NIST Net: A Linux-based network emulation tool

Article *in* ACM SIGCOMM Computer Communication Review · July 2003

DOI: 10.1145/956993.957007 · Source: DBLP

Mark Carson
National Institute of Standards and Technology
10 PUBLICATIONS **586** CITATIONS

NIST Net – A Linux-based Network Emulation Tool

Mark Carson, Darrin Santay
National Institute of Standards and Technology (NIST)
carson@nist.gov, santay@nist.gov

Abstract

Testing of network protocols and distributed applications has become increasingly complex, as the diversity of networks and underlying technologies increase, and the adaptive behavior of applications becomes more sophisticated. In this paper, we present *NIST Net*, a tool to facilitate testing and experimentation with network code through emulation. NIST Net enables experimenters to model and effect arbitrary performance dynamics (packet delay, jitter, bandwidth limitations, congestion, packet loss and duplication) on live IP packets passing through a commodity Linux-based PC router. We describe the emulation capabilities of NIST Net; examine its architecture; and discuss some of the implementation challenges encountered in building such a tool to operate at very high network data rates while imposing minimal processing overhead. Calibration results are provided to quantify the fidelity and performance of NIST Net over a wide range of offered loads (up to 1 Gbps), and a diverse set of emulated performance dynamics.

1 Introduction

Testing network protocols and applications has always been a difficult task, but current trends in Internet software and hardware complicate matters in many ways. Networks do not simply get "faster," but more diverse, carrying more diverse traffic as well. Link technologies vary widely in bandwidth, latency, and error and loss rates, and may be highly asymmetric. Overall network dynamics can fluctuate wildly, with spot congestion and failures common. The demands that applications make of networks vary widely as well, often taking on near-real-time characteristics that differ fundamentally from the best-effort delivery typically provided by current networks. Applications and protocols in consequence increasingly employ adaptive mechanisms to make more intelligent use of available network resources. But these, too, present new testing challenges: the "correct" behavior of adaptive code cannot be defined statically or often even in any simple deterministic fashion; and adaptive protocols at different levels or on different systems may interact poorly with each other in ways not easily detectable while testing in isolation.

To address this growing diversity of network hardware and software, and to provide a controlled, reproducible environment for testing network-adaptive applications, we have designed *NIST Net*, a simple, fast, Linux[1] kernel-based network emulator. NIST Net provides the ability to emulate common network effects such as packet loss, duplication or delay; router congestion; and bandwidth limitations. It is designed to offer sufficient capabilities and performance to reproduce a wide range of network behaviors (forwarding rates of up to 1 Gbps, satellite-like delays or longer, asymmetric characteristics), while requiring only commodity PC hardware and operating systems. NIST Net has been used for emulation

[1] Certain software, equipment, instruments, or materials are identified in this paper in order to specify the experimental procedure adequately. Such identification is not intended to imply recommendation or endorsement by the National Institute of Standards and Technology, nor is it intended to imply that the materials or equipment identified are necessarily the best available for the purpose. Product names mentioned in this paper are trademarks of their respective manufacturers

up to line rate over 100Mbps Ethernet with typical “throw-away” machines (200 MHz Pentium-class processors and PCI-based 10/100 Ethernet cards). On current generation machines, NIST Net has been successfully used at line rate with gigabit Ethernet cards and 622 Mbps OC-12 interfaces.

Emulation, as defined here, is a combination of two common techniques for testing network code: simulation, which we can define as a synthetic environment for running representations of code; and live testing, a real environment for running real code. In these terms, emulation is a “semi-synthetic” environment for running real code. The environment is semi-synthetic in the sense that it is a real network implementation (in the case of NIST Net, the Linux 2.xx kernel) with a supplementary means for introducing synthetic delays and faults. Emulation thus offers many of the advantages of both simulations (a controlled, reproducible environment, which is relatively quick and easy to assemble) and live testing (real code in a real environment, which obviates any questions about the fidelity of the representation). In a sense, emulation can then minimize the intellectual “investment” required for network testing. NIST Net extends this metaphor by deliberately simplifying its installation and basic usage. Thousands of people throughout the world have successfully installed and used the emulator for a wide variety of projects, even those with no prior experience with Linux. It has proven particularly useful in academic settings for class laboratories and student research projects.

In this paper, we discuss the features and design of the emulator, emphasizing the approaches taken to ensure simple installation and use, and to minimize processing overhead, while still providing useful network emulation facilities. This paper is organized as follows. Section 2 describes the principal features of the emulator. Section 3 then outlines the architecture that implements these features, with special notes on performance-related aspects. Section 4 describes the statistical routines NIST Net uses to generate its effects. Finally, section 5 presents the results of our calibration testing of NIST Net itself.

1.1 Related work

Network emulators have been implemented in the past for a variety of purposes on a variety of systems. Among them we can mention Dummynet for FreeBSD [1], the Hitbox pseudo-device for SunOS [2], the MOST Radio network emulator [3], the Ohio Network Emulator [4], the Orchestra fault injection tool [5], Trace-based mobile network emulation [6], the emulation capabilities in the Vint/NS simulator [7], and the X-kernel simulator/live test environment [8].

Of these, the NIST Net kernel design most closely resembles the SunOS-based Hitbox, while its user interface was inspired in part by that of the MOST radio network emulator. However, none of these were designed to operate at the high data rates, nor offer the same range of capabilities, as NIST Net. In particular, none of these emulators make use of a separate high-speed clock for packet scheduling, and hence they cannot achieve the same degree of precision in delay modeling. While some of the emulators, such as Dummynet, employ sophisticated queuing models for bandwidth modeling, none of them include delay models of much statistical sophistication. NIST Net’s approach of combining a relatively powerful set of capabilities with a very simple and easily-understood interface seems unique in this area.

2 Emulator features

A useful way to think of NIST Net is as a “network-in-a-box” (Figure 1) – a specialized router which emulates (statistically) the behavior of an entire network in a single hop. NIST Net selectively applies network effects (such as delay, loss, jitter) to traffic passing through it, based on user-supplied settings.

