# vBET: a VM-Based Emulation Testbed

Xuxian Jiang
Department of Computer Sciences
Purdue University
West Lafayette, IN 47907, USA
jiangx@cs.purdue.edu

Dongyan Xu
Department of Computer Sciences
Purdue University
West Lafayette, IN 47907, USA
dxu@cs.purdue.edu

## ABSTRACT

With the increasing requirement of robustness and predictability for network protocols and distributed systems, it becomes necessary to develop realistic, customizable, and scalable emulation testbeds for the testing and evaluation of network and distributed protocols. A number of recently proposed emulation testbeds have clearly demonstrated the advantage and promise of this approach. Meanwhile, more efforts are necessary to achieve higher degree of flexibility and customizability, especially for the creation of arbitrary network topology and for the customization of network-level entities.

In this paper, we present vBET, an efficient and flexible emulation testbed using the virtual machine technology. Based on Linux, vBET can be installed in a high-end desktop or a commodity server and is therefore easily deployable in a research lab. vBET creates a virtual distributed environment with both network infrastructure and end systems. Each entity, such as a router, switch, firewall, or application-level proxy, is emulated by a virtual machine running unmodified system or application software. The entities emulated by vBET are user-configurable. Furthermore, the same (physical) vBET server can be easily setup as testbed for different experiments, such as Internet routing, distributed firewalls, and peer-to-peer networks.

We describe the design, implementation, and application of vBET. For the design and implementation, we present key enabling techniques including virtual OS, virtual networking, and small-footprint file system. For the application of vBET, we demonstrate the creation of different experimental environments using vBET, including OSPF routing, distributed firewall, and Chord peer-to-peer network. These experiments reflect the versatility, customizability, and efficiency of vBET.

## 1. INTRODUCTION

There has been increasing requirement of robustness and predictability for network protocols and distributed systems, such as IP routing [19, 22] and packet scheduling [25], peer-to-peer systems [9, 11, 18], overlay networks [6, 7], and computation/data grids [8, 12]. It has become necessary to develop realistic, customizable, and scalable emulation testbeds for the testing and evaluation of these protocols and systems.

Meanwhile, traditional simulation tools, such as the widely used ns-2 [2], are more available and economical, due to their easy installation, management, and relatively low resource requirements. Unfortunately, simulation-based experiments may be deemed less convincing, due to their lack of fidelity to real-world environments. On the other hand, real-world experiments are based on realistic settings and therefore more credible. However, real-world experiments are highly complex and costly to set up, control and monitor. Between the two ends, emulation provides a good tradeoff between fidelity and cost. The goal (and challenge) of emulation testbed development is therefore to achieve controllability, configurability, reproducibility, scalability, and ease of setup and management. In particular, the emulation testbed should be *flexible* enough to create a wide range of network environments for different experiments.

Recently, a number of real-world or emulation testbeds have been successfully deployed. Representative testbeds include PlanetLab[21], Netbed[28], and ModelNet[27]. These testbeds clearly demonstrate the advantage and promise of emulation-based experimentation. Still, a number of challenges exist in the design and implementation of emulation testbeds. In this paper, we address the following challenges.

- *Easy and wide deployment* Like ns-2, it is desirable that an emulation testbed be easily deployable in any research lab equipped with commodity servers or high-end desktops. As a result, researchers will have full control over the testbed and enjoy more convenient execution and monitoring of the experiments.
- *Setup of arbitrary network topology* The testbed should support flexible setup and re-wiring of network topology. This requires a stronger virtualization of network connections by the testbed.
- *Customization of network-level entities* In addition to end-system node customization, the testbed should also

support the customization of network-level entities, such as the replacement of packet queuing discipline or IP lookup algorithm of a particular router in the emulated environment.

- *Fast start-up and tear-down of experimental environment* It is expected that the start-up or tear-down be automatically performed within *seconds*, so that researchers do not have to experience long waiting time during the experiments.

In this paper, we present vBET, a flexible and efficient emulation testbed using the virtual machine technology. Based on Linux, vBET can be installed in a high-end desktop or a commodity server and is therefore easily deployable in a research lab. vBET creates a virtual distributed environment with both network infrastructure and end systems. Each entity, such as a router, switch, firewall, or application-level proxy, is emulated by a virtual machine running unmodified system or application software. The entities emulated by vBET are configurable by users on-demand. Furthermore, the same (physical) vBET server can be easily setup as testbed for different experiments. The key enabling techniques of vBET include virtual OS, virtual topology, and small-footprint file system. We have used vBET to create network environments for experiments with OSPF routing, distributed firewall, and Chord peer-to-peer network.

The rest of the paper is organized as follows. Section 2 presents an overview of vBET, including a network topology modeling language for vBET users. Section 3 describes vBET implementation in detail. Section 4 demonstrates the creation of a number of experimental environments using vBET. Section 5 compares our work with related works. Finally, Section 6 concludes this paper.

## 2. OVERVIEW OF VBET

vBET leverages virtual machine technology to achieve scalability and local deployability: each real-world entity is emulated by an independent virtual machine, which is physically a 'slice' of the vBET server. Figure 1 shows that the same vBET server can be used to create different experimental environments: the first one is a simple three-node network for the evaluation of OSPF protocol; the second one is a typical multi-LAN environment; and the third one creates a distributed firewall environment. Each node inside the emulated environments is a virtual machine, inter-connected by virtual links. To enable the virtual machines, vBET leverages and extends User-Mode Linux (UML) [10], an open-source Linux-based virtual OS[1]. vBET is scalable with respect to the number of virtual machines in the vBET server: In our laboratory, one vBET server (a Dell PowerEdge 2650 server with Xeon 2.6GHz CPU and 2GB memory running Linux-2.4.19) can support up to 60 virtual nodes[2].

