Link Multiplexing in a Xen-based Network Emulation System

Pasquale Di Gennaro, Roberto Bifulco, and Roberto Canonico

University of Napoli Federico II, Italy
Dipartimento di Informatica e Sistemistica
Via Claudio, 21 - 80125 - Napoli, Italy
{pasquale.digennaro,roberto.bifulco,roberto.canonico}@unina.it

Abstract. Network emulation has gained wide interest in the community of network researchers to evaluate the effectiveness of new protocols and applications in controllable and realistic network scenarios. To ensure scalability, modern emulation systems rely on the use of virtualization techniques to create complex networked systems by means of the computational resources available in a cluster of computers. In the context of network emulation systems, *link multiplexing* is the problem of emulating multiple point-to-point connections on top of a single Ethernet link. In this paper we present how link multiplexing is implemented in NEPTUNE, a Xen-based network emulation system developed at University of Napoli Federico II. We compare our technique with those adopted in other network emulation systems. We also present experimental results aimed at investigating the performance limits of our system and at providing researchers with useful insights into the faithfulness of emulated scenarios.

Key words: Network Emulation, Xen, Performance Evaluation.

1 Introduction

Network emulation is an experimental methodology that is widely adopted to test innovative protocols and distributed applications in realistic and controllable scenarios. Unlike simulation, which reproduces a system's behavior by modeling all the interacting components of the system, emulation allows researchers to test a *real* implementation of a system component, by making it interact in real-time with other real world or modeled components of the system [1]. In the specific case of networked systems, emulation consists in reproducing a "virtual" network setup on top of a collection of physical devices. In particular, one of the issues of network emulation is the ability of reproducing the behavior of different communication links (such as geographic point-to-point links, shared LANs, wireless LANs, and so on) on top of a general purpose facility. One of the first emulation tools was *dummynet* [2], a software system developed by Luigi Rizzo as an extension of the FreeBSD kernel. Dummynet makes a FreeBSD system able to shape and delay the traffic flowing through it. Due to the ease of deployment, dummynet is often used in small scale testbeds to emulate the behavior of

congested wide area networks for testing of protocols and applications. A modified version of dummynet has also been recently deployed in PlanetLab Europe [3], where FreeBSD "Dummynet Boxes" have been deployed in front of a subset of PlanetLab Europe nodes. Dummynet Box can be dynamically configured, so that individual users (slivers) can independently and concurrently set up the characteristics of an emulated link for their experiments [4]. Modern network emulation systems are able to reproduce in a virtual environment the behavior of complex network topologies, and let these virtual networks interact with real applications under test. The architectures of these emulation systems are extremely different, ranging from *centralized implementations*, reproducing the emulated network within a single computer, to *distributed emulation facilities*, usually relying on clusters of PCs interconnected by programmable networking devices. We call this latter kind of systems "cluster-based network emulation systems". Since realistic evaluation scenarios often require thousands of nodes, scalability is a key requirement for network emulation systems. Different solutions have been proposed in the literature to scale-up the maximum size of emulated networks. Grau et. al classify them into parallelization, abstraction, node virtualization, and time virtualization [5]. Node virtualization has been used in several cluster-based network emulation systems, such as University of Utah's EmuLab [6] and University of Stuttgart's NET [7] [8]. NEPTUNE is a cluster based network emulation system developed at University of Napoli Federico II that makes use of Xen for node virtualization [9],[10]. Besides node multiplexing, a network emulation system needs proper techniques to emulate the behavior of different point-to-point links on top of a shared networking infrastructure. This latter problem is usually referred to as *link multiplexing*. In this paper we present and experimentally evaluate the "One Link per Virtual Interface" technique (OLVI in short) we use in NEPTUNE as the basis for link multiplexing. OLVI combines Xen bridging with the emulation features provided by the NetEm extension of the Linux kernel [11] to multiplex several point-to-point communication links, each of which with its own bandwidth and delay, on top of a single high-performance LAN. The rest of the paper is organized as follows. Section II discusses the resources multiplexing problem and how virtualization has been used in some reference emulation systems. In Section III we present NEPTUNE, and, in particular, the OLVI technique. In Section IV we present the results of a performance evaluation study aimed at investigating the limits of OLVI. Finally, in Section V we compare our results with what has been presented in other research works aimed at evaluating the performance of Xen in virtualized network infrastructures and draw our conclusions.

