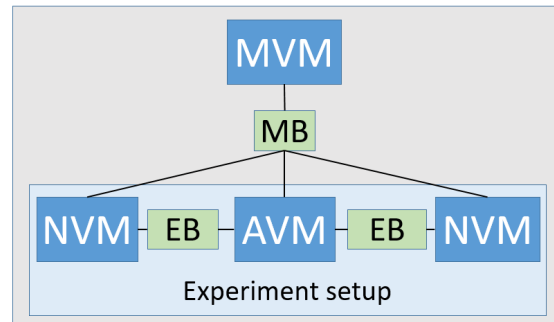


Figure 1: Experiments are executed with node and auxiliary VMs (NVMs, AVMs) that are interconnected by experiment bridges (EBs). NVMs and AVMs are denoted as experiment VMs (EVMs). A managing VM (MVM) is connected via a single managing bridge (MB) to EVMs.

3.2 VITO’s Virtualization Platform

We use a server with an Intel Xeon processor as a physical machine and GNU/Linux as host system. We leverage the Intel VT-x [6] feature to enable hardware-accelerated virtualization of the x86 platform as x86 by itself is not virtualizable. We use Linux Kernel-based Virtual Machine (KVM) [11] as hypervisor which supports VT-x. VMs are created based on QEMU and are managed using the libvirt [20] framework. Software bridges are directly run on the host using the respective 802.1d Linux kernel module. Figure 2 visualizes this concept. In comparison to software emulated machines

like in VMSimInt this enables nearly native performance for every machine [31].

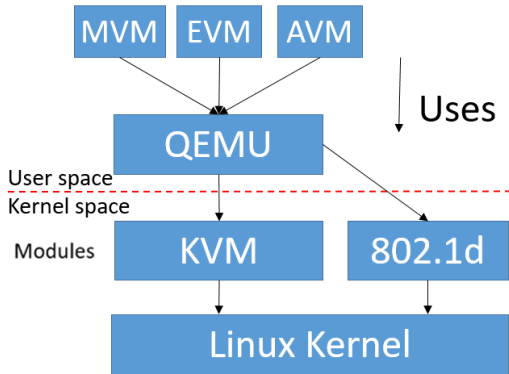


Figure 2: Virtualization experimentation platform.

3.3 Orchestration of Experiments

The experimentation platform itself has only the MVM and the MB running on top of a normal Linux as hypervisor system. The experimenter requires only access to the MVM but not to the host machine itself, i.e., he can create a virtualized testbed and perform experiments without root permissions on the host. Furthermore, he has a single point for the collection of experimental results.

An XML-based file is used for experiment description and control. The file comprises the configuration of all EVMs and EBs, the configuration of the network, and commands that need to be issued at specified times on the VMs. First, the MVM efficiently creates the testbed VMs using libvirt leveraging an XML-based template for VM creation. The CPU pinning option in the VM configuration may be activated to ensure that the VM is run on one CPU thread exclusively. This is helpful for EVMs running CPU-intense computations to ensure they have sufficient resources. Every EVM is equipped with a hard disk that is a snapshot of a template VM which will be described in Section 3.5.

The template foresees a single uplink interface per EVM which is assigned a random MAC address after creation. dnsmasq [22] assigns them IP addresses and makes them reachable via their host names which are configured in the description. As a result, the MVM can connect the EVMs via SSH and execute specified commands. The MVM creates required EBs, further interfaces on the EVMs with randomly generated MAC addresses, assigns them IP addresses, and connects them to the corresponding EBs. udev [12] rules are installed to rename interfaces according to the description. Firewall rules are installed to prevent the usage of the uplink interface for other purposes than communication with the MVM.

For the purpose of a centralized data management, a Network File System (NFS [21]) server is installed on the MVM. This allows all EVMs to write data to a shared directory.