### 2.1 Steps of vBET Emulation

Our current implementation of vBET is suitable for small-scale experiments based on Linux. Due to its limited support for resource isolation (Section 3.5), vBET is more suitable for experiments that evaluate systems or protocols qualitatively than for experiments that require quantitative accuracy. To perform emulation using vBET, there are three main steps:

- *Topology specification* A researcher will use a simple but expressive topology modeling language (Section 2.2) to specify the experiment network topology.
- *Virtual node and topology creation* Based on the topology specification, the next step is to map the logical entities in the experiment to virtual machines in vBET. During this step, vBET performs both *virtual node creation* and *virtual topology creation*. Virtual node will have the specified capabilities, such as running a specific routing protocol or performing an intrusion detection task. Meanwhile, virtual topology creation makes sure that the topology of the virtual nodes conforms to the topology specification. For example, an *OSPF* router will be mapped to a virtual node which runs OSPF software and communicates with its neighbor OSPF routers as in the topology specification.

  vBET achieves functionally accurate one-to-one mapping from each entity in the topology specification to a virtual machine. In addition, virtual switches, hubs or routers may be added when necessary to glue different virtual nodes. A configuration script is created as the result of this step, which can be re-used or updated for multiple runs of the experiment. Currently, each experiment is performed by *one* vBET server. The mapping of one experiment to multiple vBET servers under the constraints of physical vBET server topology and resources is a non-trivial problem [23], and is not yet supported by vBET.
- *Experiment run* In this step, vBET starts the experiment by invoking the configuration script created in the second step. A key feature of vBET is that each virtual node has a unique reserved IP address, and port redirect technique is employed to provide remote researchers with console access to each virtual node for runtime monitoring and management.

### 2.2 Network Topology Modeling

The vBET network topology modeling language is similar to the facilities provided by ns-2, but easier to understand. Yet it is expressive for the modeling of network and distributed environments, especially for the composition of complex network topologies based on simple ones.

Currently, the vBET network topology modeling language supports four different resource types.

- *Network* A network represents a medium for communication between network devices. A network can be logical or physical medium depending on the granularity of topological composition.
- *Network Device* A network device is a communicating entity, such as a bridge, switch, router, firewall, NAT box or even end host. A network device can generate, forward or accept real packets.

[1]UML is completely *different* from UMLinux, another virtual machine project, with respect to implementation.

[2]We use 'virtual machine' and 'virtual node' interchangeably.

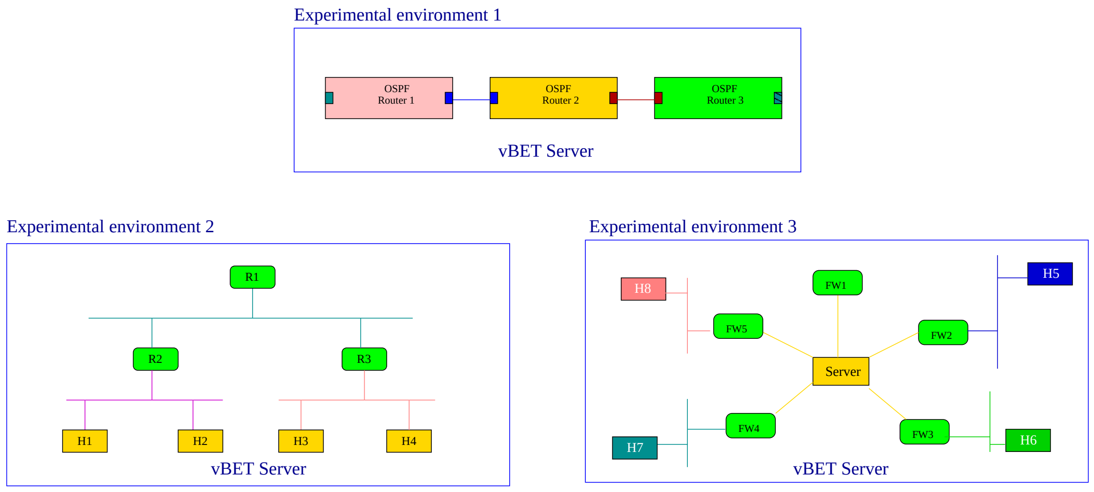


**Figure 1: Different experimental environments created in a vBET server**

- *Network Interface Card* A network interface card (NIC) is the entity that handles the actual packet sending and receiving. It has the flexibility of being dynamically attached to or detached from a network device.

- *Cable* A cable refers to the physical or emulated communication link which enables the actual packet transmission between network devices.

Based on the abstract resource types, the language further defines three pairs of primitive operations.

- *alloc/dealloc*: A simple resource, such as a network device, NIC or cable, can be allocated and deallocated for an experimental environment.

- *attach/detach*: A NIC can be attached to or detached from a network device.

- *link/unlink*: A cable can be used to link two NICs, which are attached to two network devices, respectively. A link can be broken by the unlink operation. The link (unlink) operation is achieved by performing the action of *plug* (or *unplug*) on both ends of a cable.

To illustrate the usage of vBET network topology modeling language, Figure 2 shows a simple network with three hosts connected by a switch. The corresponding topology specification is shown in Figure 3.

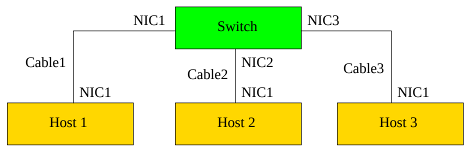


**Figure 2: Simple Ethernet topology**

```
switch1 = {attach(alloc(switch), alloc(NIC),alloc(NIC), alloc(NIC))}
host1 = attach(alloc(host), alloc(NIC))       host2 = attach(alloc(host), alloc(NIC))
host3 = attach(alloc(host), alloc(NIC))       cable3 = alloc(cable)
cable1 = alloc(cable)                         cable2 = alloc(cable)

plug(cable1, NIC(switch1)   plug(cable2, NIC(switch1))   plug(cable3, NIC(switch1))
plug(cable1, NIC(host1))    plug(cable2, NIC(host2))     plug(cable3, NIC(host3))
```

**Figure 3: Topology specification**

## 3. VBET DESIGN AND IMPLEMENTATION

In this section, we presents vBET's design goals of flexibility, scalability, and customizability, as well as its implementation details, including virtual OS, virtual networking, small footprint file system and resource isolation.

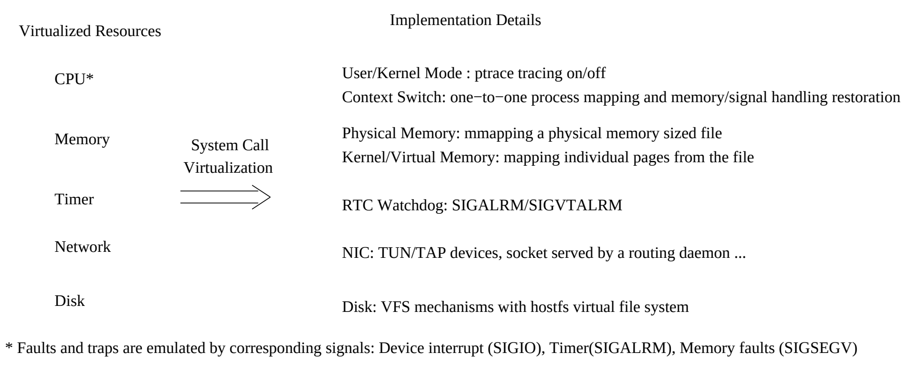
### 3.1 Design Goals

Besides easy deployment and versatility for a wide range of network and distributed experiments, the design goals of vBET also include:

- *Topology flexibility* Topology flexibility is desirable for on-demand creation of arbitrary network topologies, especially the ones composed from basic network topologies such as ring, star, or switch-enabled LAN. There should be no physical limit on the number of physical network connections for each network device.

