# Operating System Support for Integrated Network Emulation in IMUNES

Marko Zec
*ICSI Center for Internet Research and University of Zagreb*
Miljenko Mikuc
*University of Zagreb*

## Abstract

As invaluable tools in networked and distributed systems research, network emulators offer a viable alternative to live experimental / testbed networks. We present a methodology for emulating computer networks by using a general-purpose OS kernel partitioned into multiple lightweight virtual nodes, which can be interconnected via kernel-level links to form arbitrarily complex network topologies. The concept of using virtual nodes inside a kernel for fast network emulation is not new, yet previously published work generally advocated implementing kernel-level virtual nodes with capabilities limited to only certain simple functions, such as blind passing of network frames from one queue to another based on a static pre-computed path. We show that virtual nodes, offering the identical rich set of capabilities as the standard kernel does, can be implemented very efficiently by reusing the existing OS kernel code. Therefore our model not only provides each virtual node with an independent replica of the entire standard network stack, thus enabling highly realistic and detailed emulation of network routers; it also enables unmodified user-level applications to run within virtual nodes, thus becoming a powerful tool for development and testing of real-world networked applications ranging from routing protocol daemons to distributed file sharing systems.

## 1 INTRODUCTION

The need for efficient iterative testing or experimental prototype validation has been a part of the most engineering research and development efforts and processes ever since. In case of networked systems, three major classes of experimentation methodologies and tools have today become widely accepted: testbed networks, network simulation, and network emulation.

Testbed networks are experimental environments constructed of real hardware, such as routers or hosts, interconnected and configured to form a network of desired topology and other properties, such as bandwidth, delay, packet loss ratios, queuing disciplines etc. While typically offering a high level of realism in resembling the characteristics of target production networks, testbed environments tend to be difficult and time consuming to setup and maintain, but above all they are costly and therefore seldom affordable. In addition, large-scale experiments with network testbeds can present a logistic impossibility [11].

Network simulators provide for synthetic conceptual network environments. Simulators are typically highly configurable and extensible tools designed to test and evaluate network dynamics in a virtual timescale and in a controllable environment decoupled from any external traffic or systems. Unfortunately, simulations often make simplifying assumptions which can result in inaccurate representation of traffic dynamics as seen in real-world environments.

Network emulators can be considered a hybrid of testbeds and simulators – they subject real network traffic and systems to a synthetic network environment. This is however only a rough definition of an emulator, as different classes of such tools are available today. Single-hop WAN link emulators, such as Dummynet [2] or NIST [3], introduce artificial delays, bandwidth constraints, packet losses, and queuing policies in real time, however they do not provide for emulation of network topologies.

This article is focused on the problem of real-time emulation of entire computer network topologies. A fundamental requirement for a single physical machine to emulate a full network topology is the ability to support virtually simultaneous operation of multiple virtual nodes and interconnecting links, multiplexed on a single physical unit (a PC workstation). One typical approach in constructing such emulators is implementing a multi-node emulation engine as a userland process running on top of a general-purpose OS kernel. Each virtual node, running either as an isolated thread or a process, is then equipped with a private instance of virtual network interfaces and routing tables, which allow it to be interconnected either with other virtual nodes or to external networks. In effect such virtual nodes can functionally impersonate a router or a stripped-down end system. However, the overall throughput in such emulation systems can be severely limited by frequent

This work was supported in part by the Croatian Ministry of Science and Technology through research contract #IP-2003-143.

context switching and data copying from kernel to userland and vice versa, and is therefore seldom usable for real-time operation in high speed network environments.

We argue that the most pragmatic means for overcoming the performance bottlenecks inherently associated with userland emulation engines can be found in shifting of the entire traffic path in the operating system kernel. This approach not only eliminates the cost of context switching overhead, but also allows for packet handoff between virtual nodes to be accomplished by reference, thus eliminating any unnecessary data copying. Furthermore, we assess that the built-in support for maintaining multiple independent routing tables within a single kernel image, which today can be found as an optional feature in certain operating systems, does not per se provide sufficient foundation for constructing fully functional kernel-level emulator engines, despite some reports describing research efforts aiming in such a direction [9]. We show that it can be necessary to virtualize all the key components of the network stack in order to support complex emulation scenarios. Most importantly, during the implementation of our prototype we learned that the virtualization of the *complete* network stack functionality was a prerequisite for achieving a full application programming and binary interface compatibility with arbitrary user space programs relying on the network communication facilities provided by the OS kernel. An additional benefit from such an approach is a rich set of tuning options at each virtual node, such as independent controlling of queue depth limits or packet filters and classifiers, which together offer a powerful toolset in constructing emulated networked environments.

In support for our thesis, we propose and validate a framework for constructing high performance real-time IP network emulator, based on a modified 4.4BSD OS kernel. Our OS extensions allow for simultaneous setup and operation of multiple independent network stack instances within a single kernel. Each network stack instance acts as an independent virtual node (router or host), connected either to other virtual nodes via simulated links, or directly to the outside world via standard network interfaces. This allows for complex emulated IP network configurations to be constructed on a single machine. The virtualized network configuration can then be observed and analyzed at the level of each independent virtual node, link, or network interface. The emulator can easily interact with real networks through standard physical interfaces at up to gigabit speeds, depending on simulated network complexity and simulator hardware capabilities.