3.4 Execution of Experiments

After creation and configuration of all EVMs and EBs, the experiment is started by executing experimentation commands specified

in the experiment description. These may include comprehensive log operations, e.g., tcpflow [23] may log TCP state variables during experimentation and tcpdump [25] may log network activity on all non-uplink interfaces of EVMs. They need to be started prior to the actual experimentation and application log files may be moved to the NFS mount after completion. The actual experiment consists of a set of commands that are supposed to be executed at specified time instants after experiment start. For instance, a server process may be started on one EVM at the beginning of the experiment and clients are started slightly later on other EVMs. Commands may also change EVM configuration during the experiment, e.g., for modifying link bandwidths or simulating link failures. The virtual testbed is deleted after each experiment to avoid undesired side effects on future experiments. After deletion of the testbed, the MVM still has access to the entire data collection on the NFS mount, compresses the data, and provides it for download as a zipped tar-file.

3.5 Optimized Testbed Generation

Copying the hard disk consumes the major time fraction of EVM generation and deletion. To minimize this overhead, we use qcow2 [19] as an overlay disk image for VMs which is supported by qemu. With qcow2, all blocks initially refer to a read-only base image. If a block is written, a modified copy is stored in a qcow file. During an experimentation, only little data on the disk image is modified, in particular as logs are stored on the NFS mount. Thereby, qcow2 is very efficient and saves lots of copying overhead. As a result, the whole process of creating and deleting a virtual testbed takes less than a minute.

3.6 Integration and Virtualization of Physical NICs

So far, we have only considered the use of virtual interfaces. However, physical network interface cards (NICs) may provide special optimizations like TCP segmentation offloading (TSO) [30]. With VT-d [8], PCI devices can be passed-through from the host to the VM. Therefore, it may be used to pass-through a physical NIC to a VM.

VT-c [7] comprises Virtual Machine Device Queues (VMDq) [9] which enables multiple queues per NIC, i.e., a single physical NIC (physical function, PF) is virtualized into multiple virtual NICs (virtual functions, VFs). VT-d in conjunction with Single Root I/O Virtualization (SR-IOV) [17], the virtual NICs can be passed-through to VMs. To that end, hardware-dependent kernel parameters need to be set to enable all required features. In addition, libvirt has to be configured to use a special pass-through method.

With physical or virtual NICs, a VM's interface is connected to the NIC device instead of the software switch.

3.7 Modelling Characteristics of Transmission Links

The Linux tool TC [13] offers access to netem (networkemulator) [28] to modify the characteristics of a network interface. netem adds constant or variable delay to packets and packets may be randomly dropped or duplicated. TC also provides tbf [29] which implements rate control for egress traffic on an interface. tbf shapes

a traffic stream according to a token bucket while enforcing a maximum latency. The token bucket is configured with a rate and a burst size. That means, `tb` generates tokens at the configured rate, saves them up to its configured burst size, and forwards packets as soon as sufficient tokens are available. Packets that cannot be sent are queued and the latency parameter determines the maximum queue size in time. As an alternative, the queue size can also be configured in bytes. If the number of tokens suffice, several packets can be forwarded at once.

The burst size is usually configured as a small multiple of a maximum transfer unit (MTU). We apply first `netem` and then `tb` to all non-uplink interfaces of EVMs with parameters defined in the experiment description. For further details we refer to Section 4.

3.8 Scalability Considerations and General Applicability

The scalability of VITO depends on the host hardware. A typical server machine has up to 12 cores per CPU on which up to 24 CPU threads are supported. Thus, it suffices to equip one MVM and 23 EVMs with exclusive CPU threads. EVMs with only little CPU load do not need an own CPU thread so that the number of EVMs in the experiment may be even larger.

Any operating system may be used on EVMs if it supports SSH and NFS. Our release of VITO already provides a few Linux VM images for immediate use. However, own VM templates can be created by installing an operating system and provide an NFS mount and the `ssh` key for key login from the manager.

The orchestrator can be started with a list of experiments and can run multiple experiments from the queue at the same time as long as there are enough available unused CPU cores.

4 PERFORMANCE ANALYSIS WITH VITO

In this section, we investigate various uses of `netem` and `tb` for link modeling and quantify appropriate rate-specific burst sizes for `tb` configuration.