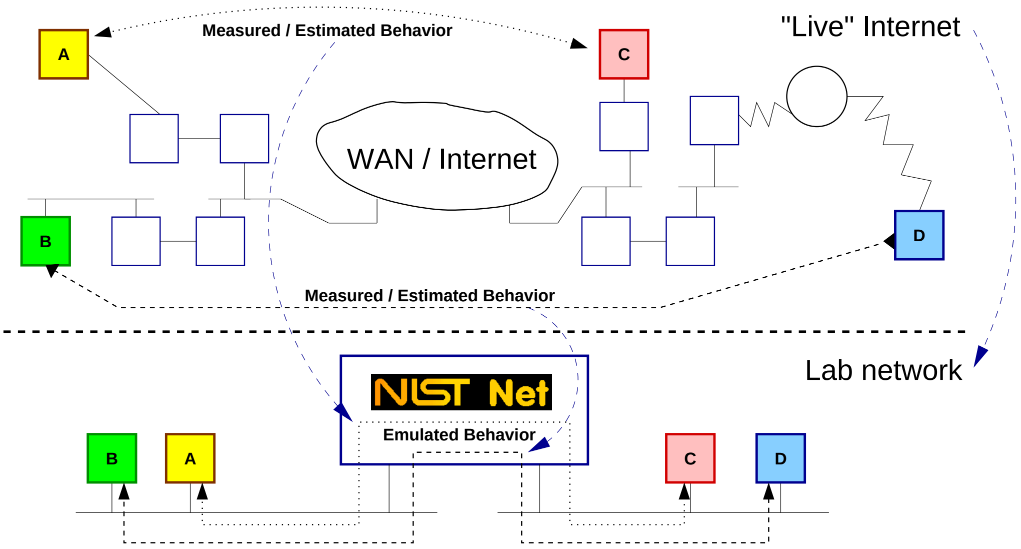


Figure 1: NIST Net as a “network in a box”

NIST Net works through a table of *emulator entries*. Each entry consists of three parts: (1) a set of specifications of which packets are matched by this entry; (2) a set of effects to be applied to matching packets; and (3) a set of statistics kept about packets which have been matched by this entry. NIST Net treats each *flow* (set of matching packets) running through it separately. Thousands of these emulator entries can be loaded at a time, each with individual network effects specified. This allows the emulation of a diverse set of network conditions even in a very small lab setup (two or three boxes). Entries may be added and changed manually, or programmatically during emulator operation, as when replaying recorded delay/loss traces.

The matching specifications in an emulator entry cover most fields of interest in IP and higher protocol headers: source and destination IP addresses, higher-level protocol identifier (such as UDP, TCP, ICMP, IGMP, IPIP), class/type of service field, source and destination ports (for UDP or TCP), type/code (for ICMP or IGMP), multicast group (for IGMP), and tunneled addresses (for IPIP). Any and all of these fields may be wildcarded, allowing for a wide range in selectivity. The packet matching code is designed to function at line rates even with large numbers (thousands) of emulator entries. The principal elements of its design are considered in section 3.1.

The set of network effects NIST Net can impose includes: packet delay, both fixed and variable (jitter); packet reordering; packet loss, both random and congestion-dependent; packet duplication; and bandwidth limitations. We briefly describe these effects here and consider a few of particular interest in more detail in section 4.

Ordinary (non-bandwidth-related) packet delay may be fixed or random, with the shape of the random distribution (run-time) settable.[2] By default, an empirically-derived “heavy-tail” distribution is used. The mean (μ), standard deviation (σ), and linear correlation (ρ) of packet-to-packet delays are all settable parameters. By appropriate setting of these parameters, packet reordering can be maximized (large σ, small or negative ρ), minimized (smaller σ, larger ρ), or even eliminated ($\sigma <<$ packet interarrival time, or ρ near 1). Packet delays may be specified with microsecond-level precision; the actual delays imposed are limited by the granularity of the clock used.

Ordinary (non-congestion-related) packet loss and duplication are similarly settable. Here a uniform

[2] In this paper, *settable* is used to refer to quantities or features which can be modified at run-time from user-level applications. *Configurable* refers to quantities or features which are set at compile time only.

distribution with parameterized mean and packet-to-packet correlations is provided.

Congestion-dependent loss is emulated through a parameterized version of Derivative Random Drop [9] (DRD). DRD drops packets with a probability that (after a minimum threshold is reached) increases linearly with the instantaneous queue length. DRD does have shortcomings compared to more complex router congestion control mechanisms such as Random Early Detection [10] (RED), principally because DRD can result in coordination of packet drops and retransmissions across multiple flows after certain types of instantaneous traffic bursts. However, these shortcomings are not relevant for NIST Net, where each flow is treated separately, and hence cross-flow-coordinated drops do not occur. In this situation, DRD's computational simplicity makes it the preferred choice.

The NIST Net implementation makes the minimum and maximum thresholds for DRD, and hence the steepness of the drop probability ramp, settable parameters. The queue length used for DRD is that associated with the individual emulator entry. Since traffic may be selectively aggregated through wildcarding in emulator entries, this allows for a variety of traffic congestion scenarios to be emulated.

Support for the router part of Explicit Congestion Notification [11] (ECN) is also implemented through DRD. In the NIST Net implementation, the congestion notification threshold is a third DRD parameter, which in normal use would be intermediate between the minimum and maximum drop probability thresholds. For queue lengths below the ECN threshold, ECN-enabled packets have their Congestion Experienced (CE) bits marked when they would otherwise be dropped. The ability to control separately to what extent ECN is used (in preference to early drop) allows for experimenting with a variety of congestion-handling schemes.

Bandwidth limits are computed and imposed on an instantaneous basis. When a packet arrives matching a bandwidth-limiting emulator entry, the theoretical amount of time the packet would take to transmit at that bandwidth is calculated. Any later packets arriving which match that entry are then delayed (at least) this time period. The first packet (the one inducing the delays for other packets) may then actually be sent at the beginning, middle or end of its time period, depending on the emulator configuration. Because bandwidth limit-related delays are cumulative, it is usually advisable to impose limits on queue lengths (through DRD parameters) as well when using bandwidth limits.

3 NIST Net emulator architecture

The emulator which implements this functionality consists of an instrumented version of a live network implementation, the Linux (2.0.xx – 2.4.xx) kernel. NIST Net consists of two main parts: (1) a loadable *kernel module*, which hooks into the normal Linux networking and real-time clock code, implements the run-time emulator proper, and exports a set of control APIs; and (2) a set of *user interfaces* which use the APIs to configure and control the operation of the kernel emulator (see Figure 2).

This organization of the emulator provides several advantages. Since all of the kernel functionality is incorporated into a loadable module, the emulator may be taken up or down, patched and reloaded during runtime, without interrupting any active connections, including those flows currently being affected by the emulator. The separation into a module also serves to insulate the NIST Net code to a large degree from changes to the base kernel.