The rest of the article is organized as follows. Section 2 explains the basic implementation concepts behind the architecture of IMUNES, an Integrated Multiprotocol Network Emulator / Simulator. In section 3 we briefly describe the implementation details of our prototype kernel-level network emulator. Section 4 discusses the performance aspects of our real-time simulation environment. A few application examples are presented in section 5. Previous and related work is outlined in section 6, followed by a conclusion and directions for future research outlined in section 7.

# 2 THE ARCHITECTURE

We propose a transparent network emulation 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 paternity to such an approach, as it has been described earlier in different variations (see section 6). However, our main contribution lies in a highly efficient implementation of virtual network infrastructure, in form of a general-purpose OS kernel providing multiple independent network stack instances, with virtually no additional per packet processing overhead compared to the unmodified OS. Unlike some other architectures, such as Modelnet [12], which provide for only static network topologies and a source-routed emulation core without any IP layer processing capabilities in virtual nodes whatsoever, our model allows for emulation of fully functional IP routers in any emulated virtual node.

## 2.1 Virtual nodes

The key element of our emulator framework is a virtual node. A virtual node is a collection of a network stack instance and zero or more associated user space processes. Each network stack instance is functionally independent of all others, so that each instance maintains its own private set of state variables such as list of network interfaces, routing tables and routing cache, set of communication sockets and associated protocol control blocks, hash tables, traffic counters etc. Furthermore, each network stack instance includes a private set of optional networking facilities, such as packet filters, traffic shapers, as well as various tunable variables controlling different aspects of network stack behavior, such as inbound packet queue depths. The basic organization of an OS kernel partitioned into multiple virtual nodes is shown in Figure 1.

If desired, arbitrary existing UNIX applications such as routing daemons, traffic generators and analyzers can be run in each virtual node. However, a node can exist even without any associated processes, for example when it serves a transient node in a statically routed virtual network. The ability to execute standard user space applications without any need for modifications, recompiling or runtime library replacements in *all emulated nodes* presents the key advantage of our

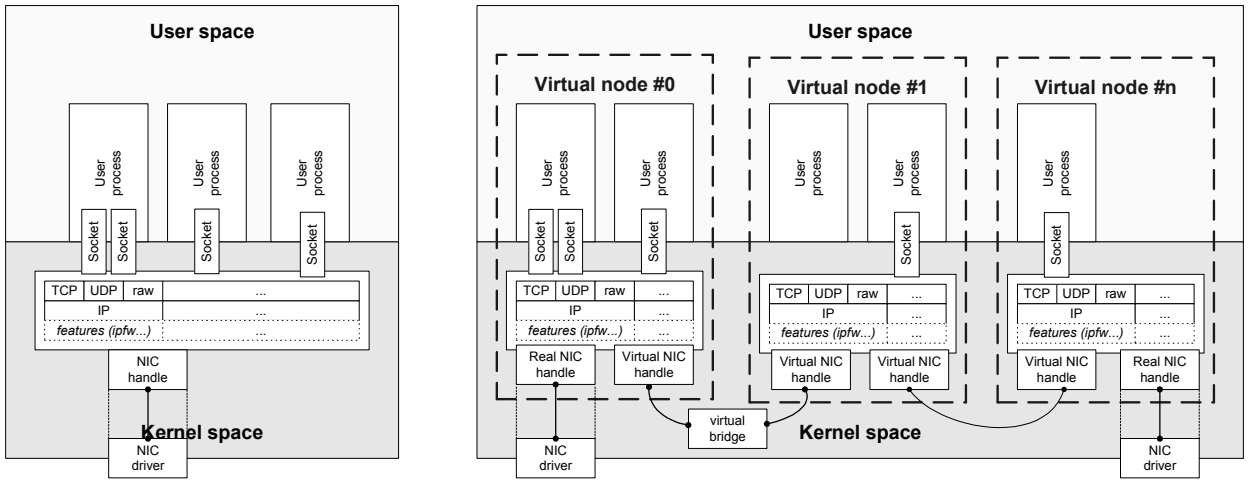


**Figure 1. Conventional organization of OS kernel and network stack (left); multiple virtual nodes / network stack instances within a single OS kernel image (right)**

framework which distinguishes it from previously published work.

Since in our model the OS kernel is modified so that the entire network stack is virtualized, each user process existing in a system must be associated with one (and only one) network stack instance at a time. During the boot up sequence the system creates a default network stack instance / virtual node, and all subsequent processes are automatically associated with the default network stack instance. Later, the administrator can instantiate additional independent network stacks, and spawn arbitrary processes within their context.

## 2.2 Virtual links

Since the basic property of each virtual node is that its network communication facilities are independent of other such nodes present in the system, it is necessary to provide explicit methods for establishing communication between isolated network stacks. Our framework relies on the standard netgraph [14] framework in the FreeBSD kernel, which provides methods and API for insertion of arbitrary traffic processing modules bellow the network interface layer. We have extended the netgraph framework to allow for interconnecting of network interfaces residing in separate virtual nodes.