4.1 Modeling Link Characteristics

VITO uses the Linux tool TC to model link characteristics. With `netem`, constant or variable delay may be added to individual packets and random packet drops, reordering, or duplication can be realized. With `tb`, a maximum data rate of the link can be enforced by delaying and dropping packets while respecting a configurable buffer size. Also active queue management (AQM) algorithms like random early detection (RED) [3] can be modelled with `tb`. The two tools `netem` and `tb` can be applied both to a single outgoing interface, but the properties of the resulting stream depend the application order.

In the following we use `netem` only for adding constant packet delay so that application order is expected to be irrelevant. We demonstrate plausible results for application order `netem/tb` and show that application order `tb/netem` does not work properly. As a workaround we propose the introduction of an auxiliary node so that `netem/tb` and `tb/netem` can be applied to a traffic stream on consecutive outgoing links which again yields plausible results. Finally, we experimentally show that the addition of an auxiliary node hardly impacts achievable throughput.

4.1.1 Combined Application of `netem/tb` on a Single Link. We investigate the impact of the application order of `netem` and `tb` on a single link. A web client (`curl`) [1] on one NVM communicates via a vIF and a software bridge with a web server (`busybox httpd`) [2] on another NVM with a vIF. The guest OS uses the Linux Kernel 4.8 with TCP Cubic. The client downloads via `http/TCP` a 100 MB file from the web server. TCP’s state variables are logged with `tcpflow` and the packet stream is monitored with `tcpdump` for analysis.

In a first experiment, the following two commands are applied to interface `eth1` to configure it with external parameters that are passed via the `%s` placeholders:

```
tc qdisc add dev eth1 root handle 1:0 \
  netem delay %sms %s corruption %s loss \
  %s reordering %s
tc qdisc add dev eth1 parent 1:1 \
  handle 10: tb rate %sMbit burst %s \
  latency %sms
```

The `netem` command effects that the packet stream is modified with delay (ms), jitter (%), corruption (%), loss (%), and reordering (%). In our experiments, only the delay is set to 100 ms and all other values are zero. The `tb` command effects that the packet stream is spaced according to a token bucket with configured rate (Mb/s), burst size (bytes), and latency (ms). This roughly models a transmission link with the specified rate and a buffer size of $rate \cdot latency / 1000$. In the following, we set $rate = 50$ Mb/s, $burst = 4542$ bytes, and $latency = 50$ ms. In a second experiment, the application order of `netem` and `tb` is interchanged. We apply the same configuration for transmission from the client to the server and vice-versa.

Table 1: Performance metrics for a download of a 100 MB file with `netem/tb` and `tb/netem` configured on a single link with bandwidth of 50 Mb/s.

tool order	download time	goodput (Mb/s)	avg. CWND (MSS)	avg. RTT (ms)
<code>netem/tb</code>	19.7s	40.6	821	231
<code>tb/netem</code>	19.3s	41.5	2013	427

Table 1 summarizes the results. While download time and goodput are rather similar for both application orders, packet loss, avg. TCP congestion window size (CWND), and avg. roundtrip time (RTT) differ significantly. For further analysis, we consider CWND and RTT over time which is illustrated in Figure 3. With `netem/tb`, the RTT varies between 200 ms and 250 ms, and the CWND between 500 and 1050. In particular, CWND and RTT decrease after the occurrence of a lost packet. This is different with `tb/netem` as no packet is lost. As a result, the CWND increases up to a maximum value of 2300 MSS and the RTT oscillates between 420 and 520 ms. RTTs in this order of magnitude are unexpected because the configured transmission and queueing delays can range only between 200 ms and 300 ms. Thus, the application order `tb/netem` causes undesired behavior and should be avoided for experimentation.

