The ENTRAPID Protocol Development Environment

X.W. Huang, R. Sharma, and S. Keshav
Cornell Network Research Group
Department of Computer Science,
Cornell University, Ithaca, NY 14853
{xwh, sharma, skeshav}@cs.cornell.edu

Abstract

As Internet services rapidly become an essential part of the global infrastructure, it is necessary for the protocols underlying critical Internet services to be robust and fail-safe. To achieve this goal, protocol developers should be able to design, implement, simulate, visualize, and validate their work in a *protocol development environment* before deployment in the field. In this paper we describe the ENTRAPID protocol development environment, give examples of its use, outline its implementation, and present a preliminary performance evaluation.

1. Introduction

A *protocol* is a set of rules and formats that governs communication between communicating peers [Zimmerman 80]. It provides the abstraction of a *service* to higher-level protocols and ultimately to network users. Developing correct, efficient, scaleable, and robust protocols greatly challenges protocol designers. They can be helped in this quest by a *protocol development environment (PDE).* This controlled environment allows developers to implement, simulate, visualize, and verify their work before deployment in the field.

We motivate the need for a good protocol development environment from the following thought experiment. Consider a researcher who wants to study the performance effect of various modifications to TCP. In the absence of a good PDE, the researcher must modify the source code for TCP in an OS kernel, then painstakingly reboot the machine to test different protocol features, or each time an implementation bug is discovered. The situation is far worse if the protocol being developed (such as an in-kernel multicast routing protocol) must run on multiple machines. Such a protocol would need multiple machines to be rebooted to test each version of the implementation code. With an appropriate PDE, however, the researcher could develop the code entirely in user space, without the need to modify the kernel. Moreover, protocols that span multiple machines could be easily tested. Of course, to be useful, the code developed in the PDE should be easily portable to an actual kernel. The environment should also accurately emulate the services available within an OS kernel, such as device interface routines and system timers. These top-level goals motivate our design. In this paper, we describe the ENTRAPID protocol development environment, giving examples of its use, and outline its application to a broad range of networking problems.

Two broad trends make research into protocol development environments particularly timely. The first is the exponential growth in the number of Internet endpoints, which requires a similar growth in the backbone. This is good news for router manufacturers, and, not suprisingly, has led to the creation of many router startups. However, these startups are learning that although hardware for rapid packet forwarding is by no means easy to develop, its complexity pales in comparison to that of router software. Routers must support a wide variety of protocols, such as RIP, OSPF, BGP, and DVMRP, each of which requires a major implementation effort. Moreover, router software must be carefully written and tested for correctness in face of failures both in the local router in distant routers. Lacking this, a router failure somewhere in the Internet, which results in the generation of spurious routing packets, could potentially cause a complete collapse of the Internet, as router after router fails [Perlman 83]. This scenario appears to be far-fetched, but was precisely the one that brought down AT&T's telephone network for the entire day in March 1991. One cannot overemphasize the need for router software stability: the cost of a mistake can be catastrophically high. However, the only way to test router software is to build small networks of routers in a lab and walk through some test scenarios. This testing is necessarily limited by the need for each test engineer to

completely control his or her own network testbed. We believe that a suitable protocol development environment, such as the one described in this paper, makes the task much easier. With ENTRAPID, engineers can embed their routing software in a software-only test network, making it easy to generate and test a variety of failure cases.

The second trend that motivates our work is the proliferation of new services in the Internet. Moreover, these services, such as stock trading, weather information, audio broadcast, and electronic commerce are far more complex than the original services of telnet, FTP, and email. Three recent developments reinforce this trend:

- Partners in the Open Signaling initiative [Opensig 98], under the auspices of an IEEE subcommittee [IEEE-PIN 98], are successfully pressing router vendors and switch manufacturers to support an open programming platform for creation of third-party services.
- Recent improvements in flow matching algorithms allow flows to be identified, and flow state to be looked up, at line speed [LS 98, SVSW 98]. Thus, future routers and switches may allow users to customize data handling on a per-flow basis.
- Third, while the Internet's open architecture has always been conducive to the creation of new services, now telephone operators too are opening up their service infrastructure, allowing third parties to develop customized services on a shared public infrastructure. Environments such as AT&T's Geoplex [Geoplex 98] and MCI's Vault [MCI 98] allow creation of services that span the telephone and the Internet, resulting in a similar proliferation of services and a similar need for protocol development environments.

These developments indicate that not only will future networks provide more services to customers, customers will potentially create specialized services for their own purposes. However, we do not yet understand how to analytically model large systems of interacting protocols. Implementation details and quirks in protocol handling code, even at lower layers of the protocol stack, can heavily influence the behavior of such systems, particularly under failure. A protocol development environment that allows exact emulation of protocols and networking subsystems is invaluable in the implementation and debugging of the protocols underlying complex services.