Our model assumes that each network interface (either physical or virtual) can be associated with one and only one network stack instance at a time. During the OS booting and autoconfiguration, real network interfaces are automatically assigned to the default network stack instance. After additional virtual nodes have been instantiated, network interfaces can be reassigned from one virtual node to another, regardless whether they are "real" or virtual / pseudo interfaces by their nature.

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

The other alternative is using virtual Ethernet interfaces, which can be bridged to form independent broadcast domains spanning multiple virtual nodes. The bridging domains can be either entirely hidden inside the kernel, or transparently connected to the outside world via one or more physical Ethernet interfaces, as shown in Figure 1.

Probably the most important property of described use of the netgraph framework for interconnecting virtual nodes is that all packets are handed over between netgraph nodes only by reference, making any expensive copying of actual packet buffers completely redundant.

## 2.3 Queuing and traffic shaping

Different queuing disciplines and traffic shaping implementations generally account for the most basic functions of any network simulator. By default any virtual link in our emulator passes all frames through a separate instance of the ng_pipe traffic shaper, which is itself implemented as a netgraph node. This implementation features a simple bandwidth limiting and delay simulation, together with bit error-rate (BER) simulation. The tuning options also include a choice between queue head or tail dropping policies on queue overflows, and a choice between FIFO, simple fair queuing (FQ) and Deficit Round Robin (DRR) buffer management methods. More advanced network-layer traffic classification and queuing policies, such as

worstcase-fair weighted-fair queuing (WF2Q) or random early detection (RED) can be simulated using dummynet [2] facility, which is integrated in the base FreeBSD system. Both ng_pipe and dummynet shapers could be mixed in the same simulation configuration, if desired.

The time precision / granularity of packet dequeuing is dependent on the frequency by which the scheduler polls the packet queues. The queue scheduler is typically run on each system clock tick, so it is necessary to configure the system to run with a higher than standard system clock rate. Our test machines have been running cloclkrate set to 1000 Hz. As an enhancement to fixed-rate scheduling, we have implemented a method for constant polling of traffic shaper queues within the OS kernel idle loop. On systems with low user-level processing load, this allows for delay time resolutions in range of only a couple of microseconds, which is otherwise unachievable with traditional system-clock interrupt-based dequeuing methods. In preliminary tests the idle loop polling approach has shown very high accuracy, provided that the system is dedicated for network topology emulation, and is executing no other user space tasks.

Another important property inherent to the proposed simulation framework is isochronous timing across all virtual nodes. Since all virtual nodes share the same system clock, 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 exceeding 100 Mbit/s.

# 3 PROTOTYPE IMPLEMENTATION

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 experiments, we have modified the FreeBSD operating system kernel to allow multiple network stacks to be simultaneously active on a system. Our initial experiments with network stack virtualization in FreeBSD have been previously documented in [16].

## 3.1 Network stack virtualization

The fundamental approach taken in implementation of the described modifications to the BSD OS kernel was introduction of a new kernel structure named *vnet*, which serves as a variant of resource container [1] for all virtualized variables and symbols. Gradually, we replaced most of the global and static symbols used by the network stack code by their equivalent counterparts residing within the independent vnet structures. Network interfaces descriptors, which have traditionally been

