V-eM: A Cluster of Virtual Machines for Robust, Detailed, and High-Performance Network Emulation

George Apostolopoulos, Constantinos Chasapis
ICS-FORTH, Greece
georgeap@ics.forth.gr, chasap@ics.forth.gr

***Abstract*— In this paper we present the design and performance evaluation of a network emulation cluster built with commodity PCs and network switches. Physical nodes host multiple emulated nodes each. Each emulated node runs inside its own virtual machine and is complete with a kernel and device drivers. A para-virtualized virtual machine monitor ensures isolation between the emulated nodes, fair access to the resources of the underlying physical node, and high performance. The load of traffic conditioning (emulating packet delays, losses and other characteristics of wide-area network links) is shared between all physical nodes by conditioning only the traffic originated by the emulated nodes they host. The above organization results in an emulation testbed that is low cost and scalable while providing strict resource isolation between emulated nodes and high emulation fidelity by allowing the emulation of kernels and other system level software. In this paper we present the main considerations behind the design of our testbed and through detailed performance evaluation we demonstrate that our approach can result in a scalable emulation system with high performance.**

I. Introduction

Network simulation and emulation have been indispensable tools for understanding the performance of network systems. Many times simulating a target network is simpler and cost effective: only a software model of the target system has to be created and the simulation needs only very modest hardware to run on. On the other hand, it can be very hard or even impossible to create an exact model of the target system and its software so usually simulations evaluate an abstraction of the target system. Over the last few years many emulation testbeds have been proposed [1], [2], [8], [3] for evaluating various types of networking environments. Most are built using commodity PCs connected through a local area network. The resources of the target network are abstracted and mapped to the resources of the physical nodes. Any networking testbed thus needs to resolve two fundamental problems: how to emulate networks with large number of links and nodes and how to emulate the traffic conditions of wide area network links in the local area network used in the testbed.

In order to achieve scaling most testbeds multiplex multiple emulated nodes on each physical node. A variety of methods has been proposed for achieving this multiplexing [4], [5], [6], [9], [10], [11], [12], [13], [14]. Each achieves a different trade-off between isolation of the emulated nodes and efficient resource usage. In many cases emulated nodes are mapped to processes on a physical node. While this approach has low overhead it results in emulated systems that are very abstract and can run only user level code. In particular, they lack system level software like kernels, device drivers and networking stacks. Furthermore, resource isolation and scheduling between the emulated nodes depends on the facilities of the underlying system. Commodity operating systems such as Linux need significant extensions and re-engineering in order to be able to provide this increased level of isolation and resource accounting [15]. On the other end of the spectrum solutions like virtualization that can provide complete isolation between emulated nodes can have a prohibitively high performance overhead.

While some emulation testbeds are by design wide-area like PlanetLab [3] most consist of nodes connected over a local area network. In order to ensure that emulated traffic encounters conditions representative of a wide area network a *network condition emulation* system is needed. To this end a variety of software and sometimes hardware systems have been proposed [16], [17], [18]. Typically such systems allow controlled introduction of traffic effects such as packet loss, delays, reordering, rate limiting, and jitter. These network condition emulators are usually resource intensive and in many cases are implemented on dedicated physical nodes as in [1] or in a cluster of physical nodes as in [2]. The number and placement of these special nodes has to be planned carefully in order to ensure there is enough processing capacity to handle the offered traffic. This usually complicates the algorithms that determine the mapping of the emulated resources to the physical ones [1]. In extreme cases, the layout of the testbed may need rewiring in order to ensure there are enough resources for a particular experiment.

In this work we propose and evaluate a different approach for building network emulators as *emulation clusters*. The emulation cluster is built with multiple identical physical nodes. Each physical nodes hosts a number of emulated nodes. We use *para-virtualization* to ensure resource isolation between the emulated nodes and keep the overheads low. Each emulated node runs inside its own virtual machine and thus can be a complete system with its kernel, networking stack and device drivers. Each cluster node provides network condition emulation only for the emulated nodes it is hosting. Point-to-point and broadcast links are emulated through manipulation of MAC addresses and using the Address Resolution Protocol (ARP) for resource discovery allowing the use low-end inexpensive Ethernet switches for interconnecting the cluster nodes. The above help us build emulation clusters that (a)

are scalable since the communication and network condition emulation load is spread evenly among all the participating nodes, (b) have low cost since we use only low-end commodity components, (c) ensure robust isolation between multiplexed emulation nodes due to their implementation as virtual machines, (d) allow experimentation with system software, kernels and device drivers, and (e) can achieve high performance due to the reduced cost of para-virtualization. We believe that our work is the first to achieve all the above goals. Most of the existing emulation testbeds satisfy some of the above goals but not all. Empower [8] and Modelnet [2] can not really achieve isolation and fairness between their emulated nodes, while in PlanetLab [3] and Emulab [1] emulated nodes can execute only user level processes making it impossible to experiment with kernels and other system level software.

In this work we will describe the architecture and the mechanisms we use to implement an emulation cluster and the design choices involved. We use Xen [9], a very popular open source para-virtualization system to implement the virtual machines for the emulated nodes and we borrow network condition emulation functionality from NISTnet [16]. Our first goal is to implement a cluster prototype, and evaluate its functionality and performance. We show that using new generation high-end commodity PC technology (dual CPU machines with multiple Gigabit Ethernet interfaces that cost under 1,000 Euros) we can implement an emulation cluster that can support a multiplexing factor of at least 10 while achieving high performance. Building a cluster that can emulate with high fidelity 500 nodes requires putting together 50 physical nodes connected with inexpensive Ethernet switches for a total cost of 100 Euros/emulated node. Such a system can be a very powerful testing platform. For example, a 500 node network is large enough to emulate the networks of large ISP providers. Various type of networking software, router operating systems and routing protocols can be tested at a real life scale while still under complete control. Performance problems that appear only in large deployments can be readily explored and resolved early in the development cycle of products. We demonstrate the capabilities of our cluster by emulating and evaluating a 81 node router network running the OSPF routing protocol.

Another goal of this work is to present a detailed performance analysis of using Xen virtual machines as virtual routers. While Xen's performance advantages over other virtualization technologies have been demonstrated for server applications [9] little is known about the performance of Xen when used as a network traffic forwarder. We believe that such virtual routers can have far reaching applications in networking that go beyond network emulation: routing stacks could be implemented as a collection of virtual routers ensuring isolation and reliability; new network functionality can be added in a router by simply adding a specialized virtual router that interfaces with the rest of the system at the IP level. Our study is a first step in understanding the performance characteristics of such virtual routers.

This paper is organized as follows: In Section II we present the design decisions for our emulation testbed. In Section III we discuss in more detail the implementation of the layer that ties the emulation cluster together. In Section IV we show the results of our performance evaluation and in Section V we show examples of how we can use such a testbed for testing and evaluating complex network applications, in this case intra-domain routing. In Section VI we present some additional considerations, in Section VII we discuss related work both in network emulation and virtualization and finally in Section VIII we summarize our main findings and present topics we will address in future work.

II. Testbed design

A. Design Goals

We want to design a distributed emulation infrastructure that fulfills the following design goals: (1) low cost through the use of commodity components for the emulation nodes and the interconnection network, (2) adequate performance, (3) the ability to scale easily and cheaply to large configurations that can emulate 100s of nodes, (4) complete isolation of the emulated nodes, (5) fairness in the emulated node's usage of the underlying physical machines resources, (6) be able to emulate complete systems that include kernel and device drivers, (7) trade-off multiplexing scale for emulation fidelity. On of the main differences from other emulation testbeds is the decision to use para-virtualization and model each emulated node as a virtual machine. While this ensures resource isolation between emulated nodes the downside is the reduced multiplexing factor. While other testbeds can host tens of emulated nodes on a single physical node we currently support up to 10 nodes. While reduced multiplexing factors may be a reasonable trade-off for achieving high fidelity emulation we plan to investigate more the scaling limits of the virtual machine approach in future work.

B. Multiplexing of physical resources

A major design decision is the mechanism that will be used for multiplexing the emulated nodes on the physical nodes. We chose to use Xen [9], an open source virtual machine monitor. Xen provides a software layer that runs on the bare hardware (the hypervisor) that is responsible for controlling access to the physical system resources. The hypervisor exports an abstraction of the real hardware to various clients. These clients are complete operating systems that implement their own scheduling memory and I/O management. The exported hardware model is not identical to the underlying hardware, thus Xen is classified as a *para-virtualization* approach and the client operating systems need to be modified in order to support the virtualized hardware model. The required modifications are rather simple [9] and many popular operating systems have already been modified, including Linux, NetBSD, and Windows XP.

Xen is hardly the only option available for resource multiplexing. Intense research and commercial interest has created many related products and research prototypes. Xen is following the virtualization approach, where resources of the physical node are split among multiple independent virtual machines

that are isolated from each other. Virtualization has been used for years on high-end commercial servers like the IBM xSeries and pSeries. Recently, virtualization products such as VMware [10], [11] and VirtualPC [13] have appeared for desktop systems. In addition to commercial products some other research virtual machine monitors are available [19], [20] but these are not as widely available as Xen and their level of maturity is unknown while Xen has multiple real life deployments and it has strong support from multiple high-end server vendors. In the same class of light weight para-virtualized virtual machine monitors we should also include Denali [21] that can achieve great scale by supporting non-standard machines with limited capabilities. This is not compatible with our goals of emulating complete mainstream systems and kernels.

A different approach for emulating multiple virtual systems on a single real system is User Mode Linux [6] or the closely related FAUmachine (known before as UMLinux) [22] where complete Linux systems run as a single or multiple processes on top of a modified Linux kernel that manages the physical hardware. The underlying operating system scheduler is responsible for managing the allocation of physical resources to the emulated systems. While these systems are popular and widely studied and deployed it has been shown that their performance is significantly lower than Xen's [9]. More relevant, when used for network emulation [23] the forwarding performance of the emulated network components is considerably less than what we are able to achieve with Xen.

Yet another approach is to make the physical node's operating system appear as dedicated to each of the emulated systems. Each emulated system can run its own set of user space processes and use communication endpoints under the illusion that it is alone on the physical hardware. FreeBSD jails [12] and Vservers [4] are widely available such mechanism that have been used in a large number of popular testbeds [1], [3], [2]. The scheduler of the underlying operating system is responsible for controlling the access of the emulated systems to physical resources and typically it can not achieve guaranteed resource isolation without extensive modifications. There also exist commercial products that follow the same approach [14] and place more emphasis on resource isolation. A main disadvantage of this approach is that it allows emulation only of user level code.

CPU emulators like Qemu [24], Bochs [5] and Plex86 [7] occupy the other end of the emulation precision spectrum. They are CPU emulators that allow emulation of a complete architecture in the form of a process that runs on the host operating system. This virtual machine can be booted with any operating system code that does not need to be modified whatsoever. The main drawback of this approach is its high overhead. The emulation fidelity provided is useful only in specialized applications.

In addition to completely virtualizing the emulated node, there are approaches that virtualize other components of the system, most often the network stack since they are geared towards networking applications and emulation. Entrapid [25] and Apline [26] create virtual networking stacks in user space while Vimage [27] virtualizes the network stack inside the kernel. Simply virtualizing the network stack is not sufficient for assuring resource isolation between the emulated nodes since it does not address resource contention for other physical resources such as memory and CPU.

Of course Xen has some limitations. The most important is that client operating systems must be modified to work under Xen. Although most of the operating systems commonly used for networking experimentation have already been ported, a network equipment vendor would need to port his operating system to work on top of Xen. According to the information in [9] the porting effort is relatively modest. Another limitation is Xen's ability to support very large multiplexing factors. In [9] the authors have successfully experimented with up to 100 virtual nodes on a server. Although we do not anticipate to be able to achieve such high multiplexing factors when virtual nodes emulate networking systems, this shows that we should be able to move beyond the rather modest multiplexing factor of 10 that we demonstrate in this work.

C. Network emulation

Wide area network links delay, misorder and loose packets but these effects are difficult to reproduce in the local area links that connect the physical nodes. Significant work has been done on systems that can reproduce wide area network traffic conditions [17], [16], [18], [28], [29]. Typically, these systems can (a) introduce delay, (b) introduce loss, (c) introduce miss-ordering, and (d) rate limit traffic. The implementation of these tools is rather complex. One reason is that they have to be realistic: the traffic artifacts they introduce should reflect what is observed in real network connections. In order to achieve this the real traffic artifacts need to be modeled so that the emulation tool can reproduce them. Another source of complexity is the need to access an accurate time source in order to introduce latencies with precision. If the timer interrupts are not frequent enough delayed packets will be grouped together and be released in large batches that may cause temporary congestion in the emulated links and they are not an accurate representation of real network delays. From previous works in the literature [8] it appears that for effective emulation the timer resolution should be below 1 msec. Current Linux systems have a timer resolution of 10 msec. Furthermore, network emulation is resource intensive. Adding a 100 millisecond delay to a packet stream of 100 Kpackets/second will require buffering 10,000 packets.

