Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institut für
Technische Informatik und
Kommunikationsnetze

ReplTrc: A Tool for Emulating Real Network Dynamics

TIK Report 231

Rainer Baumann, Ulrich Fiedler

Computer Engineering and Networks Laboratory
Swiss Federal Institute of Technology
ETH-Zentrum, Gloriastrasse 35
CH-8092 Zurich, Switzerland
{baumann,fiedler}@tik.ee.ethz.ch

3rd October 2005

Abstract

Evaluating the performance of network sensitive applications, devices, and protocols has become increasingly complex as the diversity of scenarios and speed of networks increase. This includes the evaluation of VoIP and video conferencing applications, telephony devices, and protocols in both wired and wireless scenarios. Therefore, in this paper, we describe how we design and implement ReplTrc, a network emulation tool that is capable to replay large packet delay traces with high accuracy in timing to enable performance evaluation. We assume that packet traces have previously been generated or captured with network probing, simulation, or network calculus. Thus, evaluations with ReplTrc account for all important performance characteristics such as long-range dependence and self-similarity of traffic or cross-traffic that are reflected in the packet traces. ReplTrc essentially consists of (i) a kernel module which intercepts the Linux protocol stack to delay, drop, or duplicate packets and (ii) a user space process for transferring traces into the kernel module. Moreover, the technique of replaying traces makes ReplTrc flexible to conduct tests in diverse scenarios under various network conditions. We have conducted excessive tests to ensure that ReplTrc is capable to meet the real time requirements necessary for performance testing when installed on a commodity PC.

1 Introduction

Performance testing under a wide variety of conditions is essential when developing network sensitive applications, devices, and protocols. This includes performance testing when developing IP phones, circuit emulation adapters, and gaming or video conferencing hard- and software as well as performance testing for new transport or routing protocols in ad hoc or wireless scenarios.

To perform reproducible performance tests in simple laboratory settings, [3] suggested to emulate networks, i.e. to employ a “box” that can reproduce the dynamics induced by networks (see figure 1). Given the increasing diversity and growing speed of networks, this has become difficult for two reasons: First, emulating networks is a time critical task. In many test scenarios, commodity PCs offer sufficient capabilities and performance for this task. However, kernel programming becomes necessary to avoid unpredictable scheduling delays from the operating system. This necessity is frequently not accounted for in existing emulators.

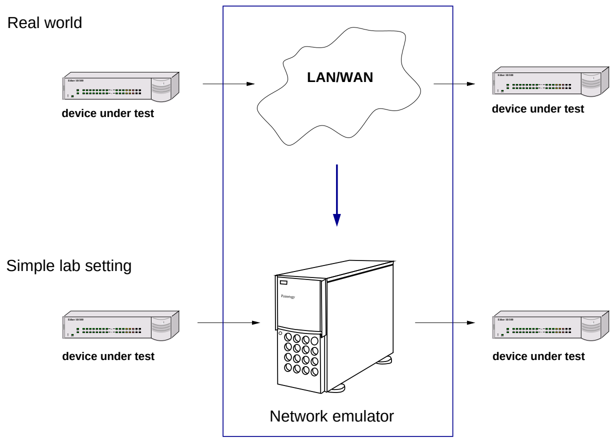

Figure 1: *Performance Evaluation*

Second, network traffic shows invariant statistical characteristics such as long-range dependence/self-similarity[1] that are known to have significant impact on performance characteristics [14]. These characteristics cannot be reproduced easily with simple integer algorithms or small tables that fit into the kernel. Presumably the only efficient way to address this is to reproduce the network dynamics from large traces that can be verified to reflect these statistical characteristics. These traces include information on packet duplication and loss in addition to packet delay and can be generated or captured with network probing, simulation, or network calculus.

Therefore, in this paper we describe how we build a network emulator that

1. employs kernel programming to protect the time critical task of network emulation against unpredictable process scheduling delays.

2. offers a capability to reproduce network dynamics from large packet traces that were previously generated or captured.

3. additionally offers a capability to generate network dynamics from a simple mathematical model.

This enables a simple way to account for important statistical characteristics in network traffic and fosters flexibility in performance testing.

The rest of this paper is structured as follows: Section 1.1 reviews other existing network emulators. Section 2 explains what long-range dependence is, why it is important to account for long-range dependence in performance evaluations and how emulators can reproduce long-range dependent network dynamics. Section 3 reviews design issues and describes the architecture and implementation of the network emulator. In section 4 we evaluate the emulator's performance limitations and delay precision before we conclude in section 5.

1.1 Related Work

A number of existing emulators do not account for the fact that network emulation is a time critical and run in the operating system's user space. The NS Emulation [1] including its Emulab front-end [11] as well as the Ohio-Network-Emulator ONE [12] and the emulator described in [8] fall into this category. A second category of existing emulators run in the operating system's kernel space however fall short

[1]We denote that when modeling network traffic the terms long-range dependence and self-similarity imply each other (see [13] for details).

in providing means that allow the researcher to adequately account for key traffic characteristics. Dummynet [18] and NIST Net [6] fall into this category. Dummynet is part of the IP firewall in FreeBSD kernel. Its functionality is capable to model constant packet delay as well as packet delay that comes from a bandwidth limitation on a single link. However, there is no built-in support to configure these delays which makes it impossible to adequately account for key network traffic characteristics without modifying the kernel. NIST Net [6] runs as a Linux kernel module. Its functionality is capable to model variable network delay. NIST Net loads a small table into the kernel that is used to generate delay values. As a consequence these values are either statistically independent or short-range dependent (see section 2 for details). Thus, NIST Net cannot account for long-range dependence and self-similarity in traffic patterns.