2 Virtualization in Network Emulation Systems

A typical network emulation system is composed of a set of physical resources (links, LAN switches, PCs) that are used to reproduce an emulated networking environment. Several strategies can be adopted to map the emulated scenarios on top of the available physical resources. While conservative allocation policies

may easily lead to underutilization, better resource utilization might be achieved if they could be decomposed in many "virtual resources", each appearing as a separate physical resource. Such a technique is usually referred to as *resource multiplexing*. In the context of network emulation we are interested in two particular forms of resource multiplexing: *node multiplexing* and *link multiplexing* [12]. Node multiplexing is the problem of emulating more than a network node on the same physical node, while link multiplexing focuses on emulating multiple point-to-point connections on top of one or more shared links.

Virtualization technologies are a widely used solution for *resource multiplexing* problems. In general terms, virtualization is a technique in which a software layer multiplexes lower-level resources for the benefit of higher level software programs and systems. Virtualization can be applied to either single physical resources of a computing system (e.g. a single device) or to a complete computing system. When applied in this latter sense, (a.k.a. *Platform Virtualization*), it allows the coexistence of multiple "Virtual Machines" in the same computing host. Platform virtualization is implemented by means of an additional software layer, called Virtual Machine Monitor (VMM) (or *hypervisor*), that acts as an intermediary between the system hardware resources and the Operating System. There are many approaches to platform virtualization: *Full Virtualization* implements in software a full virtual replica of the emulated system's hardware, so that the operating system and user applications may run on the virtual hardware exactly as they would in the original system. *Paravirtualization*, instead, makes available a software interface to virtual machines that is similar but not identical to the underlying hardware in order to improve scalability and performance over full virtualization, at the cost of requiring the guest operating system to be explicitly ported for the para-API. Finally, *Operating system-level virtualization* further improves scalability allowing a physical server to run multiple isolated operating system instances sharing the same kernel with little overhead, but at the cost of a reduced flexibility.

While node multiplexing is inherently a problem of platform virtualization, link multiplexing is a more specific problem that can be solved in different ways, at different layers of the communications stack. Several network emulation systems have been designed (or re-designed) in order to use virtualization techniques for efficient resource multiplexing. University of Utah's Emulab, as first example, implements node multiplexing by means of a modified version of FreeBSD Jail. ModelNet [13] implements node multiplexing by means of so called Virtual Nodes (VNs) which is just a process level isolation approach, while link multiplexing is implemented by combining IP aliasing, a socket interposition library and centralized Core Nodes running dummynet. University of Stuttgart's NET implements link multiplexing by combining the use of VLANs and a virtual device driver, NETShaper, which allows to dynamically configure bandwidth, delay and loss rate [14]. Besides network emulation systems, virtualization techniques have also been used for resource multiplexing in large scale distributed testbeds. The VINI project has created a virtual network infrastructure allowing experimental evaluation of protocols and services under real traffic loads, in con-

trollable network conditions [15]. VINI uses two container based virtualization technologies for node multiplexing, VServer and NetNS, in addition to Ethernet EGRE tunneling [16] for layer 2 encapsulation.

The use of virtualization techniques is also at the basis of NEPTUNE, a cluster-based network emulation system developed at University of Napoli Federico II. NEPTUNE relies on Xen [17] for node multiplexing, which implements paravirtualization by means of an hypervisor and several domains, running on top of that hypervisor. The hypervisor controls guest domains access to the physical machine's hardware resources, while for the sake of reliability and efficiency, device drivers are kept in an isolated "driver domain" (Domain 0, or dom0) with special privileges. Domain0 is created at boot time and, through it, users may create and terminate other unprivileged domains (domUs), control CPU scheduling parameters and resource allocation policies.