Due to the complexity of network emulation some emulators such as [1], [2] perform this emulation in dedicated nodes that are not used for hosting virtual nodes. This clearly increases the cost of the emulator but also the complexity of mapping real networks topologies to it. In our approach we integrate the network emulation on each physical node. Thus the question that emerges is whether the amount of resources that the link emulator consumes will have a detrimental effect on the performance of the virtual routers. In order to investigate and keeping up with our decision to use only publicly available

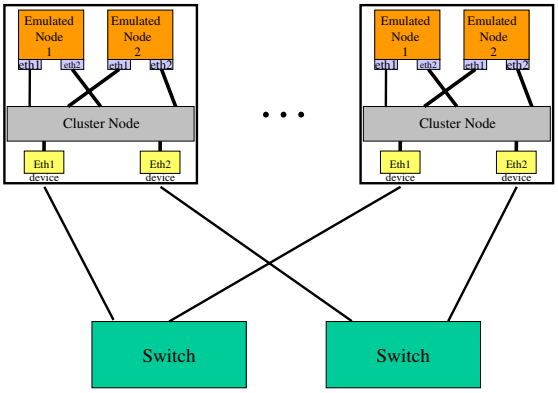


Fig. 1. High level view of the emulation cluster

software we decided to use parts of the NISTnet emulator [16] that has been quite popular and has extensive models for network delays and losses. Dummynet [17] is another popular solution but it exists only for FreeBSD. Another alternative is the queuing mechanisms available in the Linux kernel. These queuing disciplines can perform quite sophisticated link scheduling and rate limiting. A relatively new queue discipline has been implemented specifically for addressing network emulation. The *netem* queue discipline can implement delaying of packets and can introduce losses. Its accuracy though is limited by the timer accuracy of Linux and when we experimented with it we found its performance overhead to be rather significant. Other proposals for network emulation include ENDE [28] that can emulate only delays and Delayline [29] that intercepts system calls to interpose its emulation routines. The Ohio Network Emulator (ONE) [18] is one of the first one node emulators.

D. System Overview

We show a high level overview of our emulation cluster in Figure 1. The emulator is built with a number of identical physical systems that are connected through Gigabit Ethernet switches. Each physical node's hosts a number of emulated nodes and a cluster node. All are Xen guests, i.e. they are complete systems running their own kernel. These virtual systems communicate with each other through Xen's inter-guest communication mechanism. Each emulated node has an arbitrary number of network interfaces. With the current Xen code these are implemented as generic Ethernet interfaces on top of Xen's inter-guest communication mechanism using a custom device driver. This limits the network interfaces of an emulated node to only Ethernet. Still, it should not be hard to create different types of interfaces (for example ATM, Packet over Sonet, or various types of IEEE 802.11) but we do not address this further in this work. Xen's monitoring of resource usage and communication ensures that an emulated node can only talk to the cluster node and it is not aware of neither can communicate directly with the other emulated nodes. Each emulated node is assigned a part of the psychical node's memory and disk, a slice of its CPU and it booted and shutdown under the control of the cluster node. To make the handling of emulated links uniform we always ensure that two emulated nodes do not communicate directly, even if they are hosted on the same physical node. We require that traffic between them will exit the physical node on one physical interface and return through a different one. The cluster node is responsible for creating and destroying the emulated nodes and implements the functions that are necessary for the operation of the cluster. In particular it: (1) handles the interfaces of the physical node, (2) handles timer synchronization so that all emulated nodes have access to (approximately) synchronized clocks, (3) performs network condition emulation for each individual emulated interface, (4) implements the mechanisms needed to emulate point to point and broadcast links over the broadcast Ethernet interconnect.

Note that this approach is quite different than that of other emulation testbeds where the network condition emulation is centralized in one or more specialized nodes. This creates a lot of problems when assigning emulated resources to physical resources and also poses scalability challenges. With our approach, each physical node is responsible for handling the network conditioning of the traffic it originates itself allowing for a natural scaling as the cluster is extended with more physical nodes. A similar host based or distributed approach had been followed in [30]. One of the arguments against performing network conditioning at the same node that hosts the emulated nodes is that the two functions will compete for the same resources. As we will show here, thanks to the solid resource isolation provided by Xen, this is not an issue.

III. Emulation Cluster Services

Here we describe in more detail the implementation of the services provided by the cluster node. Creation and destruction of the emulated nodes is performed through the command line API provided by Xen. The configuration of the whole cluster is under the control of appropriate scripts that are created automatically based on the network topology we need to emulate. Also, since nodes exist on the same local area network we use NTP to synchronize the clocks of the physical nodes. Below we discuss in more detail the more complex cluster services, i.e. emulation of different types of links and implementation of network condition emulation.

A. Networking on the emulation nodes

In a Xen system a number of guest operating systems can run on top of the Xen hypervisor that controls their access to the physical system resources. One of the guest systems, called *xen0* or *domain0*, has increased functionality and is used for starting and stopping the other guests. It is also responsible for handling the real Ethernet interfaces. We show an example of the networking architecture of one of the physical emulation nodes in Figure 2. Virtual nodes can have multiple virtual interfaces. Each virtual interface is implemented as an Ethernet interface with its own unique

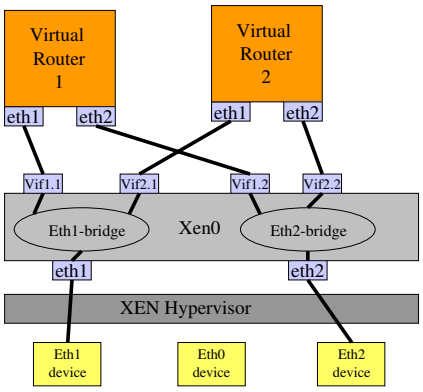


Fig. 2. Organization of the network connections on the emulation node

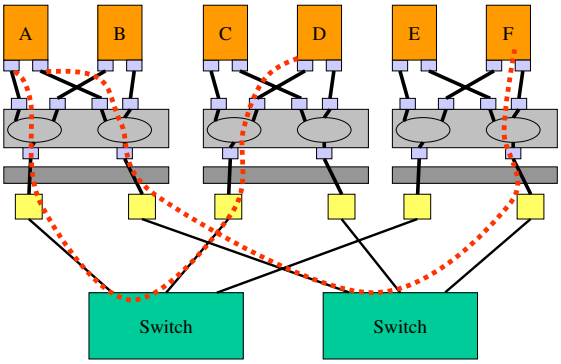


Fig. 3. Example of connections between virtual routers

Ethernet MAC address. Every virtual interface has a corresponding interface in xen0 called the virtual interface (vif). Vifs are connected to their corresponding virtual interfaces in a point to point fashion. xen0 sends traffic to a virtual router by sending it to the corresponding vif and outbound traffic from the virtual router arrives into xen0 through another vif. xen0 connects real Ethernet interfaces and vifs through one or multiple bridges. These bridges are implemented using the Linux kernel's bridge module that provides forwarding based on MAC address and MAC address learning. These bridges are responsible for forwarding traffic between the virtual and physical interfaces. The IP stack of the xen0 system never sees traffic from/to the virtual nodes.

A testbed is built by interconnecting multiple physical nodes through a set of Gigabit Ethernet switches. An example of such an interconnection is shown in Figure 3. In the simplest configuration all physical interfaces are connected to the same broadcast domain. Since the xen0 bridges extend the broadcast domain to the virtual interfaces, effectively all virtual interfaces appear to be connected to the same Ethernet switch and any two virtual interfaces can directly exchange IP traffic. When a virtual interface attempts to send IP traffic to the IP address of another virtual interface, ARP will be able to reach the destination interface and retrieve its MAC address. The packet will be sent to this destination MAC address and will arrive to the destination virtual interface since the Ethernet switches and bridges in between have discovered the location of all MAC addresses of virtual interfaces (Xen virtual interfaces explicitly advertise their MAC addresses when they are created so that switches and bridges can learn it). Links between virtual nodes can be created by simply configuring the appropriate addresses to the virtual interfaces at the two endpoints of the link.

This approach has two potential downsides. First, since the destination MAC address on each packet is that of the virtual interface, the physical interfaces need to operate in promiscuous mode. This should not cause problems in our design. Since switches and bridges have performed address learning they can direct traffic to the correct destination physical interface without interfering with other physical interfaces. The other problem is that requiring all the physical interfaces to belong to the same broadcast domain may cause scalability issues since Ethernet switches can learn a limited amount of MAC addresses. Still, for network emulation there is no need for full connectivity between physical ports. Only physical ports that host the two ends of the same emulated links have to be able to communicate. With the proper assignment of emulated links to physical interfaces it is possible to group the physical interfaces into multiple disjoint broadcast domains reducing dramatically the cost of the interconnection network.

1) Emulating point to point links: In our testbed, unicast traffic will be received only by the destination virtual interface since the unicast destination address of the packet will be resolved through ARP. But multicast traffic will be sent to a multicast MAC address and as a result will be delivered to multiple physical and virtual interfaces. Multicast MAC addresses are derived from the (multicast) destination IP address of the packet by assigning the low order 24 bits of the IP address to a MAC address that starts with the prefix 01:00:5E. In some applications such as routing where all virtual interfaces will have joined the same multicast group (for example the ALL-OSPF-ROUTERS group in OSPF routing) the packet will be actually delivered to the routing application the executes on each virtual node. This is clearly unacceptable. In order to emulate a point to point link we need to ensure that packets are delivered to the virtual interface at the remote end of the link, independent of their destination MAC address. We can achieve this if we rewrite the destination MAC address before the packet is sent over the switched network. Since the destination MAC address is known at configuration time this is simple to do. Outgoing packets have their MAC address rewritten and then sent to the outgoing Xen bridge. Of course we can rewrite the destination MAC address of only the multicast IPv4 packets. Other packets (such as ARP for example) should not be prevented from reaching the correct destination nodes. We would like to perform this destination MAC rewriting with as few changes as possible to the xen0 system. The Linux kernel already supports the ebtables [31] mechanism that allow us to do destination MAC address

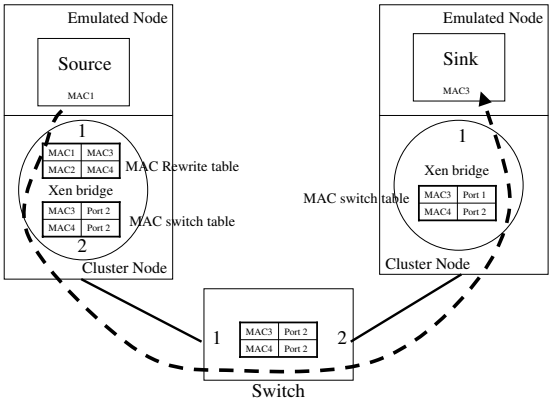


Fig. 4. Example of destination MAC address manipulation

rewriting by specifying appropriate rules. Unfortunately we found this tool to have a very high performance penalty. Even with very few rules, there was a significant (on the order of 15%) degradation of the packet forwarding performance of our system when ebtables where enabled.

This led us to implement destination MAC address rewriting inside the kernel's bridge code. By adding to the bridge code a MAC rewriting hash table which is keyed by the source MAC address it is possible to quickly find if the destination MAC address should be rewritten. We show an example of the path of a packet between two emulated nodes in Figure 4. A packet that is sent with source address MAC1 and destination MAC address MAC8 will have its destination address rewritten by the bridge module of the emulation node to MAC4 and then it will be forwarded based on the new destination MAC address. This new destination is the MAC address of the destination network interface, which is known to the network switches and the bridges of the other emulation nodes and the packet is delivered to the correct destination interface even if its original destination MAC address was a multicast one.

2) Emulating broadcast links: In some network applications we need to emulate multiple virtual interfaces that connect to the same broadcast link (for example multiple routers connected to the same Ethernet segment). In this case, multicast packets must be delivered to all the virtual interfaces that are "attached" to the emulated broadcast link. We want to be able to emulate this functionality without delivering traffic to any interfaces that do not belong to the emulated broadcast link. Even low-end Ethernet switches can perform switching for multicast MAC addresses. The switches keep track of the location of the multicast MAC addresses and when forwarding a packet destined to this MAC they replicate it to their appropriate ports. Usually switches snoop on the IGMP messages emitted by the interfaces that participate in a multicast group in order to discover the location of multicast MAC addresses. Thus we can effectively emulate a broadcast link if we map it to an IP group. When computing the mapping of resources on the emulation testbed we need to assign multicast IP addresses to all the broadcast links that are to be emulated. These IP addresses have to be selected in such a way so that their corresponding multicast MAC addresses are distinct. Then we need to ensure that each emulated interface joins the appropriate group. This will trigger IGMP messages that will allow the switches to learn the location of the emulated interface. Finally, when sending traffic to this emulated link, the source will need to rewrite the destination MAC of the packets to match the multicast MAC address associated with the broadcast link. We already have support for this in the modified bridge code in the emulator node's kernel. This mechanism can ensure that traffic sent over an emulated broadcast link will reach only the interfaces that belong to this emulated link and no other avoiding unnecessary packet replications and transmission. The downside of this mechanism is that it is not completely transparent to the emulated node since it will need to join an appropriate IP multicast group on each of its interfaces. While we currently take care of this through configuration commands, it would not be hard to modify the driver that emulates an Ethernet interface on the emulated node so that it automatically joins the right IP groups at startup. Xen already does this by forcing emulated interfaces to send ARP requests when they start so that switches can learn their location.