2 Modeling Network Dynamics

This section introduces the statistical background of correlation structures such as long-range dependence in network dynamics, discusses their implications on performance evaluations and reviews how this can be accounted for when designing network emulators.

Correlation structures can be classified according to their implications on performance evaluations. This leads to a classification into uncorrelated, short-range dependent, and long-range dependent.

We illustrate the effect of correlation structures with a discussion on the average throughput on a network link (for a formal introduction see [4]). To simplify the formula we make the technical assumption that time is discrete and the throughput has a finite mean and variance which we denote with μ and σ. Then the *autocorrelation* which denotes the relation between expected throughput X_i at time i and throughput X_j at time j can be defined with

$$\rho(i,j) = \frac{E[X_i - \mu] * E[X_j - \mu]}{\sigma^2} \tag{1}$$

The simplest case is that there are no relations between the expected throughput at time i and j. This implies that the variance of the average throughput is given by

$$var(\overline{X}) = \frac{\sigma 2}{n} \tag{2}$$

where n is the number of measurements and the autocorrelations

$$\rho(i,j) = 0 \; for \; all \; i,j \tag{3}$$

This corresponding correlation structure is called *uncorrelated*.

However, this is not very realistic. Since the autocorrelation in throughput typically only depends on the time difference (“lag”) $k = |i - j|$, which statisticians call *weak stationarity*, this can be rewritten as

$$\rho(k) = \frac{E[X_i - \mu] * E[X_{i+k} - \mu]}{\sigma^2} \tag{4}$$

It can now be shown [4] that if the sum of the correlations over all lags k is finite, then the variance of the average throughput is given by

$$var(\overline{X}) = c_1 * \frac{\sigma 2}{n} \tag{5}$$

where n is the number of measurements and c_1 is a constant correction factor and

$$\rho(k) \sim \frac{1}{k} \; for \; k \to \infty \tag{6}$$

or faster. This correlation structure is called *short-range dependence*.

However, for the case where the sum of correlations over all lags becomes infinite, it has been observed [4] that for a number of phenomenons of practical interest the variance of the average throughput is given by

$$var(\overline{X}) = c_2 * \frac{\sigma 2}{n^{\beta}} \; for \; 0 < \beta < 1 \tag{7}$$

where n is the number of measurements and c_2 is a constant correction factor and β specifies the slow down of convergence. The autocorrelations

$$\rho(k) \sim k^{-\beta} \; for \; k \to \infty, \quad 0 < \beta < 1 \tag{8}$$

This correlation structure is called *long-range dependence*. The relation to the Hurst parameter H is $\beta = 2H - 2$.

A consequence of this difference in correlation structures is that for a sufficiently large number of measurements, the average in the short-range dependent case will have a significantly smaller variance compared to the long-range dependent case. For network traffic this effects is significant since the Hurst parameter H is typically around 0.9 which implies that beta is around 0.2. Given that long-range dependence also implies trends in measurements values this difference in correlation structures also implies that estimating the average while not accounting for long-range dependence may lead to significantly inaccurate estimations.

However, accounting for long-range dependence in network traffic or cross traffic is a difficult issue when designing a network emulator. To our knowledge there are essentially two ways to account for long-range dependence. The first is to replay large packet traces that inherently reflect the corresponding statistical properties. This is simple and presumably sufficient in many cases. Verifying that packet traces account for long-range dependence can either be done in the time domain or in the frequency domain. The easiest method for that is to show that burstiness in time domain is preserved under aggregation. This can be done with plotting the functional relation

$$Var(X_i^{(m)}) = m^{2-2H} \; Var(X_i) \tag{9}$$

in a log-log plot (see figure 2 for an example) where

$$X_i^{(m)} = \frac{1}{m}(X_{(i-1)m+1} + ... + X_{im}) \tag{10}$$

defines the aggregation. However, more sophisticated methods such as Whittle or wavelet estimators (see figure 4) lead to more reliable and accurate results.

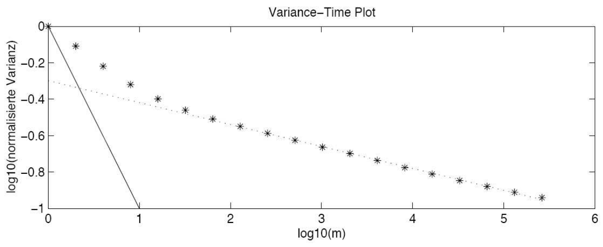

Figure 2: *Identifying long-range dependence with a variance-time plot (from [10])*

The second way to account for a long-range dependent correlation structure in a network emulation is to fit parameters of either a fractional gaussian noise process or a fractional autoregressive integrated moving average processes (fractional ARIMA) to measured data (see chapter 10.3 in LeBoudec [5]).

