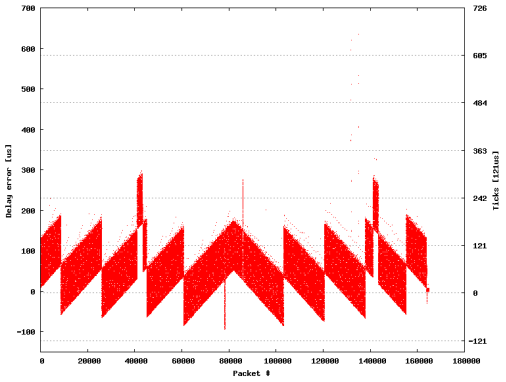
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Master's Thesis

Trace-Based Network Emulation



18th April 2005

Thomas Hug
<thomhug@ee.ethz.ch>

Tutor: Dr. Ulrich Fiedler, Rainer Baumann
Supervisor: Prof. Dr. Bernhard Plattner

Abstract

The test environment for sensitive network applications needs to be flexible. This is only possible in a lab setting containing a suitable network emulator. This thesis describes different possibilities for handling network emulation focussing mainly on emulators running on commodity PC's. It was found that none of the available open source emulators are able to correctly model network dynamics such as long range dependence. The most suitable existing emulator, NIST Net, was chosen and extended with a trace reader, which is able to read packet delays from different pregenerated sources like real network traces, simulation traces or calculus traces. A performance evaluation that proves the correct working and the sufficient speed closes the thesis.

Contents

Chapter 1

Introduction

Studying the impact of network dynamics under a wide variety of conditions is essential in order to assess the performance of network sensitive applications such as VoIP phones or circuit emulation adapters. Network emulation, i.e. the reproduction of network dynamics "in a box" (see figure 1.1), enables engineers and researchers to conduct performance studies in simple laboratory settings. In many settings, today's commodity PCs offer sufficient capabilities and performance. However, implementing network emulation requires

(i) kernel programming to prevent resource consuming user space processes from adding additional delays to the emulated network dynamic and

(ii) correct modeling of key statistical properties of network traffic such as long-range dependence among packet delays.

All known solutions for network emulation on commodity PCs fall short on either (i) or (ii). Luigi Rizzo's dummynet (FreeBSD) [4] works on the kernel level. However, it falls short on (ii), since dummynet can only do one of the following:

- add a constant delay to packets
- add the delay that comes from a bandwidth limitation on a single link
- add the delay that comes from weighted fair queuing on a router

NIST Net [1] is a network emulation package that runs as a Linux kernel module. However, it falls short on (ii), since NIST Net only uses a small distribution table to generate packet delays. Due to how the table is accessed, consecutive packet delays are either statistically independent or linearly correlated. This inherently leads to short range dependence. The simplest way

to cure this and to account for key statistical properties such as long-range dependence in network emulation is to replay packet delay traces, which reflect these properties, instead of generating packet delays.

Therefore, this thesis heads towards an implementation of a trace based network emulator that runs a Linux kernel module on a commodity PC. This emulator can then replay delay traces that were previously captured with a network protocol analyzer such a Ethereal [6], produced with a network simulator such as OpNet [7] or ns-2 [8] or generated by calculus. These packet delay traces can thus be easily verified to account for key statistical properties such as long-range dependence.

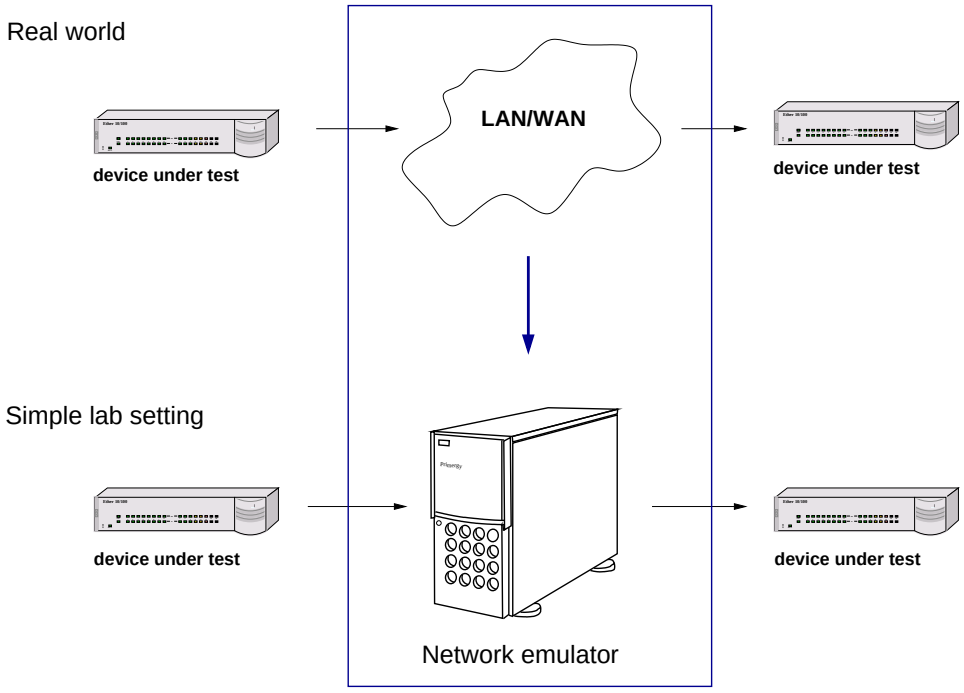


Figure 1.1: Network environment

To achieve this, NIST Net was extended with a trace reader that is able to replay network traces from a trace file.

A data format was defined in order to describe an action to be taken by NIST Net for each packet. This can result in delaying the packet with a given delay, duplicating the packet, or dropping the packet. 32 Bit per packet are used to describe these actions.

To copy the delay information from user space to kernel space, different methods were evaluated. The process file system showed to be the best solution concerning performance, complexity, and overhead.

NIST Net comes with a command line interface used to control the kernel module. This interface was extended to support trace files. Besides the normal add of a flow into NIST Net's flow table, a process that is responsible for sending the delay information from the trace files via the process file system to the kernel, is started per flow.

A buffer architecture in the kernel, which consists of two buffers per flow in the flow table of NIST Net, is responsible for keeping the delay applied to the packets by NIST Net and the reloading of the buffers from distracting each other.

A performance evaluation with focus on maximum performance and precision shows that NIST Net with our trace reader extension is fast enough to test sensitive network applications like VoIP or network devices like circuit emulation adapters. With UDP packets with a size of 54 Bytes and no payload, the software was able to process 105'000 packets/s. The resulting precision is below 2 timer ticks of the timer used by NIST Net (Δ 242μsec).

Trace-based network emulation opens a wide area of other applications than just VoIP and circuit emulation adapters. Streaming applications, network and wireless protocol testing, and network prototyping are examples of categories that the enhanced version of NIST Net can be used for.

This report is organized as follows: Chapter 2 describes existing emulators and explains why it was chosen to enhance NIST Net. In chapter 3, the functionality of NIST Net is described. Enhancement ideas and evaluations of NIST Net are explained in chapter 4. Chapter 5 describes the implementation of the enhancement in NIST Net which is tested with a performance evaluation in chapter 6. Chapter 7 with ideas for possible applications ends this report.

Chapter 2

Related Work

This chapter describes existing emulators. All of the existing emulators fall short on the correct modeling of key statistical properties of network traffic such as long-range dependence among packet delays.

2.1 Dummynet

Dummynet is a simple, yet flexible and accurate network emulator that is part of the FreeBSD firewall functionality in the kernel. It can be built with minimal modifications to an existing FreeBSD protocol stack, allowing experiments to be run on a stand-alone system. Dummynet works by intercepting communications of the protocol layer under test and emulating the effects of finite queues, bandwidth limitations and communication delays. It runs in a fully operational system, hence allowing the use of real traffic generators and protocol implementations, while solving the problem of simulating unusual environments. Dummynet allows experiments with network protocols simply with running the desired set of applications on a workstation. A FreeBSD implementation of dummynet, targeted to TCP, is available from Luigi Rizzo [4].

However, the current version has some drawbacks and limitations. It can only add a constant delay to packets, add the delay that comes from a bandwidth limitation on a single link, and add the delay that comes from weighted fair queueing on a router.

2.2 NIST Net

NIST Net is a general-purpose network emulator for emulating performance dynamics in IP networks. NIST Net is designed to allow controlled, reproducible experiments with network performance sensitive/adaptive applications and control protocols in a simple laboratory setting. By operating at

the IP level, NIST Net can emulate the critical end-to-end performance characteristics imposed by various wide area network situations (e.g., congestion loss) or by various underlying subnetwork technologies (e.g., asymmetric bandwidth situations of xDSL and cable modems).

NIST Net is implemented as a kernel module for Linux and uses a command line interface or X-System based interface to control the application. In use, the tool allows a commodity PC-based router to emulate numerous complex performance scenarios, including: tunable packet delay distributions, congestion and background loss, bandwidth limitation, and packet reordering / duplication. NIST Net also provides support for user defined packet handlers to be added to the system. Examples of the use of such packet handlers include: time stamping / data collection, interception and diversion of selected flows, generation of protocol responses from emulated clients. NIST Net uses a timer with a clock tick rate of 8192 Hz [1].

The problem with the current version are the limited capabilities for modelling network delay. Generated delay values are either independent or short-range dependent.

2.3 ONE - the Ohio Network Emulator

The Ohio Network Emulator (ONE) works on a Solaris-based workstation. The user can control various features of the network, such as propagation delay, queueing characteristics and bandwidth. In addition, the tool interfaces with Satellite Toolkit (STK) to emulate variable propagation delays based on the orbits of satellites. ONE is CPU intensive and runs as a user space program. It uses Solaris' clock with a tick rate of 1000 Hz [5].

2.4 Conclusion

None of the described network emulators are able to model correctly the key statistical properties of network traffic and use at the same time kernel programming.

NIST Net is considered the most advanced tool and was chosen for an enhancement.

Chapter 3

Functionality of NIST Net

NIST Net [1] is a network emulator and runs as a Linux kernel module. It extends the functionality of the Linux kernel by adding packet delay, packet drop, and packet duplicate functions. Packet delays can be constant or generated according a small distribution table in the kernel. NIST Net works on a per flow basis and can apply its functions to separate flows which differ in source and destination IP with their corresponding ports.

In section 3.1, the packet interception by the Linux kernel is discussed. Section 3.2 describes the buildup of NIST Net's Fast Timer. Section 3.3 finishes with a description of how the command line interface communicates to the kernel module.

3.1 IP Packet handling

Figure 3.1 shows an overview of the way a packet travels through the Linux kernel and NIST Net from the incoming to the outgoing network interface. The red box is handled by the NIST Net kernel module.

3.1.1 The Linux protocol stack

If a packet arrives on the network interface, it is copied into the queue in the devices memory. A hardware interrupt informs the kernel code, which copies the data into a socket buffer (sk_buff). The packet is then treated by the function to determine the layer 3 protocol (`eth_type_trans()`). The determined protocol is sent to a function that informs the Linux networking code (`netif_rx()`). `netif_rx()` is also responsible to find the layer 3 handler. Next, the packet is sent into a queue per processor, where a software interrupt request ensures that the correct function picks up the packet dependent of the result of `netif_rx()`. In the case of an IP packet, it is the `ip_rcv()` function. The initialization routine of the NIST Net kernel module replaces this Linux IP handler function (all `ETH_P_IP` Inter-

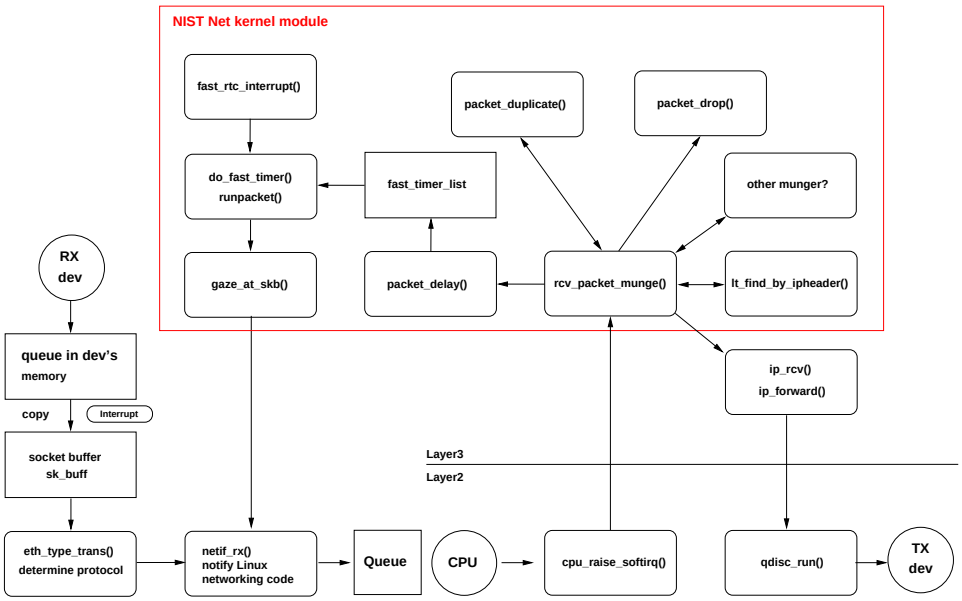


Figure 3.1: Embedded NIST Net kernel module

net Protocol packets 0x0800 (Ethernet frame's protocol field)) with its own function `rcv_packet_munge()`. Every new IP packet is handed to the NIST Net kernel module before `ip_rcv()` gets called.

3.1.2 The NIST Net kernel module

If the NIST Net kernel module is loaded, the `rcv_packet_munge()` function is called when an IP packet arrives. The function first checks whether the NIST Net emulator is on and whether the packet has not yet been checked (See 3.2 for explanation for packets arriving a second time). If the emulator is off or the packet has already been marked by the NIST Net code, the packet is immediately sent back to original IP handler function.

`lt_find_by_ipheader` looks up the corresponding entry in the NIST Net table (see section 3.3), which contains a list with source destination combinations and their corresponding action (i.e. delay, drop, duplicate).

If there are other munger modules registered in the NIST Net package (`linmunge.o` or `knistspy.o`), they also get executed. The two modules linmunge and knistspy are used to spy a single point-to-point connection or to monitor certain traffic. These extensions are not covered by this thesis.