3 NEPTUNE

NEPTUNE is an open-source cluster-based network emulation system developed at University of Napoli "Federico II" that can be used to assess new networking technologies and protocols (e.g. new QoS routing protocols and Traffic Engineering schemes in MPLS-based networks), as well as new distributed applications and architectures (e.g. multimedia peer-to-peer applications). NEPTUNE provides researchers with the ability of interactively designing multiple virtual network topologies, which are then deployed onto a cluster of real machines and used as if they were dedicated physical testbeds. NEPTUNE was designed with two goals in mind: manageability and portability. Manageability, because we wanted that NEPTUNE could have been easily deployed and managed by system administrators. Portability, since NEPTUNE is not linked to specific hardware solutions, but it can be installed on general purpose machines and its features can be conveniently extended by software developers. In NEPTUNE, an experiment is a collection of virtual nodes deployed on a subset of a cluster's physical nodes, each running a virtualization layer, and properly configured in order to reproduce a user-defined virtual network topology. To achieve higher degrees of scalability, complex systems are reproduced by allocating multiple virtual network nodes onto each of the cluster's real nodes (*node multiplexing*). Likewise, multiple virtual links are multiplexed onto the same shared physical link by associating each virtual link endpoint to a different virtual NIC (*link multiplexing*). Multiple fully isolated experiments can be run by NEPTUNE at the the same time, while providing users with the illusion of having allocated a dedicated infrastructure (virtual cluster). A role-based authentication system allows flexible definition of roles and actions allowed to each role. Roles and permissions are stored in XML files for simple editing, to allow system administrators modify policies even at run-time.

3.1 NEPTUNE Architecture

NEPTUNE's physical architecture (Figure 1) is composed of three parts: i) a set of worker nodes providing computational resources used to reproduce emulated networks, ii) a centralized repository providing storage space to worker nodes and iii) a front-end node, NeptuneManager, hosting system management software. All these physical components are connected by a switched LAN, carrying what we call "control traffic". Worker nodes are also connected by a second high-performance LAN, carrying traffic generated by users' experiments. All users (both normal users and administrators) access the system through the NeptuneManager web interface. All system functions are exposed by this interface, so users can set up and execute their emulation experiments by means of a user-friendly AJAX web-interface.

3.2 Usage Model

An experiment life-cycle begins with the definition of a virtual network topology. Once the topology is defined, an experiment can be allocated onto the cluster's physical nodes. A running experiment can be either suspended for future reallocation or definitively terminated. Allocation of experiments onto the cluster is made under control of system administrators, who need to explicitly accept users requests. Once accepted, experiment's topology allocation process starts. Such allocation process is automatic, involving tasks like virtual nodes mapping on cluster's physical nodes and IP addresses assignments.

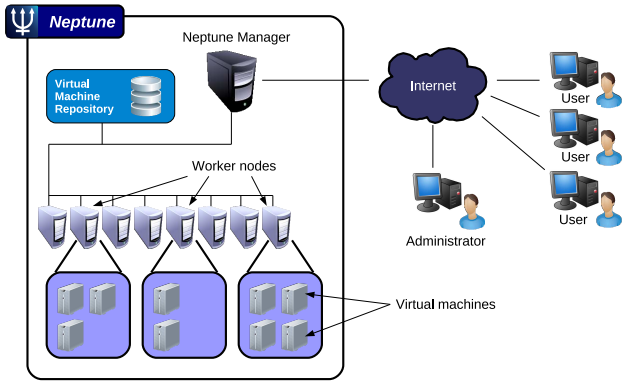

Fig. 1. NEPTUNE architecture.

To define virtual topologies, users can both write a topology description in a custom XML format or use an interactive graphic tool embedded into the web user interface. It is also possible to select pre-defined topologies for fast experiment definition, modify and in case save them as new topology templates.

To define virtual nodes software configuration, users can access via the Neptune web interface a "Virtual Nodes Template Images Repository" and select a VM template for each of the emulated nodes. VM templates, which enclose OS filesystem and in case other software, can be modified and saved as new templates for reuse.

Furthermore, users can control an experiment status and can execute actions to terminate that experiment or save it. Our current implementation of experiment status saving only creates copies of virtual nodes file systems, as saving the whole status of a running experiment is a distributed snapshot problem [18] which is actually out of the scope of our system. Commands and tools to manage and monitor virtual nodes and links are provided too. Finally, remote access is made available to each of the experiment nodes through a VPN tunnel.

3.3 Node multiplexing in NEPTUNE

Node multiplexing is implemented in NEPTUNE by means of Xen, using virtual machines as network nodes. Our current implementation relies on libvirt virtualization APIs [19], making it feasible supporting different virtualization technologies in the future. Mapping of virtual nodes onto the cluster physical nodes is described by an allocation map which can be generated either manually by a system administrator or automatically, by means of a software module implementing a Lin-Kernighan derived optimization algorithm [20]. When a virtual network is to be deployed on the physical cluster, NeptuneManager distributes Virtual Machine template instances to the physical cluster nodes. This distribution process is composed of two phases for each virtual node: 1) raw copy of the virtual machine image file containing VM template, and 2) VM creation on the target virtual machine monitor. During this last phase, virtual hardware resources are provided to the virtual node according to node definition provided by the experiment topology description. Use of Xen for node multiplexing provides a totally virtualized environment to test applications. Xen isolates virtual machines from each other and guarantees them the availability of resources assigned at VM creation time. Because of the total isolation between VMs, it is possibile to run custom operating systems on each VM and, hence, custom network stacks. This allows us to correctly emulate different network devices, e.g. routers, within a single physical host.