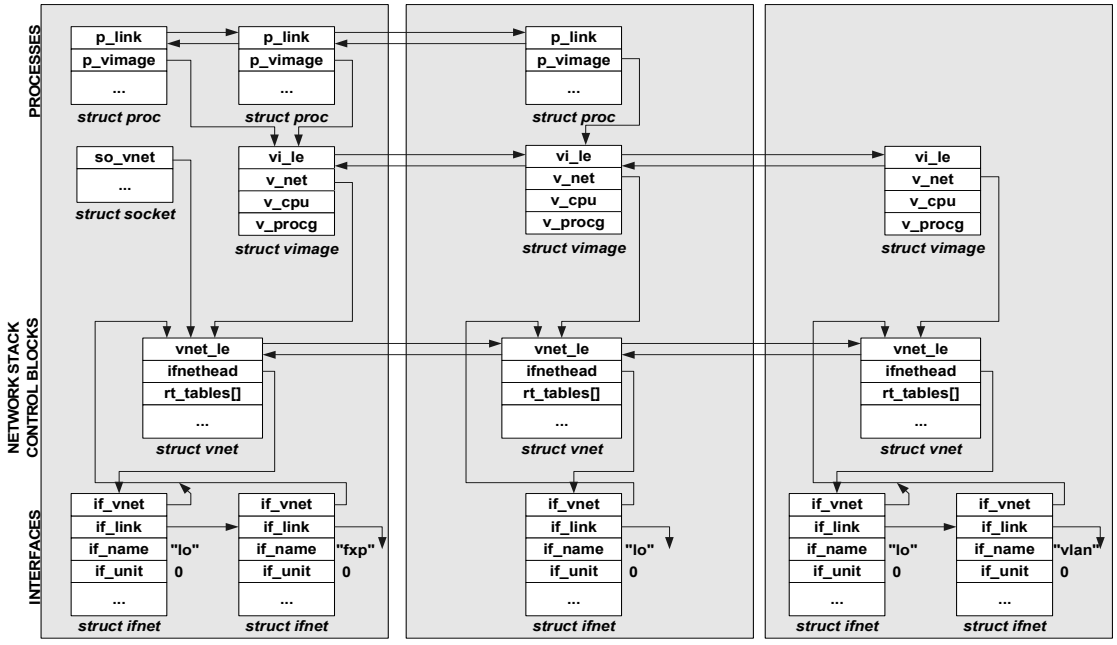


**Figure 2. Virtualized network stacks: major kernel structures**

maintained in a single linked list, are now associated with separate vnet structures, so that each network stack instance has command over its own list of network interfaces. Each network interface contains a pointer back to its vnet 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 the relations between the most important kernel data structures in a clonable network stack implementation is shown in Figure 2.

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. We decided to transparently extend each user process control block in the kernel 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.

We further combined the described tagging of user processes with already existing jail [4] 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. Latter, this concept was further extended by including feedback-driven proportional scheduling modifications to the CPU scheduler, so that the average CPU usage in each virtual node could be controlled and limited. This is an important mechanism for preventing runaway or maliciously constructed process or group of processes from monopolizing and starving all the real CPU resources. Furthermore, this also allows for system load monitoring to be performed on per virtual node basis, which provides more fine-grained control rather than accounting resource usage solely on physical machine level.

## 3.2 The management model

In early development phases we have introduced a basic command-line interface for managing virtual nodes. The command-line utility allows for creation of virtual nodes, assigning virtual or physical interfaces in appropriate nodes, and for starting of arbitrary UNIX commands or applications within a context of a target node. However, specification of complex network topologies requires more sophisticated management tools. Recently we have implemented a simple Tcl/Tk based graphical user interface (GUI) console, combined with a topology compiler for managing the kernel-level emulation engine. The management console automates the job of establishing, configuring and interconnecting the nodes in the target topology, and allows for quick access to a command line interface of each of virtual nodes during emulation execution. As GUI implementation falls outside the scope of this article, we are only presenting a snapshot of a sample network topology configured with