Using the partitioning offered by the multicast MAC addresses we can avoid more complex partitioning solutions. Technologies such as VLANs, MPLS, and tunneling over a routed network could also be used by with a higher cost. Indeed most low-end Ethernet switches do not support VLANs, while MPLS capable switches or router are considerably more expensive than low-end Ethernet switches. The main assumption is that network condition emulation is applied only at the source of the packets. This works well in most cases but not when we emulate wireless links. A wireless link is by its nature broadcast but network traffic may be different on each emulated interface attached to the link. Loss rate for example can be different for each emulated interface since packet delivery can be affected by conditions that are local to each node. Emulating such a link will require network condition emulation to be also performed at the receiver node. Implementing this should not be difficult. NISTnet can intercept packets arriving at the physical interfaces but now the destination MAC address must be used for selecting the appropriate NISTnet rule for each incoming packet.

B. Network Condition Emulation and resource isolation

A fundamental design requirement for our emulation testbed is that of resource isolation between the emulated nodes. Emulated nodes have to share the CPU, memory and networking capacity of the physical node so that they do not interfere with each other or the cluster node even when the system operates close to its capacity. Xen's hypervisor handles effectively the sharing of memory and CPU. A part of the physical memory is allocated to each emulated node when it is created. Xen also undertakes the scheduling of emulated nodes. Xen 2.7 can use multiple types of CPU scheduling. The default configuration we used in our experiments uses a Borrowed Virtual Time

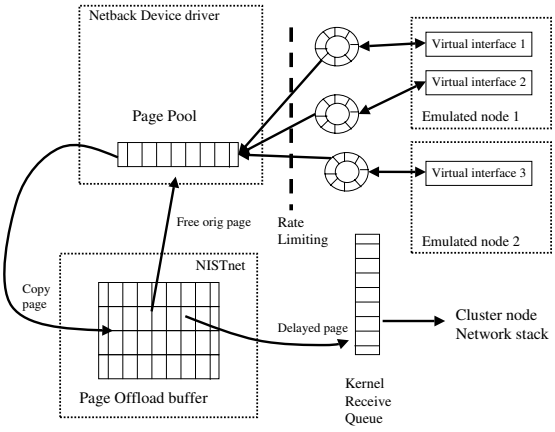


Fig. 5. Receive path on the emulation node

(BVT) scheduler that attempts to ensure that each emulated node will get its fair share of the physical CPU. The allocation of CPU is controlled with configurable CPUs weights for each emulated node. Furthermore, Xen can assign emulated nodes to different CPUs of a multi-CPU system. In our experiments we set the same weights for each of the emulated nodes. Since our systems are dual CPU, we run the cluster node in one CPU and all the emulated nodes in the other. In this work we did not consider disk I/O, thus we were not able to verify if the system can achieve fairness there.

From the above we conclude that Xen can control effectively how emulated nodes share the CPU and memory resources of the physical node and can ensure fair access to them. The challenge in our system thus is to ensure resource isolation in the networking subsystem of the cluster node in the presence of network condition emulation. In particular when some packet flows are subjected to traffic conditioning such as rate limiting or delaying there should be no impact to other unrelated packet flows. We achieve this by structuring the packet processing subsystem of the cluster node as shown in Figure 5. In a Xen system communication with the virtual interfaces of the emulated node is performed through the netback driver that executes in the cluster node. This driver handles sending and receiving packets to all the interfaces of all emulated nodes hosted on the node. Packets are exchanged with each virtual interface over a private circular queue with 256 entries. The actual data transfer is implemented through page remapping between the two domains avoiding in this way data copying. Pages for data arriving from the emulated nodes are taken from a pool of 256 pages that is shared among all virtual interfaces. The memory management system will call a page free handler when the data on the page are freed that will return the page to the pool. When there are no pages available in the pool the emulated nodes can not send more data until more pages are available. When packets are received by the driver they are placed at the kernel's packet processing queue (input_pkt_queue) using netif_rx(). The kernel processes these packets asynchronously through the RX SOFT_IRQ mechanism.

First we extend this system so that it can provide network condition emulation. As we discussed above we use NISTnet for network condition emulation. NISTnet executes in the cluster node of each physical system. NISTnet uses the Real Time Clock (RTC) available in PCs for achieving a timer interrupt rate of up to 8 KHz. The original NISTnet intercepts IP packets as they enter the system by replacing the default handler for incoming IP packets and applies traffic conditioning on a IP flow basis using a separate conditioning rule for each flow. In our case we need to apply network condition emulation only at the source of the packets, i.e. as soon as the packets enter the cluster node. In order to intercept incoming packets as soon as possible and prevent packets that will be subsequently dropped from loading the system we perform the NISTnet processing before packets are placed in the kernel's packet processing queue. The netback driver passes the packets it receives to NISTnet's packet handler. NISTnet needs to process packets at the granularity of a virtual interface since we apply the same traffic conditioning rules to all traffic arriving from the same virtual interface. Thus the unique ifindex associated with each virtual interface can be used to select the appropriate conditioning rule. In this way we can dramatically simplify the rule lookup in NISTnet; we can implement it with a simple hash of even a table.

One of the conditioning actions supported by NISTnet is rate shaping. This is accomplished by delaying packets so that packets exit the box at the desired rate. In certain cases this can be quite dangerous; if we allow very high rate packet flows to enter the cluster node and attempt to shape them there, we will create large backlogs of packets that will tax the resources of the cluster node and potentially cause resource contention and unfairness. For this reason we decided to implement rate limiting at the netback driver and control the rate that packets are entering the cluster node. Enforcing the bandwidth rate limit at the netback driver is simple given the existence of the high frequency timer used by NISTnet. With the help of this timer we maintain appropriate budgets for all the rate limited virtual interfaces. When a packet is received by the driver this budget is reduced and when it is exhausted the sending node will be blocked and will be permitted to send again only when the budged in replenished. The frequency that this budget is updated is configurable. Since now the netback driver is responsible for enforcing rate limits we disable NISTnet's machinery for rate limiting.

Implementing packet delay also presents some complexities. As we showed above the page pool used by the netback driver is shared among all emulated interfaces and limited to 256 pages. Pages are returned to the pool when the data they are carrying are released, i.e. the packets are sent out to the physical network. The limited size of this page pool can cause contention and unfairness between sending emulated nodes. If xen0 has enough resources to process all the packet traffic from and through the emulated nodes the page pool is usually not exhausted and all nodes can appear to get fair network service. This is not true though when packets are delayed before sent out. Delayed packets prevent their page to be returned

to the page pool and this will cause the page pool to be exhausted rather fast keeping even unrelated emulated nodes from sending data. The only way to remove the contention for the limited page pool is to ensure that packets are returned to this pool as fast as possible by copying the data to a new location and releasing the page. This wastes away the benefits of page remapping performed by Xen. Still, we only copy packets that need to be delayed and after we have applied rate limiting to them. We will discuss the overhead of packet copying in the evaluation section and will show it is negligible. When NISTnet determines it needs to delay a page it allocates a timer, adds it in a timer data structure and returns. When the delay period expires, the timer management code will schedule the packet for transmission by placing the packet in the kernel's input packet queue using netif_rx(). Clearly, NISTnet can not buffer and delay an unlimited amount of packets. We limit the number of packets that can be buffered to 10,000 for all virtual interfaces. When this limit is reached, packets are dropped. Since this limit is shared among all virtual interfaces in cases that there is contention we can have unfair dropping of packets among different virtual interfaces. We can deal with this by proper buffer management but it is not necessary. As we will see in the evaluation section our system can typically handle a load of 100,000 packets/sec, thus if we limit the maximum delay we can impose to 100 msec, then 10,000 is sufficient for buffering all the delayed packets. For 1,500 byte packets buffering 10,000 packets requires a reasonable 15 Mbytes of memory.

With all these modifications we have removed most of the points of contention in the packet path through the cluster node except one, the kernel's input packet queue. All packets received from the virtual interfaces will be eventually put into this queue to be processed asynchronously by the TX SOFT_IRQ. The kernel enforces a limit on the size of the queue and also a quota on how much packets from a certain device can be put in the queue in each round. Contention for this queue can cause unfairness. We could definitely introduce our own mechanisms for managing this queue but we found the mechanism already implemented by the Linux kernel to be sufficient. By setting the proper frequency of updating the receive budgets in the netback driver we can control the size of the arriving bursts from the virtual interfaces and we can ensure that the kernel's input queue is not overrun with sudden bursts of incoming packets. In our experiments we were able to verify that with the right settings packets were never dropped at this queue.

So far we have covered the path from the emulated nodes to the physical network. The path in the opposite direction is much simpler. As soon as a packet is received from the physical network it is passed to the kernel bridge and then it is sent directly to the netback driver. The driver will deliver the packet to the corresponding emulated interface. There is a dedicated circular buffer for each emulated interface avoiding in this way crosstalk between different interfaces. If the destination interface does not have enough buffers the packet is dropped. This path does not have any potential contention points where traffic directed to different interfaces can interfere with each other thus we need not worry about fairness. Emulated nodes will be able to receive packets at a rate that is determined by their CPU allocations which are controlled by Xen.

C. The assignment problem

All network emulation testbeds need to solve the problem of mapping the resources (nodes and links) of the target network to the topology of the emulation testbed. This mapping should be performed in a way that satisfies the constraints of the emulation testbed and minimize the usage of its resources. In our emulation architecture the constraints are the number of virtual nodes that can be supported on a single emulation node (10 for the version of the testbed we present in this work) and the amount of link bandwidth available for allocation to the emulated network links (two Gbits/sec in each direction). As discussed earlier, when two routers that have a link between them are assigned to the same emulation node the link is implemented as leaving the emulation node from one interface and entering through another since we want to avoid implementing links between virtual routers internally in the emulation node. Also, in order to make the network interconnect scalable, we do not require full connectivity between the Ethernet interfaces of the emulation nodes. Rather, they are split into groups that do not communicate between them.

Thus, the network mapping problem in our testbed can be formulated as follows: Given an input network as a set of routers (R), links (L), and a bandwidth requirement for each link and given an emulation setup with a set of physical nodes (N) and Interfaces (I), a partition of I into broadcast domains (P), a maximum node capacity N_{max} and a maximum interface capacity I_{max}, find a mapping from L to $I \times I$ so that all links members of L are covered, none of the constraints is violated, and the maximum utilization of any node and interface is minimized (in an attempt to spread the load evenly).

Determining this mapping is algorithmically difficult. Some versions of the problem can be shown to be NP-complete. For example if we consider the case where all interfaces belong to a single broadcast domain and $N_{max} > R$ then the problem can be mapped to the bin packing problem. The bins are the members of N, their capacity is the total capacity of the interfaces attached to each node, the items to be placed are the members of R, and their weights are the sum of the bandwidth requirements of their adjacent links.

Another version of the problem could be to ignore the link capacities. This could make sense in an emulation that is targeted at studying routing effects where link latencies and loss characteristics are more important than their overall capacity. In this case, we can again map the problem to the bin packing problem. The bins are again the members of N, their capacity is N_{max}, the items are the members of R, and their cost is 1.

The full version of the problem with the partition of the broadcast domains and the multiple constraints is also hard. While we do not prove formally we suspect it is not easier than the problems described above. Thus, our mapping algorithm is based on the worse fit heuristic. When placing a link we pick a node/interface that carries the smallest number of emulated nodes. Node selection has precedence over link selection and we start by placing the virtual nodes that have the largest sum of adjacent link bandwidth requirements. In a simple implementation, for each link we need to scan all the emulation nodes and their interfaces to select where to place the link, thus the complexity of the algorithm is $O(L * N)$.

We have implemented various front ends so that our placement algorithm can accept input from multiple popular network topology generators such a BRITE [32], GT-ITM [33] and we also can read the topologies generated by the Rocketfuel project [34].

IV. Performance Evaluation

In this section we report the results of the detailed performance evaluation we carried out on our emulation testbed. Our testbed is built by a number of dual CPU systems with AMD Opteron CPUs running at 2.4 GHz, 512 Kbyte L1 cache, and a Tyan motherboard with two Tigon3 built in Gigabit Ethernet interfaces. Each system has 1 GByte of physical memory and a 120 GByte SATA disk. Each system also has a 100 Mbit Ethernet interface that is used only for control. The cluster and the emulated nodes all run Linux version 2.6.11 modified to work under Xen. We use version 2.7 of Xen and version 3.0a of NISTnet. We focus on evaluating a network of emulated routers. All emulated systems have one input and one output interface and operate as packet forwarders. All have equal load and we allocate physical resources to them evenly. We allocate 64 Mbytes of physical memory to each emulated node and 256 Mbytes to the cluster node. All the nodes are assigned the same CPU scheduling weight. The emulation node is assigned to one of the CPUs while the emulated nodes share the second CPU. Since our application is packet forwarding, our performance metrics are packet forwarding throughput and latency of virtual routers and how do these values scale as we emulate more and more virtual routers on a single physical node.