3.4 Link Multiplexing in NEPTUNE

In several network emulation systems, link multiplexing is performed by means of Virtual LANs (VLANs). Such a solution is implemented by properly configuring the Ethernet switches and does not require any configuration and processing in the cluster nodes. This makes, however, the system configuration software extremely dependent on the characteristics of the network switches. For the above reason, we decided not to use VLANs in NEPTUNE and we adopted a network device independent solution for link multiplexing that we call "One Link per

Virtual Interface" (OLVI in short). The OLVI technique is implemented by exploiting the network virtualization mechanisms implemented in Xen. Every time a new virtual machine is instantiated, Xen creates a new pair of "connected virtual ethernet interfaces", with one end of each pair within the virtual machine and the other end within the virtual machine monitor. Virtualised network interfaces have their own ethernet MAC addresses, whose values can be assigned at virtual machine creation time [21][22]. When using OLVI technique, each point-to-point link is identified by its end points, which are virtual NICs in virtual nodes. Since a unique MAC address is assigned to each virtual NIC, virtual links are uniquely identifiable within NEPTUNE. To completely implement link emulation, virtual links need to be configured according to specific user-defined properties. Such properties are assigned to emulated links through the use of a queuing discipline and a traffic shaper that are associated to both ends of an emulated link. Queuing discipline are enforced through the traffic control module of the Linux kernel, while traffic shaping is done through NetEm, another Linux kernel module, provided by default from kernel2.6 distributions, that gives the possibility to emulate delay, loss rate, re-ordering and duplication over a link.

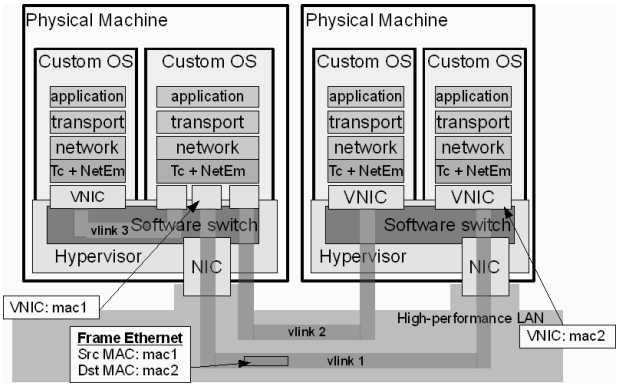

Fig. 2. Link multiplexing in NEPTUNE.

A major problem when dealing with the creation of virtual links is the need to assign IP addresses to both ends of virtual links, according to a general IP addressing scheme. In NEPTUNE users can manually define IP addresses for a link's end-point, but such a task is tedious, error prone and not viable when dealing with big topologies. For these reasons NEPTUNE also provides an algorithm that automatically assigns subnets to links and IP addresses to their end-points. Furthermore, since several experiments can be running on the same shared infrastructure, the algorithm also ensures non overlapping of address spaces used by different experiments. This latter requirement is also enforced when experiments use manual allocation of IP addresses.

4 Experimental Evaluation

The need to create repeatable experimental scenarios to test network applications and protocols is the main reason for the adoption of simulation and emulation techniques. Node and link virtualization in network emulation testbeds provide users with the possibility of trading off the amount of physical resources to be allocated to a given experiment for emulation accuracy. First target of our experimental evaluation is to identify limits of proposed multiplexing techniques and in particular the maximum network throughput a NEPTUNE's virtual node can achieve. Here we discuss experimental results demonstrating the maximum throughput obtainable when one VM is running on top of the hypervisor: these performance values are an upper bound for the case of multiple concurrent VMs running on top of the same physical hardware. To assess performance levels of NEPTUNE, first of all we need to find a reference performance value. To this end, we set up a preliminary experimental scenario (Setup #0), in which two identical machines are connected by a point-to-point 1Gbps Ethernet cable. Both nodes are HP ProLiant DL380 servers, each equipped with two Intel Pentium IV Xeon 2.8 GHz CPUs, 5 GB of PC-2100 RAM, one 100 Mbps Ethernet NIC and one Gigabit Ethernet NIC. The adopted CPUs support the Hyper-Threading Intel technology. Both hosts run native GNU/Linux (CentOS5.3) with a 2.6.18 Linux kernel.

