HAL
open science

A Comparative Study of Network Link Emulators

Lucas Nussbaum, Olivier Richard

To cite this version:

Lucas Nussbaum, Olivier Richard. A Comparative Study of Network Link Emulators. Communications and Networking Simulation Symposium (CNS'09), Mar 2009, San Diego, United States. inria-00425613

HAL Id: inria-00425613

https://inria.hal.science/inria-00425613v1

Submitted on 22 Oct 2009

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

A Comparative Study of Network Link Emulators

Lucas Nussbaum and Olivier Richard

Laboratoire d'Informatique de Grenoble – LIG
{Lucas.Nussbaum,Olivier.Richard}@imag.fr

Keywords: network, emulation, software, accuracy

Abstract

Between discrete event simulation and evaluation within real networks, network emulation is a useful tool to study and evaluate the behaviour of applications. Using a real network as a basis to simulate another network's characteristics, it enables researchers to perform experiments in a wide range of conditions. After an overview of the various available network emulators, this paper focuses on three freely available and widely used network link emulators: Dummynet, NISTNet, and the Linux Traffic Control subsystem. We start by comparing their features, then focus on the accuracy of their latency and bandwidth emulation, and discuss the way they are affected by the time source of the system. We expose several problems that cannot be ignored when using such tools. We also outline differences in their user interfaces, such as the interception point, and discuss possible solutions. This work aims at providing a complete overview of the different solutions for network emulation.

1. INTRODUCTION

The performance of distributed applications is often difficult to measure. Most evaluations are carried out either using modeling and simulation, or using evaluation on real-world systems. But both have well-known shortcomings, unfortunately. Simulations depend heavily on the quality of the underlying model, and modeling with accuracy can be a long process, especially when the system being studied is a complex, pre-existing application, whose internals are not known. The opposite solution is to execute applications on experimental testbeds, such as PlanetLab; instead of trying to understand how the application works, the application is executed, and the result of its execution is then examined. But platforms like PlanetLab provide a single configuration; it is usually not possible to modify the platform to test the application under different conditions (for example, different network conditions), leading to results that lack generalization.

Emulation is another, intermediate solution. It consists of executing the real application in a synthetic (*simulated*) environment. It allows researchers to reproduce various conditions, at a very low cost. Instead of modifying the network infrastructure, the emulation layer is configured to emulate latency, lower bandwidth, or degraded network conditions.

Several solutions for network emulation already exist. However, the accuracy of these tools has never been compared, despite them being widely used by the community. This work contributes a comparative study of network emulators, outlining their differences, their advantages and their problems.

The remaining of this paper is organized as follow. Section 2 is an overview of the existing network emulation solutions. In Section 3, we compare the features provided by Dummynet, NISTNet and Linux TC/Netem. In Section 4, we study the accuracy of the network emulation (both latency and bandwidth emulation). Section 5 outlines differences in the tools' user interfaces, while Section 6 discusses an issue with the interception point of packets with TC. Finally, Section 7 provides directions for some future work, before we conclude in Section 8.

2. NETWORK EMULATORS

Network emulators are not a new idea. In 1995, a WAN emulator was used to evaluate TCP Vegas [1]. Amongst all the emulation solutions developed since then, one can distinguish two kinds of network emulators:

Virtual network emulators aim at emulating a whole network cloud. The description of a network topology is supplied to the emulator, which typically uses a cluster of computers to emulate the network. Examples of such emulators are MicroGrid [2], Modelnet [3], Emulab [4], EMPOWER [5], IMUNES [6], V-em [7] and eWAN [8]. However, those approaches are generally quite complex, and their deployment outside of the laboratory which developed them is often very limited, because they involve a non-trivial setup phase.

Network link emulators are more simple. They delay or drop packets coming in or going out of a specific network interface to match the desired network characteristics (latency, packet loss and bandwidth). Delayline [9] is a user-level library providing such features. The Ohio Network Emulator [10] runs on Solaris and is no longer maintained. Dummynet [11] runs on FreeBSD and is integrated with FreeBSD firewall IPFW. NISTNet [12] was initially developed for Linux 2.4 and was recently ported to Linux 2.6. Linux 2.6 also provides Netem [13],

a network emulation facility built into Linux's Traffic Control (TC) subsystem. A network emulator named hxbt [14] is also available in OpenSolaris. Finally, hardware solutions exist, such as GtrcNET-1 [15] (using an FPGA) or the products from Anué [16].

In the remainder of this work, we focus on Dummynet, NISTNet and TC/Netem, for three main reasons:

- Firstly, those three solutions are of production quality, and are no longer prototypes. They are ready to be used by researchers ;
- Secondly, they are freely available on operating systems (Linux and FreeBSD) that are commonly available in laboratories ;
- Finally, they are already being used by the research community, either directly, or integrated into virtual network emulators. For example, Emulab uses Dummynet on its FreeBSD nodes and Linux/TC on its Linux nodes, while V-em uses NISTNet.