- *Node customizability* Every node in the experimental environment should be further customizable for experiments with different network services and software, such as different service disciplines and routing algorithms. Support for customization should be provided not only for end-systems, but also for network-level entities (such as routers).

- *Scalability* Instead of scaling the number of physical servers in the testbed, vBET focuses on the scalability with respect to the number of virtual machines in one physical vBET server. Current virtual machine techniques, such as VMWare [4] and the original UML [10],



Virtualized Resources
Implementation Details
CPU*
User/Kernel Mode : ptrace tracing on/off
Context Switch: one−to−one process mapping and memory/signal handling restoration
Memory
System Call Virtualization
Physical Memory: mmapping a physical memory sized file
Kernel/Virtual Memory: mapping individual pages from the file
Timer
RTC Watchdog: SIGALRM/SIGVTALRM
Network
NIC: TUN/TAP devices, socket served by a routing daemon ...
Disk
Disk: VFS mechanisms with hostfs virtual file system
* Faults and traps are emulated by corresponding signals: Device interrupt (SIGIO), Timer(SIGALRM), Memory faults (SIGSEGV)

**Figure 4: Summary of resource virtualization in UML**

are not lightweight enough to enable many virtual machines needed in a typical emulation experiment. We also expect to contribute our techniques for scalability to the existing emulation or real-world testbeds.

## 3.2 Virtual OS

To implement virtual machines, techniques at three levels are involved: *guest OS*, *virtual machine monitor*, and *host OS*. Guest OS provides a confined environment for all processes running inside it, thus achieving administration, fault/attack, and resource isolation between virtual machines [17]. Virtual machine monitor provides fundamental underlying resource virtualization. Host OS provides the ultimate physical I/O and memory access for virtual machines, and schedules virtual machine processes as regular processes based on certain scheduling policy, such as round-robin or fair queuing [26].

vBET supports Linux as the host OS, and it leverages and extends UML, an open-source virtual OS project. UML can support most Linux applications without modification inside a virtual machine, except some applications involving privileged instructions, such as *hwclock* using *iopl* and *inb/outb*. Unlike other virtual machine techniques such as VMWare [4], a UML runs directly in the unmodified *user space* of the host OS. Processes within a UML will be executed in the guest OS exactly the same way as they would be executed in a native Linux machine, which gains performance benefit in contrary to the overhead of instruction-level interpretation such as in Java VM [13]. In UML, a special thread is created to intercept the system calls made by all process inside the UML and to redirect them into the host OS kernel. An additional process context environment may be created to store or restore context information at the entry or exit point of a system call, which can reduce context switching overhead.

One major challenge in virtual OS is to achieve resource virtualization, including process address space assignment for guest kernel/user mode differentiation, system call interception and virtualization which involves context switch, virtual memory emulation, network interface emulation, and RTC timer emulation for preemptive scheduling. Figure 4 summarizes the virtualization mechanisms in UML. Interested readers are referred to [10] for more details.

We have extended the host OS (Linux) to improve UML scalability and isolation. More specifically, a small footprint root file system (Section 3.4) is implemented to improve UML scalability and a CPU scheduling algorithm (Section 3.5) is implemented to improve resource isolation between virtual machines.

## 3.3 Virtual Networking

Virtual networking enables communications between virtual nodes, and is essential to topology flexibility. In the following sections, we describe vBET networking techniques for different types of virtual nodes.

### 3.3.1 Virtual Hub/Switch

Like a regular physical hub (switch), a virtual hub (switch) enables simple packet forwarding and constructs a basic LAN environment with multicast capability. Virtual hub will forward every packet received to every available port, which may result in degraded performance due to the multiple packet copies. Virtual switch adds some intelligence to packet forwarding, so that only designated receivers will receive the packet.

We have two options to emulate virtual hub or switch: *TUN /TAP* and *socket tunneling.*

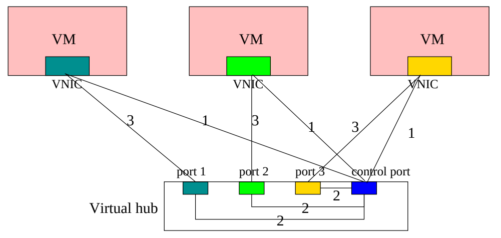
- *TUN/TAP* dynamically creates or deletes a new interface in the host OS, and thus requires routine with root privilege or *setuid* root privilege. This may not be desirable and may limit its portability.
- *Socket tunneling* encapsulates real packets as socket packet payload and provides one uniform, portable and flexible way of virtualization, at the cost of higher overhead than *TUN/TAP*. Socket tunneling will also enable emulations based on multiple vBET servers, which can not be achieved by *TUN/TAP*.

In vBET, we adopt the *socket tunneling* approach. In our current single-vBET-server environment, one temporary UNIX domain socket is created as the point of contact in a virtual hub (or switch)[3]. It receives incoming connection requests and creates one virtual port for each incoming connection. The established UNIX socket connection will serve

[3]In a multiple-vBET-server environment, a UDP socket daemon will be created instead of a UNIX domain socket.

as the 'cable' connecting to the corresponding virtual NIC of a virtual machine. UNIX socket connections do not impose a physical limit on the number of ports created, although they do incur some overhead. Various packet queuing and forwarding polices can be implemented in the UNIX socket server of a virtual hub (or switch), in order to emulate link characteristics such as bandwidth, delay, loss rate, and congestion.

Figure 5 shows the virtual hub and the creation of ports in virtual hub. The *poll* system call is used by the virtual hub to poll the arrival of packets and perform corresponding packet processing such as queuing, forwarding or dropping. Different packet queuing and forwarding polices can be applied at this point. *Poll* system call also notifies the UNIX socket daemon (emulating virtual hub or switch) of the arrival of a connection request, so that the daemon can allocate a new port to a new request. The total number of ports that can be created in the virtual hub may be configured when the UNIX socket daemon is started.