In such a scenario, we run the D-ITG suite [23] to generate network traffic and measure effective throughput in terms of generated/received packets per second (pkt/s). We used D-ITG to generate UDP constant bit rate traffic, at different rates, with packets size of 1042 bytes "on wire".

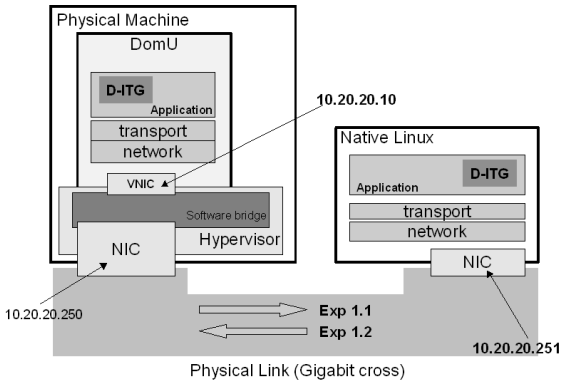

Fig. 3. Experimental setup #1.

Similar experiments were repeated among the same pair of physical machines, but in a different experimental scenario (Setup #1 shown in Figure 3): one of the two hosts runs a Xen Virtual Machine Monitor (Xen version 3.1.2) with a 2.6.18

Linux kernel plus a single domU virtual machine. Two different experiments were run in this configuration: Experiment 1.1, where domU node acted as CBR traffic generator, and Experiment 1.2, where domU acted as receiver. Due to the use of Hyper-Threading, our systems appear having four "logical" CPUs, numbered as CPU0, CPU1, CPU2 anc CPU3. The CPU enumeration order used by Xen is: hyperthreads, cores, sockets. On our system, the four CPUs are then mapped as follows:

- cpu 0 : socket 0, [core 0], hyperthread 0;
- cpu 1 : socket 0, [core 0], hyperthread 1;
- cpu 2 : socket 1, [core 0], hyperthread 0;
- cpu 3 : socket 1, [core 0], hyperthread 1.

We configured Xen by constraining domUs to use no more than one CPU at a time.

Experiment run in Setup #0 provides us with the maximum number of packets per second that our Linux-based hosts are able to receive. Since experiments run in Setup #1 show a lower throughput, we are confident that this performance penalty is caused by the use of Xen.

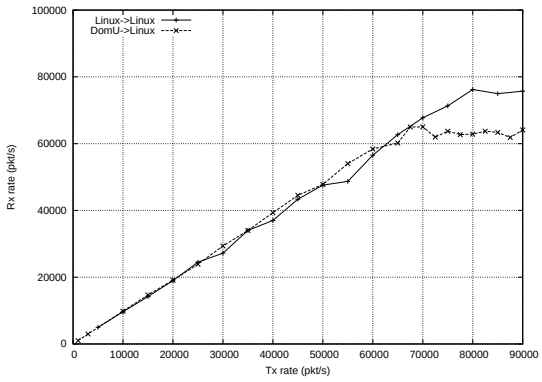

Fig. 4. DomU transmission performance.

Figure 4 shows achieved throughput for Experiment 1.1, compared to throughput measured in Setup #0. This graph demonstrated that domU performance is about 75% of the native GNU/Linux host, providing ourself with an upper bound for transmission capabilities of a NEPTUNE virtual node. Figure 5 shows achieved throughput for Experiment 1.2, again compared to throughput measured in Setup #0. This graph shows that, as receiver, domU performance is 65% of native GNU/Linux.

Figures 6 and 7 show the average CPU load measured during Experiments 1.1 and 1.2, as reported by the *virt-top* monitoring tool. The overall CPU capacity refers to the whole set of four logical CPUs, hence, due to the configuration of

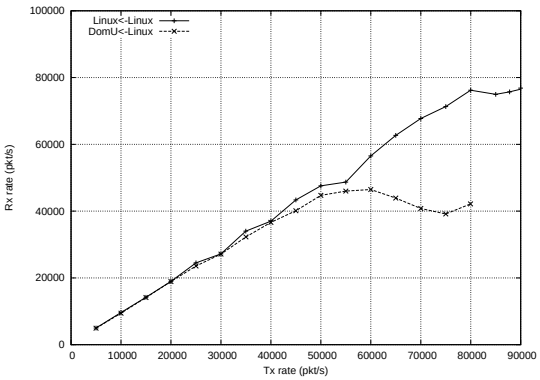

Fig. 5. DomU reception performance.