Drop and duplicate are self-explanatory. Delay values due to bandwidth limitations or customized guidelines are calculated by the `packet_delay()` function. All packets that have a delay greater than zero are put on the fast timer list (See 3.2). This list is a sorted queue of packets which have to be

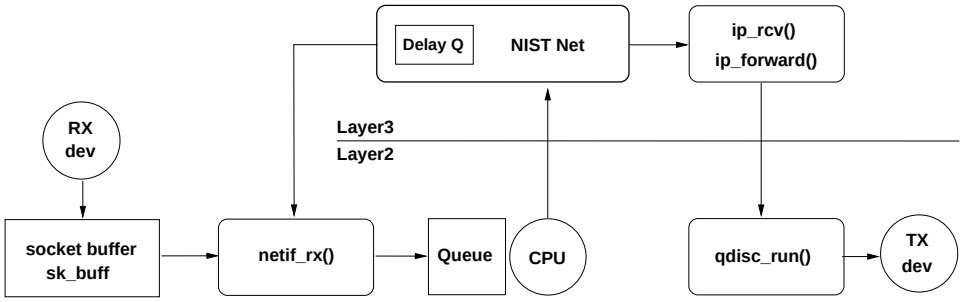


Figure 3.2: Packet interception in the Linux kernel

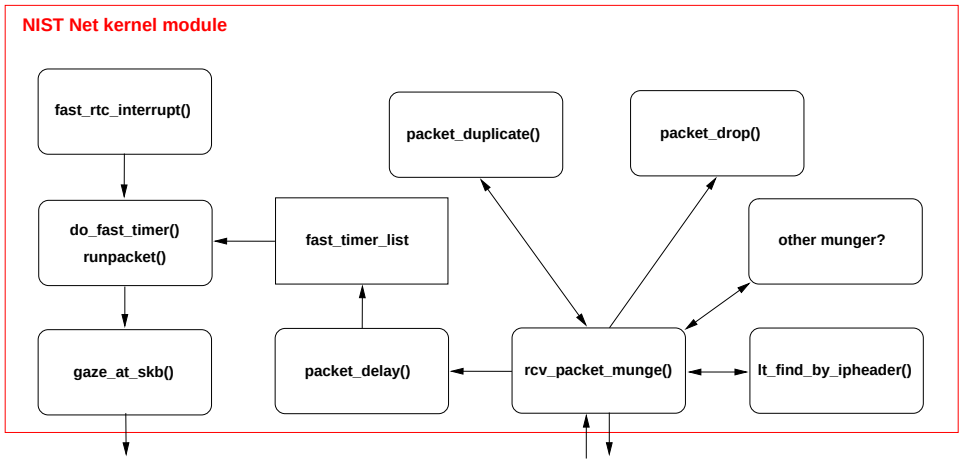


Figure 3.3: NIST Net kernel module

delayed.

The original NIST Net code is only capable of calculating delays based on random driven lookup tables. Packets with no delay are sent back to the original handler function for IP packets.

The following section describes the path the packets take from the `fast_timer_list` back to the networking part of the Linux kernel.

3.2 NIST Net's Fast Timer

To apply given packet delays to packets within a packet stream, a fine-grained timer source is needed. NIST Net uses the MC146818 real-time clock (RTC) as the timer source. After loading the NIST Net kernel module `knistnet.o`, the `init_module()` function configures the timer chip to run at 8192Hz, its highest possible interrupt rate, and registers the func-

tion `fast_rtc_interrupt()` as the corresponding interrupt handler. This setting allows a tick granularity of approximately 122 μsec [1].

`init_module()` also allocates memory for the `fast_timer_list`, the list that holds all timer entries in it. At startup, 1024 slots for packets are allocated (36K). The maximum allowed usage is 616K (17344 packets), not counting the space used up by the skbuffs.

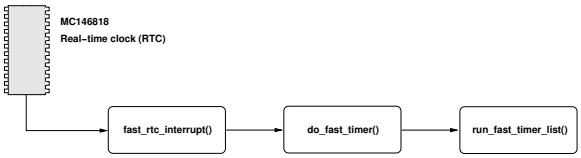


Figure 3.4: Real-time clock (RTC)

Every function call of the interrupt handler calculates the new expires value and sets it if required. It is required only when it is not already set. Finally, the `do_fast_timer()` function is called, which checks and adapts the number of lost miniticks (interrupts by the RTC) and runs the final scheduler function `run_fast_timer_list()`. The scheduler checks for expired packets in the fast timer list, detaches the expired ones and put them into a run queue.

Every packet on the runqueue is then marked as processed and sent back to to the Linux kernel's `netif_rx()` function.

3.3 Communication with Command Line Interface

The NIST Net module uses a character device for its communication with the user space. A character device is a device file used for communication purposes between user space and kernel space. This character device is registered within the kernel with a major number, which is used for the unique identification by the user space.

For every character device there are pre-defined generic functions (open, read write, release, ioctl, ...), which can be used by the user space. While registering the character device by the kernel, also the correspondent kernel function for each of the needed pre-defined generic functions will be registered. If the user space executes an open syscall on a character device, the corresponding user_open function, which is registered during the initialization of the module, gets called.

Besides the self-explanatory functions like open or release that perform the initializations and management, other functions like read is used to read out the whole NIST Net table (see below) from the data structures in the kernel and write is used to handle packet distribution tables.

Ioctl (short for Input Output ConTroL) is of importance for NIST Net. Every control action is handled by the ioctl function of the character device. It can be used to switch the emulator on or off, control debugging, flush tables, gather statistics, and/or add/remove entries to the NIST Net table.

The NIST Net table is the most important part in the kernel module. It contains a list of all information as to how IP packets have to be treated. For every table row in the CLI, there is a corresponding entry in the NIST Net table in the kernel. This entry consists of a source and destination IP address, delay time, delay sigma, bandwidth limit, drop probability, duplicate probability, DRDmin, DRDmax, and DRDcongest (see [1] for explanations of DRD).

Whenever data is to be transferred from user space to kernel space in order to update the NIST Net table or gather statistics, only a pointer to a struct in the user space is sent to the ioctl function. The ioctl function in the kernel then copies the data via `copy_to_user` or `copy_from_user` between kernel and user space. The reason for the need to invoke these copy functions is described in section 4.2.

Chapter 4

Enhancing NIST Net

4.1 Requirements

NIST Net [1] falls short on being able to model correctly some key statistical properties such as long-range dependence. This is because it makes use of only a small distribution table when generating packet delays. In order too avoid this problem, NIST Net's functionality needs to be enhanced to be able to use pregenerated delay values. The enhancement should provide NIST Net with the capability to replay packets from pregenerated trace files. It is assumed that the maximum speed of a packet stream running through NIST Net is 100 MBit/s.

Because NIST Net is running as a kernel module and trace files are stored in user space, a suitable and "fast enough" transport from user space to kernel space has to be found. An intelligent queueing algorithm within the kernel should satisfy acceptable latency for fetching packet delays for passing packets.

This chapter deals with the different types of communication between user space and kernel space and proposes a queueing algorithm to use within the kernel. A short discussion about preemption in Linux 2.4 kernels ends the chapter.

4.2 Communication between user and kernel space

Memory Mapping, Process File System, and Character Devices are discussed in the following subsections. Socket communication is only mentioned as an option, but not discussed in detail.

ProcFS and Character Devices both take advantage of `copy_from_user` while Memory Mapping accesses the data directly. The reason for the need to invoke these copy functions lies within the fact that Linux memory is segmented. This means that a pointer does not reference a unique location in memory but only a location in a memory segment. The kernel needs to

know which memory segment it is to be able to use it. There is one memory segment for the kernel and one for each of the processes.

4.2.1 Memory Mapping

The kernel sets up a special memory region that is associated with some portion of either a regular file in a disk-based file system or a block device file. Each access to a byte within a page of the memory region is translated by the kernel into an operation on the corresponding byte of the file.

The memory manager stores a mapping of virtual to physical addresses on a per-process basis, and also stores additional information on how to fetch and replace particular pages. This information is stored in a memory-map data structure that is stored in the process scheduler's task list.

Two kinds of memory mapping exist. Shared memory mapping and private memory mapping. Shared memory mapping is used when multiple processes access the same region. As soon as a process writes into a page of shared memory, the changes are visible to all other processes that map to the same file and the page on the disk is changed. With private memory mapping there is only one process writing and one or more reading. This kind of mapping is therefore more efficient because the pages have not to be immediately written to the disk if the process writes to the mapped memory.

```
mmap (caddr_t addr, size_t len, int prot, int flags,
int fd, off_t offset)
```

`mmap()` can be called from user space or from kernel space. The maximum amount of virtual address space available on 32 Bit systems for a process is 3GB, the remaining 1GB is reserved for the kernel. These limitations come from the fact that a 32 Bit system can only address 4GB. Note that the partitioning can be changed with a kernel patch to i.e. 2GB/2GB.

`mmap()` within the kernel demands the allocation with `vmalloc()` of memory first. Since `vmalloc()` is limited to the size of RAM, this is probably less than 3GB in user space.

The `VM_LOCKED` flag prevents the memory area from being swapped out. This is especially important if a user space process writes to a region of the kernel.

`mmap()` is faster than the other methods because there is no need of `copy_from_user`. But since the maximal size is fixed and the part of the region that is read next has to be in memory, memory mapping is complex to handle. If it is not possible or too complex to guarantee that the part of the region that is read is in the memory, it is mandatory to use non-swappable buffers inside the kernel. This gives memory mapping more disadvantages than advantages.

4.2.2 Process File System

The Process File System (procfs) is a special file system (/proc) in the Linux kernel. It is a virtual file system, which is not associated with a block device and exists solely in memory. The files in the procfs allow user space processes access to certain information from the kernel. It is used for process information, debug, and configuration purposes.

Instead of reading (or writing) information directly from kernel memory, procfs works with call back functions. As soon as a user process reads or writes to a file in /proc, a dedicated function is called. These functions can be registered for every file in the procfs after creation.

The following struct of a procfs entry shows the two handler functions.

```
struct proc_dir_entry* entry;
entry->read_proc = read_proc_foo;
entry->write_proc = write_proc_foo;
```

While a read from the procfs simply invokes the sprintf function in the kernel to send data to the user space, a write operation to the procfs is more complicated. Because the process accessing the procfs is normally in the user space, the write function for the corresponding procfs entry has to invoke the `copy_from_user()` function first.

An example that demonstrates the concepts and functions of procfs can be found in the process file system guide [3].

A write operation to a file in the procfs invokes the registered `write_proc` function. The most important arguments are a pointer to a buffer in user space and the size of the buffer. If the amount of data is too big for the handler function, it can just take a part of the data and return a lower value than the size of the whole buffer. This forces the kernel to call the handler function again with the rest of the data. The user space process that feeds the data to the kernel has not to care about buffer overflows. The kernel provides a buffering functionality already.

4.2.3 Device files

Kernel modules can register device files that are able to be used to communicate with the user space. A list of the major numbers of the devices that the kernel can handle is available in the file `/proc/devices`. All functions for the possible device operations (open, release, write, read, ioctl) are registered within the module. If a user space process opens the device file, the corresponding function is executed. Commands like ioctl (short for Input Output ConTroL) take arguments and it is therefore possible to hand a pointer to the kernel from the user space and transfer data to kernel space if the ioctl function in the kernel invokes the `copy_from_user` function.

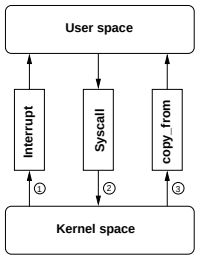


Figure 4.1: Signalling

The communication from kernel space to user space is handled by interrupts.

4.2.4 Netlink Sockets

Netlink is used to transfer information between kernel modules and user space processes. It provides bidirectional communication links and consists of a standard socket based interface for user processes and an internal kernel API for kernel modules. It was made for communication signals and not for the transport of big amounts of data. It is therefore not considered to be an option.

4.3 Measurements

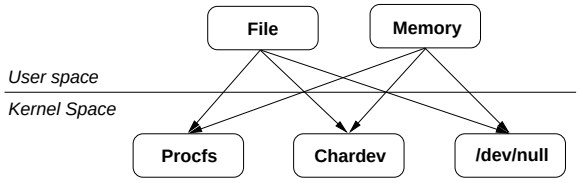


Figure 4.2: Ways to copy data

To find the suitable communication method, this section evaluates the data transfer rate of the procfs and device file alternative. Mmap isn't further tested because it's higher complexity for this purpose.

The tests were processed on a Dell Precision 340 Workstation with an Intel P4 2.0 GHz, 512 MB RAM and a 40 GB Hitachi Deskstar 120GXP running Debian Sarge with Kernel 2.4.27.

Two sample kernel modules have been programmed to support the tests. Figure 4.2 shows an overview of the alternatives that have been evaluated.

The focus was set to the direction from the user space to the kernel space, because this is the dominant one.

The task was to copy 1 GB of data from the user space to the kernel space. For each of the two methods, a kernel module was programmed that copies the data that is fed in, into a buffer inside the kernel. Additionally, the same test was repeated with the /dev/null device.

The test file was read sequentially from the hard disk. The measurement results in figure 4.3 show, that all three methods use almost the same amount of time to complete the task. The reason lies in the fact that the hard drive operates at its maximum speed and builds the limit.

The worst case used 38.26 seconds to copy 1 GB of data, which is a minimum speed of 219.25 MBit/s. With the defined trace file format (see section 5.2), a packet delay uses 32 Bit. This results in a theoretically calculated packet rate of 6.85 Mio packets/s. Because the size of an IP packet (including minimal UDP header and 802.3 header, see figure 4.4) without payload is 54 Bytes, the theoretically processing speed with this packet rate is 2.95 GBit/s.

	Procfs	**Device files**	**/dev/null**
Average	36.91s	37.29s	37.29s
Std. deviation	0.37s	0.27s	0.39s
Max	37.79s	37.70s	38.26s
Min	36.54s	36.72s	36.80s

Figure 4.3: Results for 10 measurements of copying 1 GB of data

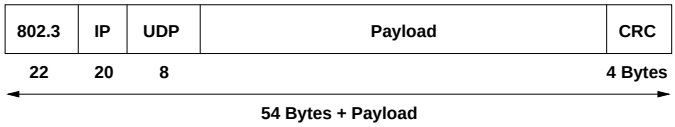


Figure 4.4: IP packet with UDP header

The implementation for copying data from user space to kernel space doesn't depend basically on the chosen alternative. Both alternatives have nearly the same data rate. The bottleneck arises from hardware limitations. Nevertheless, because the additional resources for handling IP packets were not taken into account for, the measured data and packet rates in these tests are not significant.

Figure 4.3 shows a summary of the communication techniques. Mmap is the fastest, however, it is not very flexible since it is most complex to handle. With the additional complexity it has also the most overhead. Procfs and device files are fast enough and are not too complex. Netlink sockets

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

Figure 4.5: Summary of the communication techniques

weren't further tested. Since Procfs has less complexity than device files, it is considered the best option to use.

4.4 Queueing in the kernel

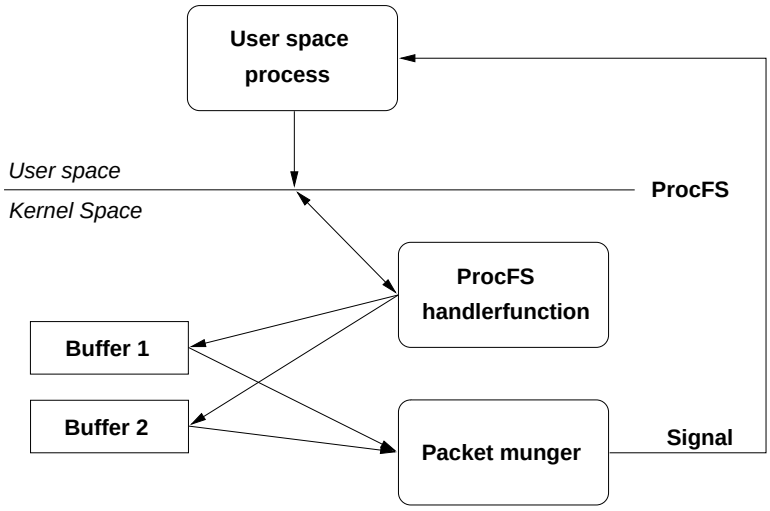


Figure 4.6: Queueing and data transport

The target of the queueing mechanism in the kernel is to provide fast access to trace file data for the NIST Net packet munger functions. A fast access without cache misses is needed to satisfy the needs of a near real time system. Every incoming IP packet that belongs to a connection in the NIST Net table uses 32 Bit information from the trace file to be treated.

