Real-Time IP Network Simulation at Gigabit Speeds

Marko Zec and Miljenko Mikuc, *Faculty of Electrical Engineering and Computing, University of Zagreb*

Abstract—In most of the network research or development projects, as well as in teaching / academia, network simulators today play an important role. Network simulation tools can be generally classified as those performing simulations off-line in form of batch-type tasks, and those operating in real-time, often with the ability to interact with the real network hardware and traffic. However, limited performance remains one of the main issues associated with most of the current real-time simulator implementations. In this article we present a novel framework for constructing high performance real-time IP network simulators, capable of operating at gigabit speeds using a commodity industry-standard PC hardware. Our implementation, based on a modified 4.4BSD operating system (OS) kernel, permits complex simulated network configurations to be constructed using basic building components: virtual nodes and links. Each virtual node can act as an entirely independent but completely functional OS image, allowing standard unmodified UNIX services and utilities to be used in the experiments, while retaining nearly the same performance properties as in the original OS.

Index Terms—Computer networks, modeling, operating systems, simulation software

I. Introduction

Network simulation tools are today extensively used in many areas of network protocol research and development, in academia and teaching, or for predicting the behavior and characteristics of complex network installations under different operating conditions. There are many driving factors in favor of using simulators, most notably simulations being much more economical than doing similar experiments with the real hardware. However, the cost factor alone is not the only advantage: in many cases (such as teaching in labs) each workplace can be presented with its own simulated network environment, which would be in most cases impossible to achieve with real network equipment, even with unlimited financial resources. Simulation results are also usually easier to collect, correlate and analyze, as information at all critical points in the observed network configuration is being logged and accessible on a single machine.

Most of the network simulation tools and products currently available fall into one of the following two categories:

1) *Offline simulators* are usually highly configurable and extensible tools designed to simulate processes in a network in virtual timescale, which is not linearly related to the real time. The simulations are executed in form of scheduled tasks, which results can be analyzed upon the simulation completion. Typical examples of such tools are Berkeley network simulator [6] and OPNet.

2) *Real-time simulators* are tools capable of creating virtual network topologies and simulating traffic processing effects in real or *scaled* timeframes. Such products are generally less flexible and extensible than their *offline* counterparts; however the major advantage of many such tools lies in their ability to interact with real network infrastructure and traffic. Therefore these types of network simulators are often referred to as *emulators*. Some tools falling in this category, such as ENTRAPID [1], Alpine [2] and Harvard NS [3], are discussed later in text.

In this article we present architecture for constructing high performance real-time IP network simulators. The framework is based on a modified 4.4BSD [4] operating system *kernel* and network stack [5]. Our modifications allow simultaneous operation of multiple independent network stacks within a single kernel. Each network stack instance can act as an independent virtual node, connected either to other virtual nodes via simulated links, or directly to the outside world via standard network interfaces. This allows complex simulated IP network configurations to be set up on a single machine and observed and analyzed at the level of each independent virtual node, link, or network interface. The simulator can easily interact with real networks through standard physical interfaces at up to gigabit speeds, depending on simulated network complexity and simulator hardware capabilities, such as CPU speed and caching efficiency, memory and peripheral bus bandwidth etc.

The rest of the article is organized as follows. Section II explains the basic implementation concepts behind *clonable* network stack architecture, which presents the foundation for building our network simulation configurations. In section III, as an example, we present a simple simulation scenario that can be achieved using the clonable network stack framework. Section IV discusses the performance aspects of our real-time simulation environment. Previous and related work is outlined in section V, followed by a conclusion and directions for future research outlined in section VI.

II. CLONABLE NETWORK STACKS AND VIRTUAL IMAGES

Real-time network simulators can be implemented either as non-transparent *clouds,* using complex mathematical models, with border outlets which present ingress / egress points to the simulated network, or as a collection of independent virtual nodes and links each one resembling traffic processing characteristics of appropriate objects in real networks. The former cloud model in some cases provides better performance (overall throughput), but at the cost of limited simulation realism or *fidelity*. Most notably, in such implementations the transient virtual nodes and links cannot be observed as independent objects, as they are all hidden behind a monolithic mathematical model and the simulation engine as its executor. In other words, a simulated multi-hop networks will be in fact presented to the outside observer as a single-hop full-mesh cloud, with the traffic processing properties aiming to resemble as closely as possible the multi-hop topology, for each pair of ingress / egress nodes at the network edge.