3. FEATURES

Dummynet, NISTNet and TC/Netem use the same principle. They capture incoming or outgoing packets, and use a set of rules and queues to store the packets, until they determine that the packet can be released to the operating system (in the case of incoming packets) or to the network (in the case of outgoing packets). However, their implementations and features differ.

Table 1 presents the features of Dummynet, NISTNet, and TC/Netem. NISTNet and Netem have very similar features, and actually share some code, but their design is totally different. While NISTNet is built as a standalone module and relies on the real-time clock for timing (which is normally only used to keep track of time when a computer is powered off), Netem is tightly integrated within the Linux Traffic Control subsystem (usually used to enforce quality of service inside networks), and relies on the same timing source as does the rest of the kernel. Also, Netem is distributed with Linux, while NISTNet is distributed separately. NISTNet is currently only available for versions of Linux lower than 2.6.14 (we successfully used it on Linux 2.6.13.5), but we ported it to Linux 2.6.26[1].

Dummynet uses a totally different code base, and has been integrated into FreeBSD since FreeBSD 4. Its main advantage over NISTNet and Netem is that it works on both incoming and outgoing packets. However, Dummynet doesn't allow to emulate degraded network conditions (packet duplication or corruption).

[1]Our patch is available on http://perso.ens-lyon.fr/lucas.nussbaum/#nistnet.

4. PERFORMANCE EVALUATION

In this section, we study the performance of Dummynet, NISTNet and Linux/TC. We investigate how closely the emulated network's characteristics match the parameters provided by the user. For given latency and bandwidth parameters, we measure the resulting latency and bandwidth on the emulated network.

4.1. Experimental setup

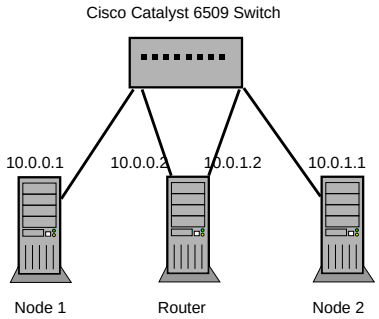

Figure 1. Experimental setup

The following experiments all use the same network and system configuration shown in Figure 1. The platform consists of 3 nodes (Dual-Opteron 2.0 GHz with 2 GB of RAM) of the GridExplorer cluster (part of the french nation-wide project Grid'5000) are used. The *Router* is configured to route packets between *Node 1* and *Node 2*. Nodes 1 and 2 are running Linux 2.6.26, while the Router uses Linux 2.6.22 or 2.6.26 (for TC/Netem), Linux 2.6.26 (for NISTNet), or FreeBSD 6.1 or 7.0 (for Dummynet). Network cards are dual-port Broadcom BCM5780 Gigabit Ethernet controllers integrated in the nodes' motherboard. Without configuring network emulation on the router, we measured a maximum bandwidth of 943 Mbps and a Round Trip Time (RTT) of about 180 μs between nodes 1 and 2.

4.2. Time source and accuracy of latency emulation

The entire focus of the industry is on bandwidth, but the true killer is latency.
Prof. M. Satyanarayanan
Keynote at ACM Mobicom '96

Latency emulation is an important aspect of network emulation. On today's networks, most of the latency is often caused directly by physical constants such as the speed of light in optical fiber, and can't be expected to be improved in the near future. How applications deal with latency is increasingly important for performance, since the available bandwidth keeps increasing.

Table 1. Features of Dummynet, NISTNet, and TC/Netem

	Dummynet	NISTNet	TC/Netem
Availability	Included in FreeBSD	Available for Linux 2.4 and 2.6 ($<$ 2.6.14), patch available for more recent versions	Included in Linux 2.6
Time resolution	system clock (up to 10 KHz)	Real time clock	system clock (up to 1 KHz) or high resolution timers
Interception point	Input and output	Input only	Output only
Latency	Yes, constant value	Yes, with optionally correlated jitter following uniform, normal, Pareto, or normal+Pareto distributions	Yes, with optionally correlated jitter following uniform, normal, Pareto, or normal+Pareto distributions
BW limitation	Yes, delay to add to packets is computed when they enter Dummynet	Yes, delay to add to packets is computed when they enter NISTNet	Yes, using the Token Bucket Filter from TC
Packet drop	Yes, but without correlation	Yes, optionally correlated	Yes, optionally correlated
Packet reordering	No	Yes, optionally correlated	Yes, optionally correlated
Packet duplication	No	Yes, optionally correlated	Yes, optionally correlated
Packet corruption	No	Yes, optionally correlated	Yes, optionally correlated

The accuracy of the emulation depends heavily on the time source used by the software. While NISTNet uses the Real Time clock configured at 8192 Hz, both Dummynet and Netem use the same timers as the rest of the kernel. On FreeBSD (Dummynet), the timer interrupt frequency is configured by the `HZ` variable of the kernel configuration, whose default value is 100 Hz.