To provide the ability to read with a constant data rate out of a buffer, a special queueing mechanism is needed. Since the data to be used originates from the user space, an intelligent coordination system must satisfy the availability of uncached data in the kernel.

Fast access to data is possible if the data is first copied to a buffer inside the kernel space. The data stream remains un-interrupted when two buffers are used and refilled alternately or when a ring buffer is used. Both

possibilities require that background jobs refill while the packet munger function of NIST Net takes data out of the buffers and empties them. While a ring buffer additionally needs a complicated coordinated handling between the reader and the writer. Therefore, it is much easier to use two buffers instead of the more complex ring buffers.

4.5 Preemption in Linux kernel 2.4

A preemptive operating system is able to stop any process at any point and reschedule it. The Linux kernel 2.4, as it is used for NIST Net, is non-preemptive for kernel tasks. The scheduler is in this case not able to reschedule a task while it is running in the kernel space. Kernel code is scheduled cooperatively, not preemptively. Kernel code runs until it finishes (returns to user space) or explicitly blocks. It isn't possible to preempt a task at any point.

With Linux kernel 2.6, preemption is possible when enabled as an option. However, rescheduling is only possible while the kernel is in a "safe to reschedule" state.

It is therefore recommended to do as less as possible in kernel space to satisfy the priority of the important tasks like treating IP packets. Open files directly in kernel space is not an option.

Chapter 5

Enhanced NIST Net Implementation

This chapter describes the enhancement of NIST Net with the trace reader extension. Different parts are discussed separately, namely the trace generation, the trace file format, the command line interface, the data transport from the user to the kernel space, the queueing in the kernel, the delay application, and the functional verification of the software.

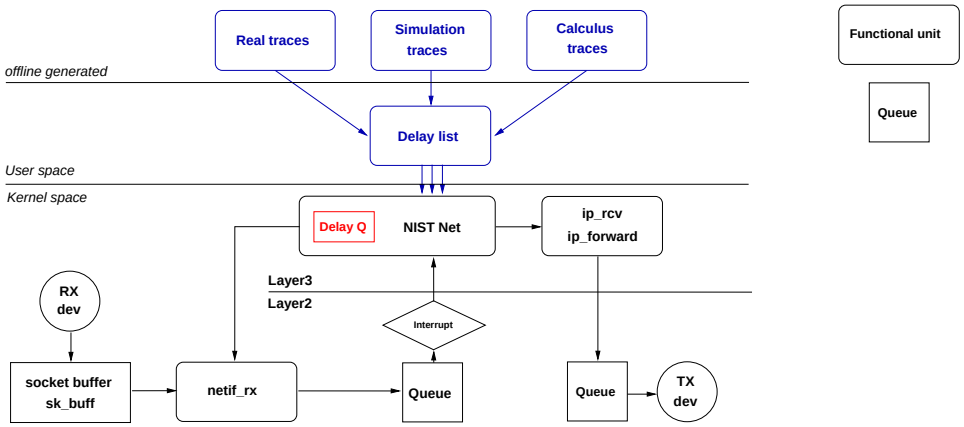


Figure 5.1: Enhanced NIST Net

Figure 5.1 shows the NIST Net kernel module with the additional parts in the user space and the different types of trace files possible. The kernel space section in the figure was described in section 3.1.2.

5.1 Trace generation

Three possibilities for generating trace files are shown at the top of figure 5.1. These traces are generated by real network probing, network simulation, and calculus. The actual generation of the traces themselves goes beyond the scope of this thesis and is not discussed here.

5.2 Trace file format

The trace file format covers the coding of information that comes from offline generated traces like packet delay, drop, and duplicate in a binary delay list. This binary delay list is then copied from the user space to the kernel space as seen in figure 5.1.

The determination of what type of the packet has to be fast. The coding is optimized to be used in combination with bit masks. This is known to be very fast.

The information for one packet consists of 32 Bit of data. Two bits are used to distinguish between a delayed packet (00), a dropped packet (01), and a duplicated packet (10). The left over case (11) is not used. One bit is used for the sign. Because of the simplicity, only signed integer values are used. The remaining 29 Bits represent the delay value. Triplicated packets have not their own coding because it is the same like two duplicated packets consecutively (see section 5.6).

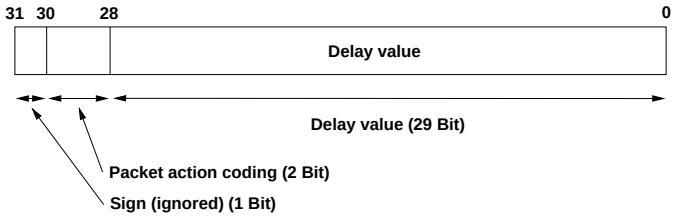


Figure 5.2: Data format

Code	Description
00	Normal packet
01	Drop packet
10	Duplicate packet
11	(not used)

Figure 5.3: Packet action coding

According to section 3.2 a timer tick is around 122μsec. Because the

time available for calculations is limited inside the kernel, it is considered to be faster to store the delay values already in a kernel compatible format. Since NIST Net makes these calculations for itself with `usec_to_minijiffy` in its `packet_delay` function, it is simpler to make use of this functionality and have the values stored in μsec. Additionally, μsec is portable because it is an absolute measurement that is the same on every architecture and on every processor

As already previously described, NIST Net uses a tick granularity of 122μsec. Using μsec for delay values allow the description of values from 0 to 536.87 seconds. This is more than enough since delay values usually do not exceed 1 second.

5.3 Command line interface

Figure 5.4 displays the entire trace reader extension. This section describes the controlling part, the command line interface (CLI). The other parts are treated in following sections in this chapter.

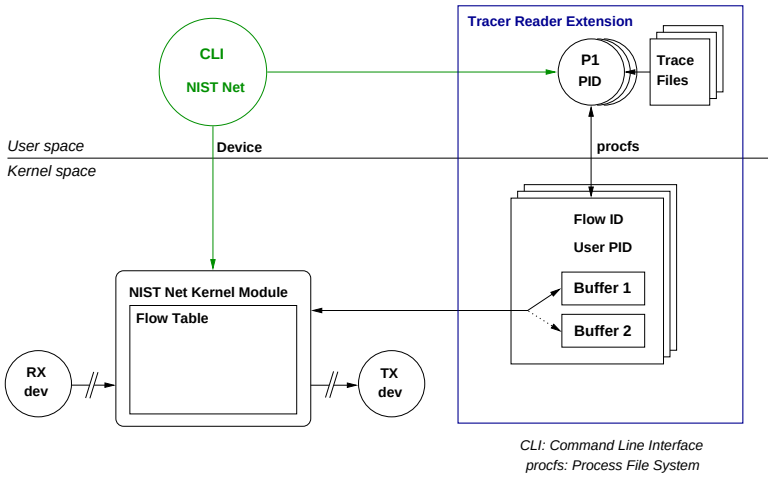


Figure 5.4: Trace reader extension

The original CLI is used for controlling the NIST Net kernel module, as soon as it is loaded. It enables/disables the entire module and adds/removes entries to/from the flow table in the module. The flow table contains an entry, one for each flow defining IP source/destination pairs with their ports and the actions to be applied on the matching packets.

Our enhanced implementation of NIST Net supports trace files. The additional action allowing a flow to get the delay values from a trace file is also controlled by the CLI. If a filename is given as an option when adding a new entry to the flow table, the CLI starts a process and allocates a

buffer environment in the kernel beside the normal procedure of just adding the source/destination pairs to the flow table. The started process that is sending data to the kernel space is called flow seed process.

When removing an entry from the flow table, the corresponding flow seed process is killed and the buffer environment in the kernel is deallocated.

The CLI can also be used to dump the flow table into user space. To see the buffer-status, a read call to `/proc/nistnet/input` provides information of about 20 internal counters for debugging. A write call to the same file resets the statistics.

5.4 Data transport user space to kernel space

The data that is stored in trace files, the format is described in section 5.2, is transported to the kernel space. The data transport is the part between the trace files and the kernel in figure 5.4.

As described in the previous section, the CLI starts a new flow seed process as soon as a new flow is entered into the NIST Net flow table. This process reads data out of a trace file, which has a size similar to a buffer in the kernel. A procfs write call fills therefore, an entire buffer.

During initialization, both buffers are filled. The starting of the flow seed process incorporates the filling of the first buffer. The first buffer then changes to the "ready" state. Because the second buffer is still empty, the flow seed process continues with a second procfs call that sends the same amount of data as it did with its first call. After that, both buffers are in the "ready" state and the kernel sends a SIGSTOP signal to the flow seed process, which keeps it from sending further data.

Since syscalls, like the write call to the procfs, are blocking, the SIGSTOP signal for the flow seed process is set before the process returns from the write call. The process stops suddenly as soon as the write call returns.

The procfs handler function in the kernel needs to know the process id of the flow seed process to send signals to. It also needs the corresponding flow id to determine which flow buffer the flow the data is for. This information is attached to each "package" that is sent to the kernel. Figure 5.5 illustrates such a "package" with several 32 Bit values in the first part, followed by the flow id and the process id.

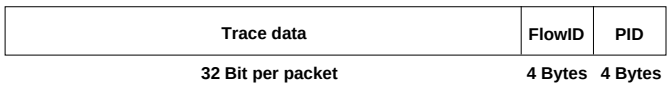


Figure 5.5: Data package with flow id and PID

The number 32 Bit values in a "package", containing packet action and packet delay, is usually above 1'000 packets. Further information can be

found in the Evaluation tests in Chapter 6.

5.5 Queueing in the kernel

Section 4.4 introduced how queueing is used and explained the reason why two buffers are used. This section explains how these buffers are implemented.

Since the enhanced NIST Net kernel module supports multiple flows in parallel, there is one buffer environment per flow. Figure 5.4 shows the buffers in the kernel space. Figure 5.6 shows one of these buffer environments in detail.

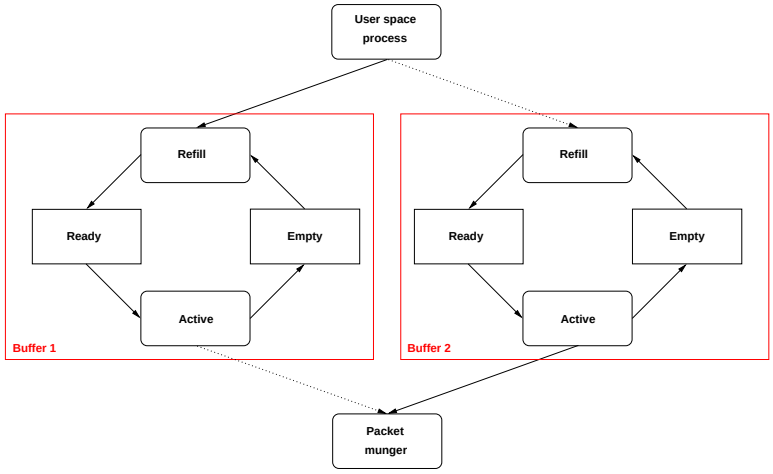


Figure 5.6: Alternately selected buffers

Section 5.4 described the process of initially filling the buffers once a new flow was added to the flow table by the CLI. Both buffers are in "ready" state after their initial load and may be used by NIST Net's packet munger function. A buffer pointer in the buffer structure points to the first buffer. Packet delays are read from the first buffer until the last packet in this buffer. After fetching the last packet of one buffer, the buffer pointer changes to the second buffer. The first buffer changes to the "empty" state, and the function sends a SIGCONT signal to the flow seed process in the user space of the corresponding flow in order to start the refilling. While the packet munger reads from the second buffer, the flow seed process sends a "package" to the process file system and the first buffer gets reloaded. Then, a SIGSTOP signal is sent to the flow seed process to prevent the process from sending more data. After finishing the reload process, the first buffer is again in the "ready" state. As soon as the second buffer becomes also "empty", the reload procedure repeats.

5.6 The delay application

The delay application is the last part in figure 5.4. It is the connection between the two kernel buffers and the NIST Net kernel module. Each packet that passes the NIST Net kernel module and matches a flow in the flow table attached to the trace reader extension, gets its delay value from trace data in a buffer. A matching packet allows the packet munger to call a function that decodes the action (delay/drop/duplicate) and the actual delay value from the 32 Bit value read from the buffer. The resulting action and delay value is then applied to the packet.

If a packet is to be duplicated, the whole sk_buf is copied and processed right after the original packet. Because the packet munger function is called a second time, the duplicated packet can have its own action and delay, different to the one of the first packet. Therefore, a triplicated packet can be generated with adding two duplicated packets consecutively.

5.7 Functional verification of the software

The extended version of NIST Net is able to make use of multiple flows. The CLI allows an operator to add and remove flows to a running system. The following tests are used to check as to whether the CLI works as it should and if the input is correctly applied to the flow table. Special attention was made to the extended part, the trace reader and its flows, when creating test cases.

Evaluation:

1. Add table entries with trace files
2. Update table entries with trace files
3. Remove table entries with trace files
4. Add/update/remove flows multiple times
5. Add/remove the same flow twice

All tests were completed successfully.

Chapter 6

Performance Evaluation

This chapter describes the performance analysis of the enhanced version of NIST Net. The steps for a performance evaluation study were processed as listed in Raj Jain [9].

Two evaluation studies build the main part of this chapter. The maximum performance test evaluates the maximum amount of packets that can be processed by NIST Net without error and the delay precision test evaluates the difference between a given and measured delay. The performance limitation of the tests is given by the hardware setup of the NIST Net box.

6.1 Evaluation Study

The goals of the study are to find the

1. Maximum amount of packets processing
2. Delay precision
3. Correct working of the software

The maximum number of packets that can be stored within the kernel is also of interest. With a limitation of a maximum delay of one second and a theoretical maximum network speed of 100 MBit/s, the machine must be able to store 12.5 MByte of packet data. Since todays machines have at least 512 MB RAM or even more, it is absolutely no problem to deal with.

The tests were processed on a Dell Precision 340 NIST Net machine, Intel Pentium 4 CPU 1.80 GHz, 512 MB RAM and a 40 GB Hitachi Deskstar 120GXP running Debian Sarge with Kernel 2.4.27. Both network interface cards were 3c905C Tornado cards from 3Com Inc.

6.2 Maximum performance

The goal of a maximum performance test is to evaluate hardware limitations. A delay of 121μsec according to the minimal tick period of the timer used by NIST Net guarantees that all NIST Net functions are processed at least once for each packet.