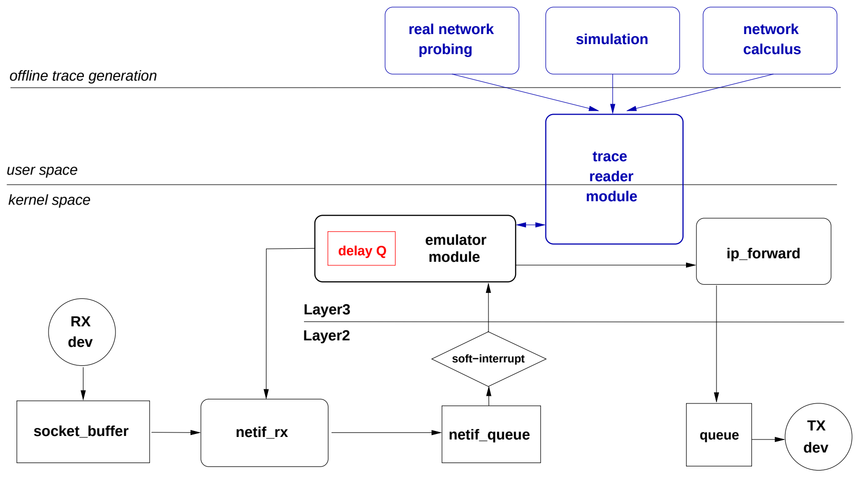

Figure 3: *Overall architecture*

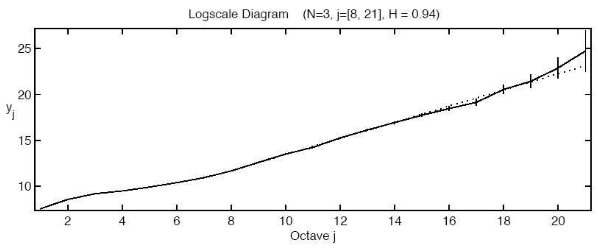

Figure 4: *Identifying long-range dependence with a wavelet estimator (from [10])*

This is an interesting alternative to replaying traces in scenarios where applications/devices under test are adaptive to network load. However, these processes are difficult to implement in a real time environment. In addition to that, literature [19] reports that it is a unsolved problem to find consistent and robust estimators for parameters to employ these processes to model network traffic. Presumably this is due to the extreme dynamics that can be found in network traffic and the fact that estimators show different sensitivity to periodicity, noise, and trends.

A very first step in accounting for correlations in a network emulation is to employ configurable random distributions for packet delay, loss, duplication and to account for correlations of consecutive values. This is what tools such as NIST Net do when generating packet delay values. Formally such correlations are called lag one correlations. Such lag one correlations in the delay, loss and duplication can be modeled with

$$d_i = \rho * d_{i-1} + (1-\rho) * A_i \tag{11}$$

where the d_i is the i-th value, ρ is the lag one correlation and A_i are independent random values from a configurable distribution. The great advantage of this simple model is that it can easily be evaluated with integer arithmetics and configurable tables of probability distributions that fit into the kernel. However, we denote that this model cannot account for long-range dependence and may thus significantly distort performance evaluations. This since the correlation at lag k computes to

$$\rho_k = \rho^k \ for\ k \geq 1 \tag{12}$$

which is short range dependent since for $k \to \infty$ these ρ_k decay faster than the $\rho(k)$ in equation 6.

3 Architecture

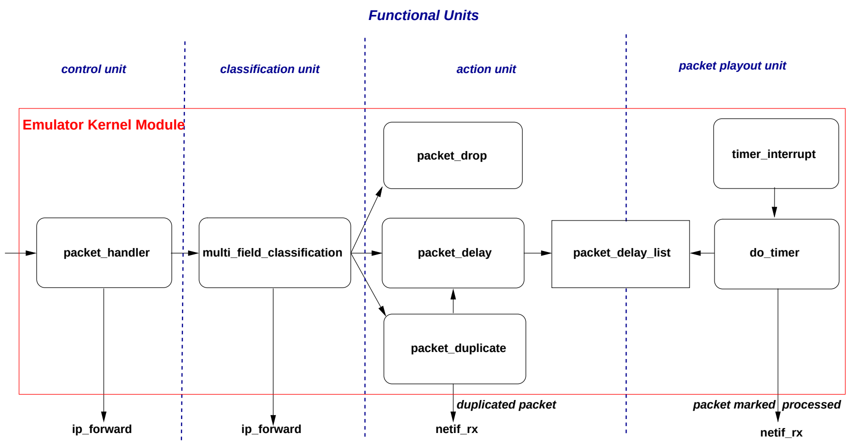

Figure 5: *Emulator kernel module*

The architecture of our network emulation tool is based on intercepting the Linux TCP/IP stack between layer 2 and 3. To achieve this, we register our emulation kernel module as a layer 3 packet handler. The kernel module then performs time critical actions on incoming packets such as delay, drop, duplicate. Once this is done, packets get marked as processed and are forwarded by the normal IP packet handler. For each flow[2] of IP packets actions can be controlled

1. either with a packet action trace that is successively loaded by a preemptive low priority user space process or

[2]A flow is characterized by the port- and IP-addresses of a source-destination host.