A. Packet forwarding performance

The IETF benchmarking methodology group has published a series of RFCs with performance measurement guidelines. In our performance evaluation we will follow the guidelines in RFC 2544 [35] and the terminology of RFC 1242 [36]. From the multiple measures of routing performance described in RFC 2544 we will use only two in this work: forwarding latency and throughput, i.e. the maximum forwarding rate that can be achieved without packet loss. We will relax the loss free requirement and we will accept a small loss rate, i.e. less than 0.05%. We observed that in the commodity software and hardware used in our emulation testbed in almost every experiment there will be some random packet loss even at very low packet rates. Apparently some of the artifacts of scheduling in the emulation node's operating systems can cause some packets to be dropped. In addition to the lost packets we also monitor the number of packets that arrive out of order. We consider both lost and packets delivered out of order when we calculate the overall loss rate. Throughput is measured for all the packet sizes mentioned in [35] but for brevity we will only show results with packet size of 64 and 1450 bytes that correspond to the minimum allowable packet size and a size close to the default Ethernet MTU of 1500 bytes. Each experiment is repeated five times. For the maximum loss free packet forwarding rate, the number we report is the median of the three best rates we observe. For latency we report the average value. In each test we send 3 million packets. Since our physical nodes had only two network interfaces we were limited to at most two 1 Gbit/flows through the routers, one for each direction of the Gigabit links. All our traffic flows are constant bit rate and consist of packets of the same size. Our virtual routers have very small routing tables; we verified that the cost of routing table lookup when forwarding traffic was insignificant.

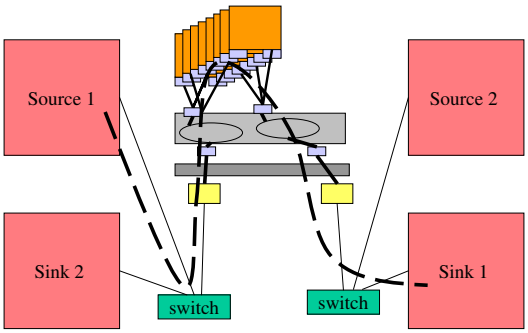


Fig. 6. Setup for routing performance measurements

Among the available traffic measurements tools [37] we decided to use a modified ttcp since it has a very simple implementation that makes it easy to change. When generating packet traffic we need to control precisely both the packet sizes and the rate of the outgoing stream. In order to achieve tight control of the sending packet rate we introduce delay through a busy loop between subsequent sends. This allows for good control on the packet sending rate without the need for timers. In our experiments we are careful to always use an unloaded machine for both the ttcp sender and receiver. The sender can split a packet stream into multiple sub-streams that all have the same packet rate. Each packet sent contains a sequence number and a stream identifier. This allows the receiver to determine packet loss and out of order packets.

For all throughput measurements we used the configuration shown in Figure 6. A pair of machines are used as traffic sources and another pair as traffic sinks. A fifth machine is connected to both source and sink machines and is the device under test (DUT). By properly setting up the routing tables

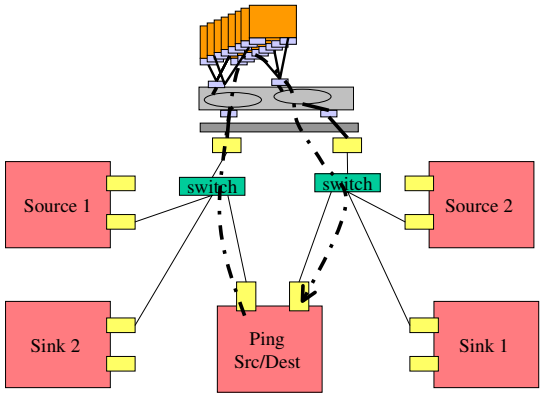


Fig. 7. Setup for measuring forwarding latencies

of the source and sink machines and the forwarding table of the DUT and its virtual routers it is possible to ensure bidirectional connectivity between the sources and the sinks and force traffic with a different destination address to be forwarded through a different virtual router. Our modified ttcp traffic generator can generate traffic streams that have varying destinations addresses so with a single ttcp process in each source machine we can generate multiple traffic flows, one for each virtual router.

In order to measure forwarding latency we use the setup shown in Figure 7. A ping machine originates ping packets that are forwarded by the virtual routers and then sent back to the ping machine. Packets are timestamped by the application that sends them out (which resides in user space) and when they are received by tcpdump. In this way by comparing sending and receiving time for each packet we can compute the time it took for the packet to arrive back at the source. This measurement will not be very accurate since the measured time includes the time it took the packet to be sent from user-space out to the network on transmission and also the time it took the packet to be delivered to the kernel on reception (interrupt, and context switches). While this imprecision does not allow us to get an accurate value of the forwarding latency we only show relatively increases in latency that can be directly attributed to the slower forwarding in the virtual routers. Ping packets are sent at a moderate rate (10 packets/second) and we send 100 packets for each virtual router. The value we report is the average of the latencies recorded for these 100 packets. The size of the ping packets is the same as the size of the packets of the workload in each case.

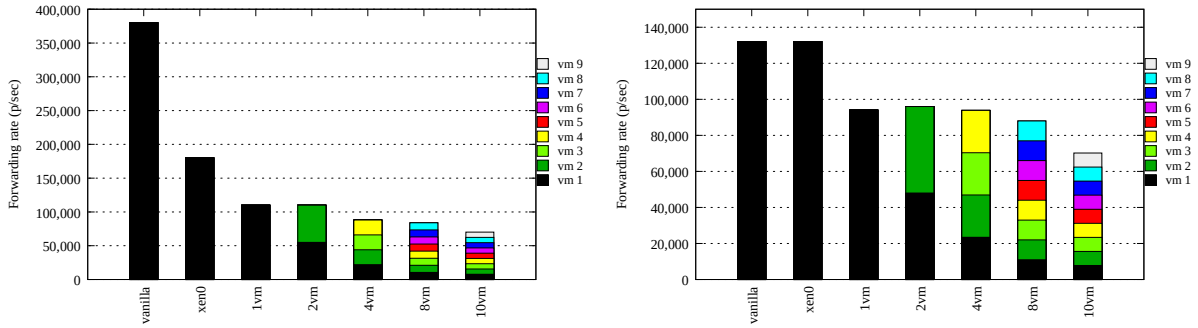
B. Baseline Experiments

Our initial experiment establishes the baseline performance of the system by running a vanilla Linux 2.6.11 kernel on the DUT. Then we characterize the effects of using the Xen hypervisor, even without having any virtual routers by running a xen0 system only on the DUT. The Xen hypervisor is responsible for handling all the events from the hardware and notify the appropriate host operating system through a soft interrupt mechanism. In networking applications Xen will have to virtualize all the interrupts from the network cards and this may have an effect on the interrupt handling capacity of the system. Finally, we perform tests varying the number of virtual routers on the DUT from 1 to 10 and measure their routing performance and the scaling of the overall system throughput as the number of virtual routers increases.

A summary of the packet forwarding rate results is shown in Figure 8 where we show the overall packet throughput for the various cases and how this throughput is split among the virtual routers. In the vanilla and xen0 cases, in order to saturate our test machine we had to use multiple streams especially in the case of 1450 byte packets since a single sender could not send packets fast enough. Two flows sent on opposite directions were sufficient to saturate the CPUs of our physical test node and achieve the maximum forwarding rate. For 1450 byte packets Xen and even the vanilla Linux system could not achieve more than 130,000 packets/sec, less than the total networking capacity available to our system which was 1 Gbit/second in each direction, i.e. 164,000 packets/second. Unmodified Linux can handle 380 Kpackets/second for 64 byte packets while a system that runs only xen0 can reach 180 Kpackets/sec. For small packets using Xen introduces a performance penalty while for larger packet sizes this penalty is almost non-existent. This behavior is consistent with the findings of [38] and is expected since with smaller packet sizes one can push much higher packet rates through the DUT causing a correspondingly higher overhead from the per packet processing.

Performance of the system drops when we introduce the first virtual router. Now xen0 will have to deliver the incoming packets to the virtual router and receive the packets output by the virtual router and forward them out of the outgoing interface. The additional CPU cost of these operations causes degradation in the packet forwarding performance of the router. This degradation is significant for the smaller packet sizes while for larger packets the reduction in performance is smaller. This is an indication that the per packet processing cost dominates the overall load when compared with the data movement cost. As we increase the number of virtual routers, we observe a further reduction of the overall packet forwarding performance but this is rather gradual and not very large. The small performance degradation is to be expected since each time control is transfered between virtual routers we have to pay the cost of the context switching between the virtual routers. As the number of virtual routers increases these effects cause a gradual reduction in the performance of the system. For a given packet rate the amount of context switching depends on how many packets a virtual router will process each time it is given control. This depends greatly on the dynamics of the interaction between the queues that carry packets between domains and how the virtual routers empty these queues. Again for larger packets this effect is smaller.

It is important to note that the performance numbers shown here are achieved when xen0 and the virtual routers reside on different CPUs of our two CPU system. Having both xen0 and



(a) 64 byte packets

(b) 1450 byte packets

Fig. 8. Packet forwarding performance

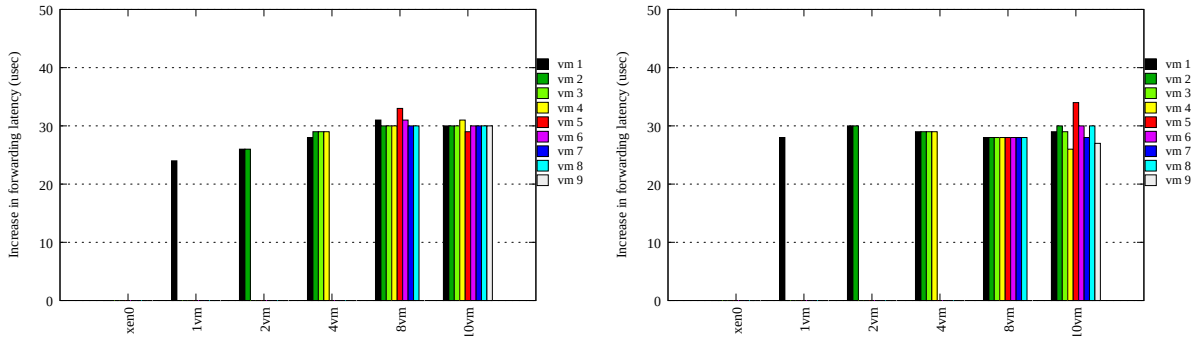
the virtual router run on the same CPU reduces performance dramatically. This is reasonable: xen0 is rather busy trying to keep up with the incoming packet traffic, thus it needs all the CPU resources it can get. Siphoning off even a small part of the CPU resources to the virtual router will adversely impact xen0 ability to process and deliver incoming packets to the virtual routers. As we add more virtual routers we keep placing them on the second CPU. Even when having multiple virtual routers, moving some of them in the first CPU still reduces performance.

In all cases we observe that the routing capacity of the system is split equally among all the virtual routers. This shows that indeed Xen fulfills its promise for controlled and fair access to the system resources. Even in cases of overload, where the offered packet rate is more than the capacity of the system, the amount of packet loss is split almost equally between the virtual routers. Our modifications on the packet paths through the cluster node removed contention points while the equal CPU weights and the same function of all the emulated nodes give each node the same opportunities to receive and process its own packets. Also by monitoring the network interface interrupts and the network interface statistics for both physical and virtual interfaces we were able to verify that even when the system operates at its load limit all incoming packets are successfully received by xen0, i.e. there are no packets dropped at the real interfaces. Both xen0 and the virtual router use polling of their network interfaces thanks to the NAPI capabilities of the Tigon device driver and the netfront device driver that implements the Ethernet device in the virtual router. This allows them to reduce the amount of time spent in interrupt processing when the network load increases. Also our setup with xen0 having a whole CPU for itself while the emulated nodes have to share the second CPU makes the emulated nodes the bottleneck; xen0 is able to process packets to and from the emulated nodes quickly, packets are usually delayed or lost when the queues for transmission to the emulated nodes exceed their maximum lengths. If we were to attempt to further improve the performance of the system we would need to consider how to make the emulated nodes more efficient.

Next we measured the forwarding latency of the system. In Figure 9 we show the increase in forwarding latency when compared to a xen0 only system when there is no load on the virtual routers. There is small increase in the latency of forwarded packets as we add more virtual routers but overall the latency is more or less constant as we add more virtual routers for both very small small and very large packet sizes. Also all virtual routers introduce the same amount of latency. Only in the 8 and 10 virtual router cases we observe some very small variation in the latencies introduced. In Figure 10 we show the corresponding results when all virtual routers forward traffic equal to their throughput, i.e. at maximum load for the respective configuration. These numbers have to be interpreted carefully since the offered traffic load is different in each configuration and latency numbers are not directly comparable. For this reason we show absolute latency values instead of the increase over the xen0 case. We observe there is a significant increase in latency when compared with the no load case but still all virtual routers introduce the same latency. It appears that adding even a single virtual router increases the latency significantly; adding more routers does have a milder effect on the forwarding latency. This is reasonable and consistent with our observations regarding the forwarding rate. Adding the first virtual router will incur a lot of the CPU overheads for the exchange of packets between xen0 and the virtual router. Adding more virtual routers will only modestly increase this overhead, especially since the offered traffic rate remains constant. For large packets latency remains more or less unchanged as we add more routers.