1: VM initiates a unix socket connection request to a specified unix socket daemon (control port)
2: The request is acceptted by the unix socket daemon, and a new port is created
3. VM establishes physical connection to virtual hub via VNIC

**Figure 5: Virtual hub and port creation procedure**

To evaluate the throughput and scalability (with respect to the number of ports) of a virtual hub/switch, we perform the following measurement as set up in Figure 6. In a vBET server (Dell PowerEdge 2650 server with Xeon 2.6GHz CPU and 2GB memory running Linux-2.4.19), we create a number of virtual nodes as senders and one virtual node as receiver - they are interconnected by one virtual switch. We then measure the aggregated throughput observed by the receiver using *ttcp*, under different number of senders (and thus different number of ports created in the virtual switch).

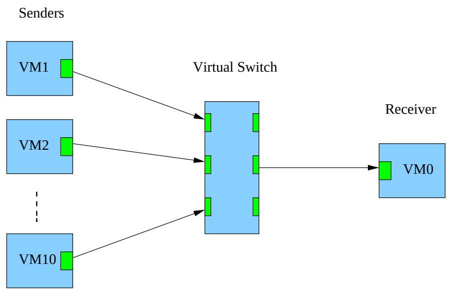


**Figure 6: Setup for virtual switch throughput measurement**

Figure 7 shows the measurement results. The maximum throughput observed by the receiver is 128Mbps, and the throughput is not seriously degraded while the number of ports created in the virtual switch increases from 3 to 11. However, when the number of ports continues to grow, the packet encapsulation overhead due to socket tunneling will increase and finally limit the scalability of vBET. As an alternative, *pipe* [24, 27] or some similar mechanism may be used to lower the overhead. Unfortunately, that will disable the emulation of *traceroute*-like network protocols; and it does not support physical multicast. Overall, for small-scale experiments, our experience shows that the overhead of socket tunneling is outweighed by the flexibility it brings to the creation of arbitrary network topology.

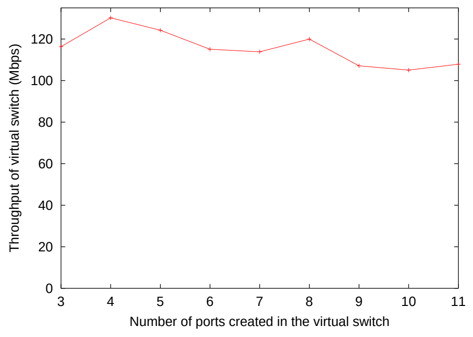


**Figure 7: Virtual switch throughput under different number of ports**

### 3.3.2 Virtual Link/NIC

A virtual link is created when establishing a connection with a virtual switch (or hub). For implementation convenience, we shift the responsibility of emulating link characteristics to virtual hub/switch.

vBET achieves the flexibility of on-demand addition or removal of virtual NIC, even when the virtual machine is active. Since NICs are virtualized, it is possible to dynamically create, configure, or delete a virtual NIC. When a virtual machine is started, it will also act as a server, which opens another domain socket (which can be a UNIX domain socket or a UDP or TCP socket) for configuration requests. When a new configuration request for adding another virtual NIC arrives, the virtual machine, after proper authentication, will configure the virtual machine kernel to accommodate the new virtual NIC by initializing and registering the corresponding device data structures and function pointers. After the virtual NIC creation, a user can configure the new network interface using some user level configuration tools such as *ifconfig* or *iproute2*. Such flexibility brings *non-stop* reconfigurability to vBET, which means that a user will not have to terminate and restart the entire experiment for reconfiguration.

In a multiple-vBET-server environment, a new challenge is to deal with the non-uniform delay on different virtual links between the same virtual hub and different vBET servers. Such a situation leads to the *testbed mapping problem* addressed in [23].

### 3.3.3 Virtual Router/Firewall

Routers perform packet forwarding, while firewalls perform packet filtering and content inspection. Via the small footprint file system (Section 3.4), it is easy to configure a virtual node, so that it possesses the specific functions to serve as a router or a firewall.

Current UML kernel (Linux 2.4.19 in vBET), especially the network subsystem, provides a convenient basis for further customization of router and firewall features. The distributed firewall experiment (Section 4.2) illustrates one example. In addition, UML provides an authentic */proc* environment and supports the *QUEUE* target for *iptables*. The *QUEUE* target in *iptables* is similar to *divert* mechanism in *ipfw*, which delivers packets to a userspace (with respect to guest OS) interface where they can be collected, modified, and inspected. *Snort-inline* [1] is one such application which inspects packets in real-time for DoS attack analysis.

## 3.4 Root File System

Upon finishing the boot sequence, the virtual machine kernel attempts to locate and mount a root file system. In order to achieve scalability with respect to the number of virtual machines in one vBET server, the root file system needs to have a small memory footprint, and it should be properly tailored by excluding the unnecessary services. In the vBET prototype, we impose a space size of 32MB for a root file system (with type *ext2*), which is reasonably small yet sufficient to include the basic services. With this small size, the root file system can be mounted in a *ramdisk* for better runtime performance.

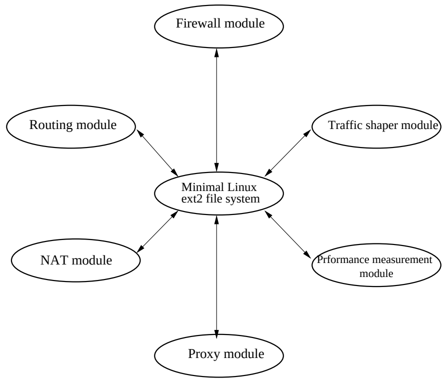


**Figure 8: Optional file system packages and the minimal Linux system**

Figure 8 illustrates six optional packages in the current vBET prototype. The *Firewall module* contains code that implements an Internet firewall subsystem, and it provides stateless or stateful packet filtering. The *NAT module* handles packet header translation and even packet payload mangling (such as for the FTP data connection) needed for network address translation. NAT can be integrated with a firewall or be used separately. The *routing module* contains code for routing protocols, including RIP, OSPF, and BGP taken from *zebra* [16] package. The *Proxy module* provides functionality of SOCKS proxy services. and *Traffic shaper module* can be employed to experiment with transmission rate monitoring and policing. *Performance measurement module* includes some performance measurement routines, such as *ttcp* [3]. All of the modules depend on the underlying *Minimal Linux file system*, which contains system-wide[4] basic configuration and daily routines and is thus required for the operation of every virtual node.

[4]Here, *system* means one virtual node.

The optional packages needed by a specific virtual node are included during the *virtual node creation* step. At the same time, a startup script is linked into */etc/rc.d/rc3.d*, which will automatically start all the required services. In vBET, it takes less than one second to start the virtual machine kernel, and another second to mount the root file system and start the customized set of services. As a result, a virtual machine in vBET can be started in as short as 2 seconds and be torn down in as short as one second.