2. by configuring probability distributions and correlations between successive actions of the same type.

The overall architecture is illustrated in figure 3.

3.1 Packet interception

Packet interception is done between layer 2 and 3 (see figure 6). The processing path of a packet from arrival until interception is as follows. When an Ethernet frame arrives at a receiving interface `RX dev`, it is temporarily stored in the device's memory before a triggered interrupt copies it into a socket buffer. The deposed frame is then unpacked and analyzed by `eth_type_trans` determining the appropriate layer 3 protocol. Then `netif_rx` transfers the packet to a queue and informs the responsible layer 3 protocol handler by triggering a software interrupt. This is the point where the emulator module is hooked in. The hook is realized by replacing the handler for standard IP packets.

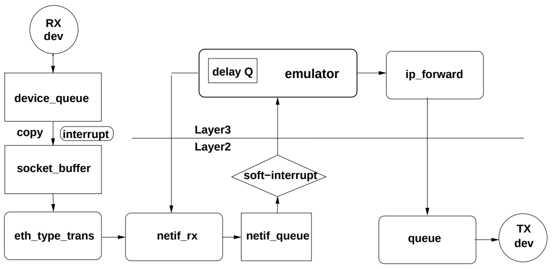

Figure 6: *Packet interception in the GNU/Linux kernel 2.4*

3.2 The emulator kernel module

The emulator module in the kernel essentially consists of a packet handler, a multi field classifier and action handlers for dropping, delaying or duplicating packets (see figure 5). The packet handler controls the emulation and a multi field classification of the IP headers that enables us to identify the flow associated with a packet. Based on this classification, the action is determined

1. either from a instruction buffer that contains a chunk of the packet action trace or
2. by looking up cumulative distribution function (CDF) tables for drop, delay and duplication probability.

Essentially, the emulator module consists of four functional units (see figure 5).

1. the control unit, implemented as of the `packet_handler` where the packets come in
2. the classification unit, implemented as of the `multi_field_classification` which determines the action to be taken
3. the action unit, containing the three action handlers for dropping, duplicating and delaying
4. the packet playout unit, containing the `interrupt_timer` and it's handler (`do_timer`)

First of all, when the `packet_handler` receives a packet, it checks if the emulator is switched off or the packet has already been processed. In this case, it forwards the packet to the IP protocol handler (`ip_forward/ip_recv`) without further processing. Otherwise the multi field packet header classification (`multi_field_classification`) is invoked for checking which action needs to be performed with this packet. This depends on the flow the packet belongs to. For this purpose, we look up a two-level hash table containing per flow information. If no entry is found, the packet is directly forwarded to the IP protocol handler. Otherwise, the action is as specified in the packet action trace or as configured in the CDF table and the corresponding handler is called. We implement handlers for the following actions.

1. dropping
2. duplicating
3. delaying

First, the `packet_drop` handler deletes the packet from the buffer and exits the module.
Second, the `packet_duplicate` handler creates a new instance of the packet. This duplicated packet is then reinjected into `netif_rx`. Thus, we can handle it separately. More over, this opens the possibility to create cascaded instances of the same packet. The original packet is transfered to the `packet_delay` handler.
Third, the `packet_delay` handler forwards the packet into a delay list (`packet_delay_list`). This is implemented with radix sort. Due packets are released from this list every $121\mu s$ ($\approx 1tick$) since this is the interrupt period of the MC146818 real-time clock. Then the packets get marked and are reinjeted into `netif_rx`. As a consequence of marking, the `packet_handler` of the emulator kernel module sends them directly to the standard `ip_forward` handler.

It turns out that the MC146818, initialized with the maximal frequency of $8192Hz$, is the limiting factor for the precision and resolution of configured delays of the emulator (see section 4).

3.3 The Trace Reader Module

The trace-reader module works on a per-flow level. It feeds packet action traces from disk into a packet action buffers that are located in the kernel (see figure 7). The module is implemented as a low priority preemptive user space process that employs the process file system and signal handling to transfer chunks of the packet action trace into the kernel. This is to make sure that it does not distort the emulation. It turns out that this is the most efficient implementation in terms of performance, run time complexity and overhead.

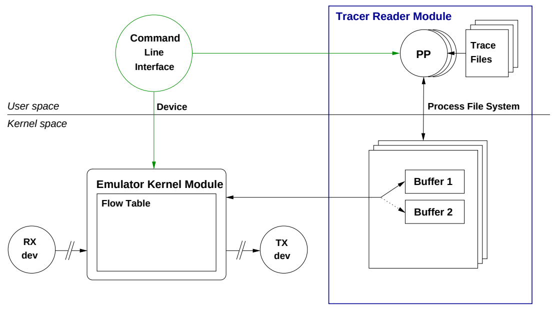

Figure 7: *Trace reader module*

3.3.1 Design alternatives

Strictly speaking, trace reading from disk into the kernel is a classical producer-consumer problem. Regarding the GNU/Linux design principles, reading from disk should be done by a process running in

user mode. The producer therefore runs in user space and sends the data to the consumer, which runs in kernel space. We assume that synchronization is performed with system signals and system calls.
To come to a design decision, we review the performance requirements of the trace reader module as well as potential mechanisms to implement the data transfer from user to kernel space. Then, we conduct tests to evaluate the suitability of these mechanisms.