The separation between emulation proper and the user interface allows multiple processes to control the emulator simultaneously. This is useful when controlling the emulator both interactively, and with parameters generated from previously-taken traces. New emulator settings may be loaded continuously, even to control currently-active flows; as discussed further below (section 3.1), kernel data structures are handled in such a way as to minimize the locking required. Two user interfaces are provided with the code: a simple command-line interface, suitable for scripting, and an interactive graphical interface

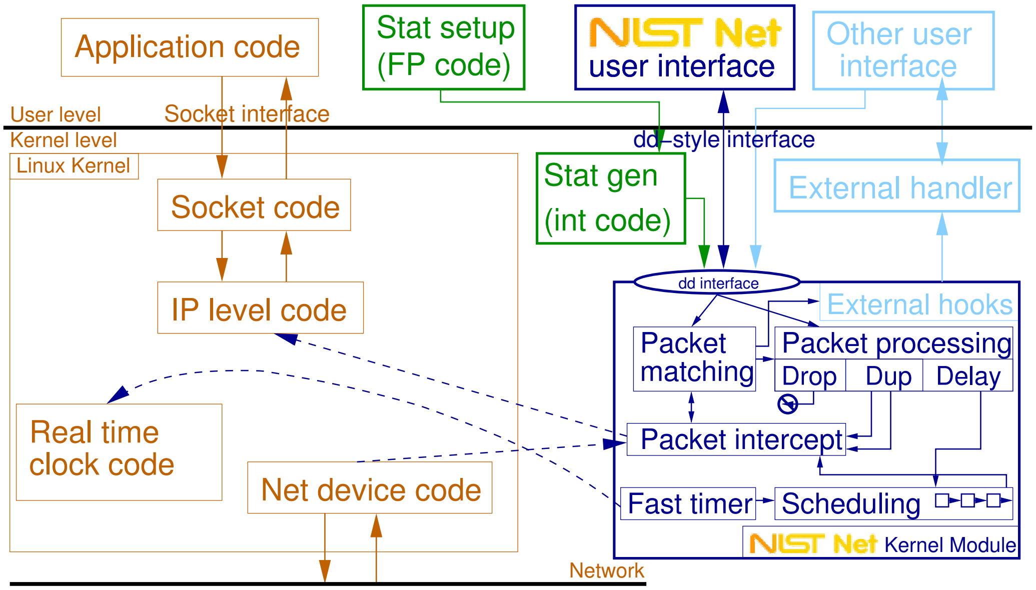
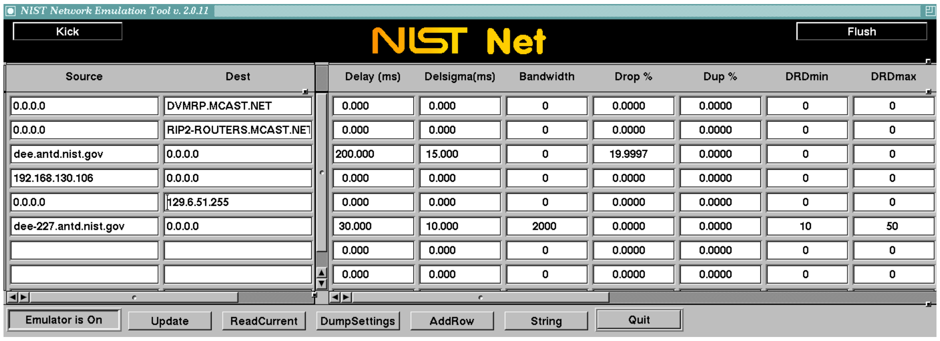


Figure 2: NIST Net Architecture



Source	Dest	Delay (ms)	Delsigma(ms)	Bandwidth	Drop %	Dup %	DRDmin	DRDmax
0.0.0.0	DVMRP.MCAST.NET	0.000	0.000	0	0.0000	0.0000	0	0
0.0.0.0	RIP2-ROUTERS.MCAST.NET	0.000	0.000	0	0.0000	0.0000	0	0
dee.antd.nist.gov	0.0.0.0	200.000	15.000	0	19.9997	0.0000	0	0
192.168.130.106	0.0.0.0	0.000	0.000	0	0.0000	0.0000	0	0
0.0.0.0	129.6.51.255	0.000	0.000	0	0.0000	0.0000	0	0
dee-227.antd.nist.gov	0.0.0.0	30.000	10.000	2000	0.0000	0.0000	10	50
		0.000	0.000	0	0.0000	0.0000	0	0
		0.000	0.000	0	0.0000	0.0000	0	0

Figure 3: NIST Net graphical user interface

(shown in Figure 3), which allows controlling and monitoring a large number of emulator entries simultaneously. Once a desired suite of emulator settings is created, it may be dumped for reloading in later runs, which simplifies repeated testing under identical conditions.

The NIST Net kernel module makes use of two hooks into the Linux kernel proper. In order to inspect all incoming packets for potential handling, the *packet intercept* code seizes control of the IP packet type handler. All IP packets received by network devices are then passed directly to the NIST Net module. After *packet matching* determines (based on the table of emulator entries) whether and how *packet processing* should affect the packet, NIST Net then (possibly after delay) passes the packet on to the Linux IP level code. The *fast timer* takes control of the system real time clock and uses it as a timer source for *scheduling* delayed packets. In the process, it reprograms the clock to interrupt at a sufficiently high rate for fine-grained packet delays.

NIST Net allows *external hooks* into its module as well. External *statistics generation code* may supply values for NIST Net's generation of network effects. (An example of such code is presented in section 4.3.) Alternatively, *external handlers* may work in concert with NIST Net or take over packet processing entirely. A variety of external handlers have been developed for such areas as flow monitoring, wireless network bit error emulation, and voice over IP testing.

In the sections which follow, we describe some of the performance-critical elements in the implementation of NIST Net.

3.1 NIST Net indexing structures

Emulator entries are stored in a table, which must be accessible both at task time (so user-level code may enter new settings) and at interrupt time (so that the packet matching code may look up entries when new packets arrive). Table lookup needs to be relatively fast to keep up with packet arrival rates, and yet needs to provide a flexible wildcard matching scheme. To meet these requirements, we use a two-level hash table to hold the emulator settings. A two-level hash table is a hash table with chaining, where the chains themselves are *redistributed* into secondary hash tables (using a secondary hash function) if they become too long (Figure 4).

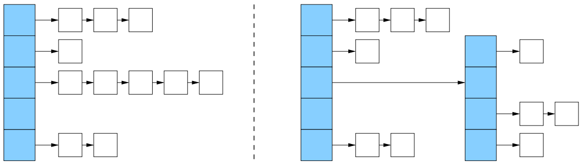


Figure 4: Redistribution in a two-level hash table

Several considerations motivated this two-level approach, as opposed to a single larger hash table: (1) Most users only use a few emulator entries at a time, for which the default hash table size (256) should be more than adequate. (2) If a larger table is needed, it is less disruptive for the reconstruction to happen piecemeal (by default, only to chains which have reached 11 entries in length) than to reallocate an entire new larger table. (3) If many entries are clustering around a single hash table slot, it may be an indication that the hash function is not separating entries well. In such a case, using a second hash function to separate the clusters is a better strategy than trying finer divisions with the same hash function. NIST Net uses a simple congruential hash function for the first level table, and a more complex function for the second [12], which, while slower, should presumably be less subject to inadvertent clustering. In experiments, this two-level hash structure has resulted in minimal overhead (usually $<< 1\mu$sec per lookup) even with several thousand emulator entries.