Two factors can be adjusted during the tests, the sender's packet rate and the flow buffer size in NIST Net. Two metrics are used to determine the maximum speed of a given flow buffer size. As soon as packets are dropped or the NIST Net flow buffers can not get refilled, the test fails.

Factors
1. Packets/s
2. Flow buffer size

Metric
a. Buffer under run
b. Drops

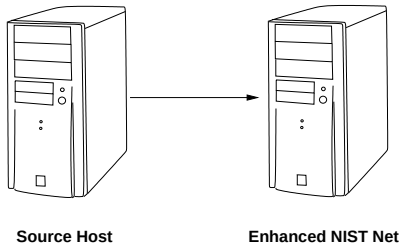


Figure 6.1: Evaluation Setup

Figure 6.1 shows the evaluation setup with two boxes. The source host sends a constant packet rate to the NIST Net box. With a network interface speed of 100 MBit/s and a minimum IP packet size of 54 Bytes (figure 4.4), a packet rate of 230'000 packets/s is the theoretical maximum.

Evaluation procedure:

1. 230'000 packets/s. Stop evaluation if no drops occur.

2. 0 - 230'000 packets/s. Evaluate beginning of occurring drops.
   Flow buffer size: 32 Kb (8192 packets)

3. 0 - 230'000 packets/s
   Flow buffer size: 16 Kb

4. 0 - 230'000 packets/s
   Flow buffer size: 64 Kb

6.2.1 Results

The configuration is able to process 105'000 packets/s.

Packets/s	CPU Load	Given Delay	NIST Net	Packet Underrun	Packet Drops
50'000	48%	121μsec	on	no	no
100'000	89%	121μsec	on	no	no
105'000	92%	121μsec	on	no	no
110'000	96%	121μsec	on	yes	yes
100'000	70%	0μsec	off		
110'000	75%	0μsec	off		

Figure 6.2: Results of maximum performance test

Packet rates above 105'000 packets per second do not work because of the heavy machine load. It makes it impossible to reload new packets in the buffers of the kernel. Kernel tasks such as interrupts always have higher priority than user level processes, therefore even processes with the highest priority do not get executed when there is too much work inside the kernel.

The tests with different sizes of flow buffers had no effect on the maximum possible packet rates and therefore are not listed. The reason for buffer under runs and drops is that the reload of the buffers is missing. The size of the buffer doesn't influence the fact as to whether or not a buffer is reloaded.

The total CPU time used without an interaction of NIST Net is shown below the double line in figure 6.2. These tests show that most of the time used is for handling interrupts, layer 2 and layer 3 processing before a packet finally enters the NIST Net code. Therefore the maximum amount of packets that can be processed depends highly on the hardware configuration used.

6.3 Delay precision

The delay precision tests measure the resulting variance added by NIST Net if NIST Net is for example used as a router. They also measure as to whether the added delays correspond to the ones given in the trace file. The controlling factors of these tests are the packet rate and the delays in the trace file to be added. The measuring metric is the delta of the delay.

Factors 1. Packets/s
2. Delay in trace file

Metric a. Δ of delay

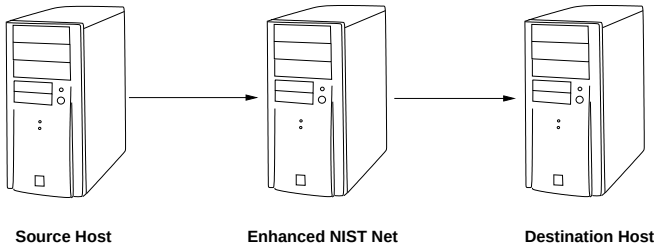


Figure 6.3: Evaluation Setup

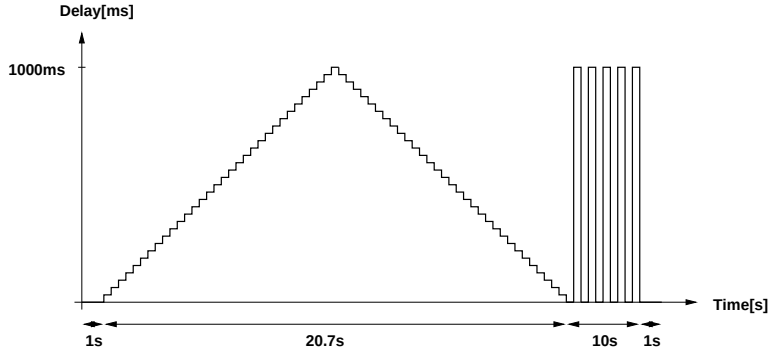


Figure 6.4: NIST Net evaluation

As seen in figure 6.3, an evaluation setup is used to process the tests. A source host sends data through a NIST Net box to a destination host. The sender and receiver are configured to use the NIST Net box as a default gateway and NIST Net itself is configured to forward packets between its interfaces.

The sender host sends an E1 stream with a constant rate of 8'000 packets/s in each test. The UDP packets (figure 4.4) have, including the payload of 32 Bytes, a total length of 86 Bytes which results in a network speed of 5.504 MBit/s [10].

The sending application is based on Real-Time Linux and writes a sequence number and timestamp in the 32 Byte payload of the UDP packet. The receiver host, which is also running Real-Time Linux, adds its own timestamp to the packet. Finally, an application on the receiver host dumps all packets into a file. This measuring application has a precision of 3μsec.

Once finished a single test, an especially for this task written software, evaluates the dump files, calculates the delay variance, and verifies the correctness of delays compared to the given delays in the trace file.

Evaluation procedure:

1. NIST Net off
2. NIST Net on, delay const 0 μsec
3. NIST Net on, delay const 121 μsec (one tick)
4. Delay times according figure 6.4
   (a) One second with 0 delay to calibrate the measuring hosts
   (b) Rising delay times, 10 packets per 121 μ sec tick
   (c) Falling delay times, 10 packets per 121μ sec tick
   (d) Fast changes to test the reordering of packets within the NIST Net Fast Timer List
5. Trace from network probing (multiplied by 10 because of too small granularity of NIST Net)

Only one pattern like the one in figure 6.4 was tested because of the limitation of the scope of this thesis. The delay is bound by a maximum of one second.

6.3.1 Results

The first test (figure 6.5) displays the delay generated by simply forwarding packets between two network cards. NIST Net is disabled and the kernel module is not loaded. The majority of the packets is forwarded without any distraction and has a low delay. The accumulation just above the majority results from the interrupt architecture. The network interface has not the highest priority and routines handling the packets get interrupted and delayed. The rest of the packets are randomly distributed and get more delays because of scheduling routines of the Linux operating system.

Figure 6.6 is similar to figure 6.5 except for the NIST Net kernel model that is loaded. Packets enter the NIST Net code but leave directly because the delay is set to 0μsec. Nevertheless, the packets path through the kernel is longer and the chance that the processing functions get distracted by a higher prioritized interrupt handler is higher. Therefore, the borders between majority around zero are not as sharp as without the loaded NIST Net kernel module.

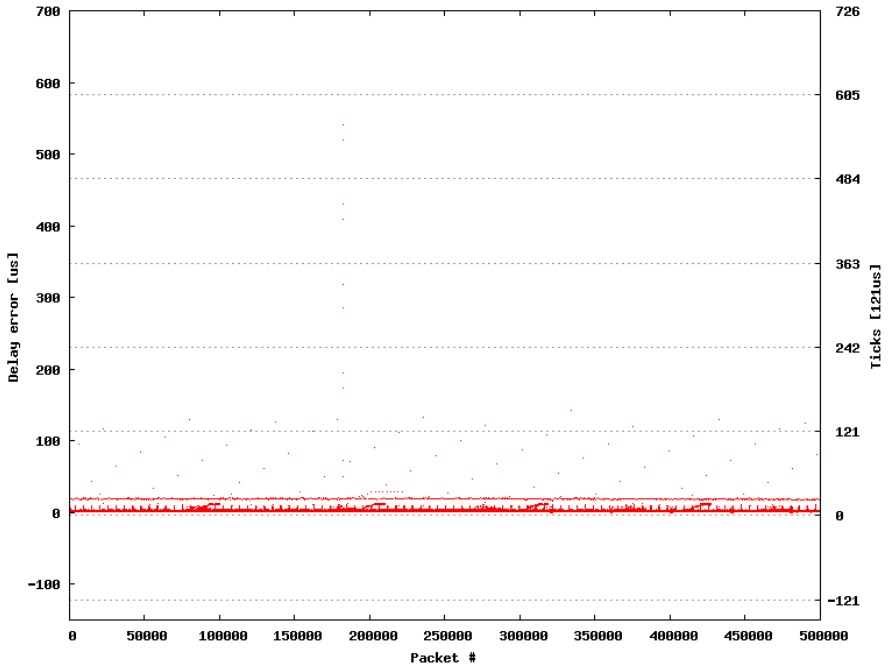


Figure 6.5: NIST Net disabled - IP forwarding delay

Figure 6.7 adds a delay of 121μsec to each packet passing through the NIST Net box. The majority of the packets get a delay error between -121 and 0μsec. All other packets get a random delay error as explained as a result from previous tests. The negative error results from the method that the timer in NIST Net is used. Packet delays get rounded off to the next timer interrupt. If a packet arrives just 1μsec before the next timer interrupt, the packet is sent to the queue for that interrupt. This results in a delay error of -120μsec. If the packet arrives just after a timer interrupt, the delay error is almost zero. Nearly zero means that the packet is delayed until the next interrupt, namely 121μsec.

Figure 6.8 shows the delay configuration for the test with increasing and decreasing delays according to the evaluation plan.

Figure 6.9 reflects the delay error resulting from using the delay configuration in figure 6.8. First, one can see the outstanding runaways resulting from the delay raises that are almost exactly as long as one timer tick. Because the timer tick is not always exactly 121μsec, the time runs through the border of one tick. The same effect can be seen in figure 6.7. Furthermore, notice how the average number of the delay errors is smaller in the middle of the figure where the delay is big. This occurs because a number of packets are stored in NIST Net's timer queue. A bigger effort is needed to update the hash tables.

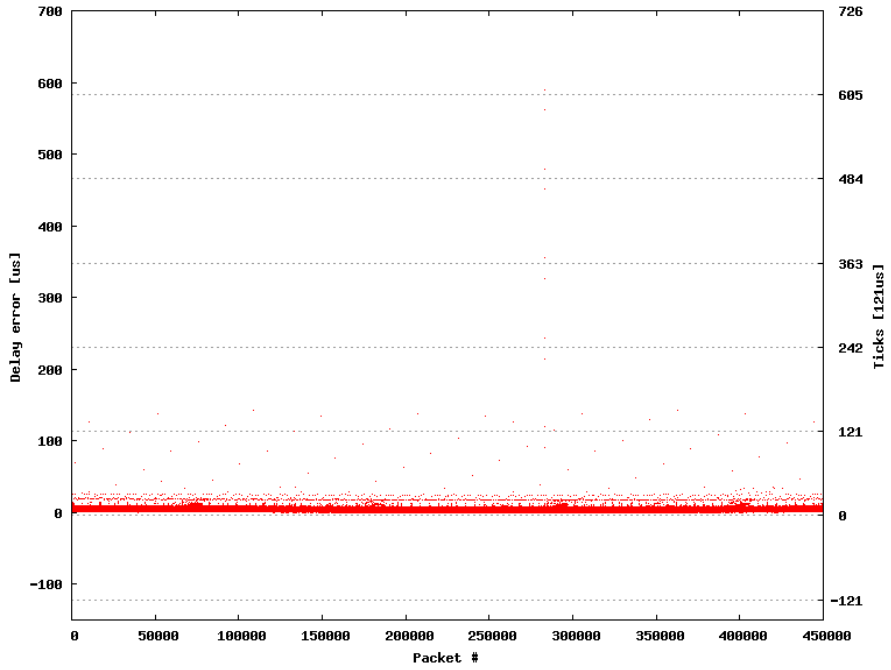


Figure 6.6: NIST Net delay set to 0μsec - IP forwarding delay

Figure 6.10 uses the same pattern as 6.9, but with a 122μsec raise instead of 121μsec. This forces the time to run through the border of one tick and displays the effects previously described.

Figure 6.11 shows a trace from real network probing. The delay values are multiplied by 10 because the original trace is almost as small as the precision of NIST Net.

The delay error of the configuration in figure 6.11 is shown in figure 6.12. The visible effects with the different interrupt levels and the negative delay errors have already been described for previous figures.

Appendix B includes the plots of all measurements including additional ones and the ones described in this section. Also, the mathematical calculations like average, standard deviation, maximum and minimum are listed in a table (see figure B.1).

6.4 Summary results

These tests prove a method to correctly apply delays to packets. The delay error has a precision of two ticks (242μsec) and is sufficient.