The major performance requirement is the timely delivery of chunks of packet action traces that derive the emulator. Moreover, the delivery should not induce undesirable additional delay in the emulator. The rate of incoming packets on a $100 MBit/s$ Ethernet link is at most 271739 packets per second when we assume that the link is flooded with empty IP packets ($46 Bytes$ with no payload neither a layer 4 protocol header, see figure 8). Thus the delivery of information from the packet action has to able to keep up with this rate.

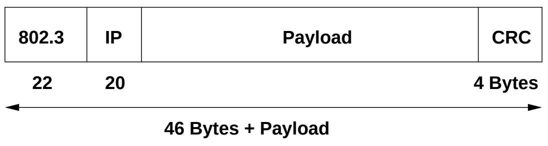

Figure 8: *Ethernet frame containing an IP packet*

Since the emulator module operates as a loadable kernel module (section 3), the large traces need to be successively transfered into this module. Regarding the GNU/Linux design principles, this should be done by a process running in user mode since a user space process runs with low priority and is preemptive. Therefore we first get granular on how to hand over large amount of data form the user to the kernel space.

On GNU/Linux operating systems, there exists several mechanisms for data transfer between user and kernel space. Popular mechanisms for handling large amount of data in an efficient way include the following.

1. transfer over memory mapping
2. transfer over the process file system
3. transfer over device files

First, the concept of memory mapping is to create a shared memory region that is accessible form both kernel an user space. This is done by allocating a virtual file on the file system and by reserving some virtual addresses in memory, using the `mmap` command. The kernel then translates operations on that file directly to the corresponding addresses which are accessible form the kernel space. Memory mapping on 32 Bit architectures is on one side limited by the amount of available physical memory (RAM) and on the other side by the dimension of the virtual address space to $4GB$. Since the kernel itself requires some areas of the address space, it is even less, usually around $3GB$. Memory mapping is fast since there is no need for unnecessary copying of data between the to address spaces. But in general, its handling is rather complex and the limitations in size militate.

The second option is to employ the Process File System (procfs), a virtual file system only existing in memory, which is usually mounted in the `/proc` directory for user-kernel-space communication. It takes the advantage of call back functions which invoke the `copy_from_user` function for transferring data between user and kernel space. The procfs has no limitations concerning the volume of data to transfer since a clever buffering functionality is already integrated. Therefore, it is very easy to handle and the overhead caused by the mechanism and the copy operation are acceptable.

The third option is to employ device files. They originally are supposed to represent physical devices. Hence, they allow us to communicate with the device driver in the kernel to transfer data using the `copy_from_user` function. Comparing to the procfs, it also uses `copy_from_user` but its interface is rather limited and is not desgined for large amount of data.

To evaluate usability in our implementation, we conduct the following performance test. We make a user process to reads a $1GB$ file from disk and transfers it directly to the kernel using the three methods. We find that on our PC all three alternatives can easily manage to transfer the packet action trace for $200MBit/s$ of data [16]. Taking into account that we require $4Byte$ information per packet, we only need $8,7MBit/s$ for the maximal number of 271739 packets per second (section 3.3). Hence performance of the transfer mechanism is not of major concern for the design. We thus implement transfer via the process file system due to it's low level of complexity (see table 1).

	Mmap	Procfs	Device files
Performance	++	+	+
Complexity	- - -	-	- -
Overhead	-	+	+

Table 1: *Design alternatives for delay trace transfer*

At this place it should be mentioned that the maximal number of packets that can be stored within the kernel module is not a design issue since the required amount of memory for delaying all packets of a $100MBit/s$ link by $1second$ is about $15,8MB$.

3.3.2 Implementation

The trace reader module is implemented based on a per flow producer-consumer architecture. The producer process (PP) runs in user space and sends the data to the consumer, which is represented as a buffering mechanism in the kernel space (see figure 7). We assume that synchronization is performed with system signals and system calls.

In our implementation, a new trace for a flow is loaded as follows. The command line interface (CLI), which can be called directly form a Linux shell, first requests a unique flow ID from the trace reader module in the kernel by executing a procfs read on a special file. With the obtained flow ID and the filename of the trace, a new producer process is initialized. Subsequently the CLI registers the flow with the corresponding producer process in the flow table of the emulator's kernel module. Associated with each entry in this table is a buffering mechanism consisting of two separate buffers from which the emulator gets its action instructions for packet handling. The two buffers enable for continuous operation. While an empty buffer is reloaded, the emulator can still read from the other. The reloading process of an empty buffer is initiated by sending a signal to the associated producer process. This process then reads the following actions from the file on the disk and refills the buffer using the process file system mechanism (see section 3.3.1).