The situation is different on Linux. In older kernel versions (until Linux 2.6.22 on `i386` and 2.6.24 on `x86_64`), Netem was using the timer interrupts (configured at 250 Hz by default), like Dummynet on FreeBSD. But in addition to being examined at each timer interrupt, Netem's queue was also examined each time a packet entered Netem, which, with important traffic, could hide problems caused by a low timer frequency.

In newer kernel versions, Netem uses a new subsystem called *High Resolution Timers* [17], allowing to obtain a much more precise timing of interrupts.

We evaluate those different solutions by measuring the RTT over time, by sending *pings* with a high frequency. If the frequency of timer interrupts is not high enough, we would observe variations in the measured RTT. Since packets can only be released by the emulator when a timer interrupt occurs, they might be released slightly too early, or slightly too late, depending on how the rounding will happen. This will cause variations in the emulated latency.

The accuracy of standard ping implementations, which use `gettimeofday()` to measure the time, was not sufficient for our purposes. We modified a *ping* implementation to use the CPU Timestamp Counter (`RDTSC` assembler instruction), to achieve both high measurement frequency (10-20 KHz) and microsecond precision.

We measured the latency over time between nodes 1 and 2 (Figure 1) when configuring the emulators to delay packets from node 1 to node 2 for 10 ms, and evaluated the following configurations for the router:

- Linux 2.6.22 with Linux/TC on `x86_64`, using timer interrupts, with a frequency of 100 Hz, 250 Hz (the default value on Linux) and 1000 Hz ;
- Linux 2.6.26 with Linux/TC on `x86_64`, using High Resolution Timers. We also verified that changing the timer interrupts frequency (100 Hz, 250 Hz, 1000 Hz) didn't change our results with this configuration ;
- Linux 2.6.26 with NISTNet ;
- FreeBSD 7.0, with a frequency of 100 Hz, 1 KHz, and 10 KHz. For some experiments, we also compared the results with FreeBSD 6.1.

Figures 2 and 3 show the results for all of those configurations. The configurations are split in 3 groups, each providing similar results, to ease comparisons. For each configuration, the plot on the left (Figure 2) gives the evolution of latency over time, measured using *pings* sent with a very high frequency, while the plot on the right (Figure 3) gives the distribution function of latency, measured with *pings* sent with a random interval.

Several configurations exhibit a sawtooth behaviour, which can easily be explained: since packets can only be dequeued when a timer interrupt happens, the duration of their stay