In contrast to the cloud approach, we propose a transparent simulation model consisting of entirely independent virtual nodes and links, which can be individually configured, interconnected, accessed and observed just as their physical counterparts in real networks. We do not claim any paternity to such approach, as it has been described earlier in different variations by numerous authors (see section V). However, our main contribution lies in a *highly efficient implementation of virtual network infrastructure*, in form of a general-purpose operating system (OS) kernel divided into multiple independent *virtual images*, as shown in Figure 1.

A typical general-purpose OS consists of multiple user processes and a kernel, which has a primary role in providing a standardized abstraction, protection and scheduling layer for accessing all the system resources – most notably the CPU, memory, file systems, diverse physical devices, as well as the interprocess and network communication facilities. In our initial experiments, we have modified an open-source, 4.4BSD-based OS kernel (FreeBSD 4.7 [7]), to allow multiple network stacks to be simultaneously active on a system. Each network stack instance was made fully independent of all others, so that each instance maintained its own private routing table, set of communication sockets and associated protocol control blocks, etc. Further, each network interface (either physical or virtual) could be associated with one and only one network stack instance at a time. The network stack virtualization experiment was extended to include optional networking facilities, such as packet filters, traffic shapers, bridging code and various *sysctl* [4] tunable variables controlling different aspects of network stack behavior.

As UNIX systems traditionally maintain only a single network stack within the kernel, an important design step was choosing the optimal method for user processes to manage multiple network stacks. One option was modification of the standard Berkeley socket interface [4], by extending the argument lists with the network stack identifier. Other variation of such approach was proposed in [8]. However, this concept has a significant drawback for all existing user programs have to be modified and recompiled in order to be able to run on the new / extended OS kernel. Therefore, an alternative approach was chosen. Each user process control block in the kernel was

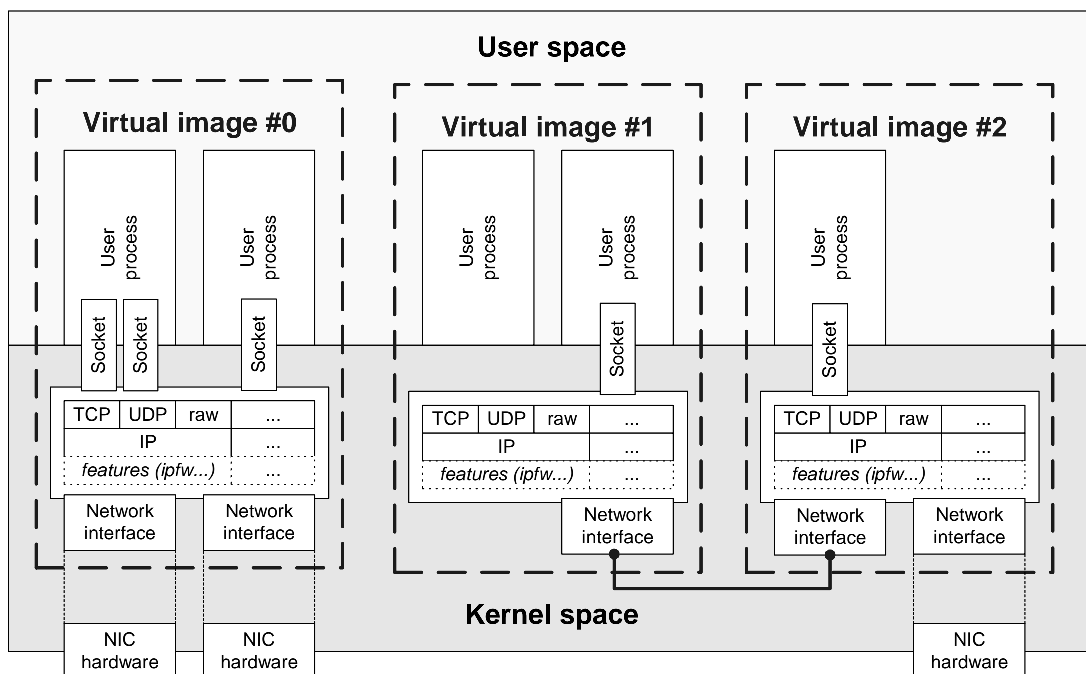


Figure 1 – operating system components separated in isolated *virtual images*

transparently extended with a tag which associates it with a network stack instance. This tag is inherited by subsequent processes from its parents without any need for intervention from the programmer. Additionally, new programming interface allowing a process to change its network stack association was introduced. This approach allowed for complete application programming and binary interface (API / ABI) compatibility to be preserved between the original and modified OS kernel, thus mitigating any need for modifications in the existing userland applications or utilities.