3.3.3 Packet trace format

For minimizing the amount of data to be transfered from disk over the user space to the kernel, a specific $4Byte$ instruction format has been developed (see figure 9). The first two bits of an instruction indicates the action to be taken (see table 2) while the remaining thirty bits specify the delay in micro seconds.

This enables us to code delays of more than $17 seconds$ which is more than enough since delay values are usually below $1 second$. As mentioned in section 3, if a packet has to be duplicated, a copy of it is replayed into the `packet_handler` and treated as the following packet. Like this, it will be processed with the ensuing instruction from the trace buffer. This offers high flexibility with minimal overhead. The delay of the replayed packet does not have to be included in the duplication instruction because it is taken from the following instruction. This trace format even makes possible to multiply packets by cascaded duplication actions.

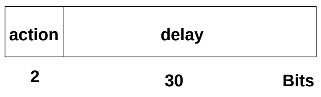

Figure 9: *Action trace instruction format*

Action	Code
Drop packet	00
Normal packet	01
Duplicate packet	10
unused	11

Table 2: *Action code in packet trace*

3.4 Configuring probability distributions

In addition to reproducing network dynamics from large packet action traces, the tool additionally offers a capability to generate network dynamics from the simple mathematical model introduced in section 2. This model employs three random distributions to generate values for delay, drop, and duplication. These distributions are coded in small tables that fit into the kernel. In addition to that the expected value μ, standard deviation $sigma$ and lag one correlation ρ can be explicitly configured for each of the three distributions. This functionality is essentially comparable to what Nist Net offers.

We thus summarize this section as follows: The architecture of our network emulation tool is based on

- intercepting the Linux TCP/IP stack between layer 2 and 3
- a emulator kernel module that is registered as a layer 3 packet handler and performs time critical task of delaying packets.
- a trace-reader module that employs the process file system to load chunks of large packet action traces into the kernel that drive the emulation.
- a simple mathematical model that offers the possibility to drive the emulation in situations where packet traces are not available.

4 Performance Evaluation

Next we analyze performance limitations and delay precision in trace-based emulation mode. For performance limitations, we essentially address the questions of how many incoming IP packets the emulator can process in trace-based emulation mode and whether the trace reader thwarts the emulator module. For delay precision, we address the question of how accurately the emulator can reproduce the delays listed in a given packet action trace. The methodology we use is taken from Jain [9].

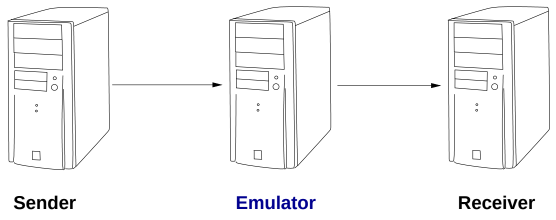

Figure 10: *Evaluation Setup*

Results reported here were achieved on a Dell Precision 340 Workstations with Intel Pentium P4 2.0GHz, 512MB RAM, 40GB Hitachi Deskstar 120GXP, two 3Com Tornado 3c905C NICs running Debian Sarge 2.4.27.

4.1 Performance limitation

With the maximum performance evaluation we show that the emulator operates untainted under stress conditions and that the trace reader module does not thwart it. For this purpose, an increasing amount of packets is sent to the emulator while monitoring the CPU consumption and the proper operation of the emulator.

For testing, the sending host transmits a constant stream of UDP/IP packets with no payload ($54 Bytes$, see figure 11) to the emulator. The most meaningful scenarios with the corresponding results are represented in table 3 and 4.

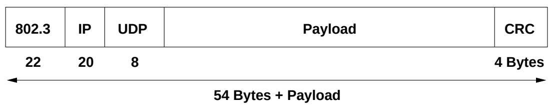

Figure 11: *Ethernet frame containing IP and UDP*

Packets [1000/s]	CPU consumption [%]	Buffer underrun
50	48	no
100	89	no
105	92	no
110	96	yes

Table 3: *Performance limitations with configured a delay of 121μs ($\approx 1 tick$)*

Packets [1000/s]	CPU consumption [%]	Buffer underrun
100	70	no
110	75	no

Table 4: *Performance reference with emulator switched off*

They show that rates below $105000 packets/s$ smoothly pass through. But for higher rates, the CPU faces its limitations. This makes it impossible for the supplier process of the trace reader module to refill the empty buffers resulting in a buffer underrun. This is because kernel tasks such as interrupts always have higher priorities than user level processes preventing them from being executed.

Hence we conclude that the hardware is the limiting factor and that the trace reader module does not thwart the network emulator.

4.2 Delay precision

In the delay precision evaluation, we show with measurements and analytical considerations that the emulator operates with a precision of $242\mu s$ ($\approx 2 ticks$). For this purpose the delay precision error of various different traces is measured and compared.

The tests were executed using a hard real-time measurement infrastructure based on commodity PCs with a precision of $3\mu s$ [17]. A sender transmits a packet stream through the emulator to a receiver (see figure 10). For this evaluation, a constant IP packet emission rate of 8000 per second containing UDP segments with a payload of $32 Bytes$ has been used because this rate is close to the emulator's resolution of the packet delay timer. For handling the stream, the following trace patterns have been employed.