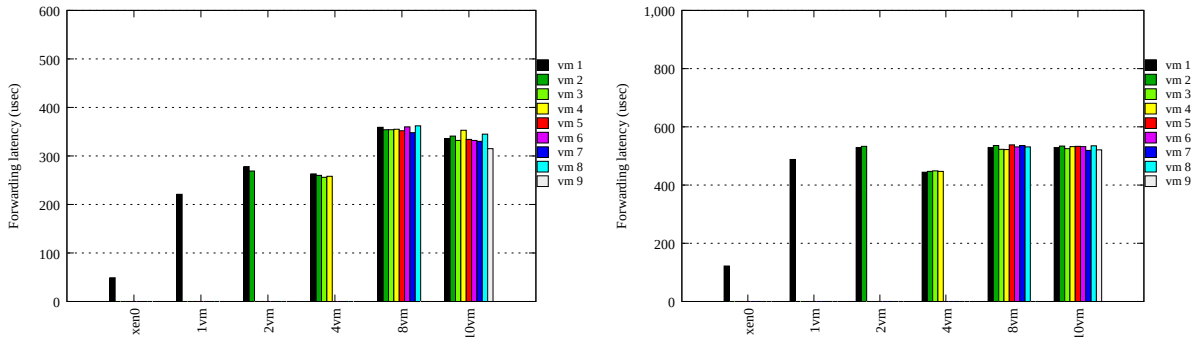
C. NISTnet performance

In this section we evaluate the performance of the emulation node when NISTnet is employed to shape and delay packet traffic. The evaluation setup and the performance metrics we use are exactly the same that we used in the previous section. NISTnet relies on frequent timer interrupts from the RTC for its operation. The interrupt frequency is configurable and can be as high as 8 KHz. The high number of interrupts may cause a drop in forwarding performance as CPU cycles are lost servicing the interrupts. Thus, our first concern is to evaluate the overhead introduced by NISTnet. Then we want to ensure that NISTnet indeed operates correctly and achieves



(a) 64 byte packets (b) 1450 byte packets

Fig. 9. Latency times no load



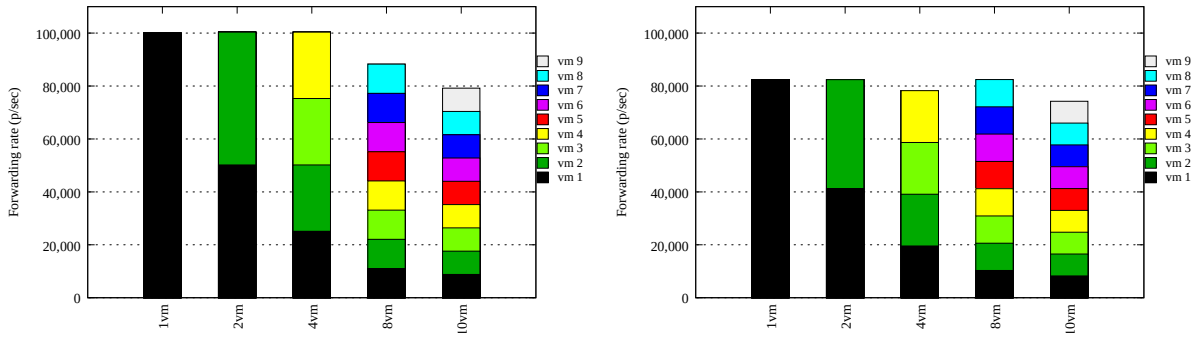
(a) 64 byte packets (b) 1450 byte packets

Fig. 10. Latency times under load

its emulation goals. In the experiments in this section NISTnet copies data that have to be delayed immediately from the pages used by the netback driver so as to avoid exhaustion of its page pool. We allow NISTnet to copy only up to 10,000 pages, if there are no more pages available incoming data are dropped.

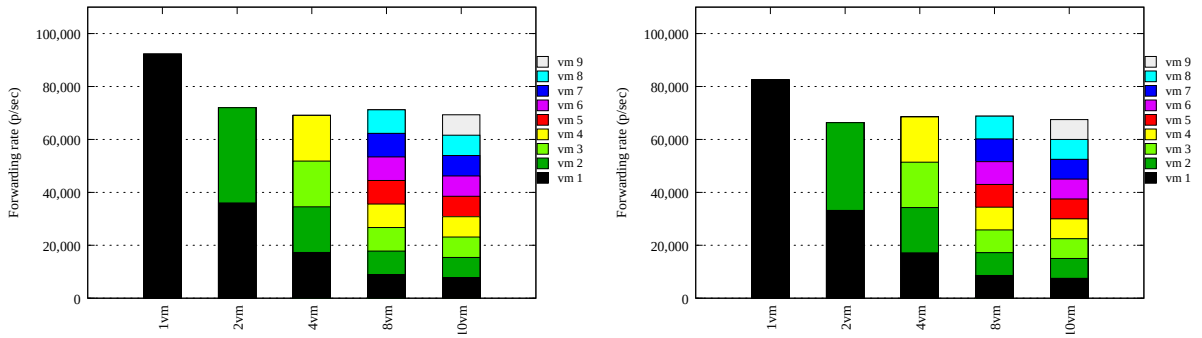
First we evaluate the overhead of the packet processing performed by the emulation node. Recall that we need to copy packets out of the page pool of the netback driver, update the rate quotas in the netback driver so that it can enforce rate limits and also intercept packets in NISTnet, and compute their latencies and drop probabilities. We verified that the additional cost of packet copying is negligible. We modified NISTnet so it copied all the packets it intercepted (instead of only the packets that have to be delayed) and we were able to observe the same peak throughput when running multiple packet streams at rates close to the maximum forwarding capacity of the emulated routers. Furthermore, we run experiments where the rate limits where set to values larger than the maximum throughput achieved in a particular configuration. Packet forwarding performance was unaffected showing that the rate limiting machinery does not incur any measurable overhead.

Next we evaluate the effects of NISTnet's packet processing overhead through two experiments. In the first we measure throughput when NISTnet is enabled but it does not have any rules configured. This will allow us to get some idea of the overhead introduced by the RTC interrupts. In the second experiment we add a 0.5 msec delay on each incoming packet. This is expressed as a separate rule for each of the emulated network interfaces. We use a very small delay value since this will not change too much the traffic pattern, will not require excessive memory for queuing packets and still will result in the maximum processing cost since each incoming packet will have to be intercepted and delayed by NISTnet. Both the experiments are performed using the maximum value of 8 KHz for the RTC clock. In Figure 11 we show the performance results when NISTnet is deployed but without performing any network emulation functions while in Figure 12 we show the same results for when NISTnet delays each packet for 0.5 msec. We show results only when virtual routers are configured since our version of NISTnet does not work for vanilla and xen0. First, we observe that the overhead of NISTnet has an impact on the performance of all cases. This is to be expected since NISTnet makes heavy use of the timer interrupts and it will have to pay the cost for the interrupt virtualization performed by Xen. Still, this performance loss is around 10% for 64 byte packets and it slowly drops off as we add more virtual routers resulting in practically the same performance as without NISTnet for 10 virtual routers. In all cases, forwarding capacity is split evenly across all virtual routers involved. When operating with the .5 msec delay there is an additional loss of performance, again in the order of 10%.



(a) 64 byte packets (b) 1450 byte packets

Fig. 11. Packet forwarding performance with NISTnet without any network emulation functions



(a) 64 byte packets (b) 1450 byte packets

Fig. 12. Packet forwarding performance with NISTnet when adding .5 msec latency

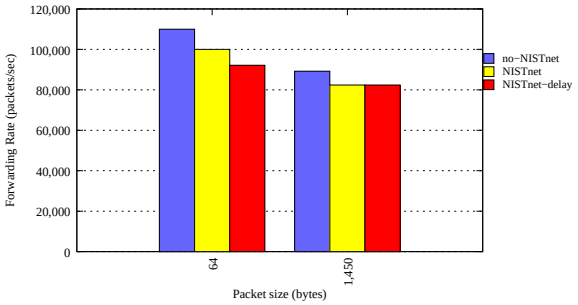


Fig. 13. Effects of NISTnet on packet forwarding throughput with one virtual router

Requested delay (msec)	min	avg	std
0	0.250	0.425	0.065
0.1	0.418	0.640	0.092
1	1.2	1.5	0.100
10	9.8	10.3	0.183
100	97.8	99	0.595

64 byte packets

Requested delay (msec)	min	avg	std
0	0.360	0.560	0.070
0.1	0.520	0.780	0.085
1	1.2	1.5	0.100
10	10.0	10.5	0.201
100	98.6	99	0.570

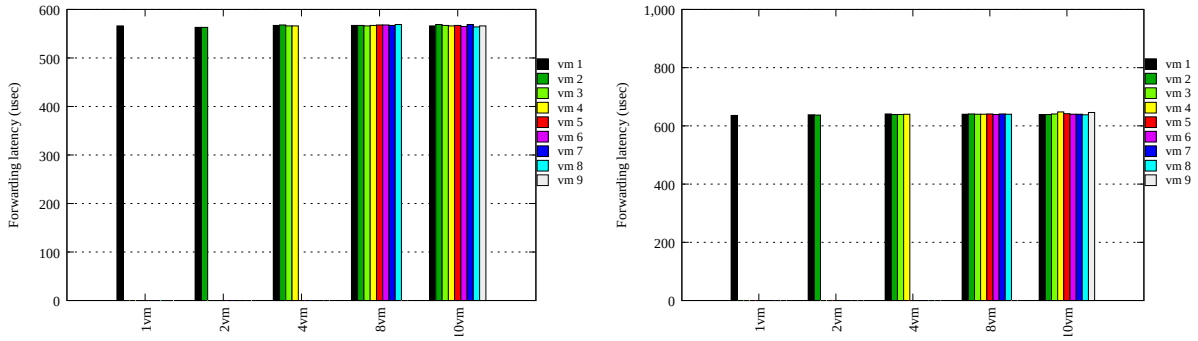
1450 byte packets

Fig. 16. Delay statistics under traffic load

Still forwarding capacity is split equally among all virtual routers. In Figure 13 we show the effects of NISTnet for different packet sizes in the case where a single virtual router is configured.

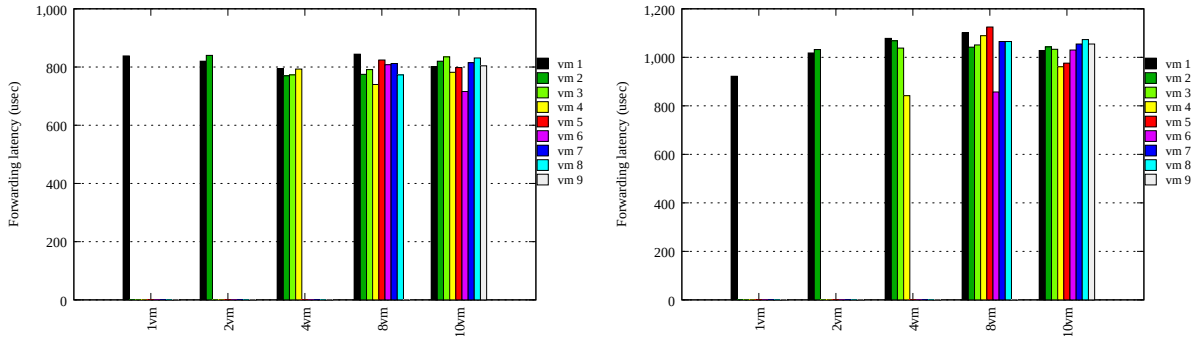
We also experimented with the timer set to interrupt at 1 KHz and we observed a drop in the packet forwarding performance. This shows that the real bottleneck is not the overhead of the timer interrupt or the management of the timer structures but instead the dynamics of the queues used for the communication between xen0 and the virtual routers. Reducing the real time clock interrupt rate results in an increased clustering of packets and the release of more packets to the system at each timer interrupt. These bursts of packets can disrupt the operation of the queues between xen0 and the virtual routers and offset any performance gains from the reduced interrupt rate.

To verify correct operation we monitor the latency of packet streams as they go through a single virtual router. We vary the number of packet streams (and thus the number of emulated routers hosted on a single physical node) between 1 and 10 and consider both the cases where there is no other load on the routers and where each handles the maximum load it can



(a) 64 byte packets (b) 1450 byte packets

Fig. 14. Minimum latency with NISTnet when adding a .5 msec latency under no traffic load



(a) 64 byte packets (b) 1450 byte packets

Fig. 15. Minimum latency with NISTnet when adding a .5 msec latency under traffic load

Requested delay	min	avg	std
0	0.070	0.080	0.012
0.1	0.2	0.260	0.041
1	1.057	1.116	0.042
10	10.087	10.170	0.100
100	100	100.1	0.220

64 byte packets

Requested delay	min	avg	std
0	0.143	0.155	0.014
0.1	0.270	0.330	0.040
1	1.126	1.193	0.050
10	10.150	10.220	0.120
100	100	100.170	0.320

1450 byte packets

Fig. 17. Delay statistics without load

support without losing packets. We show the minimum latency experienced by packets in the various configurations in Figures 14 and 15. This latency includes the time between sending the packets from the probe machine and the times the packets arrive back. All the probe packets were delayed for at least the amount we requested. We can also see the effects of the increased load on the system by their increased forwarding latency when compared with the latency when NISTnet was not used. As in the non-NISTnet case packet forwarding latency is split evenly across the multiple virtual routers. Only when under load and for 8 and 10 virtual routers there is some modest variation in the forwarding latency observed through each virtual router. From the above results we can conclude that NISTnet is able to implement correct delaying of packets even when the system is operating at its forwarding capacity. We show more information about the delay observed on the packets in the above experiments in Figures 16 and 17. We show minimum and average latency as well as standard deviation for different values of latency when both under load and without load. In the case under load, 10 virtual routers where forwarding traffic close to their maximum forwarding capacity and all had also to handle a stream of traffic that should be delayed. We show results for both 1450 and 64 byte packets for one of the delayed packet streams, the results are similar for the other 9 streams.