The described tagging of user processes was combined with already existing *jail* [11] resource protection framework in FreeBSD, which resulted in user processes associated with one network stack being effectively invisible to the other processes running on the system, and vice versa. The newly developed framework, which combined different areas of resource protection mechanisms into one entity, in fact achieved *light* virtual machine functionality. Therefore we named each collection of network stack instance and a set of its associated userland processes a *virtual image*. Latter, this concept was further extended by including modification to the CPU scheduler, in a way that each virtual image could be limited in average CPU usage, so that runaway or maliciously constructed process or group of processes might be prevented from monopolizing and starving all the real CPU resources. This also allows system load monitoring to be performed on per virtual image basis, which provides more fine-grained control rather than accounting resource usage solely on physical machine level. Finally, a basic API for managing the virtual images was implemented, accompanied by a simple userland utility.

The fundamental approach taken in implementation of the described modifications to the BSD OS kernel was introduction of a new vimage kernel structure, which serves as a container for all virtualized variables and symbols. Gradually, most of the global and static symbols used by network stack code were replaced by their equivalent counterparts residing in independent vimage structures. Network interfaces descriptors, which have traditionally been maintained in a single linked list, are now associated with vimage structures, so that each network stack instance has its own list of network interfaces. Each network interface contains a pointer back to its vimage structure, so that incoming traffic can be easily demultiplexed to the appropriate network stack, depending on the interface the traffic is received on. Basic schematic diagram outlining relations between most important kernel structures in clonable network stack implementation is shown in Figure 2. More deep coverage of implementation details falls outside the scope of this article, so curious readers are pointed to [17] which covers this matter more thoroughly.

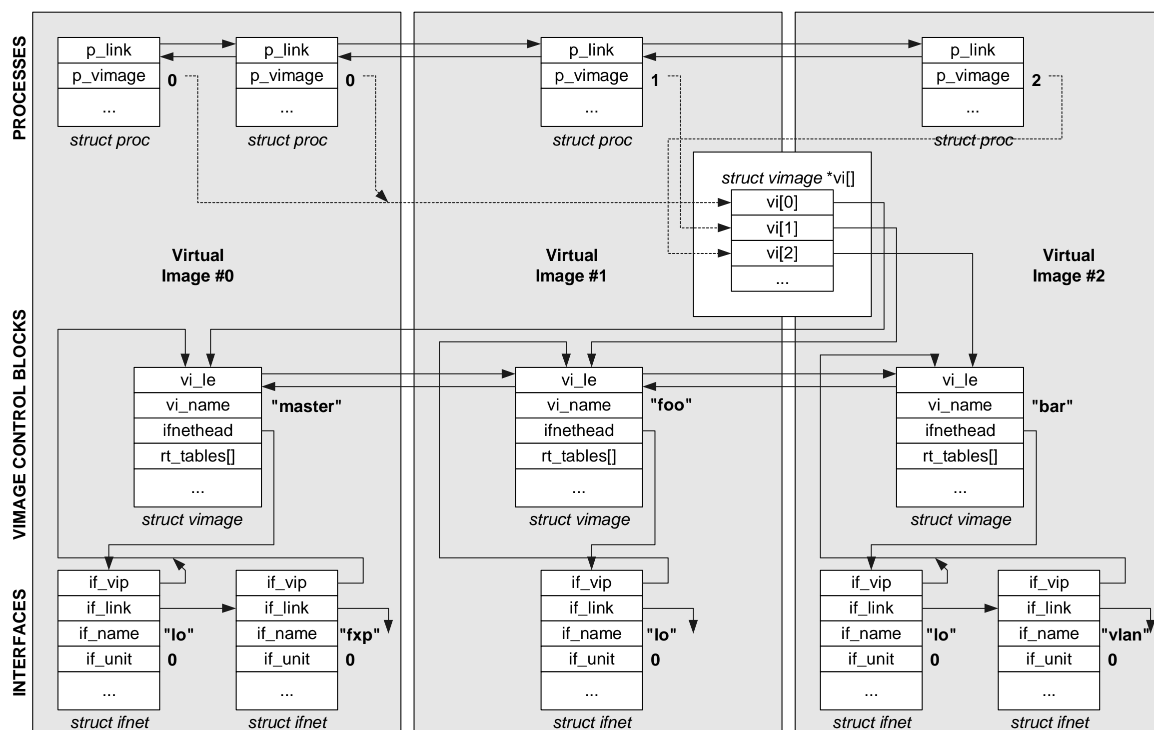


Figure 2 – links between kernel structures associated / separated throughout *virtual images*

III. Simulation Scenarios

The basic property of each virtual image is that its network communication facilities are independent of other virtual images present in the system. It is therefore necessary to provide explicit methods for establishing communication between isolated network stacks. Our framework provides two such facilities.