## 3.5 Resource Isolation

To achieve performance isolation of each virtual machine and thus accuracy of vBET-based experiments, resource guarantee or isolation must be provided to each virtual machine. This is our on-going work and we have only initial results. In vBET, resource isolation is realized by extending the host OS (Linux).

- *CPU isolation* We have implemented a coarse-grain CPU proportional sharing scheduler in the host OS kernel . The scheduler enforces the CPU share allocated to each virtual node. The CPU sharing of a virtual node is decided during the virtual node creation. Within one virtual node, all processes bear the same user id, and host OS CPU scheduler performs resource consumption accounting and enforces resource usage limitation for virtual nodes based on their user ids.

  However, proportional sharing is not sufficient for CPU isolation. We are extending the CPU scheduler to support guaranteed CPU reservation and enforcement. Furthermore, we are refining vBET's virtual node creation process in order to predict the amount of CPU needed by each virtual node, based on the real-world capacities of entities in an experiment.

- *Network traffic isolation* We are implementing both *intra-server traffic isolation* and *inter-server traffic isolation*. Intra-server traffic is generated within one vBET server, and does not consume real-world network bandwidth. Inter-server traffic incurs real-world network traffic between vBET servers (in the multiple-vBET-server environment). For intra-server traffic, we provide each upstream virtual node with traffic shaping capability by including the *traffic shaper module* in its root file system. For inter-server traffic, a traffic shaper running inside the host OS enforces the bandwidth reservation between virtual nodes in different vBET servers.

- *Memory isolation* vBET leverages the *memory usage limit* feature of UML, which limits the amount of physical memory allocated to each virtual machine. Such limit (32MB in vBET) is enforced by the UML kernel. As a result, one vBET server with 2GB memory can support up to $2G/32M = 64$ virtual machines.

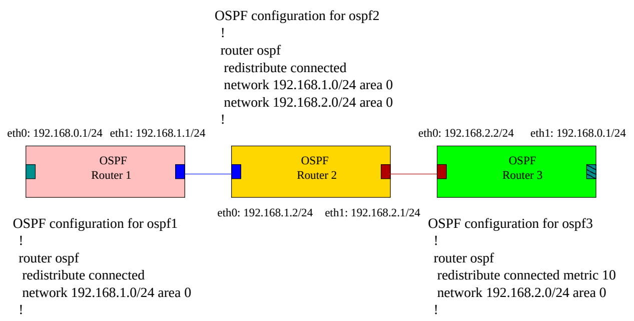
Disk isolation is another challenge for vBET. Disk access activities of different virtual machines, as well as the block cache management policy in the disk device driver may violate the performance isolation between virtual machines. vBET does not yet support disk isolation.

## 4. APPLICATION OF VBET

We have created a number of experimental environments to demonstrate the application of vBET. Three environments are presented in this section: The first experiment tests *routing flapping* in OSPF, which shows that an infrastructure-critical routing protocol can be deployed and customized in vBET virtual machines. The second experiment is the evaluation of *distributed firewall*. A star topology is created to enable the coordination of a set of distributed firewalls to protect a central server. The third experiment emulates application-level peer-to-peer lookup service, and subjects the service to different types of failures (such as peer failure and network partition) based on different network topologies.

### 4.1 Routing Flapping in OSPF

This experiment examines an interesting scenario involving the OSPF protocol, which demonstrates that baneful persistent route flaps may exist. The logical network topology for this experiment is shown in Figure 9. The corresponding topology modeling script is shown in Figure 10. To highlight the efficiency of vBET, we note that the whole system is bootstrapped within 6 seconds, and can be torn down within 4 seconds.



OSPF configuration for ospf2
```
!
router ospf
 redistribute connected
 network 192.168.1.0/24 area 0
 network 192.168.2.0/24 area 0
!
```

OSPF configuration for ospf1
```
!
router ospf
 redistribute connected
 network 192.168.1.0/24 area 0
!
```

OSPF configuration for ospf3
```
!
router ospf
 redistribute connected metric 10
 network 192.168.2.0/24 area 0
!
```

**Figure 9: Network topology for experiment with OSPF**

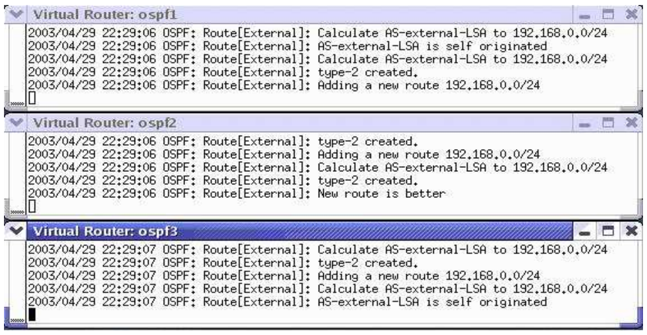
In Figure 9, there are three routers - $R_1$, $R_2$ and $R_3$, each of which is running *ospfd* from Zebra routing software [16]. Each of routers $R_1$ and $R_3$ intentionally has one interface configured to have the same IP address 192.168.0.1/24, and router $R_1$ advertises route entry for 192.168.0.1/32 with metric 20 (default value) and router $R_3$ advertises for same destination with smaller metric 10. Since router $R_3$ incurs lower cost to reach destination 192.168.0.1/24, router $R_2$ should choose $R_3$ for destination 192.168.0.1/24 when the network is stabilized. Detailed OSPF configuration and IP address assignment for each router are shown in Figure 9.

At the beginning, router $R_1$ and $R_2$ are bootstrapped, then $R_3$ is started. When routing tables are stabilized in all three routers, from the screenshot in Figure 11, we can see the router $R_2$ (i.e., ospf2 in Figure 11) adopts the route to 192.168.0.1/24 via router $R_3$ (i.e., ospf3 in Figure 11), which is correct since $R_3$ contains smaller metric for destination 192.168.0.1/24.

```
r1 = attach(alloc(router), alloc(NIC), alloc(NIC))
r2 = attach(alloc(router), alloc(NIC), alloc(NIC))
r3 = attach(alloc(router), alloc(NIC), alloc(NIC))

link(NIC(r1,2), NIC(r2, 1))
link(NIC(r2,2), NIC(r3, 1))
```