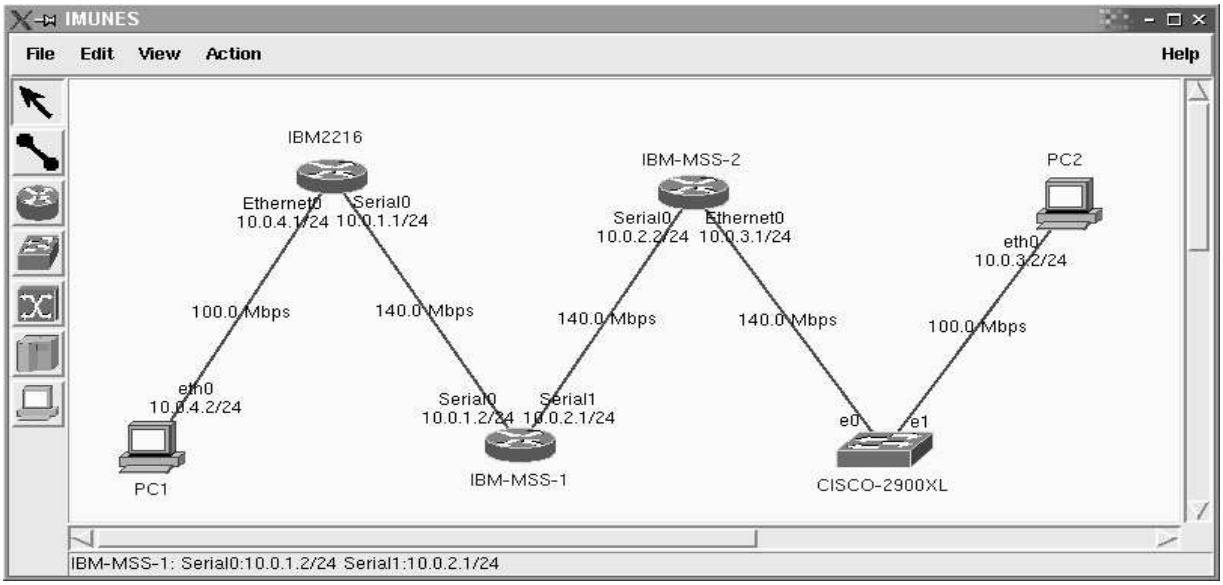


**Figure 3. An example of a simulated network topology**

```
PC2# traceroute 10.0.4.2
traceroute to 10.0.4.2 (10.0.4.2), 64 hops max, 1400 byte packets
 1  10.0.3.1 (10.0.3.1)  0.319 ms  0.259 ms  0.255 ms
 2  10.0.2.1 (10.0.2.1)  0.377 ms  0.354 ms  0.353 ms
 3  10.0.1.1 (10.0.1.1)  0.464 ms  0.451 ms  0.445 ms
 4  10.0.4.2 (10.0.4.2)  0.599 ms  0.584 ms  0.585 ms

PC2# pathchar -q 8 10.0.4.2
pathchar to 10.0.4.2 (10.0.4.2)
 mtu limited to 1500 bytes at local host
 doing 8 probes at each of 45 sizes (64 to 1500 by 32)
 0 10.0.3.2 (10.0.3.2)
 |    58 Mb/s,   29 us (265 us)
 1 10.0.3.1 (10.0.3.1)
 |   145 Mb/s,   9 us (367 us)
 2 10.0.2.1 (10.0.2.1)
 |   139 Mb/s,   7 us (468 us)
 3 10.0.1.1 (10.0.1.1)
 |    98 Mb/s,   9 us (609 us)
 4 10.0.4.2 (10.0.4.2)
4 hops, rtt 109 us (609 us), bottleneck  58 Mb/s, pipe 10820 bytes
```

```
tpx30# traceroute 10.0.4.2
traceroute to 10.0.4.2 (10.0.4.2), 64 hops max, 44 byte packets
 1  10.0.3.1 (10.0.3.1)  0.541 ms  0.368 ms  0.319 ms
 2  10.0.2.1 (10.0.2.1)  0.631 ms  0.508 ms  0.503 ms
 3  10.0.1.1 (10.0.1.1)  0.717 ms  0.667 ms  0.619 ms
 4  10.0.4.2 (10.0.4.2)  1.644 ms  1.744 ms  2.107 ms

tpx30# pathchar -q 256 10.0.4.2
pathchar to 10.0.4.2 (10.0.4.2)
 mtu limited to 255 bytes at ??
 doing 256 probes at each of 6 sizes (64 to 704 by 128)
 0 10.0.3.2 (10.0.3.2)
 |    39 Mb/s,   188 us (520 us)
 1 10.0.3.1 (10.0.3.1)
 |    72 Mb/s,   77 us (752 us)
 2 10.0.2.1 (10.0.2.1)
 |   105 Mb/s,   75 us (0.95 ms)
 3 10.0.1.1 (10.0.1.1)
 |    51 Mb/s,   65 us (1.20 ms),  +q 1.23 ms (7.87 KB)
 4 10.0.4.2 (10.0.4.2)
4 hops, rtt 809 us (1.20 ms), bottleneck  39 Mb/s, pipe 5830 bytes
```

**Figure 4. traceroute and pathchar traces over an emulated (left) and real testbed network (right)**

the GUI in Figure 3.

As an example, we have compared the fidelity of the sample emulated topology from Figure 3 with the appropriate real testbed network. Just like shown by icon labels, our physical network consisted of an IBM 2216 router, two IBM 8210 MSS LES/BUS servers operating as one-armed routers, and a Cisco 2900XL LAN switch, all connected to an IBM 8265 ATM router via OC-3c links. LAN emulated Ethernet was used for establishing connections over the ATM backbone. Both the testbed routers as well as the emulated ones were distributing routing information among themselves using the OSPF protocol. We performed a traceroute and pathchar traces through both networks and compared the results, as shown in Figure 4.

# 4 PERFORMANCE

Real-time network simulation was one of the key design goals behind the proposed concept of 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, with two major goals. One goal was to determine the difference in TCP transmission performance in the standard versus the virtualized kernel. The other goal was to determine the limitation of our framework in simulating network topologies with large hop counts.

## 4.1 Standard vs. virtualized kernel

The objective of our first test was performance comparison between the original and modified 4.4BSD network stack. The test involved measuring loopback TCP throughput using the common netperf [13] 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 also located in the same virtual image. The TCP throughput test was repeated for different maximum transmission unit (MTU) values set on the loopback interface. We used an AMD Athlon uniprocessor system with a CPU clock of 1200 MHz, a bus clock of 100 MHz and 256 Mbytes of SDRAM as a referent platform.

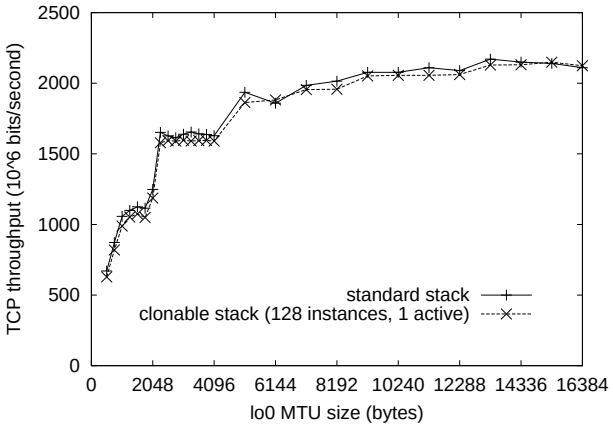


**Figure 5 – TCP loopback throughput comparison**

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 has to 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.