1. Emulator off (for reference reasons)
2. Delay configured to $0\mu s$
3. Delay configured to $121\mu s$ ($\approx 1 tick$)
4. Synthetic patterns (constant ascending delays, 10 packets per $121\mu s$ / constant declining delays, 10 packets per $121\mu s$ / fast changing delays)
5. Traces from active network probing (see figure 12) [17]

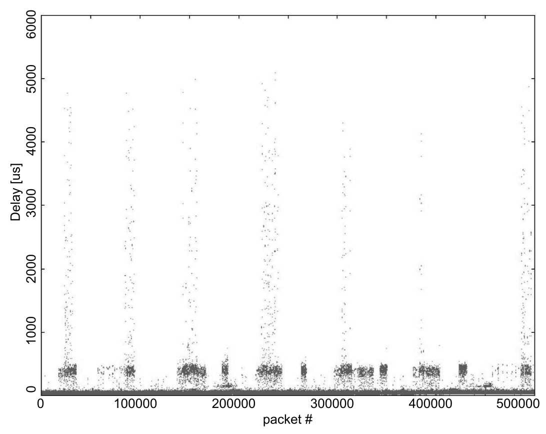

Figure 12: *Exemplary delay trace from active network probing in a switched MAN [17]*

If the emulator is switched off, the normally introduced processing delay error is not significant (see figure 13). But two phenomenas are remarkable. First, the accumulation at $20\mu s$ derives from processing of interrupts with higher priorities than the one of the network interfaces. Second, approximately 1 of of 10000 packets has a delay error up to $1 tick$. This phenomena has also been observed and extensively

discussed with the NIST Net emulator and can be ascribed to scheduling operations of the kernel [15]. As expected, running the emulator with no additional delay performs almost similar because the packets are directly forwarded by the `packet_handler` (see section 3.3.2) (see figure 14).
But, if we add a constant delay of one tick, the picture looks slightly different (see figure 15). In addition to the artifacts described above, the delay errors are randomly distributed between $-121\mu s$ and $0\mu s$ ($\approx 1tick$). This smearing can be derived completely to the emulator's timing accuracy. If a packet arrives just after a timer interrupt, it has to wait almost for a full tick before it is picked up by the emulator's timer (section 3), resulting in an effective delay of approximately $121\mu s$. But if a packet arrives shortly before a timer interrupt, it is almost immediately picked up, resulting in an effective delay of approximately $1\mu s$ or a delay error of $-120\mu s$.
This considerations concerning the timer inaccuracy and the artifacts originating from the kernel leads to a accuracy of $2ticks$ ($\approx 242\mu s$) for the emulator.

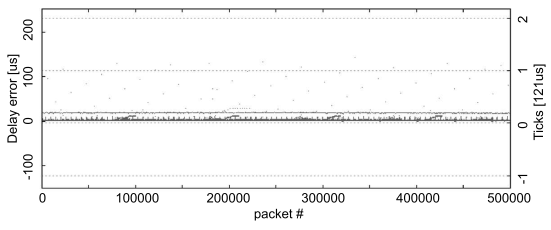

Figure 13: *Forwarding delay error - emulator off*

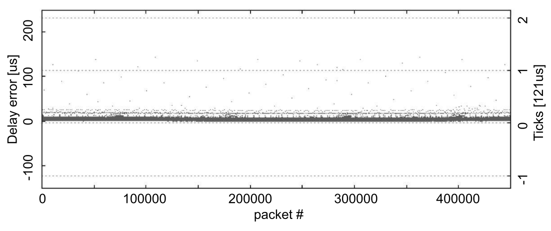

Figure 14: *Forwarding delay error - delay configured to* $0\mu s$ *delay*

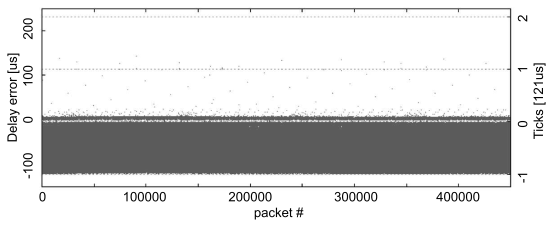

Figure 15: *Forwarding delay error - delay configured to* $121\mu s$ *delay*

In the tests with real traces from active probing in MANs [17], we used various different traces with small and large delays. We denote that cross traffic on this MANs is long-range dependent (see figure 2). Exemplary for them, following a typical result is presented. The delay errors look quiet similar to the one in figure 15 but with one exception (see figure 16). The delay errors are shifted up by $61\mu s$ ($\approx \frac{1}{2}tick$). This arises from arithmetic rounding in when translating μs to $ticks$ but has no impact on the overall concision. Rudimentary, also an influence, of the amount of buffered packets, on the delay error can be observed.