Using what is effectively an elaboration of a simple hash table makes lock handling during updates relatively trivial. To avoid the need for read locks, all updates are handled in a "safe" fashion (assuming atomic pointer writes), as suggested by Pugh [13]. Addition of new entries follows the usual scheme, but preserving consistency for readers during deletions is slightly more complex, requiring a two-step approach. The first step removes the item to be deleted from further "upstream" access as usual. The second step then leaves it pointing "downstream" but one step farther up, so that any readers which happen to be dwelling on the deleted node will be able to find the rest of the list, even in the face of multiple deletions (Figure 5). Deleted nodes are only recycled when a subsequent node allocation is requested, at which point any readers have long since exited.

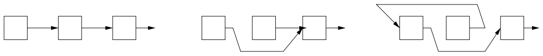


Figure 5: Read-stable node deletion

Write locking is handled at the (top) hash table slot level only; that is, each of the top-level hash table slots has an individual lock. This is a compromise position between a single overall table lock and individual node level locks. Since in the usual Linux kernel environment, the number of simultaneous writers is limited to the number of processors, this is more than adequate.

NIST Net allows selective matching of packets by most common header fields. Any and all fields may be wildcarded. Wildcarding is implemented through a (configurable) array of bitmasks, arranged from most to least specific. All emulator entries, whether wildcard or specific, are stored in the same table. Matching an incoming packet then (potentially) involves multiple passes through the table: the values in the packet header are masked by each set of bitmasks in turn, then the table is searched for the masked values. For an example, an incoming ping packet from 192.168.120.104 to 129.6.51.105 will match the following potential emulator entries (in order):

Source address	*Destination address*	*Protocol*	*Port*
192.168.120.104	129.6.51.105	icmp	echo-request
192.168.120.104	129.6.51.105	icmp	–
–	129.6.51.105	icmp	echo–request
192.168.120.104	–	icmp	echo-request
192.168.120.104	129.6.51.105	–	–
–	129.6.51.105	icmp	–
192.168.120.104	–	icmp	–
–	–	icmp	–
–	129.6.51.105	–	–
192.168.120.104	–	–	–
–	129.6.51.–	–	–
–	129.6.–.–	–	–
–	129.–.–.–	–	–
–	–	–	–

(*Port* is of course a misnomer for ICMP; for these entries, NIST Net uses the port field to store the type and code.)

NIST Net also (optionally) allows matching by the IP class/type-of-service (ToS) bits (which include the DiffServ code point). For brevity, the additional possibilities this creates have been omitted from the table above.

To decrease the overhead involved in making multiple searches through the table, we keep a small cache of previous lookups, by default of size two. If a new lookup request matches a cached request (that is, the request has the same packet header values as one of the cached ones), the cached result is returned. The idea in using only a two-entry cache is that in most cases, only a single stream is being actively affected by NIST Net at any one time. A two-element cache will then cover both directions of this stream. The cache is emptied any time table indices are modified, in case the modification might introduce a more specific match.

3.2 NIST Net timer handling

NIST Net uses a timer to replay delayed packets at the appropriate later point. To do this with sufficient fidelity requires a fine-grained (sub-millisecond level) timer source. For the initial versions of NIST Net on i386-style machines, the system i8253/4 timer chip was used as a source of interrupts (the timer "ticks"), reprogrammed to interrupt at a much higher rate than the fairly coarse Linux default of 100 Hz. Unfortunately, such reprogramming requires kernel source modifications, which over time proved to be a maintenance and installation issue. For this reason, current versions of NIST Net use the MC146818 real-time clock (RTC) as the timer interrupt source. Linux normally only makes fairly trivial use of this device, so it can be reused for NIST Net without fear of any substantial interference. NIST Net runs the RTC at its highest possible interrupt rate, 8192Hz, for a tick granularity of approximately 122 μsec.

In scheduling packets, NIST Net uses a variant of the Linux timer code, reimplemented to run on the RTC. This code maintains the list of timers to be run by a “just-in-time” radix sort. There are five levels of the sort. As a simple ad hoc optimization, the lowest level uses radix 256 (2^8), while the upper levels use radix 64 (2^6), together covering the entire 32-bit range (32 = 8 + 4*6). The basic idea is to avoid extra sort steps for short-lived timers, while keeping the upper level bins small.

At each level, the bins are maintained in a circular list; the pointer to the current list position advances as the clock ticks. Thus, at the lowest level, there are 256 nodes in the list, for timers which will go off at 0, 1, ..., 255 ticks in the future. At the next level, there are 64 nodes, for timers which will go off at 256-511, 512-767, ..., 16128-16383 ticks in the future, and so forth. The initial step in the radix sort is done when a timer is first added. Then, every time a lower-level list makes a complete circuit, the next batch of timers from the next level up is cascaded downward, which will cause them to undergo the next sorting step.

In the Linux timer implementation, each bin of timers is maintained as a stack in LIFO (“last in, first out”) fashion; with each cascade, then, the timers are all restacked. The result is that the sort is not stable; that is, timer events scheduled for the same clock tick do not necessarily occur in the same order that they were scheduled. In Linux usage, where timer events are presumably unrelated, this difference is immaterial, but in NIST Net’s usage, this had the effect of randomly reordering packets “clumped” into a single timer slot. While packet reordering does occur in real life, it does not normally occur to this extent, so this proved unintentionally harsh for some network software being tested. Hence, in the NIST Net implementation, the individual timer bins were modified to be FIFO (“first in, first out”), making the sort stable and eliminating the undesired reordering.

Unlike the Linux timer code, all the tasks whose timers have expired are gathered up into a separate, detached list before running them. This increases the preliminary processing somewhat, but has a positive overall effect on system stability and overhead, since interrupts need only be disabled once to gather the entire list and reenabled once to run all the timers, rather than disabled and reenabled repeatedly to run each one. When the delayed packets are rerun, they are reintroduced to the network software interrupt (as opposed to the RTC interrupt the timer functions run on), so that they will be processed in the correct context. The packets are marked so that NIST Net will know not to delay them again. Overall, this setup has proved stable even at extraordinarily high interrupt rates, up to a (i8253/4-based) 1.68 μsec tick granularity and gigabit Ethernet connections.