To sum up, we believe that increases in the number of router vendors and in the number and complexity of network services make protocol development environments increasingly valuable.

2. Requirements

We believe, based on a decade of experience in building and using network simulators, that an ideal PDE should have the following characteristics, in order of decreasing importance:

- Ease of use
- Exact emulation
- Controllability
- Visualization
- Extensibility
- Scalability
- Verification

2.1 Ease of use

An ideal PDE should be easy to use. In particular, it should allow developers to implement, modify, and test protocols normally resident in kernel space (such as TCP and IP) entirely in user space. It should allow developers to intuitively specify large test topologies and their associated workloads. It should also allow developers to easily select probe points.

2.2 Exact emulation

To allow rapid development, it should be easy for protocol developers to move code from the PDE to the Internet and *vice versa*. This imposes two subsidiary requirements. First, the PDE should support an Application Programmer Interface (API) that is identical to APIs commonly used on the Internet (typically Berkeley sockets and Winsock32). Second, PDE components that interact with the protocol under test should behave exactly the same as their counterparts in the Internet. The first requirement is relatively simple. The second requirement, however, is both subtle and difficult to implement. It requires, for example, that an application should experience the same packet loss, flow control, routing, and link outages as it would were it running on the Internet. In this sense, the PDE should be 'transparent' to the application developer. Although no practical PDE can achieve exact emulation, we believe that a PDE should be judged by the degree of emulation it can achieve.

2.3 Controllability

The PDE should allow a developer to set up complex network scenarios. In particular, it should allow developers to model an existing configuration, such as the one in a campus Intranet or an ISP backbone. Moreover, developers should be allowed to induce controlled errors, such as packet losses, packet corruption, line failures, and routing protocol corruption, to stress the protocol under test.

2.4 Visualization

Protocols tend to be hard to design, and inexperienced developers have difficulty understanding them, and especially their interactions. We believe that good visualization can play a key role in developing correct and efficient protocols. For instance, HTTP 1.0 opens one TCP connection per inline image. This is extremely inefficient, and it took a major effort to change HTTP, in version 1.1, to use a single connection to fetch multiple images. We conjecture that one reason for this design flaw was that HTTP designers did not realize the performance penalty paid by TCP's slow start algorithm. With a good visualization tool, the performance impact of this design decision would be immediately apparent.

2.5 Extensibility

From our experience, we have found that developers tend to highly customize their development environment. This customization helps them to quickly solve routine tasks. Besides allowing customization of the user interface, an ideal PDE should allow developers to add protocol components as required. For example, if a developer is interested in studying new multicast routing protocols that interoperate with existing multicast forwarding algorithms, the PDE should allow the developer to quickly port an existing multicast forwarding algorithm from the Internet into the PDE. The PDE should also allow developers to add new link types, such as wireless links, or even new network types, such as the telephone network, cable modem networks, and satellite networks.

2.6 Scalability

Some protocol design problems show up only in large networks. These scaling problems are often the most insidious ones, and designing scaleable protocols is almost always a matter of instinct and good judgement rather than scientific design. Good judgement, however, is a rare commodity, so we would like an ideal PDE to scale to large networks, so that scaling problems can be identified in a controlled setting.

2.7 Verification

Where possible, the PDE should allow developers to verify that their protocol does not suffer from obvious problems such as deadlock and livelock. Thus, the PDE should incorporate formal verification tools, such as those described in References [Holzmann 98, SECH 98].

3. State of the art

Protocol developers in the Internet today can choose from one of four options: (a) develop directly on the Internet, (b) use kernel extensions, (c) use a network simulator, and (d) use a router-oriented simulator. We describe these options in greater detail next.

3.1 Protocol development directly on the Internet

A developer can implement and test a protocol directly on the Internet. This is acceptable for services and protocols that do not modify the transport layer protocol (TCP) or below. For example, it is an acceptable alternative for protocols layered above HTTP. For such protocols, developing on the Internet provides ease of use, exact emulation and extensibility. However, there is little or no support for controllability (for example, changing the number of clients or servers dynamically), visualization, scalability (for example, adding a large number of clients), or verification. Thus, even in this limited environment, direct development on the Internet is not easy.

Things are harder when a protocol tries to modify or exploit the details of TCP, IP, or the MAC layer. For these types of protocols, such as the load distribution protocol in a cluster-based server [BFHR 97], wireless snoop protocols [BSK 95], or various extensions to TCP, direct development on the Internet requires extensive kernel modifications. Such modifications are not only complex, they are also non-portable and require specialized knowledge of the kernel environment. This difficulty is reflected both in the paucity of such services, and with the frequency with which implementation bugs are detected in such services (practically every TCP implementation, even after years of experience, seems to be buggy [Paxson 97]!). For such protocols, the Internet protocol development environment offers exact emulation and little else.