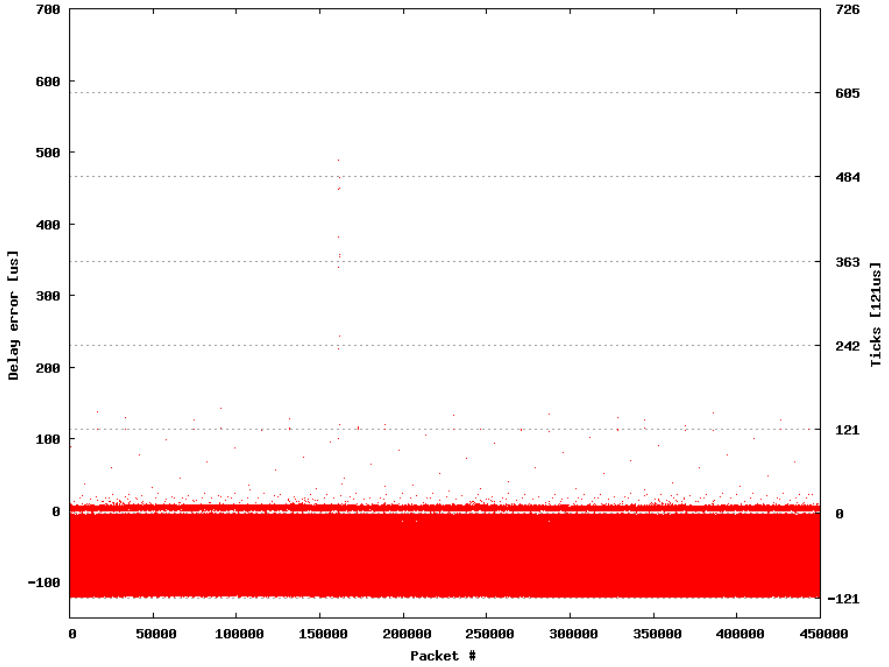


Figure 6.7: NIST Net delay set to 121μsec (1 tick) - Delay error

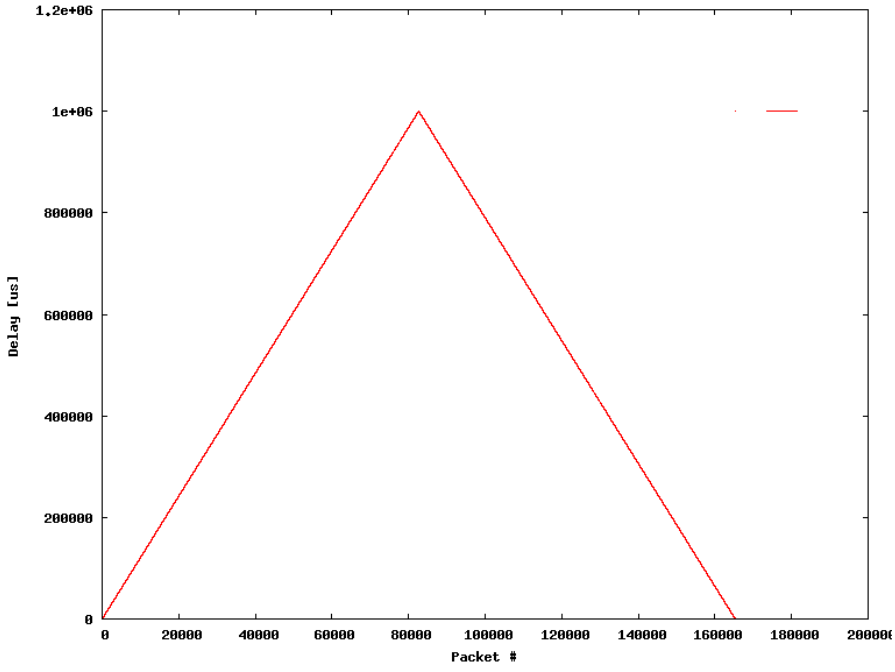


Figure 6.8: Pattern - 10 packets per 121μsec raise - Delay config

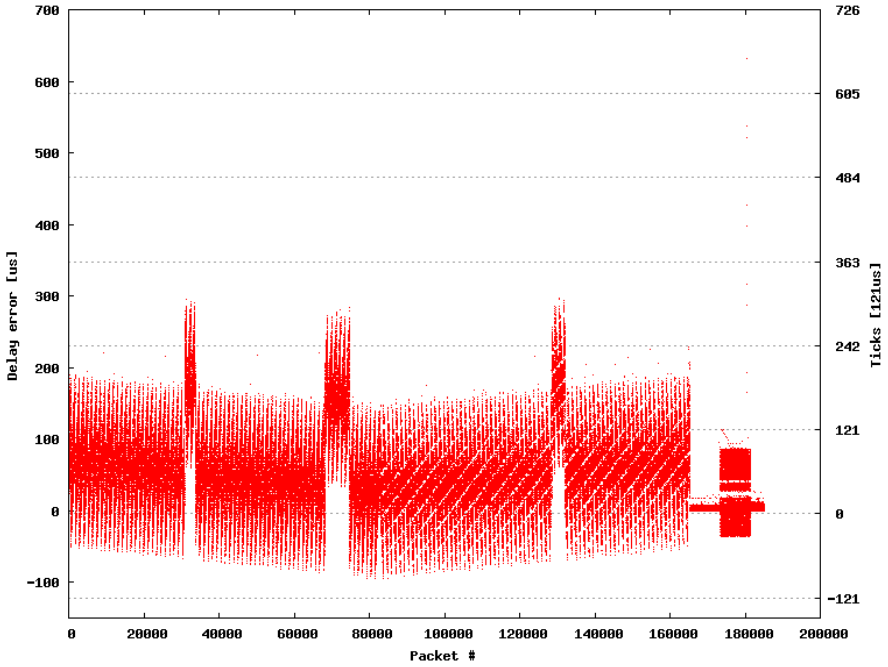


Figure 6.9: Pattern - 10 packets per 121μsec raise - Delay error

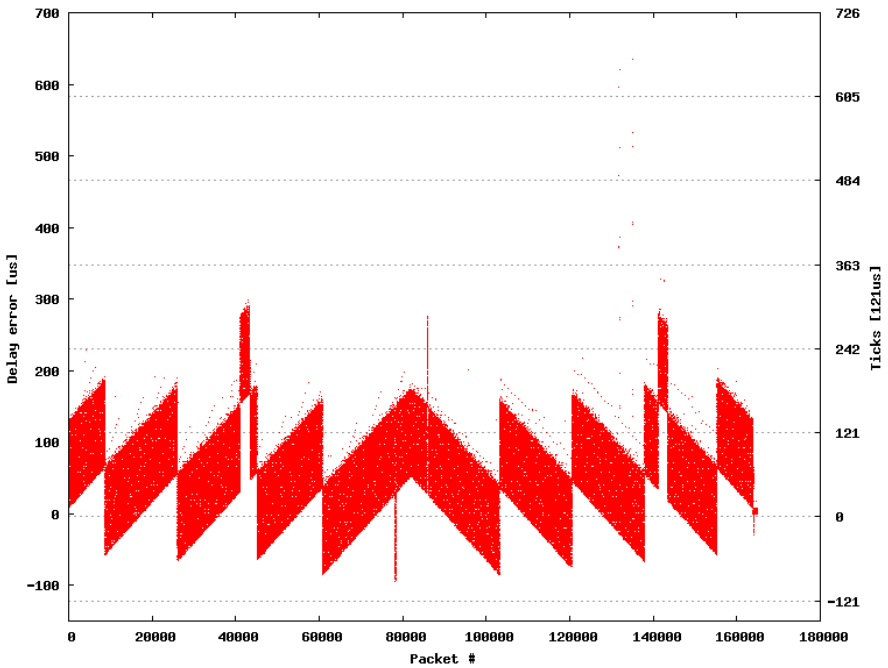


Figure 6.10: Pattern - 10 packets per 122μsec raise - Delay error

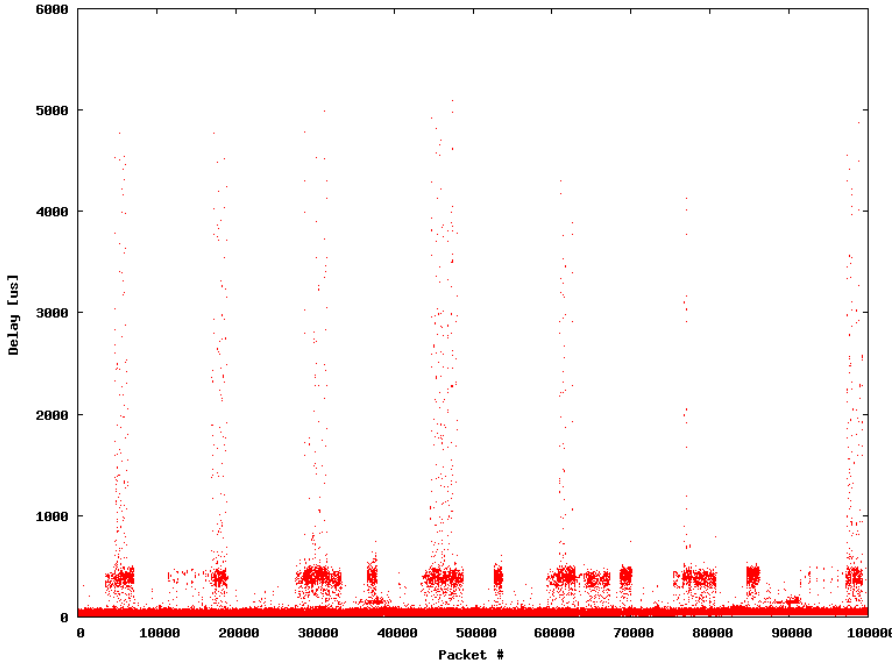


Figure 6.11: Trace from network probing (1) multiplied by 10 - Delay config

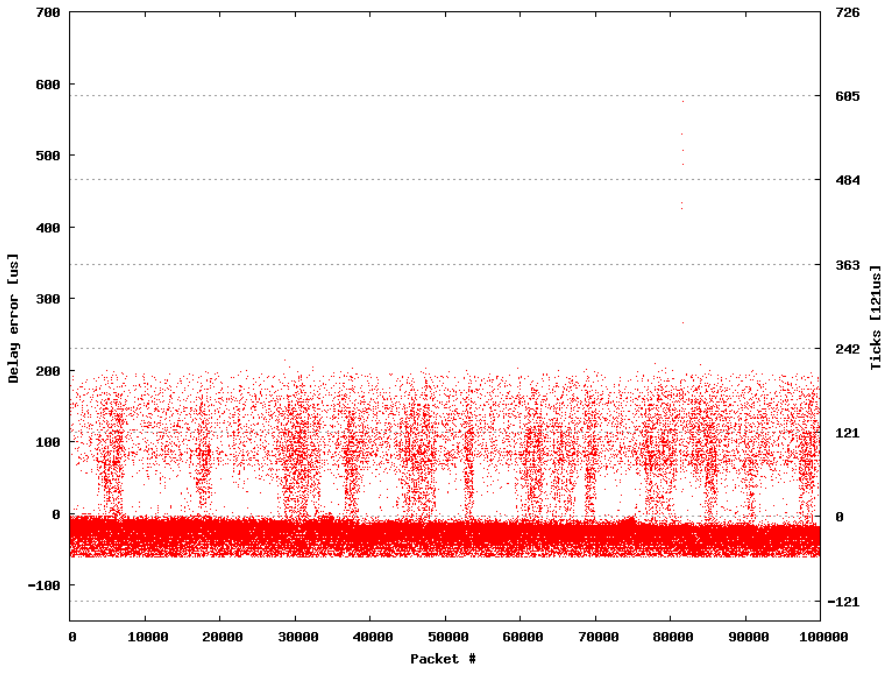


Figure 6.12: Trace from network probing (1) multiplied by 10 - Delay error

Chapter 7

Applications

Trace-based network emulation opens several new possibilities for applications/network devices. Real time applications like Voice Over Internet Protocol (VoIP) and video/audio streaming or network devices like Circuit Emulation Adapters demand a predictable network connection. These real time applications and devices need guaranteed Quality of Service (QoS).

Using a network emulator is one way to verify the robustness of an application. A trace-based emulator can provide "what-if" testing in a laboratory environment and the ability to create reliable, repeatable and standard test configurations for applications and devices.

Applications can be categorized in network applications, network prototyping, and protocols. The applications for each category are described in the following sections.

7.1 Network Applications

Figure 7.1 shows several network applications with different requirements.

Common applications like Internet browsing, data transfers, and E-Mail

Traffic	**Bandwidth requirement**	**Relative delay Tolerance**	**Time Factor**
Internet browsing	128 - 1 MBit/s	High	Buffer tolerant
Data transfer (E-Mail, Files)	128 - 1 MBit/s	High	Buffer tolerant
Chat (text)	28 KBit/s	Low	Low tolerance
Voice (phone)	64 KBit/s	Extremely low	Real time
Voice (conference)	1 MBit/s	Low	Real time
Voice & Video	5 MBit/s	Medium / Low	Real time
Video Streaming	256 - 1 MBit/s	Low	Buffer tolerant

Figure 7.1: Network applications

are not sensitive to relative delay or time and are buffer tolerant. The packet drops/duplicates/reordering is managed by the network protocol below and has no influence to the applications themselves.

Real time applications such as audio and video streaming are greatly affected by latency and packet loss. Since they are only one-way and not interactive, buffering is tolerated. With interactive applications like Voice Over Internet Protocol, Voice conference and chat, buffering is not tolerated. As real time applications merge with traditional data services on the IP network, it becomes more critical for networks to provide minimum latency and packet loss. This is especially with wireless networks the case. These real time applications need to be tested in combination with various network conditions.

7.2 Network Prototyping

Network devices can not always be tested by software simulators. A Circuit Emulation Adapter (CEA) is an example device which can not be simulated in its final hardware version. CEA's are used for example in city networks. Using a trace-based network emulator with traces from network probing in a same network environment, enables network emulation with the same condition as in real world.

Competitor's devices are often only available as "black boxes". Exact specifications for simulation are often unavailable. Therefore, testing by a network emulator is required to compare such devices or demonstrate the differences.

7.3 Protocols

Protocols for reliable data transport, real-time data, routing, peer-to-peer file sharing depend on the underlying network infrastructure. Network emulation enables the possibility to test protocols under various network conditions. It is possible to optimize protocols (i.e. wireless protocols) and test them under exactly the same conditions.

Chapter 8

Conclusion

This thesis describes the enhancement of NIST Net with a trace reader extension. The enhanced NIST Net is a fully-fledged layer 3 network emulation tool, implemented on top of the original NIST Net and Linux. It is to say that the emulator can handle several parallel E1 Voice streams, what is requested to test sensitive Voice-over-IP applications. It is able to emulate statistically correct network dynamics in a flexible way, especially long-range dependence. Packet delays, drops, and duplicates can be applied based on preloaded trace data, which is offline generated from real traces, simulation traces, or calculus traces. The performance evaluation which consisted of a maximum performance test and delay precision tests under different conditions, proved the emulator as working correctly and being able to handle the requested precision and maximal packet rate.

Chapter 9

Further Work

It is proposed that a further work would be to redesign NIST Net so that it is able to handle packets already in layer 2. This would enhance speed, flexibility, and allows usage of a wider area including layer 2 protocols like for example the spanning tree algorithm, that could be optimized with timers fitting a given network. An enhancement in speed results because the overhead of the packet processing is smaller. Layer 3 headers stay unchanged in the packets and have no influence. However, a flow-based layer 2 packet handling needs major changes to the NIST Net implementation. This includes

- change the hook where the packet munger function is called
- change all structures, hash tables functions, and the flow table to use MAC addresses instead of IP source/destination addresses
- adapt the command line interface
- change the reentering point where the packets are sent back to the Linux networking code

If all functions that determine the address of the packets are cleared out, the porting is simpler. Not like in layer 3, an emulator in layer 2 doesn't need the ability to differentiate flows if it is running as a bridge because there are usually no layer 2 flows.

The timer used by NIST Net uses a granularity of 121μsec, which is sufficient for applications like Voice-over-IP but may not suffice for more sensitive applications to real time. In combination with the timer, NIST Net could be ported to a real time Linux environment which allows much higher precision and suppresses scheduler and interrupt outliers as seen in the plots. However, a real time Linux has a different architecture and needs the porting of NIST Net's kernel module to the real time API. The expected precision on a real time Linux is between 3 and 20μsec.

A port to Linux kernel 2.6 involves the adaption to the new kernel module behavior in kernel 2.6. Kernel 2.6 comes with a different TCP/IP stack which denotes also a change to the NIST Net kernel module. A third point is the new behavior of the scheduler. The preemption that also can occur within the kernel might require changes in the module.

Appendix A

Tools

This chapter describes helper tools to convert files to be used as trace files or that can be used for debugging. Also tools to gather statistics can be found in this chapter.

A.1 txt2bin

Usage: txt2bin input file output file

txt2bin converts an ASCII input file with a signed integer value on each line in a binary file that can be used as a trace file.

A.2 bin2txt

Usage: bin2txt input file

bin2txt reads files that have been created with txt2bin. The output is human readable and can be used for debugging purposes. It splits up the 32 Bit integer value in the two control bits and the remaining 29 bits for the delay value. The values are presented in integer format. A human readable binary output is also provided for base debugging purposes.