For point-to-point style communication between virtual images, a pair of FreeBSD's standard *netgraph* [10] interface nodes can be employed, with one instance residing in each virtual image. The interface nodes can be connected back-to-back, thus forming an efficient in-kernel traffic path.

The alternative is using virtual Ethernet interfaces, which can be *bridged* to form independent broadcast domains spanning multiple virtual images. The bridging domains can be either entirely hidden inside the kernel, or connected to the outside world via one or more physical Ethernet interfaces.

Using the concept of virtual nodes and links, it is possible to configure and simulate complex network topologies, however in this article we are limited to presenting only a simple example. In our example, the simulated network consists of three virtual nodes named *east, west* and *north*, as shown in Figure 3. The nodes *east* and *west* are each assigned one of the two physical Ethernet interfaces present in the host system. The physical Ethernet interfaces are attached to two isolated LAN segments, on which external hosts *tindy1* and *tpx30* are connected. Virtual nodes *west* and *north* are linked through an internal *virtual Ethernet* segment, and *east* is linked to *north* through a point-to-point *netgraph* tunnel, which also includes an *ng_dummy* [13] traffic shaper. An instance of routing daemon is started in each virtual node, allowing dynamic propagation of routing information throughout the simulated network.

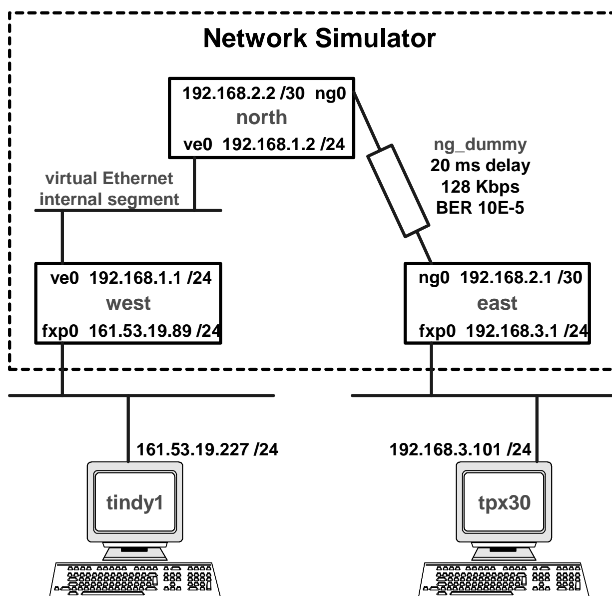


Figure 3 – a simple network simulation topology

The following shell script can be used to configure the simulator in accordance with the described topology:

```
#!/bin/csh

# create virtual images and start routing daemons
foreach vi_name (east west north)
        vimage -c $vi_name
        vimage $vi_name routed
end

# assign the eth interfaces to nodes east and west
vimage -i west fxp0
vimage west ifconfig fxp0 161.53.19.89/24
vimage -i east fxp1
vimage east ifconfig fxp0 192.168.3.1/24

# configure internal eth. interfaces
ifconfig ve0 create link 0:0:0:0:0:a
ifconfig ve1 create link 0:0:0:0:0:b
vimage -i west ve0
vimage -i north ve1
vimage west ifconfig ve0 192.168.1.1/24
vimage north ifconfig ve0 192.168.1.2/24

# do bridging between west and north
sysctl \
net.link.ether.bridge_cfg="west.ve0,north.ve0"
sysctl net.link.ether.bridge=1

# link north and east
ngctl mkpeer iface inet inet
ngctl mkpeer ng0: dummy inet upper
ngctl name ng0:inet dummy0
ngctl mkpeer dummy0: iface lower inet
vimage -i north ng0
vimage -i east ng1
vimage north ifconfig ng0 192.168.2.2 192.168.2.1
vimage east ifconfig ng0 192.168.2.1 192.168.2.2

# configure traffic shaping on link north-east
ngctl msg dummy0: setcfg \
"{ upstream={ bw=128000 dly=20 ber=100000 }}"
ngctl msg dummy0: setcfg \
"{ downstream={ bw=128000 dly=20 ber=100000 }}"
```