domUs, each domU may consume as much as 25% of the overall CPU capacity. Figure 6 shows that the traffic generating domU consumes as much CPU as possible even at low packet rates, while dom0 increases the CPU utilization as the packet rate increases. Figure 7 shows that a Xen domU acting as a packet receiver linearly increases its CPU utilization with the packet arrival rate up to a given threshold. Since incoming packets are first processed by dom0, the domU CPU load reflects the dom0 curve up to a threshold value (50 kpacket/s in Figure 7). For packet rates above this latter threshold, domU saturates its CPU utilization curve.

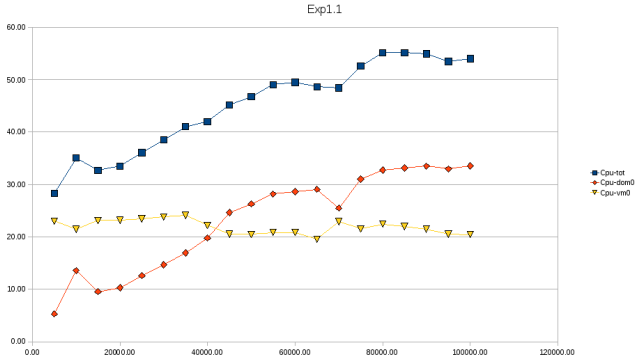

Fig. 6. DomU transmission performance.

We also carried out an experimental evaluation of the impact of NetEm on performance of the emulation layer. These experiments were performed using the same pair of physical machines as for previous experiments, but engaging a dif-

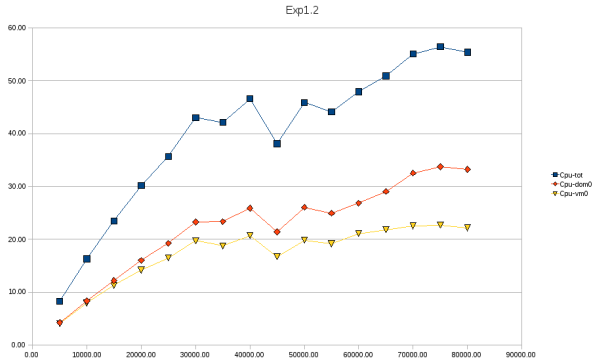

Fig. 7. DomU reception performance.

ferent scenario (Setup #2 shown in Figure 8). Since our tests were unidirectional, we only configured NetEm on the transmitting side.

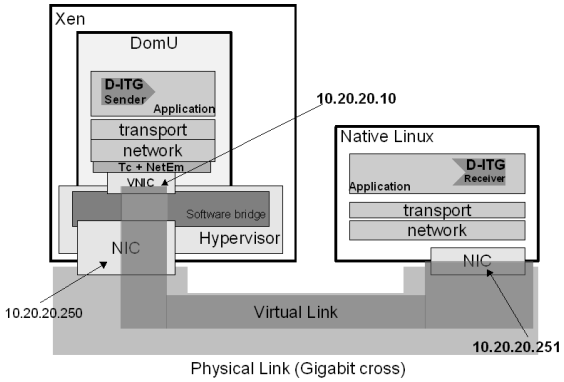

Fig. 8. Experimental setup #2.

Again, we performed two experiments. In both of them, we configured the emulation layer to emulate a virtual link of limited bandwidth, by means of a token bucket filter (TBF). Our goal was to evaluate if the combination of bandwidth limitation mechanisms and virtualization layer (Xen) produced any unexpected effects on the throughput and jitter experienced by packets carried by the emulated link. Figure 9 shows traffic received by the native GNU/Linux host, for several TBF limits applied at the sender side, when the rate of generated traffic was progressively increased. This graph demonstrates that traffic rate has been correctly shaped, delivering the expected bandwidth.

In Figure 10 we compare jitter values experienced on link when using two different configurations: in the fist one, traffic sender run on native GNU/Linux

(no virtualization layer here), while in the second one a Virtual Machine worked as sender (Figure 6). In both cases we generated 25 Mbps of UDP costant bit rate traffic over a virtual link tailored at 20 Mbps by applying a TBF shaper. Results show that the virtualization layer has no significative effects on the jitter, that maintained similar properties.