A.3 bindump

Usage: bindump input file

bindump simply shows a human readable binary output and the corresponding integer value.

A.4 headgen

Usage: headgen head delay

headgen can be used before converting values with txt2bin. txt2bin takes the values as integer, which have besides the delay values also information about drop, or duplicate. To triplicate packets, use duplicate twice in a row. See figure 5.3 for information about coding.

A.5 pps

Usage: pps

pps shows every second the amount of packet that passes NIST Net. This tool was used to determine the maximum amount of packets per second that NIST Net can process.

Appendix B

Additional Plots

This appendix includes all plots and their mathematical calculations of the delay error plots like average, standard deviation, minimum and maximum, listed in table B.1.

Figure	**Test**	**Avg.**	**Std.**	**Min**	**Max**
B.2	NIST Net off	2.22	2.05	0	540
B.3	Delay 0μsec	2.20	2.87	0	589
B.5	Delay 121μsec	-60.65	36.19	-121	488
B.7	Increasing, decreasing	58.32	62.47	-96	305
B.9	Increasing, decreasing	59.49	57.87	-95	635
B.11	Increasing, decreasing	67.84	60.66	-85	647
B.13	Network trace	-3.50	13.21	-61	582
B.15	Network trace * 10	-7.44	51.70	-61	575
B.17	Network trace * 100	72.47	50.28	-60	298
B.19	Network trace * 1000	77.78	50.75	-53	653
B.21	Network trace peak	73.26	49.96	-114	297