A simple verification the above script correctly implements the network topology presented in Figure 3 could be obtained by tracing the IP route from external host *tpx30* to *tindy1*. The external hosts must however either accept RIP updates from the simulator, or have statically configured routes in order to be able to communicate with each other.

```
tpx30>tracert 161.53.19.227
Tracing route to 161.53.19.227 over a maximum of
30 hops

  1    <1 ms    <1 ms    <1 ms  192.168.3.1
  2    39 ms    39 ms    39 ms  192.168.2.2
  3    40 ms    39 ms    39 ms  192.168.1.1
  4    39 ms    40 ms    39 ms  161.53.19.227

Trace complete.
```

Different queuing disciplines and traffic shaping implementations generally account for the most basic functions of any network simulator. In the above example, a simple *ng_dummy* traffic shaper was used. This implementation features FIFO / drop-tail queuing based bandwidth limiting and delay simulation, together with bit error-rate (BER) and rudimentary phantom traffic / congestion simulation. More advanced network-layer traffic classification and queuing policies, such as worst case-fair weighted-fair queuing (WF2Q) or random early detection (RED) can be simulated using *dummynet* [9] facility, which is integrated in the base FreeBSD system. Both *ng_dummy* and *dummynet* shapers could be mixed in the same simulation configuration, if desired.

Independently of traffic shaping / queuing facility used, it is necessary to ensure low delay jitter and smooth dequeuing of network traffic, in order to obtain acceptable simulation quality. This is particularly important at higher data rates, both because of TCP windowing scheme and overall throughput being highly sensitive on end-to-end transmission delays, and due to limited queue depths available at different stages of packet forwarding paths inside the simulator. On the other hand, forcing the system into extremely frequent polling of simulation queues could lead to CPU resources being wasted and in the end result in overall system performance being degraded. Unfortunately there's no universal rule of thumb determining the optimal polling rate, but based on our experience we can recommend using 1000 Hz or 2000 Hz for most application, which translates to delay granularity / average jitter of 1 ms / 0.5 ms and 0.5 ms / 0.25 ms respectively. For smooth simulations at speeds exceeding 100 Mbit/s even higher polling frequency may be required, however the upper limit depends on the simulator hardware performance and the topological complexity of the simulated network. The polling frequency must be fixed at the time of kernel compilation using the standard BSD kernel configuration file.

An important property inherent to the proposed simulation framework is **isochronous** timing across all virtual nodes. This simplifies correlation and comparative analyses of traffic traces performed simultaneously in different virtual nodes, in contrast to real networks, where it is extremely difficult or in many cases impossible to achieve clock synchronism for timestamping purposes among distributed network nodes, at least not with an acceptable accuracy for traffic rates in range of or exceeding100 Mbit/s.

IV. PERFORMANCE

Enabling high performance real-time network simulation was one of the key design goals behind the proposed concept of virtual OS images and clonable network stacks. Our objective was to implement the required modifications to the 4.4BSD network stack without introducing significant performance degradations, compared to the original (unmodified) stack. To determine the actual performance properties of our simulation framework, we performed a series of simple tests. The experiments and measurements were executed on the same referent hardware in two different scenarios, as shown in Figure 4.

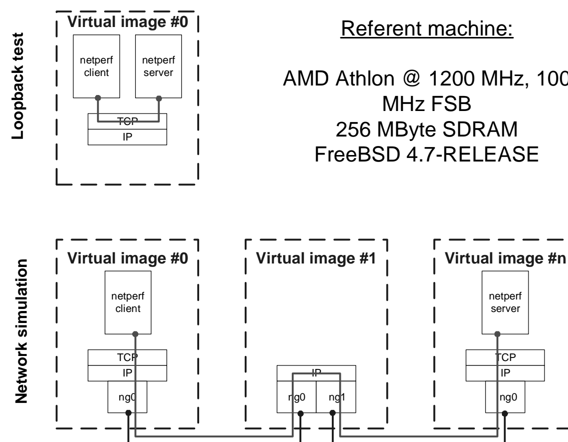


Figure 4 – "loopback" (top) and "multiple virtual hops" (bottom) testbed scenarios

The objective of the *first* test was comparing the performance between the original and modified 4.4BSD network stack. The test involved measuring loopback TCP throughput using the common *netperf* [12] throughput measurement tool, with both the sending and receiving process residing in the same machine. In case of the modified network stack, both the sender and the receiver were located in the same virtual image. The TCP throughput test was repeated for different *maximum transmission unit* (MTU) values set on the loopback interface.