4.1.2 Application of `netem/tb` on Consecutive Links. We now interconnect the client and the server NVM via another AVM and software bridge. The configuration is illustrated in Figure 4 for

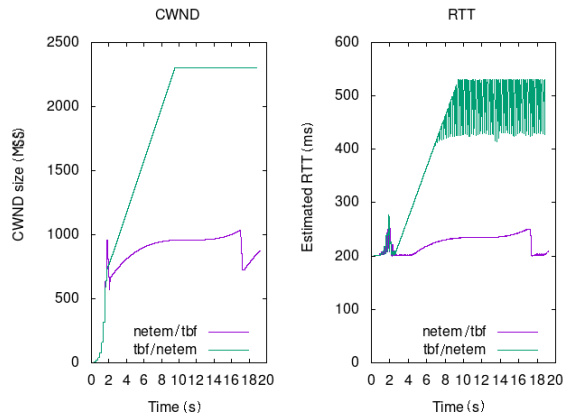


Figure 3: TCP’s congestion window and roundtrip time for a single TCP flow on a link with 100 ms delay, a bandwidth of 50 Mb/s, and a buffer with a maximum latency of 50 ms; tbf’s burst size is configured with 4500 bytes.

the application order tbf/netem. Thus, netem/tbf or tbf/netem are applied to consecutive links instead to a single one. The client and server NVM see only single links with combined properties.

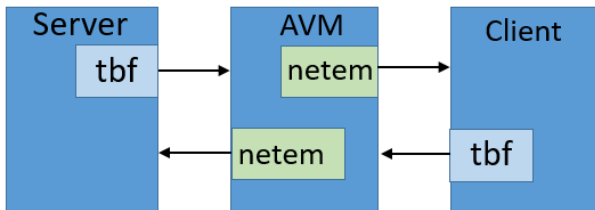


Figure 4: tbf/netem is configured individually on two consecutive links through introduction of an additional EB and AVM that just forwards traffic.

Table 2 reports the experimentation results which hardly differ from the experiment with netem/tbf. The same holds for CWND and RTT over time for which figures are omitted. Thus, netem/tbf may be applied together on a single outgoing interface. If tbf/netem is the desired application order, an auxiliary node may be used to ensure correct behavior.

Table 2: Performance metrics for a download of a 100 MB file with netem/tbf and tbf/netem individually configured on two consecutive links.

application order	download time	goodput (Mb/s)	avg. CWND (MSS)	avg. RTT (ms)
netem/tbf	19.1s	41.8	834	224
tbf/netem	18.9s	42.3	845	229

4.1.3 Impact of Auxiliary Nodes for Delay Addition. An additional AVM may add some delay. We show that this delay is so small that it hardly influences experimentation results. To that end, we perform similar experiments like above but deactivate rate control and add a delay of 0 ms and 1 ms, respectively. We perform the experiments 20 times. Table 3 summarizes the utilization. Even without any base delay, the presence of the AVM is hardly visible by the increased download time and with a base delay of 1 ms, the download time is almost the same. Thus, AVMs add so little delay that it cannot even be perceived for small positive base delay.

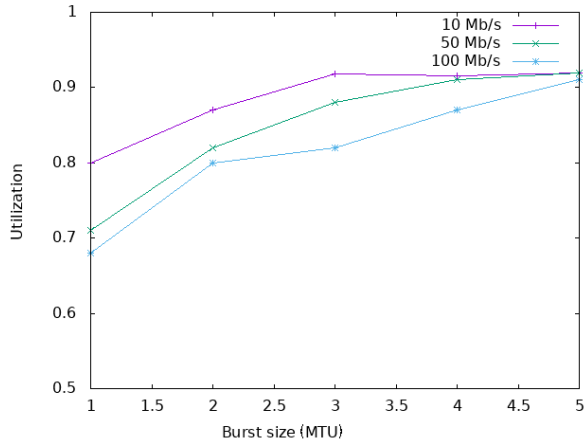
Table 3: Download time for various configurations depending on one-way delay with a configured bandwidth of 100 Mb/s and a 100 MB file size.

delay	w/o AVM	w/ single AVM	w/ jLISP
0 ms	8.68s	8.70s	9.21s
1 ms	8.72s	8.73s	9.22s