4 NIST Net statistical generation

NIST Net uses a set of statistical routines to generate the parameters for the impairments it applies to network traffic. The routines used need not be realistic models of the actual internal mechanisms of networks and routers, but should be computationally simple, and yet adaptable enough to imitate a wide range of network behaviors. In this context, *computationally simple* means two things: that generation of values needed for delays and other effects must be fast enough to keep up with packet receive rates (up to hundreds of thousands per second); and that any kernel-based calculations must be done using only integral or fixed-point arithmetic. (Linux does not support using the floating point unit in kernel mode.) Calculations done outside the kernel may be arbitrarily complex, but kernel memory constraints limit how much data (such as distribution table values) can be stored in the kernel at any one time. *Adaptable* here means that the routines should be parameterized in terms of quantities (such as mean, standard deviation, correlation) that are easily understood and easily derived from sources such as traffic traces, so that NIST Net can be made to imitate the behavior demonstrated by those traces. This section describes the techniques used to generate packet delay values; analogous techniques are used for other network effects. We present first the methods used in the released version of NIST Net, outline some of the limitations of these methods, and then describe the improved methods used in a new version of NIST Net currently under development.

4.1 Simple packet delays

The typical distribution of packet delays through the Internet features a "heavy tail" – a skewing toward the right (longer delay times) when compared to the symmetric normal distribution. As an example, the solid line in Figure 6 depicts the distribution of roundtrip ping times for a three-hour connection between machines in Gaithersburg, MD (ran.antd.nist.gov) and Boulder, CO (time.nist.gov).

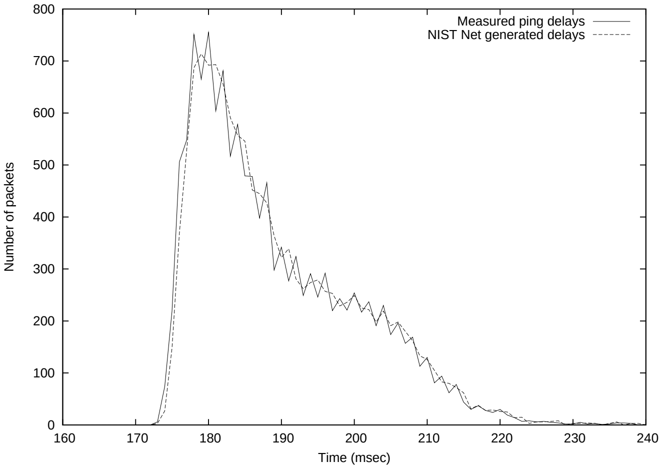


Figure 6: Real vs. NIST Net-synthesized delay distributions

Informally, this behavior makes sense – there are far more ways for things to go "wrong" for a packet (worse than average) than there are for things to go "right" (better than average). Various network simulation models have been devised to produce this delay pattern, and it can even be reasonably well approximated by closed-form analytical expressions. However, simulations cannot generally be run quickly enough to generate individual packet delays in real time, where packets can be arriving at the rate of thousands per second; and, while several closed-form approximations are provided as sample delay generation methods in the distributed NIST Net code, these do not really have the flexibility to match a very wide variety of delay patterns.

Instead, as our standard method, we use a much simpler approximation – a random lookup in a previously generated table. The approach used is based on a tabular approximation of the inverse of the cumulative distribution function (cdf), essentially as described in the simulation literature [14] [15].

In creating the tabular approximation, we use an evenly-spaced table of moderate size (normally 4096) which covers the range from -4.0 to +4.0 standard deviations. (In actuality, to avoid all use of floating point arithmetic, the table values are internally stored as 16-bit integers, and all internal calculations are carried out in fixed point.) In filling out the inverse table, we use linear interpolation. While in general it is more accurate to use Stieltjes partitioning [16], where the width of table entries is chosen based on the rate of variation of the cdf, in our case the cdf has fairly smooth behavior except at the extremities of the range, so that the simpler evenly-space table with linear interpolation suffices (as is shown in Figure 6).

The inverse table is normalized to produce values with mean of 0 and standard deviation 1. These values are then transformed via scaling and translation to yield the desired mean and standard deviation.

4.2 Simple correlation and μ and σ "correction"

The approach described above produces a distribution of the correct "shape," but without further modification, it misses many other features of real delay distributions. The most important of these is that successive packet delay values tend to be highly correlated. As an example, in the three-hour connection test mentioned above, the linear correlation (ρ) between successive packet delays over a three hour period was 0.478, fairly high, but within a typical range.

The released version of NIST Net takes a straightforward approach to reproducing this correlation – if the successively generated uncorrelated values are $x_1, x_2, x_3, ...$, then NIST Net uses $y_1, y_2, y_3, ...$, where

$$y_i = x_i * (1 - \rho) + \rho * y_{i-1}$$

Given that the x_i are uncorrelated, it follows immediately that the linear correlation between successive values of the y_i is indeed ρ. Unfortunately, this simple-minded method of forcing correlation will tend to dampen variance (assuming positive correlation). As a simple expedient, NIST Net "corrects" for this by using a larger σ initially; that is, rather than use $\mu + \sigma * x$, it uses $\mu + c(\rho) * \sigma * x$, where $c(\rho)$ is a suitable "correction" factor. This correction factor (which, as indicated, is dependent on ρ) is determined empirically *a priori*.

This method also causes some minor drift in the generated μ values, which NIST Net "corrects" for in a similar fashion. Thus, the values actually generated for a given μ, σ, ρ are

$$z_i = y_i * (1 - \rho) + \rho * y_{i-1}$$

where

$$y_i = d(\sigma, \rho) * \mu + c(\rho) * \sigma * x_i$$

and x_i is the value returned from the distribution table. Since the $d()$ and $c()$ factors are fixed, once μ, σ, ρ are given, they are all calculated in advance (during emulator configuration).

4.3 Improved generation methods

The method described above is computationally trivial and adequate for most purposes, but clearly can be improved upon. Its most obvious drawback is that it only considers correlation between successive delays and not longer-term correlation. Longer-term correlations as generated by NIST Net drop off exponentially at a much faster rate than real delays (see Figure 7). At a user level, this results in a perceptibly greater degree of "roughness" in the generated distribution – a greater degree of variability over randomly-chosen segments of the generated delays than is seen on average in real delay data.