The results of this test, as shown in Figure 5, suggest that the extensions / modifications to the network stack had only a slight impact on the maximum TCP throughput. For MTU value of 1500 octets, the throughput achieved using the modified kernel was around 93% of values observed on the standard system. However, it should be noted that during the test traffic passed through the network stack twice: once when data was transmitted by the sending process, and once when the same data was received by the other process. It is clear that the one-way throughput degradation must be even less significant, and can be estimated as square root of the obtained throughput ratio between standard and modified stack for both sending and receiving side processing. Therefore, for MTU=1500 we can estimate one-way maximum TCP throughput of the modified network stack to be around 96.5% of the standard (unmodified) system.

In the *second* test scenario we measured the maximum TCP throughput as a function of the number of virtual hops. As shown in Figure 6a, the throughput asymptotically decreases with each additional transient hop. This is a natural consequence of the fact that the simulator has to

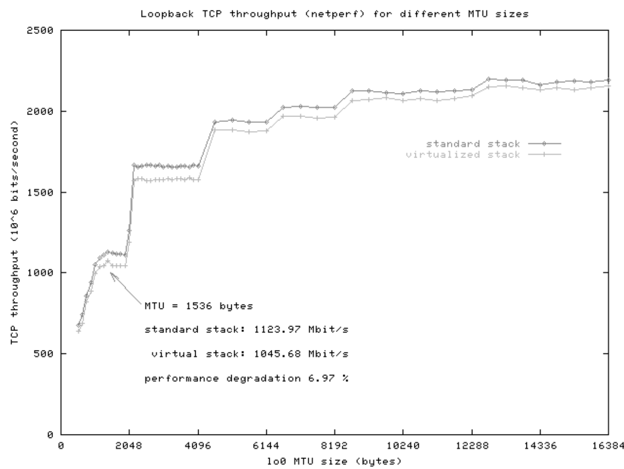


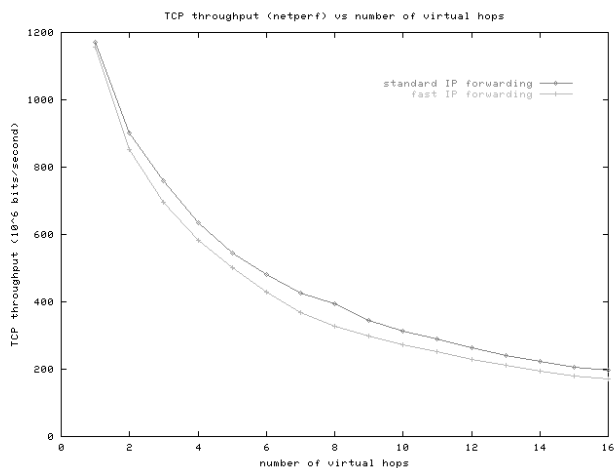
Figure 5 – maximum loopback TCP throughput on standard and modified network stack

perform the complete set of IP switching tasks as defined in [14] for each single packet in each individual virtual node. It should be noted that in described scenario the simulator not only switched the packets between the virtual nodes, but also terminated the TCP sessions on edge nodes. Despite this additional processing load, the simulator was able to operate around gigabit data rates for the smaller number of transient virtual hops. As our referent platform was relatively limited in terms of RAM bandwidth, CPU speed and cache size, compared to current top-notch IA-32 systems available on the market, migrating to more powerful hardware would certainly yield even better results.

This last test involved measurement of round trip time (RTT) as a function of number of virtual hops. The results show that each virtual hop approximately linearly contributes to the increase of overall RTT, as shown in Figure 6b. From the presented data it can be easily calculated that in one node the process of switching a single packet lasts approximately 1.25 microseconds in average, which translates to a maximum theoretical aggregate throughput of 800.000 packets per second for the whole system. However, in reality the impact of many different factors, such as device driver and interrupt processing overhead [15], system bus congestions, CPU cache trashing etc. would result in lower effective throughputs.

The tests have proven our framework is capable of performing simulation at gigabit speeds for network topologies with limited end-to-end hop count. In case more complex topological configurations need to be simulated, it would be possible to combine multiple machines into a network simulation cluster, if CPU power on a single box would become a bottleneck. Independent logical links could be multiplexed over a single physical link between different machines, as the support for virtual LANs (VLANs) has been incorporated into the simulation framework.

The clustering approach described above provides a simple methodology for ensuring aggregate throughput scaling in cases where CPU power is scarce. Unfortunately, the current simulator implementation cannot take any advantage of symmetric multiprocessing (SMP) hardware, since FreeBSD version 4.7, our simulator is based on, does not support multiple simultaneous threads of execution within the kernel. In the future this is likely to change with the migration to FreeBSD 5.0 code base, which provides much finer-grained locking of critical sections within the kernel, and thereby the foundation for presumably efficient SMP support.