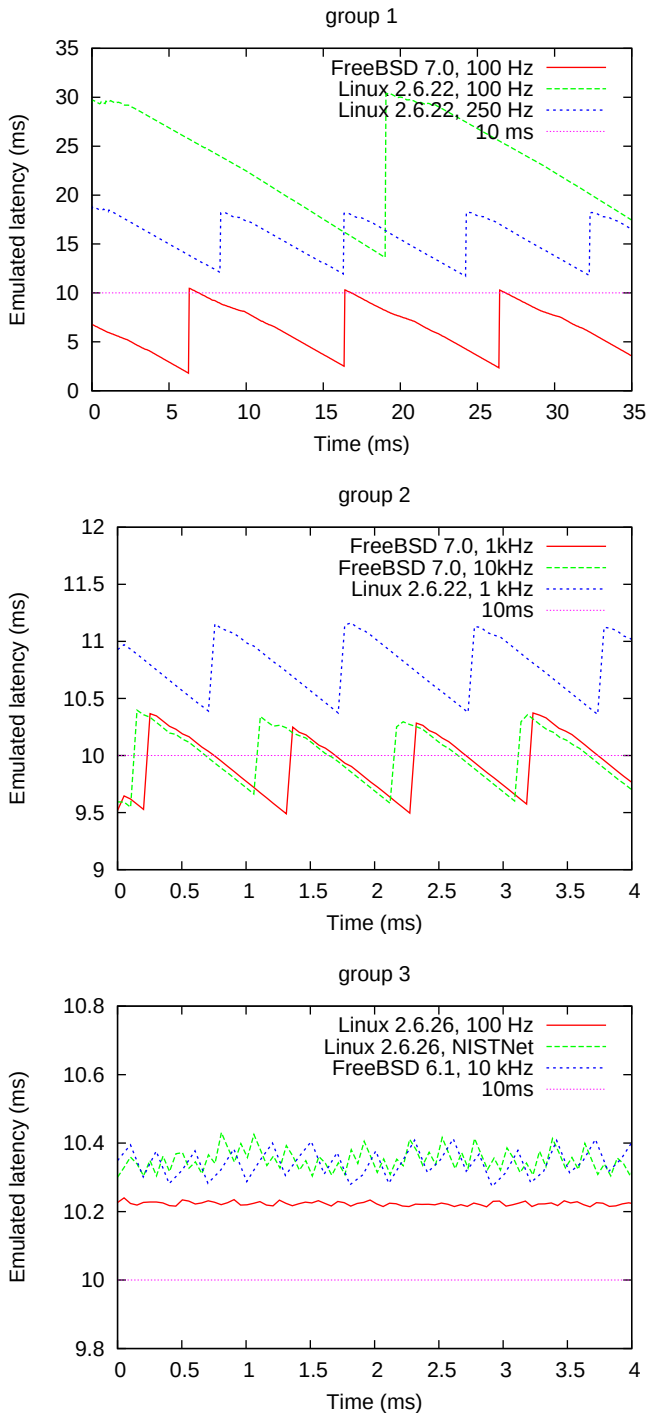

Figure 2. Evolution over time of latency emulated by the *router* node, for pings sent from *node1* to *node2*. The emulated latency varies over time.

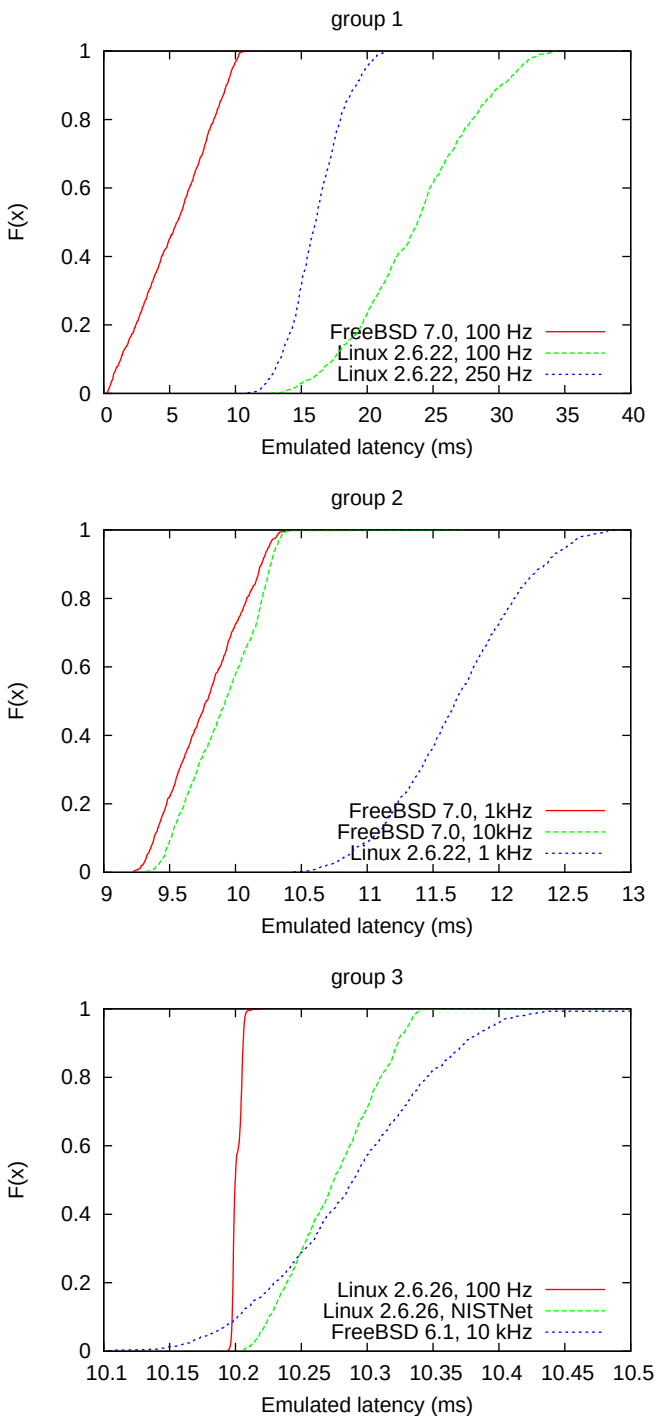

Figure 3. Latency emulation. Empirical cumulative definition function of emulated latencies, for packets sent at random intervals.

in the emulator's queue will depend on their arrival time. Packets arriving long before the next timer interrupt will stay longer in the queue than packets arriving just before a timer interrupt.

This sawtooth behaviour could create a bias in experimental results. At network-level, equipments (routers, switchers) might not be able to handle a sudden burst of packets, and cause packet drops. At application-level, those bursts of packets will increase the need for large buffers, and might desynchronize processes that would otherwise be synchronized.