**Figure 10: Network topology modeling script for the OSPF experiment**



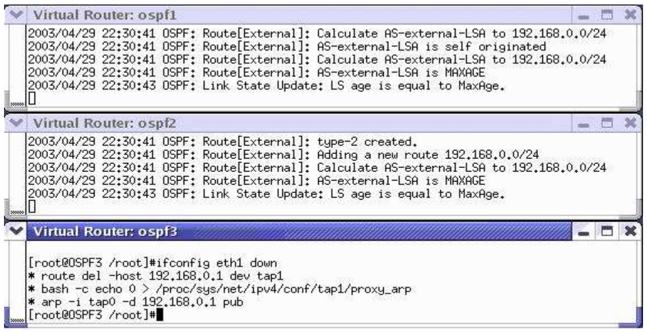
```
2003/04/29 22:29:06 OSPF: Route[External]: Calculate AS-external-LSA to 192.168.0.0/24
2003/04/29 22:29:06 OSPF: Route[External]: AS-external-LSA is self originated
2003/04/29 22:29:06 OSPF: Route[External]: Calculate AS-external-LSA to 192.168.0.0/24
2003/04/29 22:29:06 OSPF: Route[External]: type-2 created.
2003/04/29 22:29:06 OSPF: Route[External]: Adding a new route 192.168.0.0/24
```

```
2003/04/29 22:29:06 OSPF: Route[External]: type-2 created.
2003/04/29 22:29:06 OSPF: Route[External]: Adding a new route 192.168.0.0/24
2003/04/29 22:29:06 OSPF: Route[External]: Calculate AS-external-LSA to 192.168.0.0/24
2003/04/29 22:29:06 OSPF: Route[External]: type-2 created.
2003/04/29 22:29:06 OSPF: Route[External]: New route is better
```

```
2003/04/29 22:29:07 OSPF: Route[External]: Calculate AS-external-LSA to 192.168.0.0/24
2003/04/29 22:29:07 OSPF: Route[External]: type-2 created.
2003/04/29 22:29:07 OSPF: Route[External]: Adding a new route 192.168.0.0/24
2003/04/29 22:29:07 OSPF: Route[External]: Calculate AS-external-LSA to 192.168.0.0/24
2003/04/29 22:29:07 OSPF: Route[External]: AS-external-LSA is self originated
```

**Figure 11: Screenshot of OSPF experiment when all routers are working**

When we intentionally disable interface 192.168.0.1/24 in router $R_3$, $R_3$ notifies $R_2$ that the corresponding link state age has reached $MaxAge$, and thus is considered not usable anymore. As a result, a more expensive route (with metric 20) to 192.168.0.1/24 via router $R_1$ (ospf1 in Figure 12) is adopted by router $R_2$.



```
2003/04/29 22:30:41 OSPF: Route[External]: Calculate AS-external-LSA to 192.168.0.0/24
2003/04/29 22:30:41 OSPF: Route[External]: AS-external-LSA is self originated
2003/04/29 22:30:41 OSPF: Route[External]: Calculate AS-external-LSA to 192.168.0.0/24
2003/04/29 22:30:41 OSPF: Route[External]: AS-external-LSA is MAXAGE
2003/04/29 22:30:43 OSPF: Link State Update: LS age is equal to MaxAge.
```

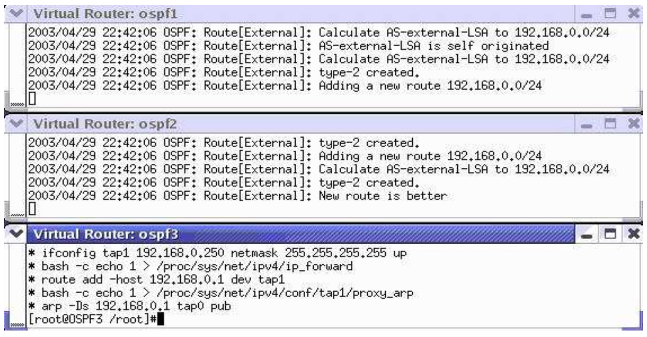
```
2003/04/29 22:30:41 OSPF: Route[External]: type-2 created.
2003/04/29 22:30:41 OSPF: Route[External]: Adding a new route 192.168.0.0/24
2003/04/29 22:30:41 OSPF: Route[External]: Calculate AS-external-LSA to 192.168.0.0/24
2003/04/29 22:30:41 OSPF: Route[External]: AS-external-LSA is MAXAGE
2003/04/29 22:30:43 OSPF: Link State Update: LS age is equal to MaxAge.
```

```
[root@OSPF3 /root]#ifconfig eth1 down
* route del -host 192.168.0.1 dev tap1
* bash -c echo 0 > /proc/sys/net/ipv4/conf/tap1/proxy_arp
* arp -i tap0 -d 192.168.0.1 pub
[root@OSPF3 /root]#
```

**Figure 12: Screenshot of OSPF experiment when the better route to 192.168.0.1/24 is down**

Then, we re-enable interface 192.168.0.1/24 at router $R_3$ to examine the routing flapping effect in OSPF. Figure 13 shows that router $R_2$ has found the new route via router $R_3$ is better than current route via router $R_1$ for destination 192.168.0.1/24. As a result, better route for the destination is updated in router $R_2$.

After several rounds of enabling and disabling interface 192.168.0.1/24 at router $R_3$, persistent route flapping effect is clearly exhibited. Route flapping is a baneful phenomenon and needs to be eliminated for more stable and robust networks. Also it suggests that more advanced OSPF route flap damping features should be introduced and that the OSPF protocol performance may be improved by learning from the



```
2003/04/29 22:42:06 OSPF: Route[External]: Calculate AS-external-LSA to 192.168.0.0/24
2003/04/29 22:42:06 OSPF: Route[External]: AS-external-LSA is self originated
2003/04/29 22:42:06 OSPF: Route[External]: Calculate AS-external-LSA to 192.168.0.0/24
2003/04/29 22:42:06 OSPF: Route[External]: type-2 created.
2003/04/29 22:42:06 OSPF: Route[External]: Adding a new route 192.168.0.0/24
```

```
2003/04/29 22:42:06 OSPF: Route[External]: type-2 created.
2003/04/29 22:42:06 OSPF: Route[External]: Adding a new route 192.168.0.0/24
2003/04/29 22:42:06 OSPF: Route[External]: Calculate AS-external-LSA to 192.168.0.0/24
2003/04/29 22:42:06 OSPF: Route[External]: type-2 created.
2003/04/29 22:42:06 OSPF: Route[External]: New route is better
```

```
* ifconfig tap1 192.168.0.250 netmask 255.255.255.255 up
* bash -c echo 1 > /proc/sys/net/ipv4/ip_forward
* route add -host 192.168.0.1 dev tap1
* bash -c echo 1 > /proc/sys/net/ipv4/conf/tap1/proxy_arp
* arp -Ds 192.168.0.1 tap0 pub
[root@OSPF3 /root]#
```

**Figure 13: Screenshot of OSPF experiment when better route to 192.168.0.1/24 is up again**

routing flapping history and being lazy and conservative in adopting new or better route entries. The trace logs as well as a video clip showing the experiment can be accessed at http://www.cs.purdue.edu/homes/jiangx/vBET.