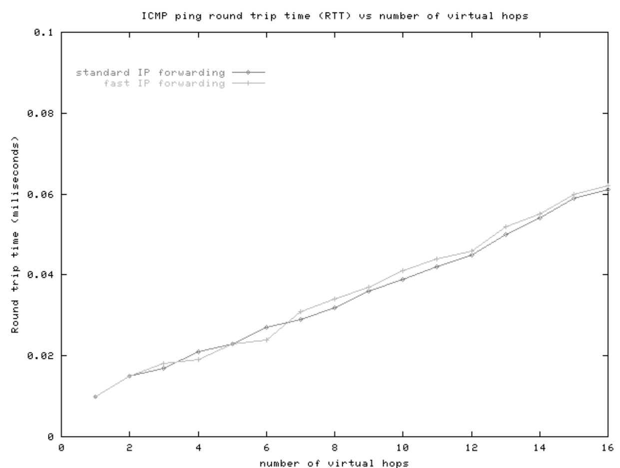


Figure 6 - a) maximum TCP throughput as a function of number of virtual hops (left); b) round trip time for ICMP echo / reply packets as a function of number of virtual hops (right)

V. Related Work

As mentioned earlier in the text, the idea of constructing network simulators by reusing and extending an existing network stack implementation is not entirely new. Further we outline the main concepts behind some of the previously published works in this area.

The *ENTRAPID* [1] protocol development environment introduced a model of multiple virtualized networking kernels, which in effect present a reentrant variants of the standard 4.4BSD network stack in multiple instances, running as threads in a specialized user process. Such attempt of network stack virtualization in the user space successfully accomplished its primary goal of providing a flexible network simulation tool, but at the cost of poor overall performance compared to the real in-kernel network stack implementation on the same hardware, combined with very limited compatibility to the standard software applications relying on network communication. Another example that followed this approach is the *Alpine* simulator project [2].

The *Harvard network simulator* [3] created the illusion of having multiple independent kernel routing tables by providing transparent IP address remapping between user and kernel space. While the kernel still maintained a single routing table with unique (non-overlapping) entries, a translation table had to be established for each virtual node, which had to be consulted on each userland-to-kernel network transaction. Although such approach provided far better performance than the ENTRAPID architecture, it still had many limitations. The performance was significantly constrained by numerous translation lookups that had to be performed on each kernel-to-userland packet transition, and vice versa. However, probably the biggest advantage of the Harvard architecture is the ability to use the existing UNIX network applications in virtualized environment without any modifications, while the ENTRAPID and Alpine required at best some porting efforts and replacement of standard system libraries, up to the point when porting become entirely impossible. The Harvard simulator is conceptually the closest to our framework, because of the ability to transparently support unmodified userland applications on the modified kernel.

Entirely different approach was taken in design and implementation of the *Netbed* [18] simulation environment. A large cluster of PC-s, running FreeBSD or Linux operating systems, is used to simulate (or more precisely to *create*) complex experimental networks. The network topologies are constructed by combining real PCs as network nodes distributed in either local or remote physical clusters, together with either real or emulated WAN links, using *dummynet* traffic shapers for simulating WAN effects. The main operational issue of such system is management of huge number of nodes that have to be individually reconfigured for inclusion in specific experimental network, which becomes growingly difficult task as the PC clusters get more geographically distributed. Netbed software currently controls 168 and another 50 PCs at two research centers respectively. Each PC is equipped with five network cards, which are all connected together via high-end LAN switches, either for simulation or control and management purposes. It is obvious that construction and maintenance of such system also requires significant financial resources, as well as a dedicated team of operators.

VI. Conclusions and Future Research

We have presented a highly efficient network simulation framework that, in our opinion, includes many of the best properties of previously known real-time simulation systems. However, our framework is not limited only to network simulation applications. As the implemented extensions and modifications did not hinder the general-purpose nature of the underlying operating system and its kernel, our platform can potentially find use in creating diverse virtual hosting or virtual private network (VPN) provisioning appliances.

Our simulator framework is redistributable under a liberal BSD-style license. It can be downloaded from http://www.tel.fer.hr/zec/BSD/vimage/ in form of patches against the FreeBSD 4.7-RELEASE kernel source, plus sources for an additional management utility. Accordingly, the simulator framework could be conveniently evaluated and adjusted or extended for special purpose applications, if desired.