Figure B.1: Mathematical figures for the difference plots

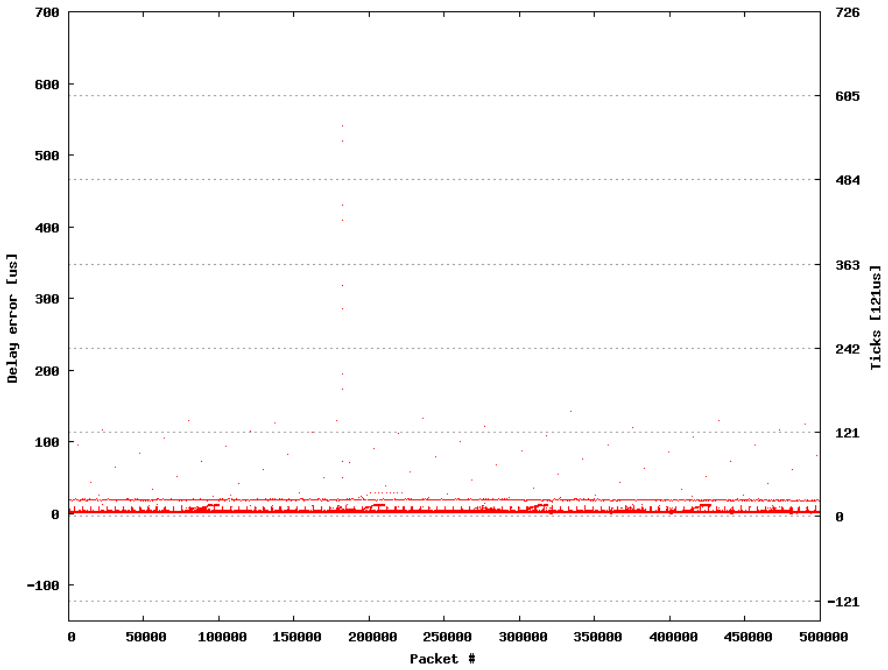


Figure B.2: NIST Net disabled - IP forwarding delay

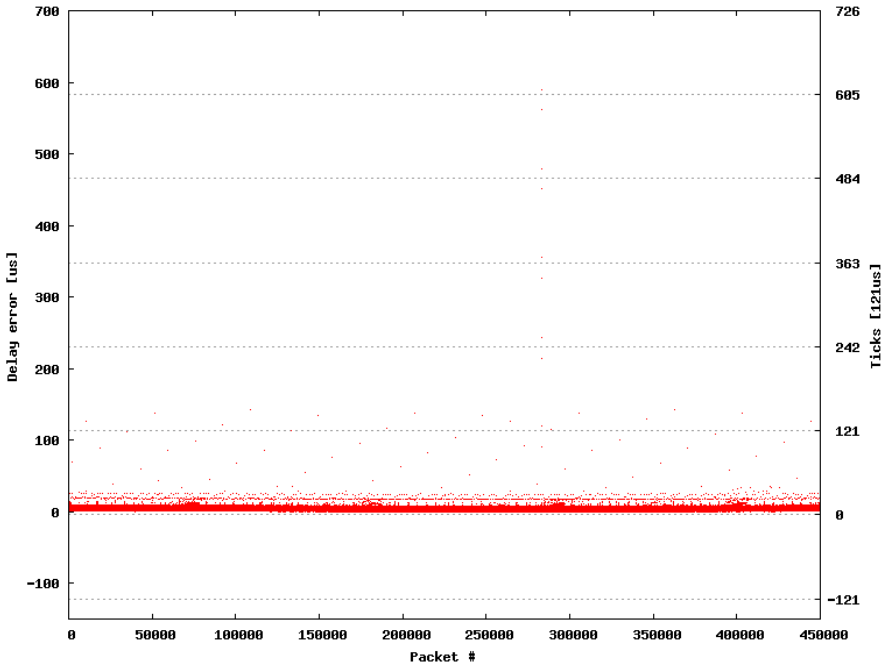


Figure B.3: NIST Net delay set to 0μsec - IP forwarding delay

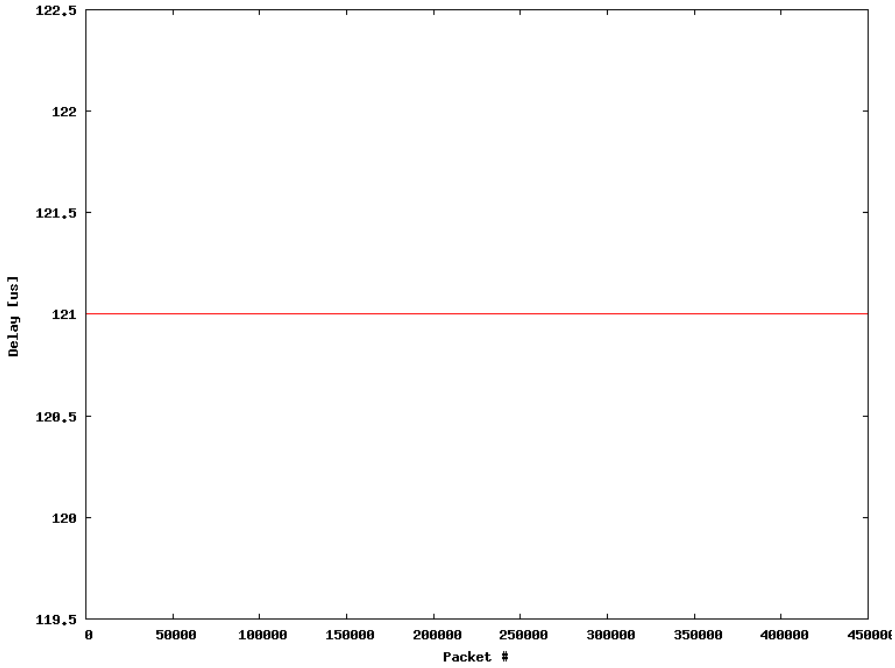


Figure B.4: NIST Net delay set to 121μsec (1 tick) - Delay config

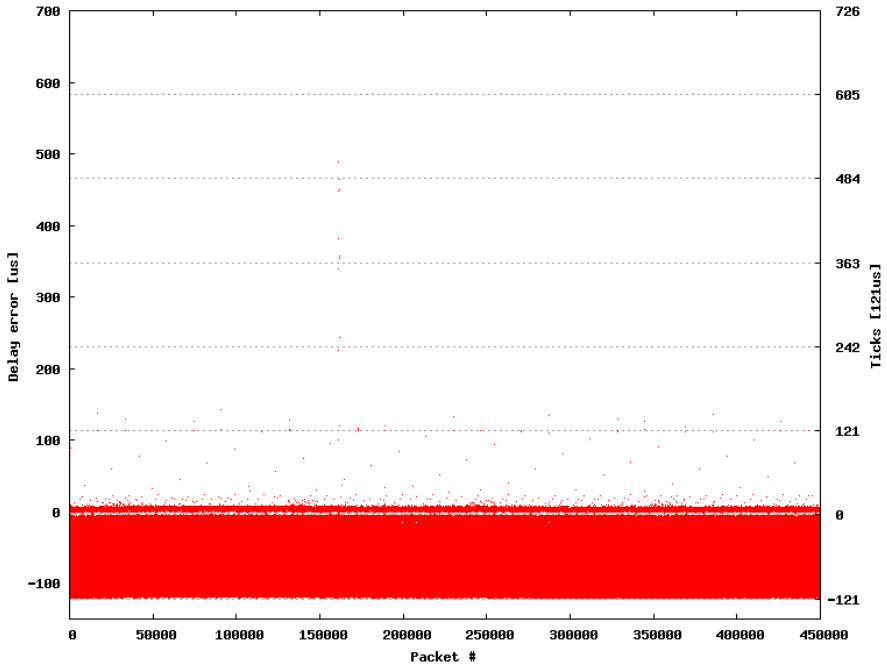


Figure B.5: NIST Net delay set to 121μsec (1 tick) - Delay error

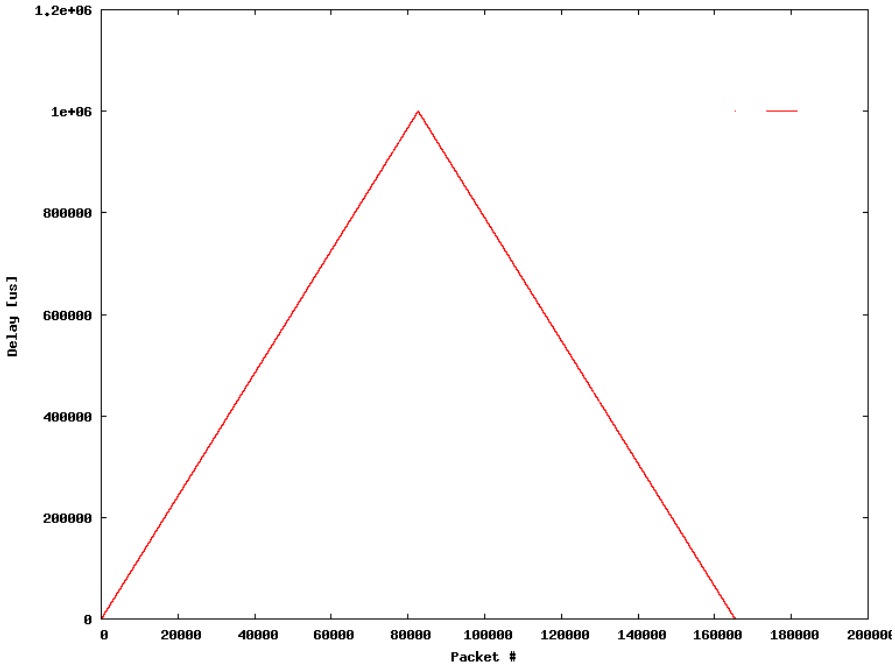


Figure B.6: Pattern - 10 packets per 121μsec raise - Delay config

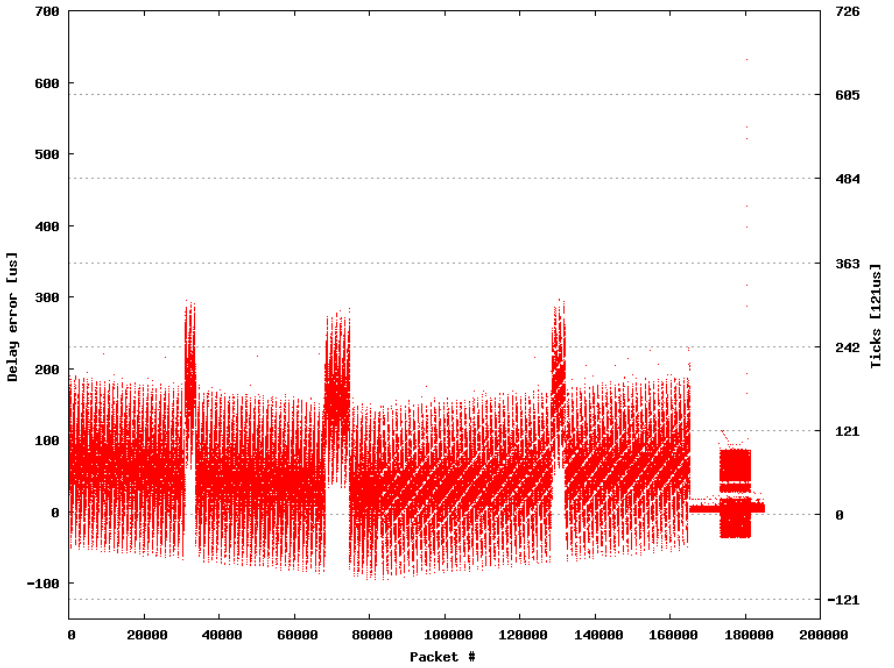


Figure B.7: Pattern - 10 packets per 121μsec raise - Delay error

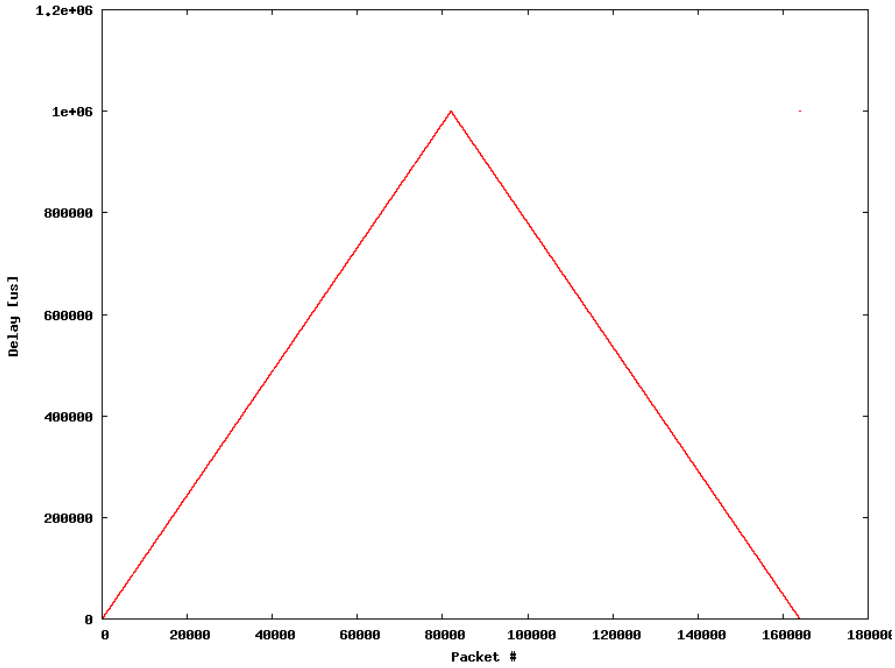


Figure B.8: Pattern - 10 packets per 122μsec raise - Delay config

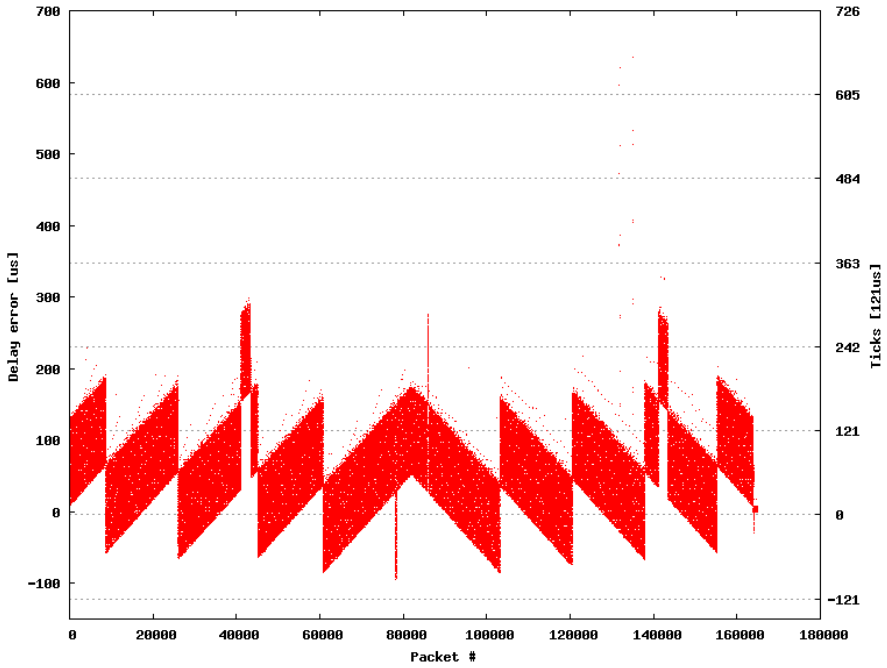


Figure B.9: Pattern - 10 packets per 122μsec raise - Delay error

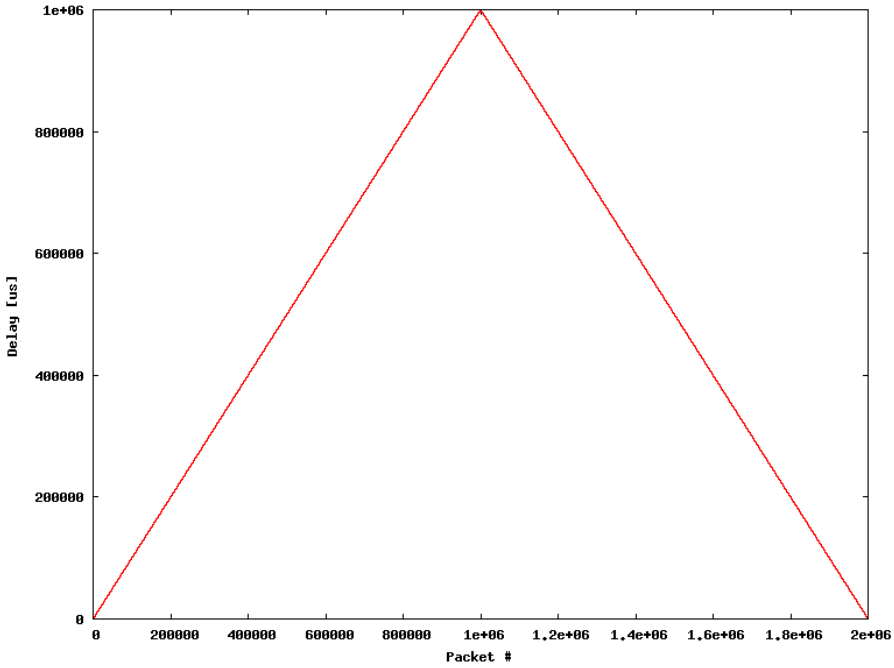


Figure B.10: Pattern - 1 packet per 1μsec raise - Delay config

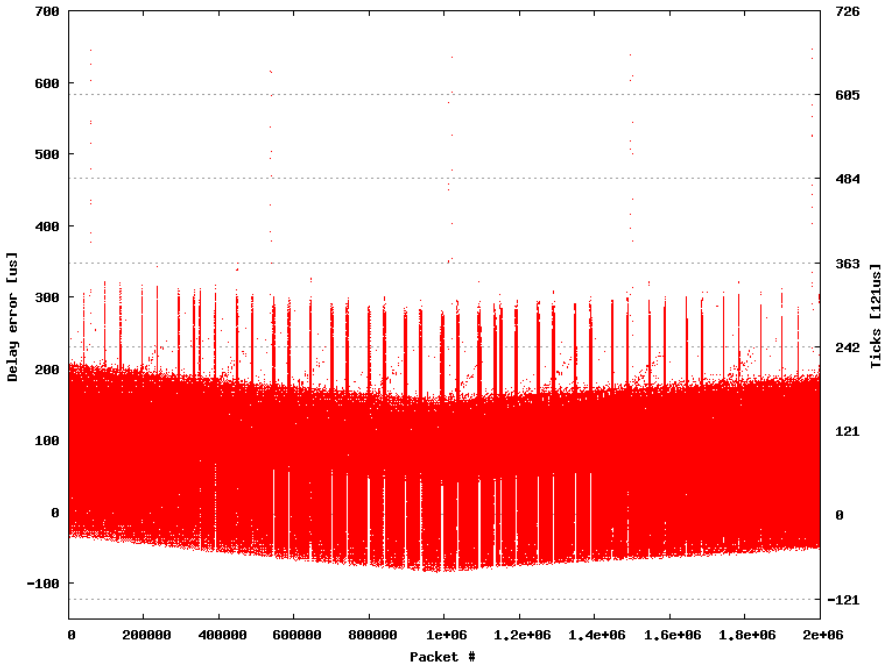


Figure B.11: Pattern - 1 packets per 1μsec raise - Delay error

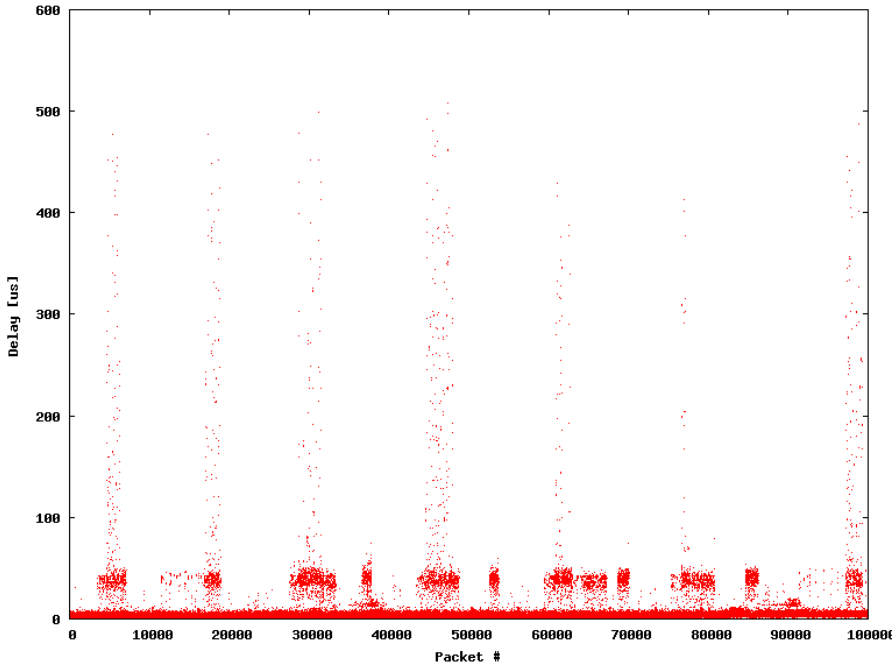


Figure B.12: Trace from network probing (1) - Delay config

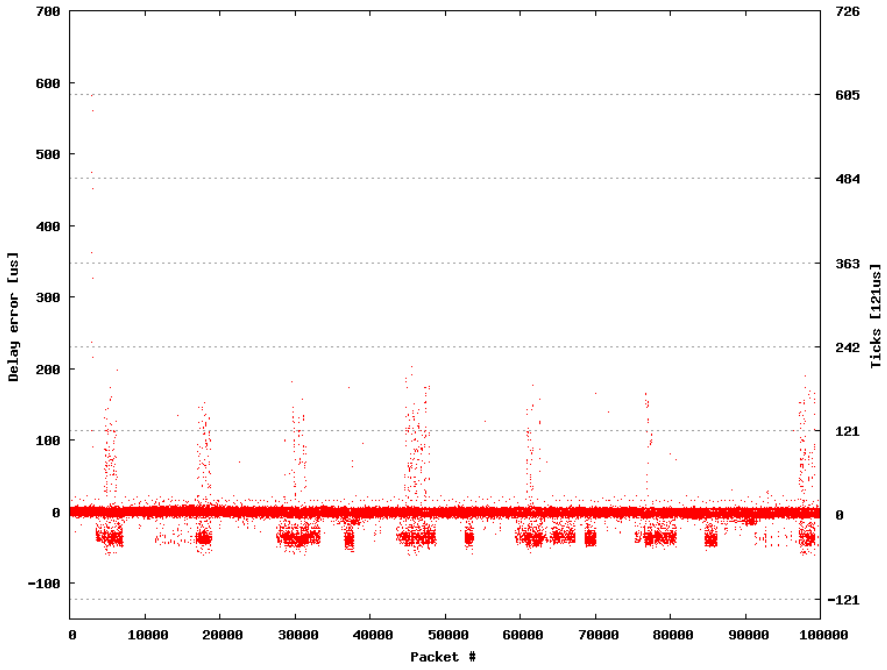


Figure B.13: Trace from network probing (1) - Delay error

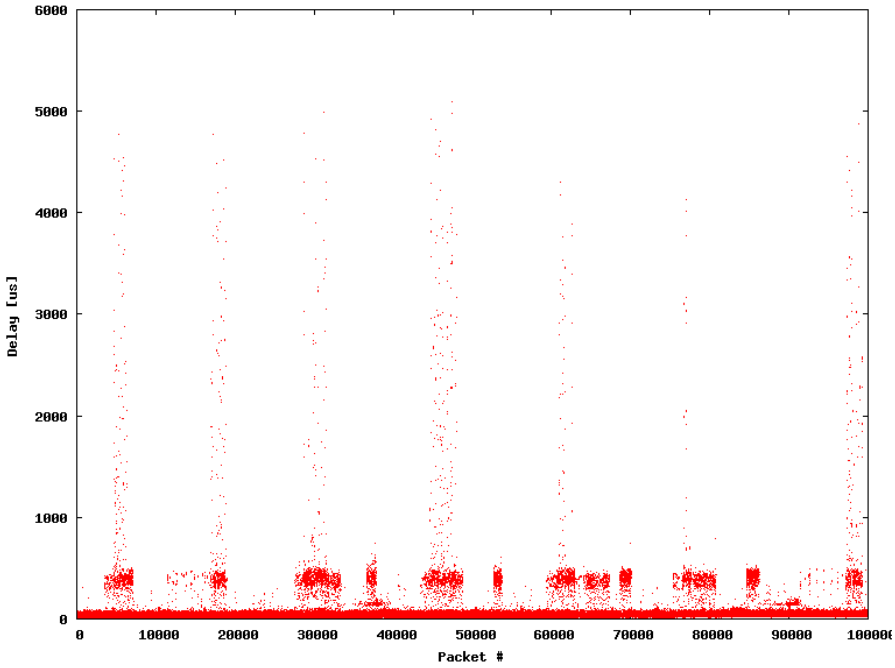


Figure B.14: Trace from network probing (1) multiplied by 10 - Delay config

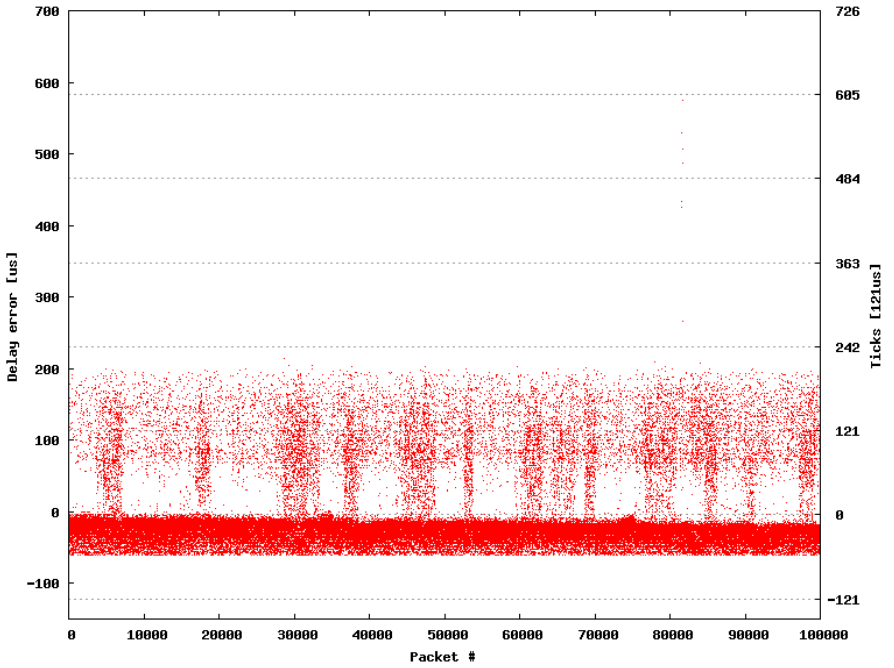


Figure B.15: Trace from network probing (1) multiplied by 10 - Delay error

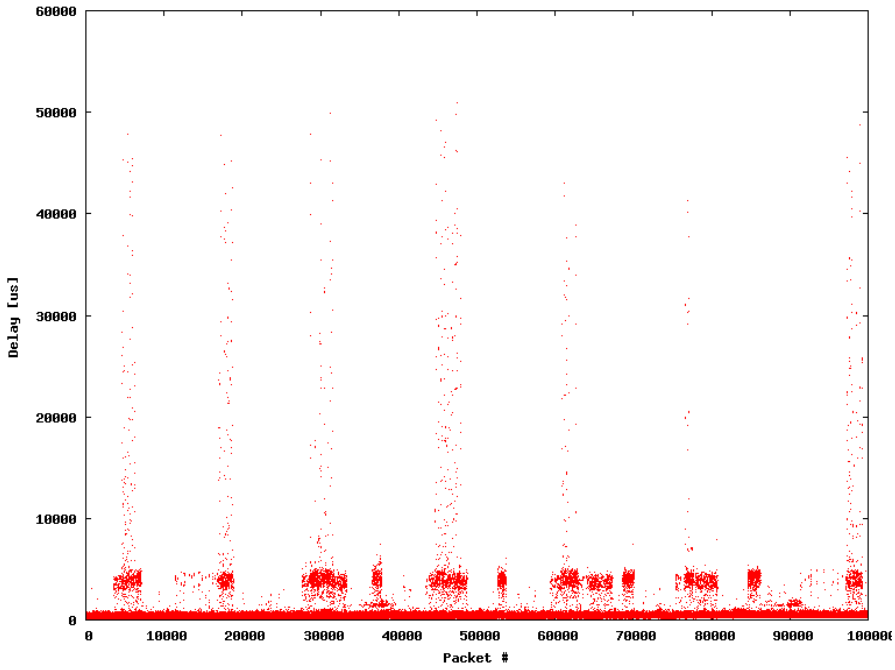


Figure B.16: Trace from network probing (1) multiplied by 100 - Delay config

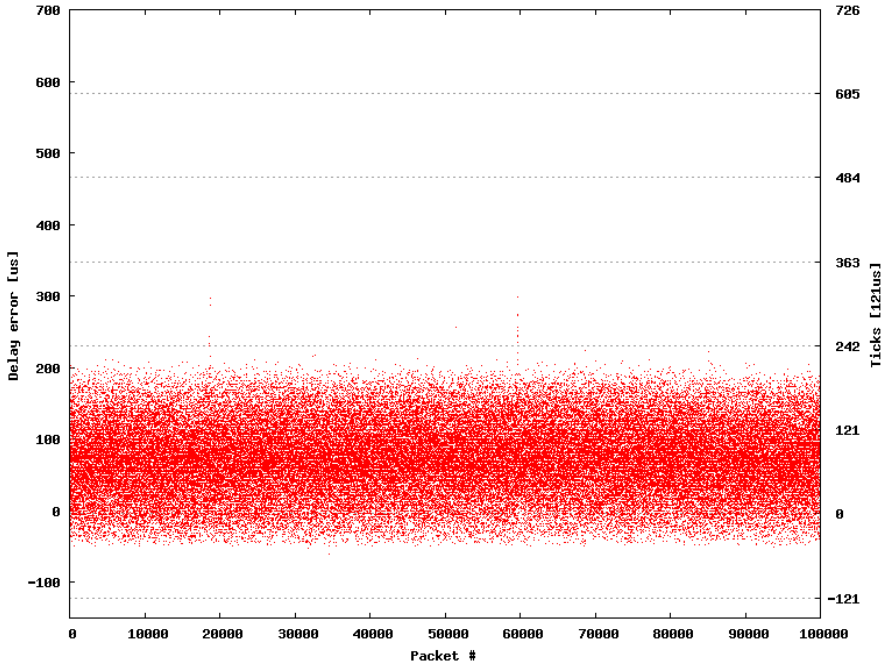


Figure B.17: Trace from network probing (1) multiplied by 100 - Delay error

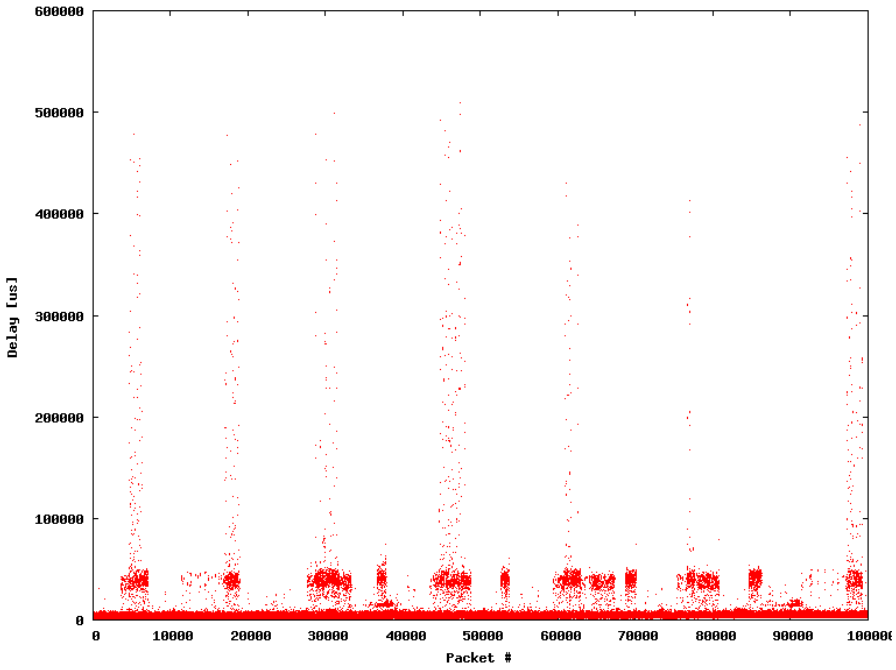


Figure B.18: Trace from network probing (1) multiplied by 1000 - Delay config

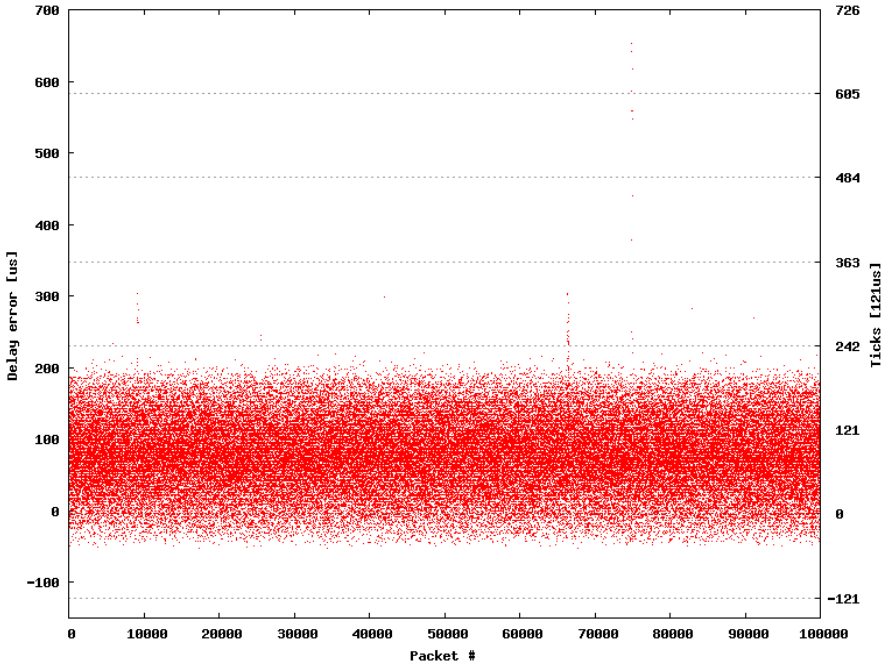


Figure B.19: Trace from network probing (1) multiplied by 1000 - Delay error

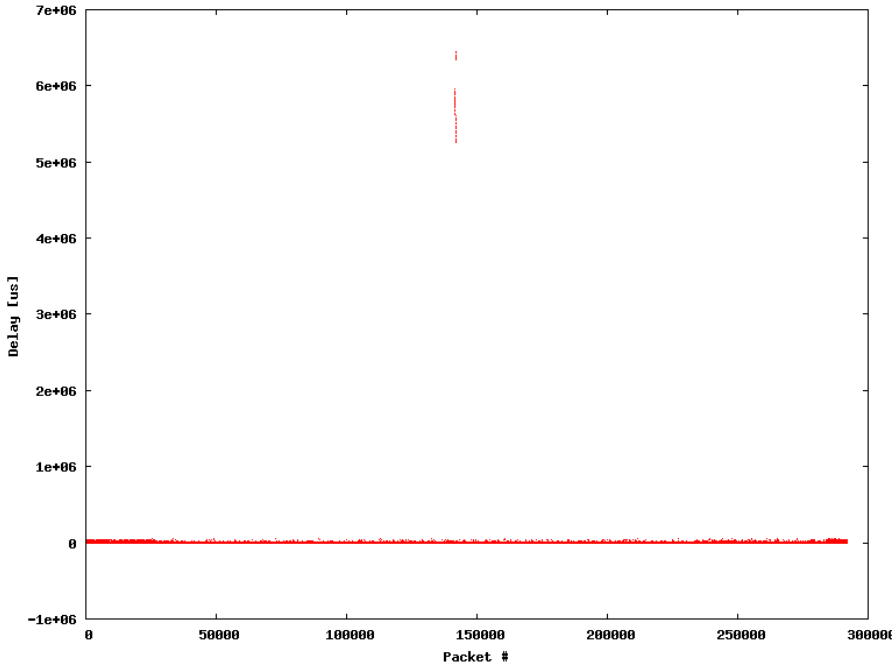


Figure B.20: Trace from network probing (2) - Delay config

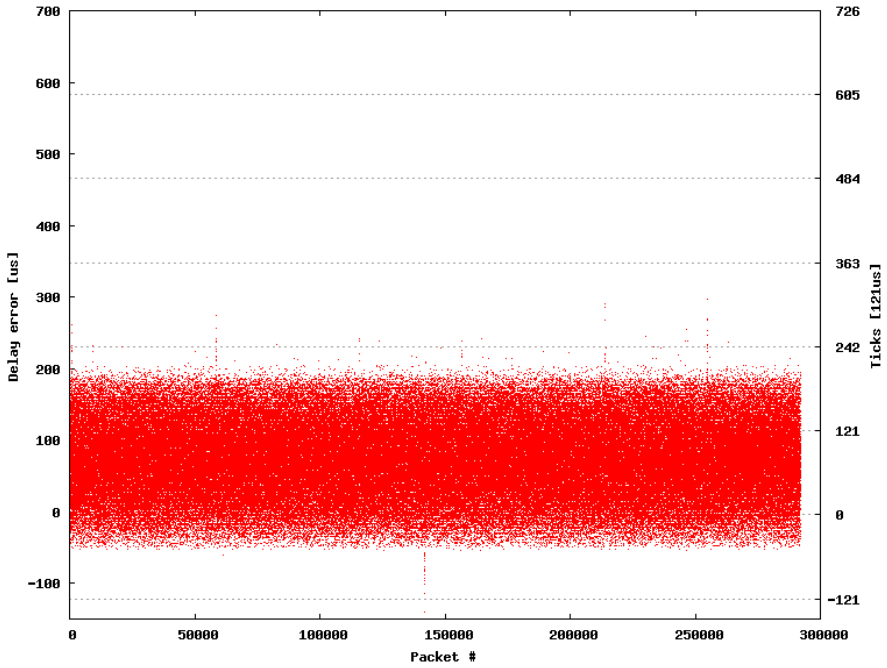


Figure B.21: Trace from network probing (2) - Delay error

Appendix C

Assignment

Objectives

The overall objective is to develop a trace-based network emulator for (i) IP networks and (ii) Ethernet-based networks that enhances the Linux kernel. To achieve this objective, we suggest to enhance and modify NIST Net. Then, the objectives of this thesis become

- reengineering the implementation of NIST Net, i.e. figuring out
  how NIST Net intercepts the protocol stack,
  how it accesses the inverse cdf table to generate packet delay values,
  and how it employs timer interrupts and queue structures to replay delayed IP packets.

- modifying the generation of delay values from accessing an inverse cdf table to accessing values from previously loaded trace. The modification should be capable to handle loss (and reordering) events as listed in the trace.

- enhancing NIST Net for trace-based network emulation on the layer 2 (Ethernet).

Tasks

- Get familiar with concept and implementation of NIST Net. Read [1], review NIST Net's web pages (`http://dns.antd.nist.gov/nistnet`), subscribe the mailing list, and go through all README files that come with the NIST Net software package.

- Install NIST Net on Debian Linux. Package and kernel configuration files can be obtained from the advisor.

- Identify and review relevant parts of NIST Net's code, e.g. review how NIST Net writes data from user space to kernel space and how NIST Net accesses the table in kernel space.