## 4.2 Distributed Firewall

This experiment enables the creation of a topology shown in Figure 14, in which several network connections are established for the provision of a web service. A set of *edge firewalls* examining incoming requests for the service need to be coordinated to ensure that the total request load for the service does not exceed the server's capability. The whole system is bootstrapped within 13 seconds, and can be torn down within 7 seconds.

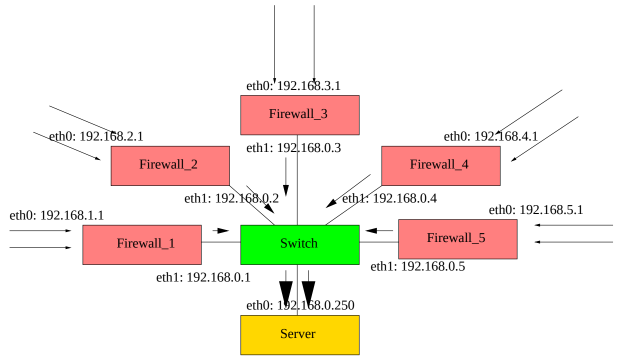


**Figure 14: A distributed firewall environment**

The virtual network topology is modeled by the script in Figure 15.

```
server = attach(alloc(host), alloc(NIC))
fw1 = attach(alloc(firewall), alloc(NIC), ...)
fw2 = attach(alloc(firewall), alloc(NIC), ...)
fw3 = attach(alloc(firewall), alloc(NIC), ...)
fw4 = attach(alloc(firewall), alloc(NIC), ...)
fw5 = attach(alloc(firewall), alloc(NIC), ...)

link(alloc(switch), server, NIC(fw1, 1), NIC(fw2,1), NIC(fw3,1), NIC(fw4,1), NIC(fw5,1))
```

**Figure 15: Network topology modeling script for the distributed firewall environment**

We create and compare two simple scenarios to demonstrate the effectiveness of distributed firewall. In the first scenario, every firewall forwards requests to the central server without any limitation, similar to a DDoS attack on the server. In the second scenario, every firewall restricts the traffic toward the central server in order to prevent the DDoS attack proactively.

We measure the in-bound traffic rate observed by the server in both scenarios. In the first scenario, the server receives traffic at a rate of 84.66Mbps. In the second scenario, traffic destined to the server is limited to 640kbps at each firewall with *tc* command and as a result, the server only receives traffic at a rate of 2.8Mbps. The individual traffic rates via the five firewalls are shown in Table 1.

| *Firewall* | *Amount of traffic* |
|---|---|
| FW1 | 578.8kbps |
| FW2 | 579.9kbps |
| FW3 | 579.4kbps |
| FW4 | 579.9kbps |
| FW5 | 580.0kbps |

**Table 1: Inbound data (web service requests) rate regulated by each firewall**

Though conceptually simple and straightforward, such experiment is difficult, if not impossible, to setup and experiment with in real world without a dedicated testbed.

## 4.3 P2P Network

Finally, we create Chord, a peer-to-peer overlay lookup service using vBET. The peer-to-peer network is deployed over nodes in a network topology depicted in Figure 16. The entire network is bootstrapped within 2 minutes, and can be torn down within 1 minute.

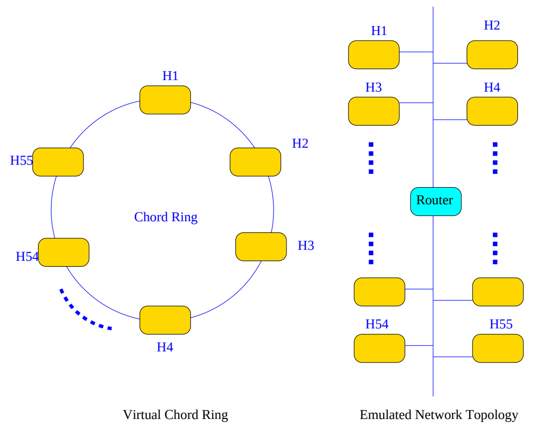


**Figure 16: A Chord P2P overlay network with 55 peers**

In Figure 16, there are 55 Chord peers, which are instantiated around the Chord ring. One Chord node assumes the responsibility as bootstrapping node. The 55 chord nodes resides in two LANs, which are connected by a virtual router. Such topology is useful to demonstrate and validate the reaction of Chord to different types of failures such as network partition.

More complicated topologies can be created to evaluate different aspects of Chord-based peer-to-peer services, such as CFS [9] - a peer-to-peer storage service based on Chord. Our purpose here is to show vBET's flexibility of creating experimental network topology and its customizability for every node inside the topology.

## 5. RELATED WORK

There have been some previous efforts in investigating the use of emulation in the context of their specific research [14, 15, 20]. Recently, several general-purpose emulation testbeds have been proposed and deployed [5, 21, 27, 28]. vBET shares the same goal of providing high-fidelity emulation or real environment for the experimentation with distributed systems and network protocols.

PlanetLab [21] is in the process of deploying hundreds of nodes across the Internet to create Point of Presence (or PoP), so that wide-area distributed systems can be deployed and evaluated using real Internet traffic. Though extremely valuable because of real deployment, the experiment environment is difficult for an individual researcher to gain full control. And core routers are not allowed to run customized software. As a result, it may not be practical to experiment with infrastructure-critical systems and protocols, such as distributed routing and distributed firewall[5]. vBET complements PlanetLab: the former can be installed in a PoP of the latter, so that experiments with infrastructure-critical systems can be carried out locally.

Netbed [28] allows users to configure a subset of network resources for isolated distributed systems and networking experiments. It provides an integrated environment that allows users to set up target operating systems and network configurations. Again, vBET can be integrated with Netbed, especially by contributing the scalability (with respect to the number of virtual nodes within one physical machine) and flexibility (of topology setup) features to the latter.

ModelNet [27] targets scalable and flexible emulation environment, which can perform full hop-by-hop network emulation. The basic mechanism for scalability is the *pipe* technique from *dummynet* [24]. As a result, it is difficult for ModelNet to support *traceroute*-like applications. One important property of ModelNet is that it balances scalability and efficiency by employing both emulation and simulation techniques. vBET, on the other hand, provides complete control and customization over network-level entities, so that third-party software for routing, advanced stateful packet filtering and other network services can be executed by the network-level entities.