At the time of this writing, the simulator code is only *a framework* as described in this article, and not a complete software package. There are many possible directions for further research and improvement. As the simulator currently provides support only for IPv4 protocol suite, one logical development step would be inclusion of the emerging IPv6 protocol into the simulator code, as well as other commonly used protocols, such as IPX, AppleTalk etc. At the moment support for simulation of data-link layer media other than Ethernet and simple point-to-point is not included. With reasonable efforts, it should be relatively simple to build a basic Frame Relay switch emulator using the netgraph framework.

Currently, the only option for constructing and managing simulated network environments is the use of UNIX shell commands and scripts, similar to the example given in section III. To simplify the configuration of more complex topologies, it would be desirable to implement a graphical user interface (GUI) for managing the simulator. A parser for definition files in common format (such as *ns*) generated by some already existing tools could present a less demanding alternative to development of proprietary GUI.

Further, in order to stay synchronized with the latest advances in OS technology (especially in SMP support), as

the development emphasis in FreeBSD shifts towards the next major release / branch, our simulator should be migrated to FreeBSD 5.0 kernel. This is not a trivial task due to major architectural differences between 4.7 and 5.0 branches. However, such migration could also present an opportunity to restructure the resource partitioning scheme from the current monolithic one into a more modular model, in line with the concept of *resource containers* [16]. The modular model would allow system administrators to freely combine selected system resources, such as network stack instance, CPU time share limit, user processes, file systems, virtual / real memory etc. into a custom virtual image, in contrast to the current implementation where all the resources contained in a virtual image are hard-coded and cannot be configured at the run time.

Finally, the simulator should be thoroughly tested and evaluated with real Gigabit Ethernet equipment. Unfortunately, such hardware (high-end PCs, Gigabit Ethernet network cards and LAN switches) was not available at the time of preparing this article.

REFERENCES

[1] X. W. Huang, R. Sharma, S. Keshaw, "*The ENTRAPID Protocol Development Environment"*, Proc. IEEE INFOCOM '99, 1999.

[2] D. Ely, S. Savage, D. Wetherall: *"Alpine: A User-Level Infrastructure for Network Protocol Development"*, Proc. 3rd USENIX Symposium on Internet Technologies and Systems, 2001.

[3] S. Y. Wang, H. T. Kung: *"A Simple Methodology for Constructing Extensible and High-Fidelity TCP/IP Network Simulators"*, Proc. IEEE INFOCOM '99, 1999.

[4] M. K. McKusick et al.: "*The design and implementation of the 4.4BSD operating system"*, Addison-Wesley, 1996.

[5] G. R. Wright, W. R. Stevens: *"TCP/IP Illustrated, Volume 2: The Implementation"* Addison-Wesley, 1994.

[6] S. McCanne, S. Floyd: *"The Network Simulator NS-2"*, online document, http://www.isi.edu/nsnam/ns/

[7] The FreeBSD Documentation Project: *"FreeBSD Handbook"*, online document, http://www.freebsd.org/doc/en_US.ISO8859-1/books/handbook/index.html

[8] R. Scandariato, F. Risso: *"Advanced VPN support on FreeBSD systems"*, Proc. 2nd European BSD Conference, 2002.

[9] L. Rizzo: *"Dummynet: a simple approach to the evaluation of network protocols"*, ACM Computer Communication Review, 1997.

[10] J. Elischer, A. Cobbs: *"The Netgraph Networking System"*, online document, ftp://ftp.whistle.com/pub/archie/netgraph/index.html

[11] P. H. Kamp, R. Watson: *"Jails: Confining the Omnipotent root"*, Proc. 2nd International SANE Conference, 2000.

[12] netperf: http://www.netperf.org/

[13] *"ng_dummy: A simple netgraph traffic shaping node"*, online document, http://www.tel.fer.hr/zec/BSD/ng_dummy/

[14] F. Baker et al.: *"Requirements for IP Version 4 Routers",* Internet Standard RFC1812, 1995.

[15] M. Zec, M. Mikuc, M. Žagar: *"Estimating the Impact of Interrupt Coalescing Delays on Steady State TCP Throughput"*, Proc. SoftCOM Conference, 2002.

[16] G. Banga, P. Druschel, J. C. Mogul: *"Resource containers: A new facility for resource management in server systems"*, Proc. Symposium on Operating System Design and Implementation (OSDI), 1999.

[17] M. Zec: *"Implementing Clonable Network Stacks in the FreeBSD kernel"*, to appear in Proc. USENIX Annual Technical Conference, 2003.

[18] B. White et al.: *"An Integrated Experimental Environment for Distributed Systems and Networks"*, Proc. 5th Symposium on Operating Systems Design and Implementation (OSDI), 2002.