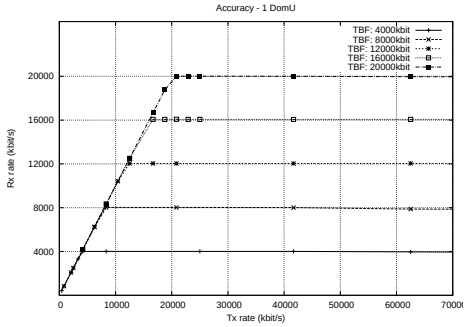

Fig. 9. Token Bucket bw limitation.

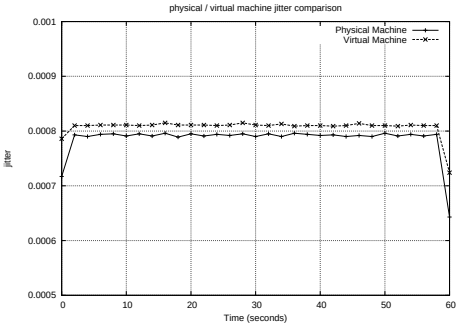

Fig. 10. Jitter.

5 Conclusions

Several papers presented performance evaluation analysis of Xen networking. Our work is not specifically aimed at evaluating Xen performance. Nonetheless, since Virtual Machine Monitor layer is a major component of our network emulator architecture, experiments we presented in this paper are strongly related to those presented in other works. In [24], packet forwarding performance of Xen's dom0 and domUs are evaluated and compared to native GNU/Linux. Results show that dom0, in the absence of concurrent domUs, has comparable performance (within 5%) of native GNU/Linux. On the other hand, a remarkable performance drop is experienced by domUs, especially in the case of multiple concurrently active domUs running on top of the VMM. This paper also compares performance of Xen's bridging and routing configurations in terms of packet forwarding within domUs. This last work is of less interest for our purposes, since we assumed the use of bridging configuration for Xen in the NEPTUNE architecture. Other papers ([25][26]) have evaluated overhead caused by the VMM layer, for previous versions of Xen (v2.x). In [25] a detailed profiling of Xen shows that the execution of I/O operations in a domU has a higher instructions count with respect to both native GNU/Linux and dom0, which explains throughput degradation. Here, authors also examine how performance can be negatively affected by the use of a general virtual NIC driver that causes domUs to execute operations like TCP segmentation in software, ignoring physical NIC TCO (TCP segmentation offload) capabilities, which are instead used by native GNU/Linux

and dom0. Our experiments show similar results to those presented in previous papers. The relevance of results we provide is mainly to identify an upper bound to the capabilities of our emulation system, in order to guarantee correctness of the emulation. More thorough investigations are currently being performed, to evaluate how other parameters, such as packet size, cpu load, number of concurrent virtual machines per physical node and so on, may affect accuracy of network emulation in NEPTUNE.

6 Acknowledgements

The research leading to these results has been partially funded by the European Community's Seventh Framework Programme (FP7/2007-2013) under grant agreement n224263-OneLab2.

References

1. Breslau, L., Estrin, D., Fall, K., Floyd, S., Heidemann, J., Helmy, A., Huang, P., McCanne, S., Varadhan, K., Xu, Y., Yu, H.: Advances in network simulation. IEEE Computer **33**(5) (May 2000) 59–67
2. Rizzo, L.: Dummynet: a simple approach to the evaluation of network protocols. SIGCOMM Comput. Commun. Rev. **27**(1) (1997) 31–41
3. PlanetLab: PlanetLab Europe. `http://www.planet-lab.eu/`
4. M. Carbone, G. Cecchetti, L.R.: Integrated dummynet and planetlab. `http://www.onelab.eu/images/PDFs/Deliverables/d4e.2.pdf` OneLab Project FP6-2004-IST-4 Public Deliverable D4E.2.
5. Grau, A., Maier, S., Herrmann, K., Rothermel, K.: Time jails: A hybrid approach to scalable network emulation. In: PADS '08: Proceedings of the 22nd Workshop on Principles of Advanced and Distributed Simulation, Washington, DC, USA, IEEE Computer Society (2008) 7–14
6. Hibler, M., Ricci, R., Stoller, L., Duerig, J., Guruprasad, S., Stack, T., Webb, K., Lepreau, J.: Large-scale virtualization in the emulab network testbed. In: ATC'08: USENIX 2008 Annual Technical Conference on Annual Technical Conference, Berkeley, CA, USA, USENIX Association (2008) 113–128
7. Herrscher, D., Leonhardi, A., Rothermel, K.: On node virtualization for scalable network emulation. In: Proceedings of the 2005 International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS '05). (July 2005)
8. Maier, S., Herrscher, D., Rothermel, K.: Experiences with node virtualization for scalable network emulation. Comput. Commun. **30**(5) (2007) 943–956
9. Di Gennaro, P., Bifulco, R., Canonico, R., Ventre, G.: Neptune: Network emulation for protocol tuning and evaluation Poster presented at the 2nd ICST International Conference on Simulation Tools and Techniques (SIMUTOOLS09), Rome, March 2009.
10. Di Gennaro, P., Bifulco, R., Canonico, R.: Neptune project sources homepage. `http://code.google.com/p/neptune-network-emulator/`
11. Hemminger, S.: Network emulation with netem. `http://linux-net.osdl.org/index.php/Netem` In Linux Conf Au, April 2005.