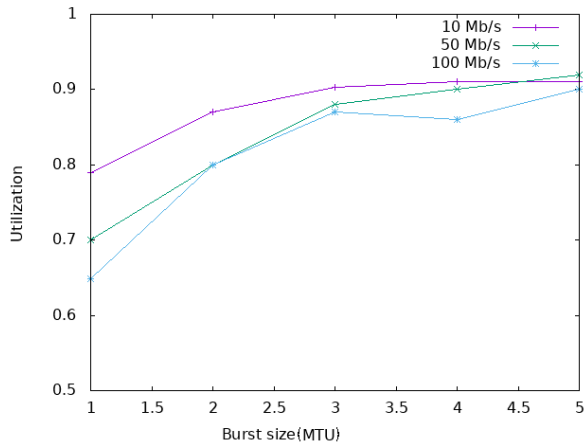
4.2 Configuration of Rate-Specific Burst Sizes

We first think of a simple rate controller that ensures a maximum bit rate of C . After transmission of a packet with size B , it transmits the next packet not earlier than after $\frac{B}{C}$ time. If the machine performs that task only slightly late, this reduces the maximum achievable data rate. To cope with the problem of late transmission, tbf uses a token bucket description for spacing. tbf continuously generates tokens and saves them in a bucket which is limited by its burst size. Packets are queued for transmission. A packet is sent if the number of tokens in the bucket is at least the packet size. If the number of tokens does not suffice, the machine retries again when enough additional tokens have arrived. On the one hand, this mechanism assures that transmission capacity is not lost if a packet is sent slightly later than possible, on the other hand it allows transmission of multiple packets so that packet bursts may be transmitted. Therefore, the burst size should be set only so large that the full transmission capacity can be leveraged, but also as little as possible to keep bursts small as the intention of a spacer is a smooth traffic stream.

In the following we evaluate the required burst size for various bandwidths. Figure 5(a) compiles the utilization for various burst sizes for a server NVM with little CPU load. The figure shows that large bandwidths require large burst sizes to minimize download times by fully leveraging the configured bandwidth. We perform the same experiment with a server NVM that performs other tasks in parallel so that its CPU load is close to 100%. The results in Figure 5(b) show that almost the same download time values are achieved because TC is granted high priority. Both experiments were repeated 100 times which resulted in a confidence interval of less than one percent in each direction for an alpha value of 0.95. We derive from Figure 5(b) recommendations about minimum burst sizes for specific values of configured bandwidths that are needed so that the full transmission speed can be achieved with tbf. We summarize these values in Table 4. The values may depend on software and hardware. Especially with different Linux kernel



(a) Idle server with an avg. CPU load of 2%.



(b) Busy server with an avg. CPU load of 98%.

Figure 5: Impact of configured burst sizes on utilization depending on configured bandwidths.

versions a different behavior could occur which leads to adaptations for the VITO systems since both the kernel and iproute2 are under active development.

Table 4: Recommended burst size for various values of configured bandwidth.

bandwidth (Mb/s)	burst size (MTU)
≤ 10	3
≤ 50	4
≤ 100	5
> 100	7

5 CONCLUSION

In this paper we proposed the Virtual Testbed Orchestration (VITO) platform which allows automation of real-world networking experiments on a single server. Physical or virtual hardware components like additional hosts or NICs may be integrated for even more realistic experiments. Networking nodes are modelled by virtual machines (VMs), leveraging hardware virtualization technologies, that are interconnected by software bridges and link characteristics are enforced by the Linux tool TC. The supportable experiment size in terms of networking nodes depends on the server machine, a lower bound is one networking node per CPU thread. VITO node VMs can accommodate any guest operating systems with SSH and NFS support. An experimental performance analysis showed how netem and tbf needs to be configured for modelling link characteristics demonstrated that the overhead of auxiliary VMs tends toward zeros, and gave recommendations for the configuration of burst sizes.