In the latest version of NIST Net, we take advantage of the ability to graft on external statistics generation code, to introduce a more sophisticated method which better handles longer term correlations. The external method itself uses a two-level approach. The first level is based on Riedi's hybrid multifractal wavelet model (MWM) [17]. Originally devised to model bandwidth utilization at routers, MWM also works well at modeling end-to-end network delay patterns. As seen in Figure 7, MWM

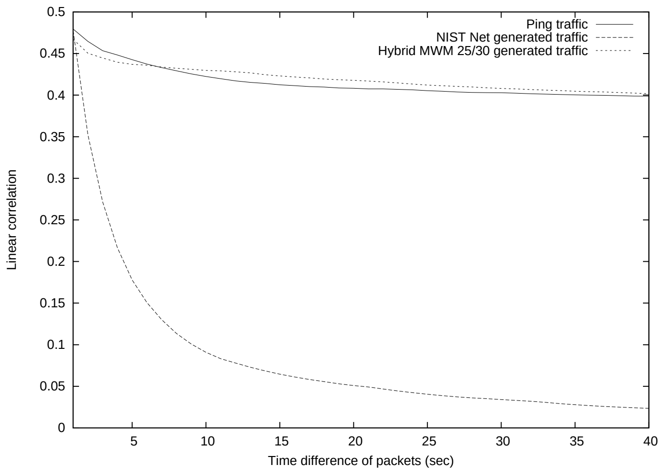


Figure 7: Long-term correlation of actual vs. generated delay values

tracks longer-term correlations quite well. The MWM model is run at user level, periodically sending new delay parameters to the kernel-level integral generator. In this configuration, the MWM model is used only to generate medium-term delay patterns (a range of seconds to minutes), which imposes only a very modest speed of generation requirement it can easily handle.

However, straightforward use of MWM does have limitations in modeling delays over longer time periods (hours or more). MWM works best when the distribution it is modeling is relatively stable over the entire modeled time period, but can give anomalous results when this is fails to be the case. As an example, Figure 8, shows (on the left) the trace of the Gaithersburg-Boulder delay data used throughout here. These data were collected for a period of several hours, beginning around 4:30pm on a weekday afternoon and continuing into the evening. There is thus a steady dropoff in the average delay in the first part of the data, as people leave work and network utilization decreases. There is also an increase in the variance of the data in the latter half, as automated network backups and other bulk transfers are scheduled. When fed directly to MWM for analysis and generation, these effects are not well-handled, as can be seen in the graph on the right in Figure 8. Here, the increased variance is spread evenly throughout the generated trace, and the changes in average delay show up at a few scattered points.

To overcome these limitations and still allow overall delay modeling to be done in an automated fashion, we add a front-end long-term analyzer as a second level on top of the MWM analyzer. The front

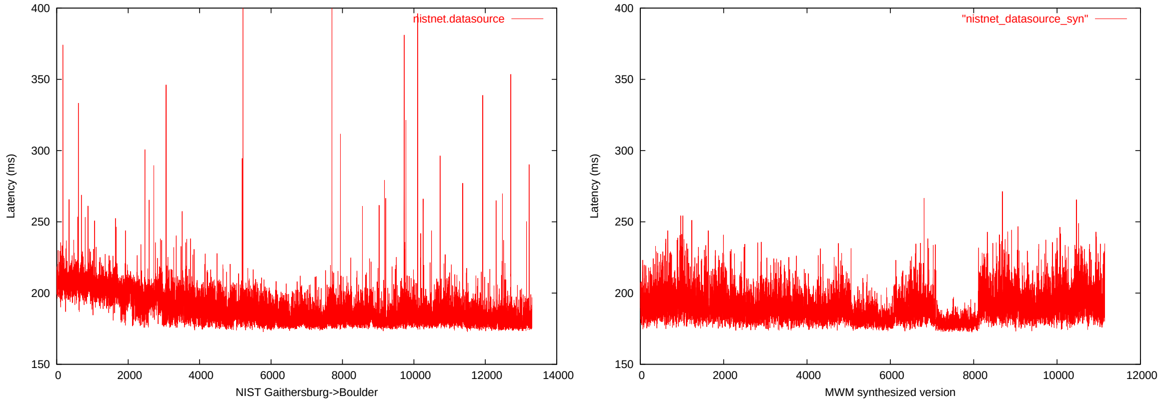


Figure 8: Trace of actual vs. MWM-generated delay values

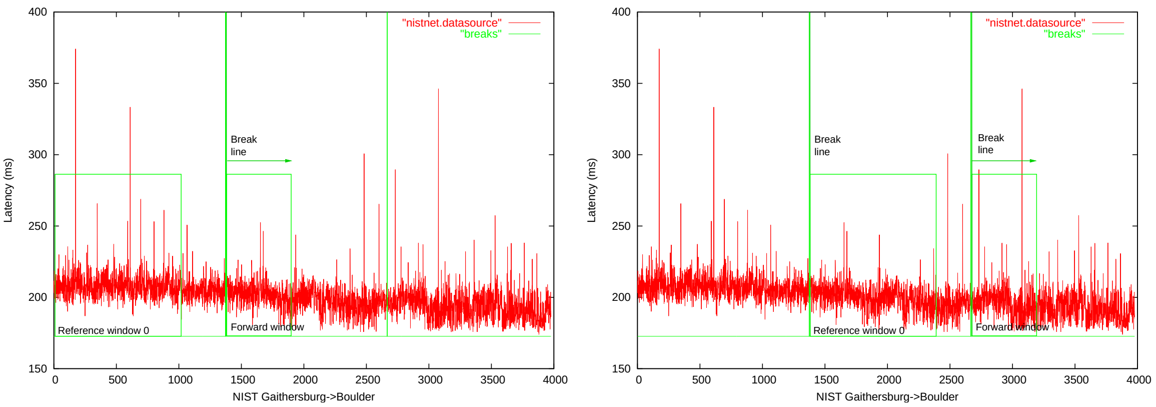


Figure 9: Break operation

end divides up long-term delay trace data into medium-term pieces which show relatively consistent statistical behavior (in terms of means and standard deviations). The front end works by advancing a window through the delay data being analyzed; when the statistical characteristics of the data in this window differ sufficiently from that seen in a reference window earlier in the data, the front end then marks a break at that point, and advances its reference window (see Figure 9).

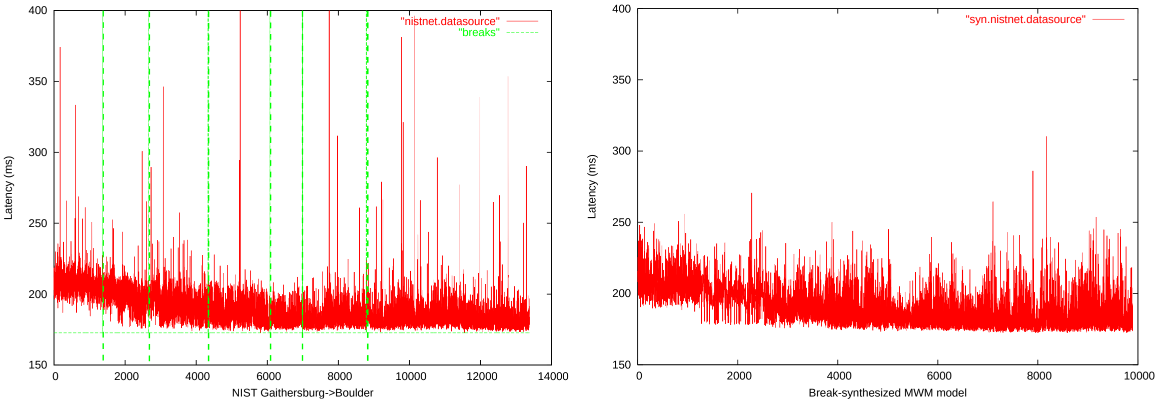


Figure 10: Overall break pattern and resulting MWM-regenerated delay values

The front end then submits each piece separately to the MWM model for analysis and generation, with the generated pieces then (optionally) reassembled (see Figure 10).