## 4.2 Hop count limitations

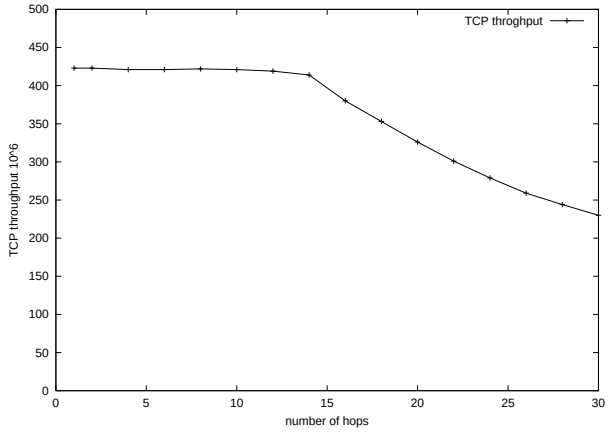


**Figure 6 – TCP throughput vs. emulated hops**

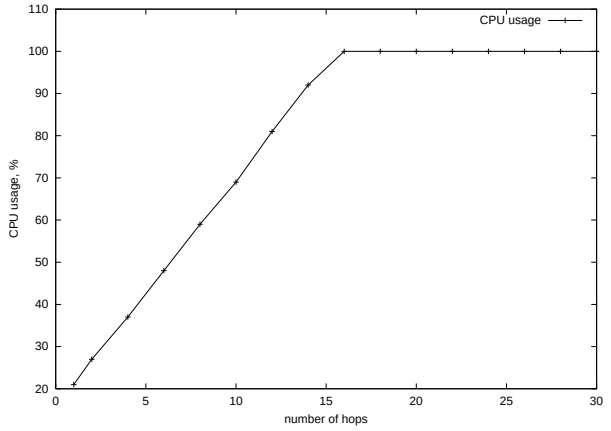


**Figure 7 – CPU load vs. emulated hops**

In the second test scenario we measured the maximum TCP throughput as a function of the number of virtual hops. We exposed a dedicated box emulating topologies of different end-to-end hop counts to a single external TCP stream. The observed TCP stream was limited in throughput to around 420 Mbytes/s due to the hardware limitations of the transmitting machine. In this test our emulator box was an IBM xSeries server with a 2.8 GHz Pentium-4 CPU and a dual gigabit Ethernet NIC. As shown in Figure 6, the obtainable throughput remained flat for emulated topologies of up to 15 hops, when it begun to decrease with each additional transient hop, dropping to around 220 Mbit/s for the longest observed topology of 30 subsequent hops. Figure 7 shows how the CPU load on the emulator box increased linearly with the number of virtual hops in the same test, leading to a partial system livelock [17] when the emulator capacity became too slow for the given TCP load and the emulated topology. This is a natural consequence of the fact that the simulator has to perform the complete set of IP switching tasks as defined in [15] for each single packet in each individual virtual node.

The obtained test results show that the IMUNES architecture can be capable of real-time operation in 100 Mbit/s to 1 Gbit/s range for emulated network topologies with limited number of end-to-end hops. This is far better of what can be achieved with traditional user-level emulators. As an example, the authors of ENTRAPID [5] measured TCP throughputs of up to 2 Mbit/s for processes communicating between separated virtual nodes. In IMUNES, the TCP throughput in similar tests easily reaches Gbit/s speeds. This is an improvement of almost three orders of magnitude, although it should be noted that the original ENTRAPID measurements were performed five years ago on a 300 MHz Pentium II PC, whereas our test machine was significantly more powerful, with a CPU running at 2.8 GHz.

A comparison with VMWare workstation [21] when used as a platform for constructing virtual network topologies could be even more interesting. We have generated a chain-style simulated network topology using both IMUNES and VMWare on the same physical machine, and have compared TCP throughput and average per-packet round trip time for traffic flows both originating and terminating in simulated virtual nodes. The results presented in Table 1 clearly outline the performance advantages of a lightweight or pseudo-VM over the traditional VM model in network emulation / simulation applications.

In cases where highly complex topological configurations need to be simulated, it is possible to combine multiple machines into a network emulation cluster. This could allow for obtaining high overall throughput and performance if the CPU power or memory bandwidth on a single box would become a bottleneck. When constructing such distributed or clustered simulators, multiple independent logical links could be multiplexed over a single physical link between different machines, since the support for IEEE 802.1Q virtual LAN (VLAN) tagging has been incorporated into the simulation framework. However, our framework currently does not yet implement any facilities for automating the process of distributing the virtual nodes in a cluster.

| | TCP troughputput (Mbit/s) MTU=1500 | | | TCP troughputput (Mbit/s) MTU=500 | | | ICMP ping Round Trip Time (miliseconds) | | |
|---|---|---|---|---|---|---|---|---|---|
| | IMUNES | VMWare | %faster | IMUNES | VMWare | %faster | IMUNES | VMWare | %faster |
| Local | 768 | 520 | 48% | 419 | 297 | 41% | 0,008 | 0,044 | 82% |
| Direct | 627 | 62 | 911% | 287 | 23 | 1148% | 0,013 | 0,272 | 95% |
| 1 hop | 452 | 9,8 | 4512% | 196 | 8,7 | 2153% | 0,022 | 0,613 | 96% |
| 2 hops | 383 | 6,3 | 5979% | 160 | 5,4 | 2863% | 0,030 | 1,001 | 97% |