We should not forget that NISTnet simply adds the requested delay to the packets it intercepts. This means that in the resulting packet stream packets will be delayed for at least the desired amount but their observed delay will also include the forwarding latency through the emulated router. Since this latency is not negligible it may become important for small desired values of latency. For example in an unloaded router the minimum forwarding latency is around 0.070 msec for 64 bytes and 0.140 msec for 1450 bytes. If the router is under load

Test	Peak rate	Requested rate 1	10	100
64 bytes	12.9	1.5	10.6	12.9
1450 bytes	255	1.1	9.83	98

Fig. 18. Accuracy of bandwidth limiting, all rates in MBit/sec

Latency	Rate limit 1	10	100
0.1	1.1	10	98.5
1	1.1	10	98.3
10	1.1	10	98.4
100	1.1	9.9	97

Fig. 19. Effects of latency on packet rate, latency in msec, rate in MBits/sec

then this latency climbs to 0.250 and 0.360 msec respectively. Thus attempting to delay such packet streams for 0.1 msec does not make much sense. For larger values the effects of the forwarding latency are less noticeable.

We measured the accuracy of bandwidth limiting using the same configuration and considered a case with four packet streams each going through its own emulated router with a rate close to the peak forwarding capacity of the emulated routers; only one of the streams where rate limited. Recall that we perform rate limiting in the netback driver. For each virtual interface that is to be rate limited, we maintain a byte budget that is replenished periodically using NISTnet's high accuracy clock. When this budget is exhausted the interface is not allowed to send more traffic until the budget is replenished. In our experiments we found that it is desirable to update the packet budgets frequently otherwise we end up receiving big batches of packets which can interfere with the kernel's incoming packet queue. Using a low update frequency (10 updates/sec) we observed drops when adding packets to the kernel's packet queue. An update frequency of 100 updates/second appears to avoid this problem without any adverse effect on overhead. Using this value we show the rates we achieved under different rate limits in Figure 18 for both 64 and 1450 byte packets. The peak rate achieved in this configuration without rate limiting is also shown for reference. The regulated rate is rather close to the requested rate with less accuracy for slower rates and smaller packets. Since the available budget is expressed in bytes and not in packets, packetization will have an effect on the accuracy of the rate limiting. In our implementation a packet is always received if there is even one byte of budget available. Thus we have a tendency to exceed the requested rate for smaller rates.

Finally, we evaluated the accuracy of enforcing a given loss rate. In tests without load, NISTnet was able to drop the right amount of packets from a stream to emulate the requested traffic load. We also performed experiments under load, using 4 packet streams with only one subjected to packet dropping. All streams were sent at a rate close to the maximum that the emulated routers can support in this configuration. In this case also we were to verify that the right percentage of packets are dropped with zero effect on the traffic of the other streams. While deriving the right models for determining packet drop probabilities is complex its implementation is rather straightforward and does have minimal impact on the cluster node.

In a realistic configuration both latency and rate limiting will be applied to an emulated link. In Figure 19 we show what are the maximum rates we can achieve when enforcing delays on a rate limited stream for 1450 byte packets. Recall that when delaying packets we copy them immediately from the pages they were received and we add them in a pool that can hold only up to 10,000 delayed packets. Packets that can not be copied are dropped. If NISTnet would run out of room for copying incoming packets we would expect a drop in the forwarding rate of the delayed packet flow. In our experiments 10,000 pages were enough to avoid packet dropping and avoid any noticeable drop in the forwarding rate achieved when delaying even high rate flows. Indeed, as can be seen in Figure 8 our system can handle between 85,000 and 120,000 packets/second depending on their size. If we want to support adding delays of up to 100 msec, 10,000 pages is sufficiently large to buffer all delayed traffic and prevent dropping packets. Since this limit is not reached we do not have to worry about contention for these pages and the potential unfairness that can result.

As a test of overall performance isolation we run an experiment with the maximum of ten emulated nodes each with one incoming and one outgoing interface operating as packet forwarders. For half of them we rate limited their traffic to 100 MBits/second and applied various values of latency up to 100 msec. The 10th node was a CPU hog running 200 CPU intensive processes. The system was loaded with the maximum rate of packets it could handle for both 64 and 1450 byte packets. In all cases the system was able to correctly shape and delay the traffic while flows that were not rate limited or delayed achieved the same rate they would on a system where there was no traffic conditioning employed. The CPU hog was not able to affect the operation of the rest of the system.

V. Using the Testbed

In this section we discuss and show how our testbed can help dramatically with both testing and performance evaluation of networked applications. In this paper we will focus on routing applications. We will use a popular open source routing suite Quagga [39]. We will emulate realistic network topologies taken from the Rocketfuel [34] project complete with their link latency and link cost values. We will focus on the OSPF routing protocol and consider that the whole network runs OSPF in a single area. We will study correctness of the protocol operation, discover and study bugs and measure routing protocol convergence performance. We will perform most of our tests using the ISP1755 topology that has 87 routers, 161 links. We will configure all links as both point-to-point and

non-broadcast each representing a single network. Since we will not have any external routes all routers will carry exactly 161 routes each. For point-to-point links OSPF sends all routing traffic to the ALL-OPSF-ROUTERS multicast address while in non-broadcast links the traffic is unicasted to the remote endpoint of the link.

A. Measuring the effects of virtualization - timer synchronization

Virtualization will give us the illusion that we have a network built from multiple independent systems but we still need to be aware of the limitations of sharing the same physical hardware among multiple virtual systems. There are two main ways that this sharing will affect the operation of the system. Events that would occur simultaneously on two independent systems will be serialized on two virtual systems that share the same physical node. In addition, while a given operation will take an amount of time t to complete on each of two independent systems now will take at least $2 * t$ if the two systems share the same physical node. Being aware of these limitations is important when one attempts to perform timing related measurements on our testbed. For example, a very important measure of IGP performance is convergence time, i.e. the time until all routers have agreed on the same set of routes and have updated their forwarding tables accordingly. An apparently simple way to measure this convergence time in our testbed would be to have each virtual router add an entry in its system log each time it modifies its forwarding table. By comparing times in these logs we can measure the time between introducing a disturbance in the network and the time that the last router updated its forwarding table.

The first question arising is what is the clock synchronization achievable on our testbed? Since all physical nodes are connected on the same local area network, NTP can achieve very reasonable clock synchronization. Using the highest possible update intervals NTP can synchronize the clocks of the physical nodes within 300 usec of each other. We measure this by logging the time each physical node receives a broadcast packet. Assuming that each node receives the packet roughly at the same time (there is only one unloaded Ethernet switch in between) and that the packet receive processing at each node will be the same (all nodes are also non loaded) the time differences we observe are due to clock differences. Virtual routers get their clocks from the underlying physical node. On i386 systems, time is maintained by the real time clock hardware. An oscillator feeds a counter and a latch controls at what counter values a timer interrupt will be generated. The interrupt frequency is usually low (one every 10 msec in the latest Linux kernels) but the gettimeofday() system call can directly read the counter and get a much more accurate time reading. Virtual routers can also read the values of the real counter (but can not write to it) so they have the same accuracy as the underlying physical system.

Next we instrument virtual routers that run on the same physical system to enter a time-stamp in their log each time they receive a specific broadcast packet. For 10 virtual routers

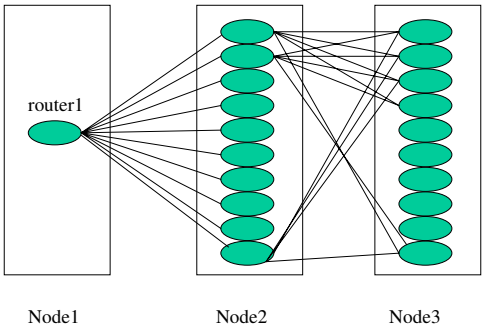


Fig. 20. Setup for measuring propagation latencies due to multiplexing of virtual routers

that are not loaded, the time difference between the first and the last entry in the log is less than 2 msec. Under load close to their maximum throughput this time increases to less than 2.5 msec. With 5 routers under no load the maximum difference is below 600 usec. When under load this time becomes 1 msec. This time captures the overhead of context switching between the virtual routers, delivering the packet to them and starting their execution. Of course, this assumes that Xen will be able to switch to the destination virtual router immediately. If the CPU where the virtual router has to run is busy with another virtual machine then delivery will delay until the scheduling quantum expires. In our configuration Xen uses a 5 msec quantum thus in a 10 virtual router system delivery of an event can delay up to 45 msec. Similarly, computational work is interleaved between the virtual machines. When 5 virtual machines have to process simultaneously an event that requires 12.5 msec for its processing, the last virtual router to complete the computations does so 64 msec after the arrival of the event on the physical system. The earlier completion is 52 msec after the arrival of the event. Events are delivered to their corresponding virtual routers at times 0, 5, 10, 15 and 20 msec after the arrival of the event as dictated by the quantum used by Xen for scheduling the virtual routers.

Now we can consider the effects of all the above to the measurement of the convergence time in the network. Assume a link fails on one virtual node. The node will flood an updated LSA to its neighbors. In Quagga each node will receive the update packet and will forward it immediately to its neighbors and will schedule a route re-computation for a later time. Thus the convergence time depends on the route re-computation time as well as on when the last router will receive the update. To investigate what happens in high load cases we consider an example with the topology shown in Figure 20. There a single virtual router on node1 is connected to all the 10 virtual routers on node2. Each of these virtual routers is connected to a set of other 10 virtual routers that exist on node3. Router1 triggers updates and we observe what happens on the links of node2. Updates from router1 arrive at a rapid rate about 15 usec apart.

Then we observe that all virtual routers have their chance to send their updates and this is done in one go, i.e. sending of updates is not interrupted. Thus the last update leaves the box roughly 4.5 msec after the first update was received. At node1 we observed 9 context switches that cost about 2.7 msec and also the transmission times for the 10 sets of 10 packet trains was 1.5 msec that adds up to 4.5 msec. Guided by the above experiment, in an example topology with 87 routers, 161 links, 8 physical nodes and a diameter of 4 the time it would take for an update to cross the network can range between the absolute minimum of 1.2 msec to a maximum of 18 msec.

The same is true for the routing table re-computation. All virtual nodes may end up performing their route computation at the same time. Computing routes for a 161 node network takes less than 6 msec when performed by a virtual router that is alone on a physical node. Thus, the computation of routes on a physical node that has 10 virtual routers may take up to 60 msec. Adding the uncertainty for the route computation with that of the flooding times we see that we can not hope to achieve convergence time measurements with an accuracy better than 100 msec on our testbed. This inaccuracy can be rather limiting in some cases. For example, the convergence time of the network has important implications for the formation of routing loops in the period that routers have not a synchronized view of the network. One may want to study the formation of these loops and characterize them using our testbed. Previous studies based on network traces [40] have observed routing loops that last as little as 50 msec. Using our testbed we would not be able to accurately observe routing loops of this duration.

B. Testing and debugging the routing protocol

Our testbed allows us to deploy protocols at a large scale while maintaining realism. In addition to performance evaluation this infrastructure can be invaluable for testing and discovering bugs. Still, even if it is possible to test the protocols in a variety of scenarios, reproducing bugs may not be simple. In our various tests we observed erroneous behavior from the routing protocols we had deployed. In order to be able to reproduce problems at will we developed our own logging/replay infrastructure for the particular routing stack we used and we also evaluated Jockey [41] an open source tool the can record/replay the execution of a process. Both logging infrastructures record all the input to the routing processes and its internal events. When we observed erroneous behavior or we had a process crash we could repeat the sequence of events that caused the error as many times as we needed to find the error. In these replays the routing process that failed was run inside the debugger. Using this record/replay infrastructure we were able to study a number of bugs in the latest version of the Quagga implementation of OSPF. These are simple and few bugs, on the other hand we only tried some very basic OSPF configuration with a single area and no complex OSPF features. Doing a full testing of this particular OSPF implementation is beyond the scope of this paper. We tested the latest version of Quagga available (0.98.5) using the ISP1755 topology configured as a single area. We configured our links to be both point-to-point and NBMA. In our tests we discovered the following problems:

(a) Quagga reads packets by posting a read handler that is called when packets arrive at the OSPF socket. When the read handler is invoked it re-posts itself so that the next packet can be read. In certain cases though, for example when the packet is received from a wrong interface or it is not possible to allocate memory for the packet, the read handler exits before re-posting itself. As a result, the OSPF process can not receive any packets afterward and it has to be restarted. The fix for this problem is very simple.