While better-matching than the original MWM model, the revised model still shows some weaknesses, especially in its portrayal of the magnitude of extreme values. To better handle these, we are currently working to employ other NIST work on extreme value modeling [18]. As part of our validation effort for this modeling approach, we are running this two-level model on the network trace data generated by the Active Measurement Project at NLANR [19]. One result of this validation effort will be a catalog of network behaviors, stored as sets of MWM generators, which can then be employed as desired in NIST Net emulations.

5 Calibration

To calibrate the fidelity and performance limits of NIST Net, we conducted a series of tests using a SmartBits 6000B Performance Analysis System [20]. This was used to drive loads of up to 1 Gbit/sec through a NIST Net system and then measure the resulting latency (Figure 11). The NIST Net platform under test had a 1666 MHz Athlon processor, 1 GB RAM, and dual NetGear 622T 64 bit/66 MHz PCI-based copper gigabit Ethernet interfaces. The operating system used was Red Hat 7.3, with Linux kernel version 2.4.18. The range of tests covered offered loads from 100 kbps to 1 Gbps, with packet sizes from 76 to 1518 bytes, NIST Net delay settings were from 0 to 10 seconds, with a variety of standard deviations and correlation constants applied. The results indicate that NIST Net runs and performs predictably up to gigabit line rates, even with packet rates in excess of 1 million packets/sec. Complete results are available at the NIST Net web site [21]; we will here consider a representative sample.

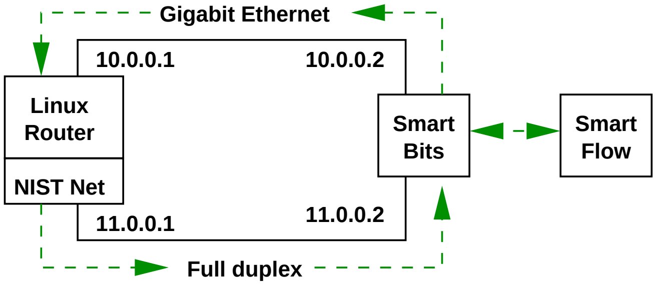


Figure 11: NIST Net testbed configuration

The table below shows the measured statistical characteristics of a 1 Mbit/sec stream of 76-byte packets (approximately 1644 packets/sec), being sent through the Linux router without the NIST Net module loaded (Control); with the NIST Net module loaded and matching packets, but imposing 0 msec delay; and with NIST Net imposing delays of 1, 10, 100 and 1000 msec:

Delay (msec)	*Mean latency (µsec)*	*Std dev (µsec)*
Control	15.65	3.89
0	17.90	6.23
1	1064.35	35.68
10	10097.78	35.52
100	100063.40	35.80
1000	1000081.54	35.42

When NIST Net is merely matching packets but imposing no delays, the effects are predictably small: a constant additional overhead (over and above the native Linux overhead) of approximately 2.2 μsec, and an increase in the standard deviation of the latency of a similar amount. Closer examination of packet traces shows the increased variance in the latency is due mainly to the addition of the 8192 Hz RTC interrupt; periodically, this will interrupt packet processing, adding a small but measurable amount to the overall latency. The amount of this inherent overhead will of course vary with the performance of the system running NIST Net; earlier investigations had shown that for a 200 MHz Pentium-class system with 10/100 Ethernet, the comparable overhead was on the order of 5-6 μsec.

Imposing constant delays with NIST Net has two noticeable (additional) effects: a constant additional overhead of approximately 50-80 μsec, and an increase in the standard deviation to approximately 35.5 μsec. Both are explained by the NIST Net tick granularity (approximately 122 μsec). When NIST Net delays a packet, it rounds up the delay time to the next multiple of this value. Assuming for simplicity's sake that packet arrivals are uncorrelated with respect to the NIST Net clock, the effect then is statistically equivalent to adding a uniformly distributed random variable with range [0:122 μsec] to the packet latency. Such a random variable has mean approximately 61 μsec and standard deviation approximately 35.5 μsec. Overall, then, when NIST Net is imposing delays, the actual observed latency is approximately $\mathrm{intended\ delay} + O(18)\mu\mathrm{sec} + U(122)\mu sec$ where $U(122)$ is a uniform random variable with range $[0 : 122]$.

At data rates above 100 Mbit/sec, other effects intrude. The most significant of these is "clumping": when the packet arrival rate exceeds the NIST Net clock interrupt rate, several packets can end up being "clumped" into the same timer slot. If packet clumps are large enough, then, when these delayed packets are rescheduled, there can be spot demands for output bandwidth exceeding the line rate of the output interface. In this case, packets can begin queuing at the interface level as well. As an example. total average measured latency increases by approximately 20-40% (over control levels) when constant delays are imposed at 500 Mbit/sec. NIST Net still functions reliably at these levels, but quantitative experiments need to take such lower-level queuing into account.

Although secondary in size to timer/interface-based clumping, NIST Net's own internal mechanisms begin to have a measurable effect at these very high rates. Here are the results of preliminary performance profiling of the most notable of these, using the Linux system clock as the time source. All of these tests involved data rates of roughly 100 to 200 Mbit/sec and packet rates of roughly 20,000/sec, with delays sufficient to queue several thousand packets at any one time (from 100 to 1000 msec).

Packet lookup in the emulator entry table (described in section 3.1) generally requires only trivial amounts of time, even with thousands of emulator entries. With two or fewer active streams, lookup overhead is unmeasurably small (less than 1 μsec), due to the caching mentioned above (section 3.1). When the number of active streams increases, lookup overhead increases somewhat, reaching a level of 3-4 μsec with 12 active streams and 1500 emulator entries.

Timer queue management (described in section 3.2) has more significant effects. Simple insertion or deletion of timer events (packets to be delayed) takes unmeasurably small time, even amid thousands of already-queued packets. The significant time expenditure is at each periodic radix sort step, when all the timers at the next higher level are cascaded downward in the radix sort tree. Interestingly, the overhead is highest (on the order of 30-50 μsec per cascade) when delays are constant, and drops considerably (to around 8-15 μsec) when large delay variances are introduced. This is presumably because larger variances scatter the timer entries more widely through the radix tree, so that individual cascades tend to be smaller.