**Table 1. Performance comparison between IMUNES and VMWare based simulated network environments.**

### 4.3 Scalability

It would be difficult to provide a universally significant scalability test for any network emulation framework. We have already seen that the achievable throughput in IMUNES depends on the number of emulated hops that packets have to traverse. Such a phenomenon is not unique to our platform, as it can be observed in all alternative emulation environments. The other scaling issue is the size of network topology that can be supported on a single physical node. The kernel memory footprint of a virtual node in IMUNES is only around 70 Kbytes large, which translates to a possibility to create thousands of virtual nodes on a single commodity workstation. Nevertheless, the actual limit on the total number of nodes heavily depends on both the size of routing tables and user space applications running in each node. For example, virtual routers with static routing do not require any user space processes to be active in their context. On the other hand, running complete sets of routing protocol daemons can present a significant burden to both the CPU and available memory; however the actual limits will also vary on case by case basis. As an example, a virtual node running Zebra / Quagga RIP routing daemon consumes only around 1 MByte of RAM, so even on our weakest machine with only 256 Mbytes of RAM we had no problems in emulating network topologies consisting of 200 virtual nodes. To illustrate that this can be considered quite an achievement, due to the RAM shortage we could not run more than four stripped-down FreeBSD OS instances under the VMWare workstation on the same hardware.

## 5 APPLICATION EXAMPLES

We consider the IMUNES framework to be a general-purpose network emulation tool, so pointing out to specific application scenarios can be somewhat misleading. Nevertheless, as an illustration, we provide two (rather similar) examples, for which we believe cannot be run as efficiently, or might not even be supported at all, on many of the alternative emulation platforms.

Verkaik et. al. have been developing extensions to the BGP routing protocol [18] to allow for reduction in the number of routed objects in the default-free zone of the Internet view. Their prototype implementation was based on the *zebra* open-source routing software suite, which they modified to include support for mentioned BGP extensions. For purposes of testing and prototype validation, they successfully created a virtual network topology consisting of 14 virtual nodes, using an earlier version of our tool.

The XORP project aims for developing an extensible and open routing platform for both research and production environments [19]. Using IMUNES, we have successfully generated virtual network topologies where a couple of nodes were running XORP, while other virtual routers ran zebra, while the whole emulated network received a routing view feed from an external environment. Furthermore, the flexibility of IMUNES allows each virtual node to run a different version of XORP, thus serving as a viable platform for regression testing.

## 6 RELATED WORK

As already mentioned, the idea of constructing network simulators by reusing and extending an existing OS network stack is not new, nor is the concept of implementing virtual nodes within the kernel for the purpose of constructing high-speed network emulators. However, our methodology significantly departs from such previous work. Further we outline the main concepts behind some of the previously published ideas and concepts in this field, and discuss the similarities and differences with our architecture.

The ENTRAPID [5] protocol development environment introduced a model of multiple virtualized networking kernels, which in effect present reentrant variants of the standard 4.4BSD network stack in multiple instances, running as threads in a specialized user process. Such an attempt of network stack virtualization in the user space successfully accomplished its primary goal of providing a flexible network simulation tool. However, this is achieved 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. Other

examples that followed this approach are the Alpine simulator project [6] and Virtual Routers [8].

The Harvard network simulator [7] 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 an 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. In this aspect our framework is conceptually similar to the Harvard simulator, because of the ability to transparently support unmodified userland applications on the modified kernel in multiple virtual node instances.

A distributed approach was taken in design and implementation of the Netbed [10] 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 a 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.

The EMPOWER [9] framework extends the concepts of distributed network emulation adopted by Netbed by allowing single physical nodes to host one or more virtual nodes in a single emulated network. Such functionality is achieved through utilizing multiple independent routing tables in the Linux kernel. However, the authors report that this model has many deficiencies and limitations; most notably it requires special hacks for establishing simple point-to-point links between two virtual nodes.

Modelnet [12] presents a recent framework for kernel-level network topology emulation by routing packets through a tree of bandwidth / delay emulating pipes. As they enter the emulator, inbound packets are prepended by a list of pipes they have to traverse through. The described model is efficient in its implementation and scales well with relatively large number of virtual nodes and links, however this is achieved at the cost of accuracy since all routing calculation / processing is performed only once as the packets enter the core, and never again when packets are traveling through virtual links. Modelnet is limited to emulating network topologies at the link layer only, since no IP layer processing is performed in any virtual node along the internal path. It cannot emulate any functionality of IP nodes in its core, nor can it execute userland applications such as routing daemons or traffic analyzers in its virtual nodes. Routes through the emulated topology are therefore predetermined and static, which makes Modelnet unsuitable for experiments which involve dynamic network routing or adaptation to topology changes. As an illustration, standard tools such as traceroute or pathchar cannot be used to explore the topology of a Modelnet core.

Traditional virtual machine frameworks can also be used for constructing emulated network environments, but the example of VMWare workstation have already shown that such platforms can suffer from severe performance and scalability issues when used as network emulators. A novel VM platform that could offer reasonable performance levels in network emulation applications is Xen [20]. Instead of providing emulation of real hardware (e.g. network interface cards) to the operating systems running above the VM supervisor, which can be quite inefficient due to frequent data copying operations, Xen offers a simplified virtual interface abstraction which basically consists of two buffer descriptors, one for transmit and one for receive, which are used for passing the network frames to/from the guest OS by reference. Although not reported in the original paper on Xen, we can imagine such an approach being used for efficient passing of network frames between guest OS instances.