12. Canonico, R., Di Gennaro, P., Manetti, V., Ventre, G.: Virtualization techniques in network emulation systems. In Boug, L., Forsell, M., Traff, J.L., Streit, A., Ziegler, W., Alexander, M., Childs, S., eds.: Euro-Par Workshops. Volume 4854 of Lecture Notes in Computer Science., Springer (2007) 144–153
13. Vahdat, A., Yocum, K., Walsh, K., Mahadevan, P., Kostic, D., Chase, J., Becker, D.: Scalability and accuracy in a large-scale network emulator. In: Proceedings of the Fifth Symposium on Operating Systems Design and Implementation (OSDI), USENIX Assoc (2002) Dept. of Comput. Sci., Duke Univ., Durham, NC, USA.
14. Herrscher, D., Rothermel, K.: A dynamic network scenario emulation tool. In: Proceedings of the 11th International Conference on Computer Communications and Networks (ICCCN 2002). (October 2002)
15. Bavier, A., Feamster, N., Huang, M., Peterson, L., Rexford, J.: In vini veritas: realistic and controlled network experimentation. SIGCOMM Comput. Commun. Rev. **36**(4) (2006) 3–14
16. Bhatia, S., Motiwala, M., Mhlbauer, W., Mundada, Y., Valancius, V., Bavier, A., Feamster, N., Peterson, L., Rexford, J.: Trellis: A platform for building flexible, fast virtual networks on commodity hardware. In: Proceedings of ROADS 2008. (2008)
17. Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., Warfield, A.: Xen and the art of virtualization. In: Procs. of the 19th ACM Symposium on Operating Systems Principles, SOSP'03, ACM Press (2003) 164–177
18. Burtsev, A., Radhakrishnan, P., Hibler, M., Lepreau, J.: Transparent checkpoints of closed distributed systems in emulab. In: EuroSys '09: Proceedings of the fourth ACM european conference on Computer systems, New York, NY, USA, ACM (2009) 173–186
19. RedHat, et others: LibVirt virtualization API. `http://www.libvirt.org`
20. Ricci, R., Alfeld, C., Lepreau, J.: A solver for the network testbed mapping problem. SIGCOMM Comput. Commun. Rev. **33**(2) (2003) 65–81
21. Xen: Xen wiki. `http://wiki.xensource.com/xenwiki/XenNetworking`
22. Menon, A., Cox, A.L., Zwaenepoel, W.: Optimizing network virtualization in xen. (2006) 15–28
23. Botta, A., Dainotti, A., Pescapé, A.: Multi-protocol and multi-platform traffic generation and measurement INFOCOM 2007 DEMO Session, Anchorage (Alaska, USA), May 2007.
24. Egi, N., Greenhalgh, A., Handley, M., Hoerdt, M., Mathy, L., Schooley, T.: Evaluating xen for router virtualization. In: Computer Communications and Networks, 2007. ICCCN 2007. Proceedings of 16th International Conference on. (2007) 1256–1261
25. Menon, A., Santos, J.R., Turner, Y., Janakiraman, j.G., Zwaenepoel, W.: Diagnosing performance overheads in the xen virtual machine environment. In: VEE '05: Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments, New York, NY, USA, ACM Press (2005) 13–23
26. Cherkasova, L., Gardner, R.: Measuring cpu overhead for I/O processing in the xen virtual machine monitor. (2005) 387–390