With Linux 2.6.22 and FreeBSD 7.0, one can clearly see the influence of the timer frequency. Increasing the frequency makes the emulation more accurate. With a low frequency (100 Hz or 250 Hz), the variations of latency are very important. For example, on Linux, and with a clock configured at 100 Hz, the emulated latency varies between 13 ms and 30 ms when the user configures a latency of 10 ms.

With FreeBSD 7.0, one can also see that the accuracy doesn't improve when the timer frequency is changed from 1 KHz to 10 KHz. With FreeBSD 6.1 (Figure 2, group 3), it is not the case. A frequency of 10 KHz provides latency emulation that is 10 times more precise than with a clock at 1 KHz.

Because of differences in algorithms used to emulate latency, one can also see that the emulated latency is always higher than the one configured with Linux TC. On the contrary, with Dummynet, it is lower than the configured latency most of the time.

Finally, 3 solutions provide reasonable performance (most of the remaining difference between the emulated latency and the configured latency can be explained by the physical latency of the experiment's network):

- NISTNet, because it doesn't use the same timer interrupts as the rest of the system, but a clock configured at 8192 Hz ;
- FreeBSD 6.1 configured with a timer frequency of 10 KHz ;
- Linux with *High Resolution Timers*.

However, increasing the interrupt frequency with NISTNet and FreeBSD 6.1 is not cost-free, because it implies that the interrupt handling routine is executed much more often, causing useless context switchs between userspace and kernelspace, and cache trashing.

We used a simple CPU-intensive program (an extremely simple calculation with no memory pressure - Ackermann's function - is performed 50 billion times) to show the influence of the timer frequency on performance. The execution times of this program on FreeBSD using different timer frequency settings are detailed in Table 2. At 10 KHz, one can see a 3.3% slowdown in the system's speed, which, in some circumstances, could become a problem.

Table 2. Average execution time of a CPU-intensive program on FreeBSD using different timer interrupt frequency settings.

HZ value	Execution time	Overhead
100 Hz	81.5s	
1000 Hz	81.7s	0.2%
10000 Hz	84.2s	3.3%

It is also worth noting that NISTNet suffers from the same problem, even if it is using a separate clock for timing. After loading the NISTNet kernel module, the execution of the same program took 86.8s (overhead of 6.3%).

High Resolution Timers don't suffer from the same problems. When they are enabled, but not used, they don't slow down the rest of the system. However, when they are used, they increase the number of interrupts. Since they are more precise, packets won't be sent in groups, but separately, with a different timer interrupt for each packet.

4.3. Bandwidth limitation

Bandwidth limitation is the other important aspect of network emulation. Many of today's network links have very limited bandwidth, or an asymmetric bandwidth, such as broadband or 3G networks. Most experimental testbeds don't include systems hosted on such connections, so it is important for researchers to be able to emulate such links.

The implementation of bandwidth limitation differs in network emulators. While NISTNet and Dummynet simply compute the delay to add to a specific packet based on the configured bandwidth and the current state of the queue, TC uses a Token-Bucket algorithm to shape traffic.

In this experiment, we compared the desired rate with the one measured using `iperf`. Using `iperf` adds a small bias to the measurement, because `iperf` measures the available bandwidth using a TCP stream, while the bandwidth limitation sets the bandwidth available for IP packets. The experiment was carried out on Ethernet, thus the interface MTU (Maximum Transmission Unit) was set to 1500 bytes. The IP and TCP headers are using 52 bytes, so the measured bandwidth was corrected by 3.6% to include the IP and TCP headers.

Figures 4 and 5 compares the corrected measured bandwidth using Dummynet (with a timer interrupt frequency of 10 KHz), NISTNet, and TC/Netem (with Linux 2.6.26). Differences between the achieved bandwidth are limited, but Dummynet and NISTNet are slightly more accurate than TC. When looking more closely at the results when the desired bandwidth is high (between 500 Mbps and 1 Gbps, second