Because timer expiration times are predetermined on an absolute basis, timer queue management overhead does not translate directly into per-packet overhead times, but clearly incurring the higher overhead levels every timer tick (122 μsec) can have a deleterious effect on overall system performance. In such a high-stress environment, NIST Net should only be used on otherwise unloaded systems to achieve consistent results.

6 Conclusion

In this paper, we have described the design and implementation of the NIST Net emulator, in particular the techniques employed to allow line-rate operation of emulation functions while still providing useful models of real end-to-end network performance dynamics. We presented some of the initial results of our ongoing efforts to improve the fidelity of the NIST Net statistical models, and to calibrate emulator functions.

Our experiences and the feedback from NIST Net's users indicate the value of the emulation approach for testing emerging network technology. Since its initial release, NIST Net has been obtained by nearly twenty thousand people around the world, and has been used for a wide variety of testing purposes, including for voice over IP, mobile network emulation, adaptive video transmissions, satellite and underseas radio link emulation, and interactive network gaming.

Because of its relative ease of installation and use, NIST Net has been widely used in academic and research laboratories (see [22], [23], [24], [25] for a few examples). In particular, the ability to specify complex network behaviors through a few easily-understood statistical parameters has made NIST Net a useful tool in a classroom setting, being used for a wide variety of student projects.

The NIST Net code, documentation, and calibration results are all in the public domain and are available through its web site [21].

7 Acknowledgments

The work described here was supported in part by the Defense Advanced Research Project Agency's Intelligent Collaboration and Visualization (ICV) and Network Measurement and Simulation (NMS) projects. We would also like to thank our colleagues at NIST, in particular Doug Montgomery, John Lu and Kevin Mills, for their valuable advice throughout the course of this project and during the preparation of this manuscript.

References

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

[2] Jong Suk Ahn, Peter B. Danzig, Zhen Liu, and Limin Yan. Evaluation of TCP Vegas: Emulation and experiment. In *SIGCOMM '95*, 1995. http://excalibur.usc.edu/research/vegas/doc/vegas.html.

[3] Nigel Davies, Gordon Blair, Keith Cheverst, and Adrian Friday. A network emulator to support the development of adaptive applications. In *Proceedings of the 2nd Usenix Symposium on Mobile and Location Independent Computing*, 1995. http://www.comp.lancs.ac.uk/computing/research/mpg/most/emulator.html.

[4] Mark Allman, Adam Caldwell, and Shawn Ostermann. ONE: The Ohio network emulator. Technical Report TR-19972, Ohio University, 1997. http://irg.cs.ohiou.edu/one/tr19972.ps.

[5] Scott Dawson, Farnam Jahanian, and Todd Mitton. ORCHESTRA: A fault injection environment for distributed systems. Technical Report CSE-TR-318-96, University of Michigan, 1996. ftp://rtcl.eecs.umich.edu/outgoing/sdawson/CSE-TR-318-96.ps.gz.

[6] B. Noble, M. Satyanarayanan, G. Nguyen, and R. Katz. Trace-based mobile network emulation. In *SIGCOMM '97*, 1997. http://www.cs.cmu.edu/afs/cs/project/coda/Web/docdir/sigcomm97.pdf.

[7] Kevin Fall. Network emulation in the Vint/NS simulator. In *ISCC99*, 1999. http://www.cs.berkeley.edu/ kfall/papers/iscc99.ps.

[8] L. Brakmo and L. Peterson. Experiences with network simulation. In *SIGMETRICS '96*, 1996. http://www.cs.arizona.edu/xkernel/www/people/brakmo.html.

[9] Mark Gaynor. Proactive packet dropping methods for TCP gateways. http://www.eecs.harvard.edu/ gaynor/final.ps, November 1996.

[10] Sally Floyd and Van Jacobson. Random early detection gateways for congestion avoidance. *IEEE/ACM Transactions on Networking*, 1(4):397–413, August 1993.

[11] K.K Ramakrishnan and Sally Floyd. A proposal to add explicit congestion notification (ECN) to IP. *RFC 2481*, 1999. http://www.ietf.org/rfc/rfc2481.txt.

[12] Robert J. Jenkins, Jr. Hash functions for hash table lookup. http://burtleburtle.net/bob/hash/evahash.html, 1997.

[13] William Pugh. Concurrent maintenance of skip lists. Technical Report UMIACS-TR-90-80,CS-TR-2222.1, University of Maryland, 1990. ftp://ftp.cs.umd.edu/pub/papers/papers/ncstrl.umcp/CS-TR-2222/CS-TR-2222.ps.Z.

[14] H.C. Chen and Y. Asau. On generating random variates from an empirical distribution. *AIEE Transactions*, 6:163–166, 1974.

[15] Paul Bratley, Bennett L. Fox, and Linus E. Schrage. *A Guide To Simulation*, page 149. Springer Verlag, 1987.

[16] J.H. Ahrens and K.D. Kohrt. Computer methods for efficient sampling from largely arbitrary statistical distributions. *Computing*, 26:19–31, 1981.

[17] R. H. Riedi, M. S. Crouse, V. J. Ribeiro, and R. G. Baraniuk. A multifractal wavelet model with application to network traffic. In *IEEE Transactions on Information Theory (Special Issue on Multiscale Signal Analysis and Modeling)*, volume 45, pages 992–1018, April 1999. http://www.dsp.rice.edu/publications/pub/riedibmw.ps.gz.

[18] Z.Q. John Lu and Nell Sedransk. Generalized pareto mixture models for network traffic with applications to performance evaluation, October 2002. manuscript in preparation.

[19] Todd Hansen, Jose Otero, Tony McGregor, and Hans-Werner Braun. Active measurement data analysis techniques, March 2000. http://watt.nlanr.net/.

[20] Spirent Communications. http://smartbits.spirentcom.com/.

[21] NIST Net web site. http://www.antd.nist.gov/nistnet.

[22] S. Das, M. Gerla, S. S. Lee, G. Pau, K. Yamada, and H.Yu. Practical qos network system with fault tolerance. http://www.cs.ucla.edu/ nrl/hpi/papers/2002-spects-0.pdf.

[23] Phil Kearns. http://www.cs.wm.edu/ kearns/001lab.d/labuse.html.

[24] Bhaskaran Raman, Yan Chen, Weidong Cui, and Randy Katz. Wide area network emulation on the millennium, June 2001. http://www.cs.berkeley.edu/ bhaskar/iceberg/pres/jun2001-retreat/wane-mill.ppt.

[25] Tascnets.com software. http://www.tascnets.com/newtascnets/Facilities/Documents/Main.html.