Umlsim[5] is another recent effort parallel to ours which exploits the virtual machine technology to create simulation/emulation platforms. Instead of focusing on a specific application or protocol (such as TCP), vBET aims at providing a general-purpose testbed for different applications and protocols.

[5]PlanetLab is able to emulate routers - at the overlay level.

## 6. CONCLUSION

Emulation testbeds are expected to be easily and widely deployable. In an emulation testbed, arbitrary network topology can be created for different experimental systems and protocols. Furthermore, it is desirable to support customization of network-level entities, such as fault injection, parameter configuration, and modification or replacement of key software component such as the routing module. In this paper, we present the design and implementation of vBET, as our initial efforts towards meeting these goals. The salient features of vBET include easy deployment, customizability, and efficiency. Our experiments with different network and distributed systems demonstrate the versatility of vBET. Furthermore, vBET complements existing emulation or real-world testbeds and can be readily integrated into the existing systems.

The current version of vBET has limitation in the scale and quantitative accuracy of experiments. To overcome the limitation, we are enhancing vBET to support resource usage prediction and guaranteed reservation. We will also extend vBET to a multiple-vBET-server environment based on existing solutions to the testbed mapping problem, so that better scalability and resource utilization can be achieved.

## Acknowledgments

We would like to thank the anonymous reviewers for their highly constructive and detailed comments. We also thank Weichao Wang and Minseok Kwon for helpful discussions. This work was supported in part by a grant from the e-Enterprise Center at Discovery Park, Purdue University.

## 7. REFERENCES

[1] Snort-inline. *http://www.snort.org/*.

[2] The Network Simulator ns-2 . *http://www.isi.edu/nsnam/ns/*.

[3] ttcp. *ftp://ftp1.sunet.se/pub/network/monitoring/ttcp*.

[4] VMWare. *http://www.vmware.com*.

[5] W. Almesberger. UML simulator. *http://www.almesberger.net/umlsim/*.

[6] D. G. Andersen, H. Balakrishnan, M. F. Kaashoek, and R. Morris. Resilient Overlay Networks. *Proc. 18th ACM SOSP, Banff, Canada*, Oct. 2001.

[7] R. Braynard, D. Kostic, A. Rodriguez, J. Chase, and A. Vahdat. Opus: an Overlay Peer Utility Service. *Proceedings of the 5th International Conference on Open Architectures and Network Programming (OPENARCH)*, June 2002.

[8] A. Chervenak, I. Foster, C. S. C. Kesselman, and S. Tuecke. The Data Grid: Towards an Architecture for the Distributed Management and Analysis of Large Scientific Data Sets. *Proceedings NetStore'99*, Oct. 1999.

[9] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. *Proceedings of the 18th ACM Symposium on Operating Systems Principles(SOSP'1)*, Oct. 2001.

[10] J. Dike. User Mode Linux. *http://user-mode-linux.sourceforge.net*.

[11] P. Druschel and A. Rowstron. PAST: A large-scale, persistent peer-to-peer storage utility. *HotOS VIII, Schoss Elmau, Germany*, May 2001.

[12] I. Foster, C. Kesselman, J. Nick, and S. Tuecke. The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration. *Open Grid Service Infrastructure WG, Global Grid Forum*, June 2002.

[13] E. Gagnon and L. Hendren. SableVM: A Research Framework for the Efficient Execution of Java Bytecode. *Java Virtual Machine Research and Technology Symposium (JVM '01)*, Apr. 2001.

[14] G.Banga, J. Mogul, and P. Druschel. A scalable and explicit event delivery mechanism for UNIX. *Proceedings of the USENIX 1999 Annual Technical Conference, Monterey, CA*, June 1999.

[15] H.Yu and A. Vahdat. The Costs and Limits of Availability for Replicated Services. *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles (SOSP)*, Oct. 2001.

[16] K. Ishiguro. Zebra. *http://www.zebra.org/*.

[17] X. Jiang and D. Xu. SODA: a Service-On-Demand Architecture for Application Service Hosting Utility Platforms . *Proceedings of The 12th IEEE International Symposium on High Performance Distributed Computing (HPDC-12), Seattle, WA*, June 2003.

[18] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. OceanStore: An Architecture for Global-Scale Persistent Storage. *Proceedings of the Ninth international Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2000)*, Nov. 2000.

[19] J. Moy. OSPF Version 2. *http://www.ietf.org/rfc/rfc2328.txt*, Apr. 1998.

[20] B. Noble, M. Satyanarayanan, G. Nguyen, and R. Katz. Trace-Based Mobile Network Emulation. *Proceedings of ACM SIGCOMM 1997, Cannes, France*, Sept. 1997.

[21] L. Peterson, T. Anderson, D. Culler, and T. Roscoe. A Blueprint for Introducing Disruptive Technology into the Internet. *Proceedings of ACM HotNets-I Workshop, Princeton, New Jersey, USA*, Oct. 2002.

[22] Y. Rekhter and T. Li. Border Gateway Protocol 4 (BGP-4). *http://www.ietf.org/rfc/rfc1771.txt*, Mar. 1995.

[23] R. Ricci, C. Alfeld, and J. Lepreau. A Solver for the Network Testbed Mapping Problem . *ACM SIGCOMM Computer Communication Review*, Apr. 2003.

[24] L. Rizzo. Dummynet and Forward Error Correction. *Proceedings of the USENIX Annual Technical Conference*, June 1998.

[25] I. Stoica, S. Shenker, and H. Zhang. Core-Stateless Fair Queueing: Achieving Approximately Fair Bandwidth Allocations in High Speed Networks. *Proceedings of SIGCOMM'98*, Sept. 1998.

[26] V. Sundaram, A. Chandra, P. Goyal, and P. Shenoy. Application Performance in the QLinux Multimedia Operating System. *Proceedings of the Eighth ACM Conference on Multimedia*, Nov. 2000.

[27] A. Vahdat, K. Yocum, K. Walsh, P. Mahadevan, D. Kostic, J. Chase, and D. Becker. Scalability and Accuracy in a Large-Scale Network Emulator. *Proceedings of 5th Symposium on Operating Systems Design and Implementation (OSDI)*, Dec. 2002.

[28] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An Integrated Experimental Environment for Distributed Systems and Networks. *Proceedings of 5th Symposium on Operating Systems Design and Implementation (OSDI)*, Dec. 2002.