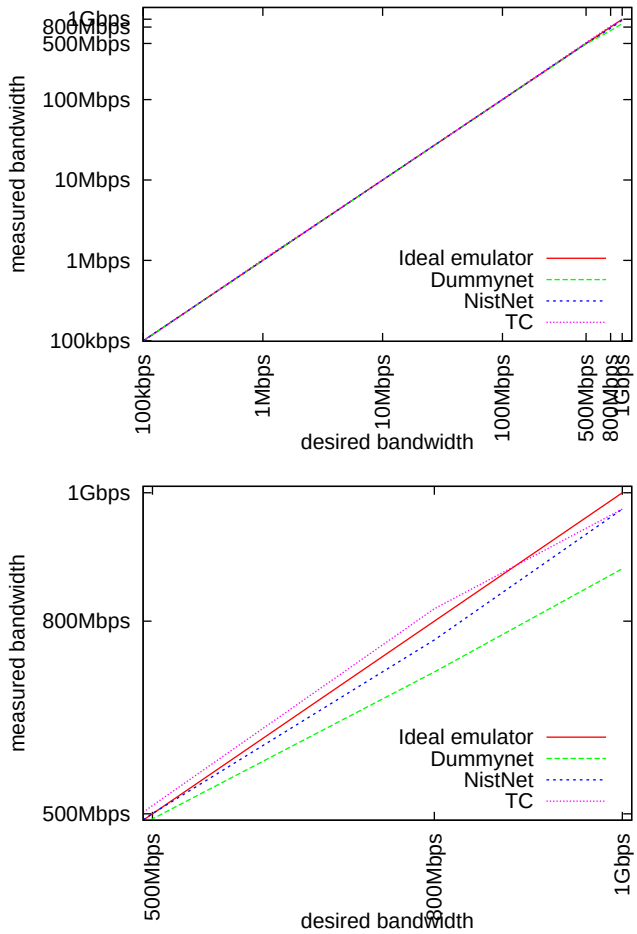

Figure 4. Measured bandwidth when the bandwidth configured in the emulator varies between 100 kbps et 1 Gbps (logarithmic scale, first graph), then between 500 Mbps et 1 Gbps (second graph)

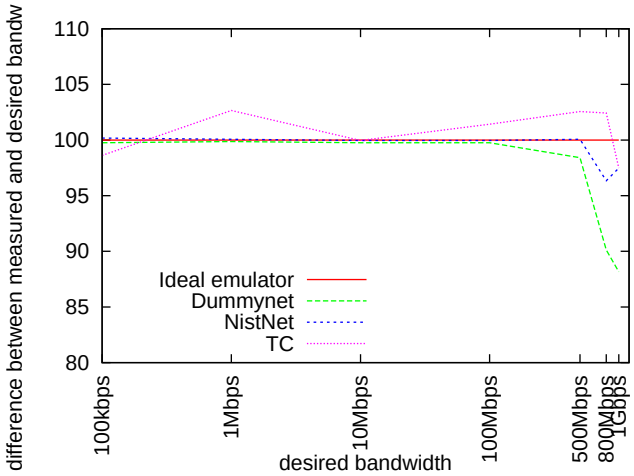

Figure 5. Difference between measured bandwidth, and bandwidth configured in the emulator

graph of Figure 4), one can see that Dummynet didn't allow us to achieve the desired bandwidth.

Bandwidth limitation also suffers from the problem described in Section 4.2.. The frequency of timer interrupts will cause the emulated traffic to be *bursty*. Data will not go through the emulation layer continuously, but by larger amount of data when the emulator is given the opportunity to send data. With Dummynet and NISTNet, the emulator only sends packets to the network when a timer interrupt occurs, leading to important burstiness. As an example, to limit the bandwidth to 100 Mbps with a timer frequency of 1 KHz, 100 Kbit of data (or 8.33 ethernet frames of 1500 bytes) will have to be transmitted at each timer interrupt. Also, if, on average, the emulator has to transmit 1.5 frames per timer interrupt, it will alternate between sending 1 frame and 2 frames, leading to unrealistic traffic.

This could create a bias in measurements if the packets are used to test congestion control mechanisms, for example. Increasing the frequency of the timer source would reduce the problems. Linux TC uses another approach to alleviate those problems. Instead of only sending frames during timer interrupts, it also checks if some frames should be sent when data is received by the emulator. This allows it to send data much more frequently, especially under heavy traffic, but requires that the emulator uses a more precise source to compute time, like the CPU's TimeStamp Counter. Dummynet currently uses the *jiffies* counter (number of timer interrupt since the sytem booted) for all the computations, so it's not possible to get sub-jiffie precision.

Also, it is worth noting that, when using Linux TC with *High Resolution Timers*, this problem doesn't occur.

Length of the waiting queue

Another problem related to the frequency of interrupts is the size of the queue used by the emulator to store packets until they are sent. To be able to achieve the desired bandwidth, the size of the queue must meet the following condition:

$$queue_size \geq emulated_bandwidth * interrupt_frequency$$

Otherwise, the emulator won't be able to suddenly receive a large number of packets, and to send them on the network one by one, to achieve the configured bandwidth.

This problem is particularly important with FreeBSD, because the maximum size of the queue doesn't allow one to emulate networks with a large bandwidth with the default timer frequency (100 Hz): the maximum size of the queue is 100 packets, limiting the emulated bandwidth to 120 Mbit/s in the most favorable case (all packets in the queue are 1500-byte packets). This problem disappears when the interrupt frequency is increased.