REFERENCES

- [1] Daniel Stenberg . 2017. cURL. <https://curl.haxx.se/>. (2017).
- [2] Denys Vlasenko et al. 2017. BusyBox. <https://busybox.net/>. (2017).
- [3] Sally Floyd and Van Jacobson. 1993. Random Early Detection Gateways for Congestion Avoidance. *IEEE/ACM Trans. Netw.* (Aug. 1993).
- [4] IEEE Computer Society. 2004. IEEE 802.1D Standard. <http://standards.ieee.org/getieee802/download/802.1D-1998.pdf>. (2004).
- [5] iMinds. 2014. Virtual Wall – Generic Test Environment For Advanced Network, Distributed Software and Service Evaluation, and Scalability Research. (2014). <http://www.iminds.be/en/develop-test/ilab-t/virtual-wall>
- [6] Intel Corp. 2006. Intel Virtualization Technology (VT-x). (2006).
- [7] Intel Corp. 2012. Intel Virtualization Technology for Connectivity (VT-c). (2012).
- [8] Intel Corp. 2012. Intel Virtualization Technology for Directed I/O (VT-d) Architecture Specification. (2012).
- [9] Intel LAN Access Division. 2008. *Intel VMDq Technology*. Whitepaper. INTEL.
- [10] Keysight Technologies. 2017. Triangle Project. <http://www.triangle-project.eu/project/>. (2017).
- [11] Avi Kivity et al. 2007. KVM: The Linux Virtual Machine Monitor. In *Ottawa Linux Symposium*.
- [12] Greg Kroah-Hartman. 2003. udev – A Userspace Implementation of devfs. In *Ottawa Linux Symposium*.
- [13] Alexey Kuznetsov and Stephen Hemminger. 2012. iproute2: Utilities for Controlling TCP/IP Networking and Traffic. (2012).
- [14] Mininet Project. 2017. Mininet. <http://mininet.org/>. (2017).
- [15] ns-3 project. 2015. ns3 Direct Code Execution. <https://www.nsnam.org/overview/projects/direct-code-execution/>. (2015).
- [16] Open Networking Foundation members. 2016. OpenFlow Switch Specification. (2016).
- [17] PCI SIG. 2010. Single Root I/O Virtualization and Sharing Specification 1.1. (2010).
- [18] QEMU team. 2014. QEMU 2. <http://wiki.qemu.org/ChangeLog/2.0>. (2014).
- [19] QEMU Team. 2017. qcow2. <http://git.qemu-project.org>. (2017).
- [20] Red Hat. 2012. libvirt: The Virtualization API. <http://libvirt.org>. (2012).
- [21] S. Shepler et al. 2003. Network File System (NFS) version 4 Protocol. RFC 3530. (April 2003).
- [22] Simon Kelley. 2017. dnsmasq. <http://www.thekelleys.org.uk/dnsmasq/doc.html>. (2017).
- [23] Simson L. Garfinkel. 2017. tcpflow. <https://github.com/simsong/tcpflow>. (2017).
- [24] Spirent. 2017. Spirent Velocity. <https://www.spirent.com/Products/velocity>. (2017).
- [25] Tcpdump Team. 2017. Tcpdump. <http://tcpdump.org>. (2017).
- [26] TETCOS. 2017. NetSim. <http://tetcos.com/>. (2017).
- [27] The FreeBSD Project. 2017. FreeBSD. www.freebsd.org. (2017).
- [28] The Linux foundation. 2017. netem. <https://wiki.linuxfoundation.org/networking/netem>. (2017).
- [29] The Linux foundation. 2017. tbf. <https://linux.die.net/man/8/tc-tbf>. (2017).
- [30] The Linux foundation. 2017. tso. <https://wiki.linuxfoundation.org/networking/gso>. (2017).
- [31] VMware. 2009. Performance Evaluation of Intel EPT Hardware Assist. <https://www.vmware.com/pdf/PerfESXintel-EPT-eval.pdf>. (2009).
- [32] Thomas Werthmann et al. 2014. VMSimInt: A Network Simulation Tool Supporting Integration of Arbitrary Kernels and Applications (*SIMUTools '14*).