- Document NIST Net current implementation and make suggestions on how to implement trace-based emulation on the IP layer.
- Document your implementation
- Review the protocol stack in Debian.
- Make suggestions on how trace-based Layer 2 (Ethernet) network emulation can be added to NIST Net.
- Implement your suggestions and document your implementation.

Deliverables and Organization

- If possible, student and advisor meet or telephone on a weekly basis to discuss progress of work and next steps. If problems/questions arise that can not be solved independently, the student should not hesitate to contact the advisor anytime.
- At the end of the third week, a detailed time schedule of the semester thesis must be given and discussed with the advisor.
- In regular intervals (e.g. every two or three months) intermediate reports are due. These reports are linked to short presentations of 15 minutes to the professor and the advisor. In these presentations, the student has to discuss major aspects of the ongoing work including results, problems, and remaining work.
- At the end of this thesis, a presentation of 15 minutes must be given either in teleconference or in the communication systems group meeting. The presentation should carefully introduce settings and background of the work. Moreover, it should contain an overview of the major results and conclusions from the work.
- We encourage to write all reports in English. However, reports can also be written in German. The final report must contain a summary, the assignment and the time schedule. Its structure should include an introduction, a methods/design section, a results section and a conclusion section. Moreover, the final report should include a complete documentation of all produced software. Related work must be correctly referenced. See `http://www.tik.ee.ethz.ch/~flury/tips.html` for more tips on thesis writing. Three hard copies of the final report must be delivered to TIK.

- Any software which is produced in relation with this thesis needs to be delivered before ending the thesis. This includes all source code and a documentation. The source code may then be published as open source. Moreover, the PDF and the source code employed to generate the final report also have to be delivered. This includes data to draw the figures Preferred format for delivery is a CDROM.

Appendix D

Timetable

Week	Subject
1	Related work
2	Reverse engineering NIST Net, documentation
3	
4	
5	Design evaluation, documentation
6	
7	
8	
9	Implementation proof of concept
10	Intermediate report
11	
12	Implementation of concept
13	
14	
15	Vacation
16	
17	Intermediate report
18	
19	Performance tests
20	
21	
22	
23	Documentation
24	Documentation
25	Presentation, revise documentation
26	Reserve

Bibliography

[1] Mark Carson and Darrin Santay.. *NIST Net: a Linux-based network emulation tool*, Computer Communication Review (ACM SIGCOMM), 33(3):111-126, 2003.

[2] Wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Mmap

[3] Procfs guide. http://www.kernelnewbies.org/documents/kdoc/procfs-guide/lkprocfsguide.html

[4] Luigi Rizzo. Dummynet: a simple approach to the evaluation of network protocols. *ACM Computer Communication Rview*, 27(1):31-41, 1997. http://citeseer.ist.psu.edu/rizzo97dummynet.html

[5] ONE - the Ohio Network Emulator. http://masaka.cs.ohiou.edu/one/

[6] URL. Ethereal - A Network Protocol Analyzer. http://www.ethereal.com, 2004.

[7] MIL3 Inc. Opnet modeler, v3. 1996.

[8] L. Breslau, D. Estrin, K. Fall, S. Floyd, J. Heidenmann, A. Helmy, P. Huang, S. McCanne, K. Varadhan, Y. Xu, and H. Yu. Advances in Network Simulations. *IEEE Computer, May 2000.*

[9] Raj Jain, *The art of computer systems performance analysis*, 26, 1991.

[10] Rainer Baumann, Ulrich Fiedler, *TIK Report 217*, 2005.