Configuration of TC's Token Bucket Filter

The Token Bucket Filter used with TC to limit the bandwidth is also a source of discrepancies between the configured rate and the measured rate. Since its original goal was to be used for providing Quality of Service (QoS) inside networks, it uses a complex algorithm. This algorithm allows bursts of packets to go through at a rate faster than the configured rate ; if the line is idle, there is no need to delay a very short but very intensive connection. This is of course not a good idea with network emulation, but a work-around exists with the `peakrate` parameter, that adds a second TBF with a very small bucket, to avoid bursting.

However, this second token bucket adds complexity to the configuration of the Token Bucket Filter. This makes it very difficult to determine settings that will be emulated at the desired bandwidth. By contrast, configuration of Dummynet or NISTNet is easier. It is important that one verifies that the settings are correct before conducting the experiment.

5. USER INTERFACES

While the performance and the accuracy of network emulators are important, their usability is also an important aspect to consider.

Both Dummynet and NISTNet use a rule-based configuration, similar to the configuration of firewalls, which make them easy to understand, especially for users already familiar with firewall configurations. However, they lack support for complex hierarchical sets of rules, which could be a problem if the user is trying to emulate a complex network topology.

Linux TC uses another approach. Its configuration is done with a hierarchical set of *qdiscs* (queueing disciplines) and classes. It is more powerful, but also more difficult to understand.

6. INTERCEPTION POINT

One important advantage of Dummynet over NISTNet and TC is that it can capture both incoming and outgoing packets. NISTNet only allows emulation of incoming packets, while TC only allows emulation of outgoing packets, which is logical since it was designed as a traffic shaper, not as an emulator. However, in many cases, it is necessary to perform emulation of incoming packets as well, for example if the user wants to perform emulation of the system where the application is running (without using an intermediate router).

A solution exists for TC with the `ifb` device (*Intermediate Functional Block*), which is a dummy (software-only) network device. It is possible to redirect all incoming packets to the ifb device, and to apply emulation parameters when packets exit the `ifb` device. Figure 6 shows how to apply 50 ms of latency to incoming packets.

However, one can question the overhead caused by such a convoluted solution.

```
# initialize ifb
modprobe ifb
ifconfig ifb0 up

# add an ingress qdisc to process
# incoming packets
tc qdisc add dev eth0 ingress

# redirect everything to ifb0
tc filter add dev eth0 parent ffff:\
 protocol ip prio 10\
 u32 match ip src 0.0.0.0/0 flowid ffff:\
 action mirred egress redirect dev ifb0

# set up netem on ifb0
tc qdisc add dev ifb0 root\
 netem delay 50ms
```

Figure 6. Using the `ifb` dummy device to apply emulation parameters on incoming packets

Using a GtrcNET-1, an FPGA-based hardware network emulator and measurement tool, we measured the time taken by packets to traverse a computer acting as a router. In the first case, no TC configuration was used. In the second case, an IFB device was added, and incoming packets were redirected to it, but the IFB device didn't perform any emulation. Figure 7 shows that the difference between the two cases is minor (about 5.2 μs), and probably negligible in most cases.

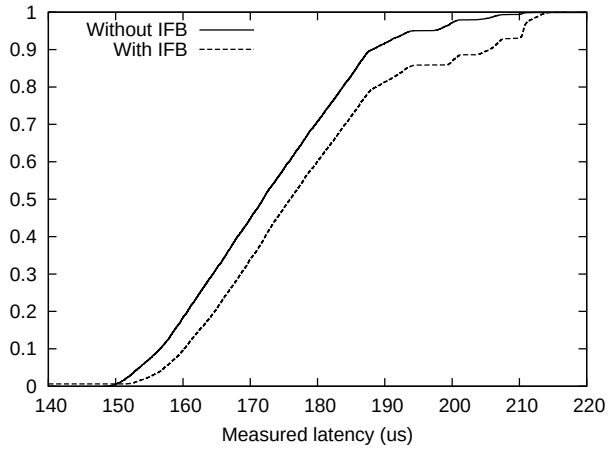

Figure 7. ECDF of the traversal time of a computer acting as a router, with and without redirecting incoming packets to an IFB device

The CPU overhead is unfortunately more important. Under very heavy network load (1 Gbps, small UDP packets generated using `iperf`), our test system showed that the CPU was used about 40% of the time without IFB. When IFB was added, it increased to about 50%. This could be a problem

when the application is running on the same node as the emulator, since increased CPU usage of the emulator might affect the application's performance.

7. FUTURE WORK

One aspect that was voluntarily ignored by this study is the cost of network emulation on the router. If the router is dedicated to network emulation, this is unlikely to become a problem. However, in many cases, it is interesting to execute the application under study directly on the sytem on which network emulation takes place. In that case, network emulation could affect the results significantly.