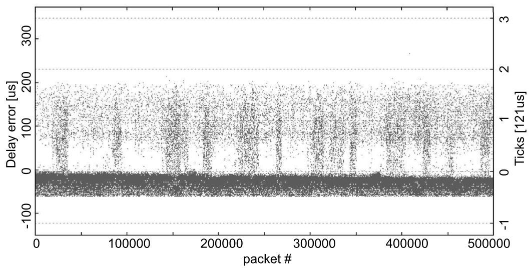

Figure 16: *Forwarding delay error - exemplary delay trace from active network probing in a switched MAN [17]*

We summarize our results on performance limitations and delay precision as follows: The performance on how many incoming IP packets the emulator can process is essentially limited by the CPU of the underlying hardware and not by the emulator itself. The trace-reader module does not thwart the emulator. The delay precision that specifies how accurately the emulator can reproduce the delays listed in a given packet action trace can be given with $242\mu s$ which is $2ticks$ of the Linux operating system.

5 Conclusion

In this paper, we have described the design and implementation of a network emulation tool that is capable to reproduce dynamics of real networks to test the performance of network sensitive applications, devices, and protocols. Due to the ability to replay large packet traces, the tool is capable to account for traffic characteristics such as long-range dependence which are key in performance evaluations. The tool's architecture is based on a combination of a kernel kernel modul that performs the time critical task of network emulation and a user space module that successively transfers packet traces into the kernel to drive the emulation. Moreover, the tool also supports to drive the emulation by configuring probability distributions. An evaluation of the tool's performance shows that it can reproduce packet delays with an accuracy that is of the order of 100 microseconds which is significantly lower than typical network delays.

Our experience and the feedback from our industry partner indicate that our emulation tool can be employed for a wide variety of testing purposes including the development and evaluation of applications, devices, and protocols for VoIP, circuit emulation, video streaming and conferencing, and distributed simulation/interactive gaming in both wired and wireless scenarios.

In the future, we plan to port the emulator onto real-time operating systems such as RTLinux [7] or RTAI [2] which leads to an accuracy of 10 microseconds when reproducing packet delays. Moreover, we plan to extend the emulator with functionality to emulate Ethernets, WiMax etc. on layer two.

Finally we would like to stress that in addition to the conceptual issues discussed in this paper, the emulator is publicly available via *www.tik.ee.ethz.ch/netemul*.

6 Acknowledgments

We would like to thank Thomas Hug for implementing this emulator, our head of the research group Prof. Dr. Bernhard Plattner and our industry partner Siemens Schweiz. Additional acknowledgments go to KTI/CTI Innovation Promotion Agency of the Swiss Confederation for funding this project.

References

[1] Network emulation with the ns simulator. http://www.isi.edu/nsnam/ns/ns-emulation.html.

[2] RTAI, Real-Time Application Interface.

[3] Jong Suk Ahn, Peter B. Danzig, Zhen Liu, and Limin Yan. Evaluation of TCP vegas: Emulation and experiment. In *SIGCOMM*, pages 185–205, 1995.

[4] J. Beran. *Statistics for long-memory processes*. Chapman and Hall, 1994.

[5] Jean-Yves Le Boudec. Performance evaluation lecture notes (methods, practice and theory for the performance evaluation of computer and communication systems). Lecture Notes EPFL, 2005. http://ica1www.epfl.ch/perfeval/.

[6] M. Carson and D. Santay. Nist net: A linux-based network emulation tool. In *ACM SIGCOMM Computer Commununications Review*, volume 33, pages 111–126, 2003.

[7] FSMLabs. RTLinux, Real-Time Linux.

[8] Brain Noble Mahadev Satyanarayanan Giao T. Nguyen, Randy H. Katz. A trace-based approach for modelling wireless channel behavior. In *ACM Press, Series-Proceeding-Article*, pages 597–604.

[9] Raj Jain. *The Art of Computer Systems Performance Analysis*. John Wiley & Sons, 1991.

[10] Oliver Thomas Lamparter. *Ein Softwaresystem fuer die Verkehrsdatenanalyse in Kommunikationsnetzwerken*. PhD thesis, ETH Zurich, 2003.

[11] The University of Utah. Emulab - network emulation testbed.

[12] Ohio University's Internetworking Research Group. ONE - the Ohio Network Emulator. http://masaka.cs.ohiou.edu/one/.

[13] K. Park and W. Willinger. Self-similar network traffic: An Overview. In *Self-Similar Network Traffic and Performance Evaluation*, chapter Introduction. Wiley-Interscience, NY, 2000.

[14] Kihong Park and Walter Willinger. *Self-Similar Network Traffic and Performance Evaluation*. John Wiley & Sons, Inc., New York, NY, USA, 2000.

[15] A. Pasztor and D. Veitch. A precision infrastructure for active probing. In *Proceedings of the PAM*.

[16] Ulrich Fiedler Rainer Baumann. A Tool for Emulating Real Network Dynamics on a PC. TIK Report 218. Technical report, ETH Zurich, Mai 2005.

[17] Rainer Baumann, Ulrich Fiedler. Active Probing of an E1 TDM Stream. TIK Report 217, Mai 2005.

[18] Luigi Rizzo. Dummynet: a simple approach to the evaluation of network protocols. *ACM Computer Communication Review*, 27(1):31–41, 1997.

[19] Mart Molle Thomas Karagiannis and Michalis Faloutsos. Long-range dependence, ten years of internet traffic modeling. pages 1089–7801. IEEE Computer Society, September 2004.