(b) The handler for deleting MAXAGE LSAs (function max_age_remover() in file ospf_lsa.c) can cause a crash in certain configurations. If routers are not configured with a router-id, they pick as an id the lowest IP address of their interfaces. If this interface fails, the routers will have to change their router id and re-advertise this to the network. This causes a crash in the OSPF process. Although running an OSPF network without configured router-ids is not recommended this is nevertheless an important issue.

(c) When we performed tests with NBMA links we occasionally noticed that after the network converged some routers were missing routes, especially after node restarts. If two nodes with a common link were restarted, then the route corresponding to this link disappeared for the network until the next time routers refreshed their LSAs, i.e. 30 minutes later. We traced this issue to a problem in the flooding procedure that rejects certain OSPF packets when they are received on an NBMA link. Since our interfaces are NMBA this results in the purging for network LSAs and prevents the proper convergence of the network. This seems to be a coding error since it is not described in section 13.4 of the OSPFv2 RFC [42] that details the checks to be performed in such a case.

In order to be able to reproduce crashes and determine the root causes of the problems we observed we had logging enabled while the routing protocols were executing. An important issue with this logging/replay is the log rate, i.e. the rate that data are added to the logs. If this rate is very high, maintaining logs for large periods of execution will be problematic. Then a check-pointing scheme becomes necessary: When a process is check-pointed, it can be restarted from the checkpoint and the previous logs are not needed anymore. With Jockey in the 1755 network we observed log rates in the order of 260 Kbytes/second in steady state. This is clearly too large since it will result in about 1 GByte of data every hour. In certain cases, given knowledge of the application operation it is possible to significantly reduce the log rates. For example, in a routing application, many packets are repeated (e.g. OPSF hello packets) while other packets have common parts (headers, IP addresses and so on). Also, typical routing daemons are structured around a central select loop that is dispatching the other protocol operations. These applications call select() repeatedly with short timeouts, thus select may not return any active descriptors. Furthermore, such applications use time heavily, but frequent calls to gettimeofday result in

small increments of time between successive calls that can be stored in the log more efficiently as time differences. In future work we will investigate if the above optimizations allow us to achieve one order of magnitude reduction to the above logging rate.

C. Routing protocol convergence measurements

In order to evaluate the performance of the testbed and the routing protocol under study, we investigate one of the most important metrics of routing protocol operation: convergence time. While routers update their routing tables they can be become inconsistent and traffic can be delayed, misrouted or lost as a result. Thus it is important for the network to converge quickly to a consistent state after a change in its topology. Convergence time includes the time to detect a change in the topology (link failure, router failure, new link or node added) the time to flood this update to all the nodes in the network and the time for each node to compute new routing tables and update their forwarding tables. We measure the convergence time of the OSPF routing protocol on the ISP1755 network. We introduce topology disturbances in the form of link failures and recoveries as well as node restarts, node failures and recoveries. Each time a router updates its forwarding table it enters a timestamped entry in its log. By looking at the last of these entries for each router we can tell when the particular router converged. As we discussed at length before, we can not measure convergence time to an accuracy less than approximately 100 msec thus we focus on an accuracy of one second. We repeat the experiments four times and we report the average results. The router dead interval is configured to 40 seconds, the SPF computation is scheduled 1 second after the first notification of a network change and consecutive SPF computations can not be scheduled before 1 second elapses after the last computation. This is necessary to avoid continuous re-computation of the routing table after receiving each link state update. When interfaces change state the kernel notifies the routing protocols thus there is no need for Quagga to periodically poll for the interface state.

We perform a number of experiments where we fail or restart a number of links and routers and we measure the time it takes the network to re-converge. Convergence time is the time between the time of earliest modification of a routing table and the latest across all the network. We break down this time into detection time which is the time until the (first) change will be reflected in the topology, i.e. until a routing adjacency will be terminated or be re-established and the remaining time. In the link failure scenario we fail 5 links that belong to different routers. We fail the links by bringing down both their endpoints at the routers at their ends. This will ensure that the detection of the failure occurs as fast as possible. In this scenario, all routers finish route re-computation within the same second that the failures occurred. The link failure is detected immediately by the routers adjacent to the link which also flood the information to all other routers. Flooding is very fast and all routers update their forwarding tables as soon as they can execute an SPF computation. In the link up test we bring these links up again. This time convergence takes longer. As soon as the neighboring routers establish an adjacency over the link they will advertise it to the rest of the network and then after some time all routers will update their forwarding tables. In our experiments adjacency is established 41 seconds after the link is operational and convergence is completed 25 seconds later. Apparently Quagga is not very aggressive when establishing a new adjacency.

Test	Detection	Convergence	Convergence few links
link down	0	2	2
link up	41	25	9
router down	26	17	2
router up	3	49	39
router restart	3	52	49

Fig. 21. Summary of convergence times, times in seconds

In the router related tests we stop and start 6 routers. When routers are brought down Quagga will detect the failure of the link through the OSPF HELLO timeout mechanism that can take between 30 and 40 seconds to detect that a link has failed. On the average in our experiments the first router to bring down the adjacency did so around 26 seconds after the router failure. After this the rest of the network took 17 seconds to converge. Bringing routers up, adjacencies with the new router are formed 3 seconds after the routers become active. In the case of a new router Quagga appears to be much more aggressive when setting up the routing adjacency. After this adjacency is setup it still takes the network 49 seconds to converge. Restarting routers will cause the tear down of the old adjacency immediately when the router becomes active again. The re-establishment of the adjacency is very fast (similar to the case where a new router starts) but convergence will happen 52 seconds later.

We also repeated these measurements when failing only a single link or router at a time. We list the convergence numbers in the last column of Table 21. In these measurements the detection time does not change but the convergence time is larger in the link up and router down cases. In these cases, there is more variability in the times it will take to establish adjacencies over the new links or time out the adjacencies with the dead routers and this is reflected as more time for the network convergence. In all the other cases, detection is very fast so independent of the number of affected links and nodes convergence will take about the same time.

Of course, the above measurements reflect the particular design choices and mechanisms implemented in Quagga. Achieving fast routing protocol convergence has received a lot of attention and there exist established guidelines for achieving fast convergence even in the millisecond range [43]. The measurements in the above paragraph were meant as a demonstration of how we can use our testbed to study such issues. We easily identified two areas that contribute to overall convergence time: detection of the change (especially of the

failure) and the rate limiting on the SPF computation. These are well understood and there have been proposals to deal with both [43], [44].

VI. Discussion

A. Forwarding performance considerations

The results we presented here for the vanilla and the xen0 systems were derived running the uniprocessor kernel although we used a dual processor system. We also run experiments with the vanilla system with the 2.6.11 SMP kernel configured for 2 CPUs. Using the SMP kernel did not achieve any significant increase in throughput. For 64 byte packets we observed a small increase and were able to reach 410 Kpackets/second before the CPUs saturated compared to 380 Kpackets/second in the uniprocessor case. For 1450 byte packets we did not observe any increase compared to the uniprocessor case, but in this case there was 40% CPU capacity available. Apparently data movement overheads and CPU cache misses limited the performance of our system in this case. In the SMP experiments we have configured the interrupt affinity so that each physical interface delivers interrupts to a different CPU. We also run experiments where we allowed the kernel to dynamically reassign interrupts to CPUs but without any effect on the observed performance. Like in all other cases we used traffic patterns that crossed the machine entering and exiting the system through different physical interfaces. In this case there is not much interrupt affinity and we did not expect the interrupt assignment to make any difference. Overheads of data movement inside the machine and especially cache misses can have important effects on the performance of the system as discussed in [45]. In this work we have taken the simple approach of assigning all virtual routers to one CPU and keeping xen0 on the other. While this arrangement achieved adequate performance it is not clear if it can be further improved. One other arrangement would be to split the virtual routers among the two CPUs with each CPU responsible for handling traffic for one of the physical interfaces. When a virtual router is assigned to a CPU, we make sure that all its virtual interfaces are assigned to the corresponding physical interface. In this way we try to achieve interrupt and data affinity in the CPUs and reduce the performance overheads from data movement in our system. We will study similar alternative organizations and their performance in future work.

Another important issue is the ability of the system to avoid livelock under heavy load. Older Linux kernels when faced with a very high incoming packet rate would spend most of their CPU on interrupt processing resulting in a collapse of the packet forwarding rate. Newer Linux kernels fix this problem using the New Network API or NAPI that switches to polling the devices when the interrupt rate becomes too high. This reduces the CPU time spent in interrupt processing and stabilizes forwarding performance under heavy load. In our experiments we were able to verify that this indeed is the behavior of the system in overload. In a system that uses virtual routers though, the situation is different. As we discussed in the evaluation section, although the bottleneck of the system is the exchange of traffic between xen0 and the virtual router(s) xen0 has enough CPU resources to receive all packets even when the capacity of this interface is exceeded. As more packets enter the system CPU resources are wasted for processing them while they will eventually be dropped at the queues to the emulated nodes. Thus, the livelock situation returns. Indeed, while we are able to achieve 100 Kpackets/second for 64 byte packets with a single virtual router, when the offered load is increased the achievable throughput is reduced to 60 Kpackets/second and then to 35 Kpackets/second as the offered load is increased further. Even at this high levels of offered load, xen0 is able to receive all the incoming packets. Effectively the interface polling mechanism employed by NAPI becomes inactive since it is not triggered. Although we have multiple packet drops due to inefficiencies in the communication between xen0 and virtual routers, the CPU that runs xen0 is not saturated thus the kernel remains responsive and keeps admitting more packets into the system.

B. Extending the cluster

Finally, it may be desirable to use a physical node as part of an emulated network. While in other testbeds that use centralized network condition emulation this is trivial, this is not the case in our design. The main reason is that certain emulation services, most importantly the network condition emulation is performed at the source of the emulated traffic. Attaching a physical node will require this node to run a network condition emulator that is compatible with the one used in the rest of the cluster. To a certain degree this is a disadvantage of the distributed network condition emulation. Workarounds could be devised in cases where it would be absolutely necessary to incorporate a physical node in an emulation.

A final consideration is how to allow physical nodes that are not on the same LAN to participate in an emulation. These could be isolated nodes or complete emulation clusters. Establishing connectivity between the clusters is probably the simplest part of the problem since Ethernet frames can be encapsulated over IP and exchanged between the different sites. Technologies such as Layer 2 Virtual Private Networks for example can give the appearance that multiple wide area sites are connected through a large layer two switch that performs learning and spanning tree computation. Such extensions are beyond the scope of this paper though.

C. Scalability

There can be some limitations on the scalability of our scheme, especially with respect to using a broadcast Ethernet network for the interconnection of the physical nodes. In particular scalability could suffer from the following:

- Limited amount of MAC addresses on the switches: Typically low-end switches support 4K MAC addresses, thus in large configurations the total number of virtual interfaces could exceed this number. The solution to this problem is the partitioning of the switched space across

multiple disjoint switches. During the mapping phase we can ensure that virtual interfaces will be assigned in such a way so that the MAC address limit per switch will not be exceeded. Note that MAC addresses will still be distinct for each virtual interface. We could reuse MAC addresses if necessary since each interface of a physical node has a different software bridge associated with it. So, in principle two virtual interfaces could be assigned the same MAC address even if they are located in emulated nodes that are on the same physical node, as long as they belong to a different emulated node and they use a different interface of the physical node. Still, since the MAC address space is large enough for even the largest configuration, MAC address reuse is not necessary.

- Amount of broadcast traffic: When traffic is destined to a multicast destination, traffic is sent to a multicast MAC address. By rewriting the destination MAC address of packets we direct all this traffic only to the proper destination physical node. For broadcast links we map the destination MAC to an appropriate multicast MAC address. While older switches treated the multicast MAC addresses as broadcast, newer switches that support IGMP snooping can selectively forward multicast traffic to its appropriate destinations avoiding redundant packet traffic.
- We have only considered physical nodes with two interfaces. This limits the load on the system and can show improved performance of our scheme. Other work [45] has showed that on systems with more than 2 Gigabit interfaces, Xen can cause much more noticeable performance degradation. Although two interfaces may be enough for creating large emulated topologies, when we add more interfaces to our physical nodes, we should expect that the overall performance of the system will not improve linearly. We are in the process of acquiring additional interfaces for our system so we can evaluate the performance of physical nodes with multiple Gigabit interfaces.
- We only support a limited amount of emulated nodes per physical node: This is caused by the artificial limitation imposed by the particular method for configuring virtual disks for our virtual routers. We chose the option that provides higher performance but is limited by the number of maximum disk partitions allowed by Linux. This is not a serious limitation and there has been reports of modifications to the Linux kernel that removes this limitation. We will evaluate such larger multiplexing factors in future work.

VII. Related Work

Multiple network emulation testbeds have been discussed in the literature each with different scalability goals and application focus. Emulab [1] consists of real nodes connected through local and wide area real links. To achieve scale multiple emulated nodes can be hosted on a single real node using extensions of the FreeBSD *jail* [12] mechanism that is based on standard kernel services. Jails can not easily guarantee resource isolation between the emulated nodes and and misbehaving nodes can compromise the operation of other emulated nodes that share the same physical node. Packet loss, delays and other artifacts are emulated with the help of the Dummynet software executing on dedicated real nodes. In Emulab network switches are also considered part of the topology and are mapped into real switches or physical nodes. Multiple emulated nodes that communicate with each other can be mapped on the same physical node, something we avoid in our emulator. In such case a different mechanism needs to be used in Emulab to emulate link delays. A custom encapsulation of network packets is used to ensure that packets are delivered to the correct destination interfaces over an unmodified Ethernet switch infrastructure.