Regardless on performance and scalability issues, it is important to note that both instantiating and managing guest OS instances requires a considerable amount of time and often manual efforts in all traditional VM systems. In contrast to that, virtual nodes in IMUNES can be created in almost negligible time: using a completely unoptimized interpreter-based tool IMUNES can instantiate a 200 node network topology in less than 30 seconds.

# 7 CONCLUSIONS

We have presented a flexible, versatile, and efficient network emulation framework that in our opinion supersets many of the key features of previously known real-time emulation systems. Most notably, our emulator allows each virtual node to operate as an independent yet fully functional IP network router, or as an UNIX end node, while making no sacrifices regarding the overall throughput and performance compared to the unmodified OS running on the same hardware. The full compatibility with the standard UNIX OS functionality and applications makes our emulator a viable tool for broad range of tests and emulation scenarios on all network layers above and including the IP layer. Currently the framework provides for simulation of Ethernet and point-to-point links. With reasonable efforts, it should be relatively simple to build a basic Frame Relay switch emulator using the netgraph framework.

It is worth mentioning that 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 supporting diverse virtual hosting scenarios, or in monitoring or even provisioning virtual private networks (VPN).

There are many possible directions for further improvements and research. As the simulator currently provides support only for IPv4 protocol suite, a 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. Implementing algorithms and support for distributing virtual nodes in a cluster could be another challenge.

The emulator code is freely redistributable under a liberal BSD/MIT-style license, and can be downloaded from http://tel.fer.hr/imunes/.

# 8 ACKNOWLEDGMENTS

We would like to thank the following people in providing valuable feedback or helping us in other ways: Joseph Touch, Venkata Pingali, Patrick Verkaik, Guido van Rooij, Julian Elischer, Orion Hodson, Atanu Ghosh, Pavlin Radoslavov, Sally Floyd, Nikola Musa, Luigi Rizzo, Ian Dowse, Daniel Gajski and Nikola Djurak. We also wish to thank the anonymous reviewers for their constructive suggestions.

# 9 REFERENCES

[1] Banga, G., Druschel, P. and Mogul, J. Resource containers: A new facility for resource management in server systems. *In Proc. of the Symposium on Operating System Design and Implementation*, 1999.

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

[3] NIST Network Emulation Tool, http//snad.ncsl.nist.gov/itg/nistnet/

[4] Kamp, P., Watson, R. Jails: Confining the Omnipotent root. *in Proc. of the 2nd International SANE Conference*, 2000.

[5] Huang, X., Sharma, R. and Keshaw, S. The ENTRAPID Protocol Development Environment. *In Proc. of the 1999. IEEE INFOCOM.*

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

[7] Wang, S. and Kung, H. A Simple Methodology for Constructing Extensible and High-Fidelity TCP/IP Network Simulators. *In Proc. of the 1999. IEEE INFOCOM.*

[8] Baumgertner, F., Braun, T. and Bhargava, B. Virtual Routers: A Tool for Emulating IP Routers. *In Proc. of the 27th Annual IEEE Conference on Local Computer Networks (LCN 2002)*, Tampa, USA, 2002.

[9] Zheng, P. and Ni, L. EMPOWER: A Network Emulator for Wireline and Wireless Networks, *In Proc of the IEEE INFOCOM*, 2003.

[10] White, B., Lepreau, J., Stoller, L., Ricci, R., Guruprasad, S., Newbold, M., Hibler, M., Barb, C., Joglekar, A. An Integrated Experimental Environment for Distributed Systems and Networks. *in Proc. of the OSDI*, 2002.

[11] Fall, K. Network emulation in the Vint/NS Simulator. *Proc. of the fourth IEEE ISCC*, 1999.

[12] Vahdat, A., Yocum, K., Walsh, K. et. al. Scalability and Accuracy in a Large-Scale Network Emulator. *in Proc. of the OSDI*, 2002.

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

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

[15] Baker, F. et al. Requirements for IP Version 4 Routers, Internet Standard RFC1812, 1995.

[16] Zec, M. Implementing a Clonable Network Stack in the FreeBSD Kernel, *Proc. of the 2003. USENIX Annual Technical Conference, FreeNIX track*, June 2003.

[17] Mogul, J. C., Ramakrishnan K. K. Eliminating receive livelock in an interrupt-driven kernel, *ACM Transactions on Computer Systems*, 1997.

[18] Verkaik, P. et. al., Beyond CIDR Aggregation, CAIDA technical report 2004-1, Feb. 2004.

[19] Handley, M., Hodson, O., and Kohler, E., XORP: An open platform for network research, *in 1st Workshop on Hot Topics in Networks (HotNets-I)*, October 2002.

[20] Bahram, P. et. al., Xen and the Art of Virtualization, *in Proc. of SOSP'03,* October 2003.

[21] VMWare Inc., http://www.vmware.com/