Secondly, for some experiments, it might be necessary to configure many concurrent queues (for example, on a router emulating the network links of a high number of systems). The performance of network emulators might become a problem when used to emulate a high number of different links. In particular, the algorithm for matching packets and queues will then be of high importance, and should be examined.

8. CONCLUSION

Network emulators allow one to easily perform experiments under various network conditions, enabling researchers to evaluate their algorithms in different environments. However, the fact that different solutions exist, and that they had never been compared before, limited their widespread use.

This work focuses on three network link emulators: Dummynet, NISTNet and the Linux Traffic Control (TC) subsystem, which are freely available in widely used operating systems. Those three emulators have also been used as building blocks for large-scale emulation platforms like Emulab. We contribute a detailed comparison of those tools, including a study of the accuracy of latency and bandwidth emulation. Our work pinpoints several some issues. First, latency emulation exhibits a sawtooth behaviour that could create a bias in experiments. A high-frequency timer source mitigates this problem, but increasing the timer frequency causes an overhead which might be a problem in some experiments. We demonstrate how recent changes in the Linux kernel (*high resolution timers*) allow to improve that situation. Second, we describe how bandwidth emulation, while being of reasonable quality in all three emulators, also suffers from problems. Dummynet doesn't allow one to achieve very high emulated bandwidth, and the timer frequency might lead to burstiness if the emulated bandwidth is important, leading to unrealistic traffic.

Finally, we provide a set of configurations that, for several reasons, don't exhibit some of those problems. It is important that users are aware of those problems, and validate their emulators' settings before performing experiments. Network emulators are powerful tools, but should not be treated as *black boxes*.

REFERENCES

[1] Jong Suk Ahn, Peter B. Danzig, Zhen Liu, and Limin Yan. Evaluation of tcp vegas: emulation and experiment. *SIGCOMM Comput. Commun. Rev.*, 25(4), 1995.

[2] H. J. Song, X. Liu, D. Jakobsen, R. Bhagwan, X. Zhang, K. Taura, and A. Chien. The MicroGrid: a scientific tool for modeling computational grids. In *Supercomputing '00: Proceedings of the 2000 ACM/IEEE conference on Supercomputing*, 2000.

[3] Amin Vahdat, Ken Yocum, Kevin Walsh, Priya Mahadevan, Dejan Kostic, Jeff Chase, and David Becker. Scalability and accuracy in a large-scale network emulator. In *OSDI '02*, 2002.

[4] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. An integrated experimental environment for distributed systems and networks. *SIGOPS Oper. Syst. Rev.*, 36(SI), 2002.

[5] Pei Zheng and Lionel M. Ni. Empower: A cluster architecture supporting network emulation. *IEEE Trans. Parallel Distrib. Syst.*, 15(7), 2004.

[6] M. Zec and M. Mikuc. Operating system support for integrated network emulation in IMUNES. In *Proceedings of the 1st Workshop on Operating System and Architectural Support for the on demand IT InfraStructure*, 2004.

[7] George Apostolopoulos and Constantinos Hassapis. V-em: A cluster of virtual machines for robust, detailed, and high-performance network emulation. In *MASCOTS '06*, 2006.

[8] P. Vicat-Blanc Primet, R. Takano, Y. Kodama, T. Kudoh, O. Gluck, and C. Otal. Large scale gigabit emulated testbed for grid transport evaluation. In *PFLDnet 2006*, 2006.

[9] David B. Ingham and Graham D. Parrington. Delayline: A wide-area network emulation tool. *Computing Systems*, 7(3), 1994.

[10] Mark Allman, Adam Caldwell, and Shawn Ostermann. ONE: The ohio network emulator. Technical Report TR-19972, Ohio University, August 1997.

[11] Luigi Rizzo. Dummynet: a simple approach to the evaluation of network protocols. *ACM Computer Communication Review*, 27(1), 1997.

[12] Mark Carson and Darrin Santay. NIST Net: a Linux-based network emulation tool. *SIGCOMM Comput. Commun. Rev.*, 33(3), 2003.

[13] Stephen Hemminger. Network emulation with NetEm. In *linux.conf.au 2005*, 2005.

[14] Hxbt: WAN emulator for solaris. http://www.opensolaris.org/os/community/networking/readme.hxbt.txt.

[15] Y. Kodama, T. Kudoh, R. Takano, H. Sato, O. Tatebe, and S. Sekiguchi. Gnet-1: gigabit ethernet network testbed. In *CLUSTER '04*, 2004.

[16] Anué systems. http://www.anuesystems.com.

[17] Thomas Gleixner and Douglas Niehaus. Hrtimers and beyond: Transforming the linux time subsystems. In *Proceedings of the Ottawa Linux Symposium*, 2006.