PlanetLab [3] consists of real nodes placed in different locations and connected through wide area links with each real node housing multiple emulated nodes. PlanetLab reports that resource isolation and sharing is one of the most challenging aspects of their testbed operation [15]. Currently PlanetLab nodes run a Linux operating system that includes significant modifications in order to ensure resource isolation between the logical nodes sharing the node. Processor scheduling has been replaced and capabilities for controlling access to resources have been implemented in the kernel. We were able to achieve all this functionality with significantly less effort and modifications to the Linux kernel. Each logical node gets access to the network through a special socket type, more special socket types are needed to generate or receive ICMP messages and be able to receive all packet traffic. In our case all these are handled transparently since all the emulated nodes have device level access to the underlying network. It would be interesting in the future to compare the performance of one of our nodes with that of a PlanetLab's node.

Empower [8] and Modelnet [2] use local area links to connect real nodes. In both network conditions are emulated in an emulation core that consists of multiple dedicated real nodes with specialized support for network emulation. Empower performs this emulation through virtual routers that reside in the emulation core and each can introduce delays, losses and shape traffic. Modelnet implements this emulation by passing traffic through multiple "pipes" inside the emulation core where each pipe can shape traffic and introduce delays and losses. Emulated nodes can be created on physical nodes that are outside the emulation core. In all the above testbeds a physical node can typically host few tens of emulated nodes. The multiplexing methods used do not allow for resource isolation between the emulated nodes.

Network Emulation Testbed [46] is a emulation testbed that is uses a virtualized network stack in order to multiplex multiple emulated nodes on a physical host. Emulated nodes are still processes of the underlying operating system and thus there is no guaranteed resource isolation between them. Finally, the work that is closer related to ours is vBET [23] where the authors build an emulation testbed using virtual-

ization. Their nodes are complete virtual machines using the UML virtualization approach. The major problem with this approach is the low performance due to the high overheads of UML and the lack of proper performance isolation between the emulated nodes. The authors address the latter partially through a different CPU scheduler for the host operating system but clearly contention can arise for other shared system resources such as buffers, queue elements and so on. In order to resolve these one has to either implement the detailed resource capabilities framework of PlanetLab or use completely isolated virtual machines like we do in this work. IMUNES [47] is a testbed based on virtualization of the networking stack inside the kernel. Emulated nodes are modeled as operating system processes thus suffering from the problem of limited resource isolation.

Finally, other emulation testbeds have more specific goals that our work here: ORBIT [48] focuses on emulating wireless networks and is using specialized hardware to speed up the emulation of wireless links. NS Emulation [49] shows how to combine simulation and emulation in order to achieve maximum emulation flexibility.

VIII. Conclusions and future work

In this work we first demonstrate how to combine open source para-virtualization technology and publicly available network emulation software to create a scalable testbed where we can evaluate large network configurations while maintaining full realism with respect to the complexity of the emulated nodes. We showed how to integrate network condition emulation and Xen's mechanisms for exchanging packets between domain so as to ensure isolation and eliminate bottlenecks when packets flow through the cluster node. We measured the performance of our emulation system on an applications where we create multiple virtual routers on each physical node and we use them for forwarding packet traffic. Using modern commodity hardware there is enough capacity for supporting at 10 virtual routers along with network condition emulation in a single physical node. Routing performance decreases as the number of virtual routers increases but even with 10 virtual routers and network condition emulation for 1450 byte packets we can still achieve an aggregate packet forwarding rate of about 60% of the maximum forwarding rate that the physical node can achieve when running vanilla Linux. This is around 1 Gbit/second and it scales linearly with the number of nodes in the cluster. With only a 10 node cluster we can handle aggregate data rates of 10 Gbits/second and emulate systems with up to 100 nodes. This performance is achieved while maintaining strict resource isolation. Misbehaving or overloaded emulated nodes do not affect the service received by other emulated nodes and can not affect the performance of the cluster node. Moreover, resources are shared fairly among the virtual routers. We were able to verify that the forwarding throughput and latency achieved by each virtual router was similar even when the system was operating close to its capacity. We learned that in in our system the bottleneck for the packet forwarding performance is the virtual routers and sending packets to them. When our system is overloaded most of the packet drops happens on the queues that send packets to the virtual routers. The cluster node was able to handle the offered load. This is certainly an artifact of assigning a whole CPU on the cluster node and sharing the other CPUs among the emulated routers.

We chose to evaluate a virtual router application because virtual routers present an interesting architectural approach. Since they use the services of the underlying virtualization layer they have a simple interface with the rest of the system allowing the virtualization layer to enforce effective isolation and resource scheduling. Indeed, we were able to verify that Xen can ensure fair sharing of system resources under overload. This provides a simple and safe mechanism for extending the functionality of an existing router. Effectively this provides an inherently safer approach to active networks. Combined with some other unique capabilities of virtual machines such as the ability to migrate even running virtual machines and to perform detailed logging and replaying one could envision multiple applications for virtual routers. On the other hand, for this to become a reality the performance of virtual machine routers and the overheads imposed by virtualization need to be understood better. This work is a first contribution in this direction. In our experiments we found the bottleneck to be the virtual routers while the cluster node had enough capacity to handle all the traffic it saw. Most traffic losses happened when the cluster node had to forward a packet to a virtual router that did not have enough resources to receive it. This certainly is an artifact of dedicating one of the CPUs of our system to the cluster node. In Future work we will evaluate single CPU configurations and alternative organizations for the virtual routers using tools similar to that of [45].

Finally, two important issues require more investigation. The first is to study if it is possible to achieve better structuring of the virtual machines with respect to interrupt and data affinity. It may be possible to group virtual routers into sets that only deal with data arriving and leaving a single physical interface and assign them to the same CPU that receives the interrupts from this interface. We will study these alternatives in combination with a detailed low level performance evaluation of the operation of the system. The second issue is the reemergence of livelock due to the loss of the load feedback that activates NAPI. In an overloaded system the kernel will pull packets from the network device queues slowly shedding some load as incoming packets are dropped at the network interface. In our case, the system reaches saturation while not necessarily being CPU limited since the bottleneck is moving packets to the emulated routers. As a result the xen0 kernel is still able to continue admitting incoming packets exacerbating the performance problems and resulting in significant loss of performance when load exceeds the capacity of the system. This behavior can be corrected if we provide better feedback to the kernel so that it can go into interrupt polling earlier as we approach the capacity of the virtual routers.

REFERENCES

[1] B. White et al., An Integrated Experimental Environment for Distributed Systems and Networks, in Proceedings of 5th Symposium on Operating Systems Design and Implementation (OSDI), 2002

[2] A. Vahdat et al., Scalability and Accuracy in a Large-Scale Network Emulator, in Proceedings of 5th Symposium on Operating Systems Design and Implementation (OSDI), 2002

[3] The PlanetLab project, http://planet-lab.org

[4] The Linux Virtual Servers Project, http://linux-vserver.org, accessed September 2005

[5] The Bochs IA-32 Emulator Project, by Kevin Lawton et al., http://bochs.sourceforge.net/

[6] The User-mode Linux Kernel, http://user-mode-linux.sourceforge.net/

[7] The new Plex86 project, http://www.plex86.org/

[8] L. Ni and P. Zheng, EMPOWER: A Network Emulator for Wireline and Wireless Networks, in IEEE InfoCom 2003, San Francisco, March 2003

[9] P. Barham et al., Xen and the Art of Virtualization, in Proceedings of the Symposium on Operating System Principles (SOSP), October 2003

[10] VMWare ESX, http://www.vmware.com/products/esx

[11] VMWare Workstation, http://www.vmware.com/

[12] FreeBSD Jails, jail(2) of the FreeBSD on line manual

[13] The VirtualPC PC virtualizer, http://www.microsoft.com/windowsxp/virtualpc/

[14] Virtuozzo for Windows and Linux Server Virtualization, http://www.swsoft.com/en/products/virtuozzo

[15] A. Bavier et al., Operating System Support for Planetary-Scale Network Services, in proceedings of NSDI'04, San Francisco, USA, March 2004

[16] The NISTnet project, htpp://snad.ncsl.nist.gov/itg/nistnet

[17] L. Rizzo, Dummynet:A Simple Approach to the Evaluation of Network Protocols, ACM Computer Communication Review, January 1997

[18] M. Allman, A. Caldwell, and S. Ostermann, ONE The Ohio Network Emulator, Technical Report TR-19972, Ohio University, 1997

[19] T. Garfinkel et al., Terra: A Virtual Machine Based Platform for Trusted Computing, in the Proceedings of the Symposium on Operating Systems Principles (SOSP) 2003, New York, October 2003

[20] The L4Ka Virtual Machine Technology, http://l4ka.org/projects/virtualization,

[21] A. Whitaker and M. Shaw and S. D. Gribble, Scale and Performance in the Denali Isolation Kernel, in Proceedings of the Fifth Symposium on Operating Systems Design and Implementation (OSDI 2002), Boston, December 2002

[22] K. Buchaker, and V. Shieh, Framework for Testing the Fault Tolerance of Systems Including OS and Network Aspects, in Proceedings of the Third IEEE International High-Assurance System Engineering Symposium, HASE 2001, Florida, 2001

[23] X. Jiang and D. Xu, vBET: a VM-based Emulation Testbed, in Proceedings of the ACM SIGCOMM 2003 Workshops, Karlsruhe, August 2003

[24] The Qemu project, http://fabrice.bellard.free.fr/qemu/

[25] X. W. Huang and R. Sharma and Srinivasan Keshav, The ENTRAPID Protocol Development Environment, in Proceedings of INFOCOM 99, 1999

[26] D. Ely et al., Alpine: A User-Level Infrastructure for Network Protocol Development, in Proceedings of the 3rd USENIX Symposium on Internet Technologies and Systems, USITS 2001, San Fransisco, March 2001

[27] M. Zec, Implementing a Clonable Network Stack in the FreeBSD Kernel, in Proceedings of the 2003 USENIX Annual Technical Conference, San Antonio, June 2003

[28] I. Yeom and A. L. N. Reddy, ENDE: An End-to-End Network Delay Emulator, technical report, Department of Electrical Engineering, Texas A&M University, 1998

[29] D. B. Ingham and G. D. Parrington, Delayline: A Wide-area Network Emulation Tool, Computing Systems, USENIX, 7(3):313-332, 1994

[30] E. Nahum et al., The Effects of Wide-Area Conditions on WWW Server Performance, in Proceedings of the 2001 ACM SIGMETRICS international conference on Measurement and modeling of computer systems table of contents, Cambridge, Massachusetts, 2001

[31] Ebtables, http://ebtables.sourceforge.net

[32] A. Medina, A. Lakhina, I. Matta and J. Byers, BRITE: An Approach to Universal Topology Generation, in Proceedings of International Workshop on Modeling, Analysis and Simulation of Computer and Telecommunications Systems (MASCOTS), August 2001

[33] K. Calvert, M. Doar and E. Zegura, Modeling Internet Topology, IEEE communications Magazine, June 1997

[34] N. Spring and R. Mahajan and D. Wetherall, Measuring ISP Topologies with Rocketfuel, in Proceedings of ACM/SIGCOMM '02, August 2002

[35] S. Bradner, J. McQuaid, Benchmarking Methodology for Network Interconnect Devices, Request for Comments 2544, March 1999

[36] S. Bradner, Editor, Benchmarking Terminology for Network Interconnection Devices, Request for Comments: 1242, July 1991

[37] CAIDA, Performance Tools Taxonomy, http://www.caida.org/tools/taxonomy

[38] K. Fraser et al., Safe Hardware Access with the Xen Virtual Machine Monitor, in the OASIS ASPLOS 2004 workshop

[39] Quagga Software Routing Suite, http://www.quagga.net/

[40] A. Sridharan et al., On the Correlation between Route Dynamics and Routing Loops, in proceedings of IMC'03, Florida, USA, October 2003

[41] Jockey execution record/replay library, http://home.gna.org/jockey

[42] J. Moy, OSPF Version 2, Request for Comments: 2328, STD: 54, April 1998

[43] Alaettinoglu, C., Jacobson, V., and Yu, H., Towards Millisecond IGP Convergence, NANOG 20, October 2000

[44] Bidirectional Forwarding Detection (BFD) IETF Chapter, http://www.ietf.org/html.charters/bfd-charter.html

[45] A. Menon et al., Diagnosing Performance Overheads in the Xen Virtual Machine Environment, in proceedings of VEE'05, Chicago, USA, June 2005

[46] D. Herrscher and K. Rothermel, A Dynamic Network Scenario Emulation Tool, in Proceedings of the 11th International Conference on Computers and Networks, ICCCN 2002, Miami, October 2002

[47] IMUNES - An Integrated Multiprotocol Network Emulator/Simulator, http://www.tel.fer.hr/imunes

[48] The ORBIT Testbed, http://oribt-lab.org

[49] K. Fall, Network Emulation in the Vint/NS Simulator, in Proceedings of the Fourth IEEE Symposium on Computers and Communications, ISCC 1999, Egypt, July 1999