3.2 Kernel extensions

A second approach to protocol development is to insert ‘hooks’ into a kernel and expose these hooks in user space, so that the kernel-resident protocol behavior can be customized at the user level. This general idea has been exploited in a number of systems including U-Net [VBBV 95], the USC TCP-Vegas testbed [ADLY 95], and the NIST emulator [NIST 98]. The key benefit of the approach is that it allows protocol developers who want to modify or exploit TCP, IP, or a MAC protocol the same ease of development as protocol developers dealing with higher layer protocols. There is some loss of emulation, because the exact timing of events is lost, but for most purposes the emulation is sufficiently accurate. The system is also extensible, since the same hooks can be used for a variety of purposes. However, this environment fails to meet our other criteria. First, the PDE is not controllable or scalable, since it does not allow developers to develop protocols that span multiple kernels: such protocols must be developed on two separate machines. Moreover, most of these environments have little or no support for visualization and verification.

Recently, developers at Torrent Networks have built an extension to FreeBSD that provides exact emulation, extensibility, controllability, and scalability [Torrent 97]. In their approach, a single FreeBSD kernel maintains multiple copies of the kernel’s networking state. While this still requires a protocol developer to deal with kernel-level debugging, protocols that span multiple kernels can be developed and tested on a single machine. This greatly eases the development of routing protocols, which, by their nature, span multiple machines.

3.3 Network simulators

A typical network simulator provides the programmer with the abstraction of multiple threads of control and lightweight inter-thread communication. Threads implement protocols described either by a finite-state machine, native C or C++ code, or a combination of the two. Simulator packages typically come with a set of pre-coded modules, with the ability to customize these modules or add new ones. Some network simulators provide extensive support for visualization and animation (such as the nam package used with ns [ns 98]). Examples of widely used network simulators include, in the public domain: ns [ns 98], VINT [VINT 98], REAL[Keshav 97], and MARS[ASDM 94], and commercially: OPNET[MIL3 98] and BONeS [Cadence 98].

Although network simulators are usually used to test protocol performance, they can also be used as protocol development environments. Given a sufficiently accurate emulation of a network and protocol stack, developers can leverage this controlled, reproducible environment to stress-test protocols using microbenchmarks. However, most current network simulators omit many details, thus losing exact emulation, to gain ease of use, controllability, extensibility, and scalability. Thus, transitioning 'real' code into a simulator is not trivial. For example, porting TCP into any network simulator package is hard, and it is reasonable to question how accurately a simulator's TCP implementation matches the behavior, of say, the implementation of TCP in the NetBSD kernel (which is the de facto industry standard). Thus, network simulators are very close to the ideal protocol development environment: their main failing is the lack of support for exact emulation. Some of the simulators listed above also do not scale beyond a few hundred nodes.

3.4 Router simulators

Router simulators are special-purpose network simulators that allow routing configurations to be tested before deployment and we include them in this survey only for the sake of completeness. Examples of such simulators are the NetSys simulator from Cisco [Cisco 98] and the MRT simulator from Merit [Merit 98]. We note that these simulators are not extensible: they are meant to test routing, and that is all they do. Thus, they are not useful for general purpose protocol development. On the other hand, they appear to be easy to use, provide an exact emulation, support visualization, and are scaleable.

The relative merits of the four approaches are compared in the table below. Note that no single approach satisfies all the criteria for an ideal protocol development environment.

	Develop directly on the Internet	Kernel extensions	General-purpose simulators	Router simulators
Examples		NIST emulator, Torrent simulator	ns, REAL, MARS, OPNET, BONeS	NetSys
Ease of use		*	*	*
Exact emulation	*	*		*
Controllability		* (Torrent)	*	*
Extensibility	*	*	*	
Visualization			*	*
Scalability		* (Torrent)	*	*
Verification				*

4. Design

Our design for the ENTRAPID protocol development environment can be viewed as combining the best features of the multi-kernel approach from Torrent and general-purpose network simulation. Figure 1 outlines the architecture of the system. At the top level, ENTRAPID can be viewed as a process that runs entirely in user space, and can interact both with other processes and with physical network interfaces. Its *switch box* component listens for commands and supplies status information. Developers connect to the switch box with a telnet connection to give configuration commands in a simple language (described in Section 5). The switch box also generates status and monitoring information for use by an external visualization engine.

The ENTRAPID process supports multiple *Virtualized Networking Kernels (VNKs)*. Each VNK (pronounced 'vink') exactly implements the networking services found in the 4.4 BSD kernel. Multiple *virtualized processes* can run on each VNK. Each virtualized process carries out a user-level protocol and redirects its system calls to the VNK. VNKs can be connected using *wires* that represent communication links. Examples of wires are Ethernet busses and point-to-point links. The final abstraction in ENTRAPID

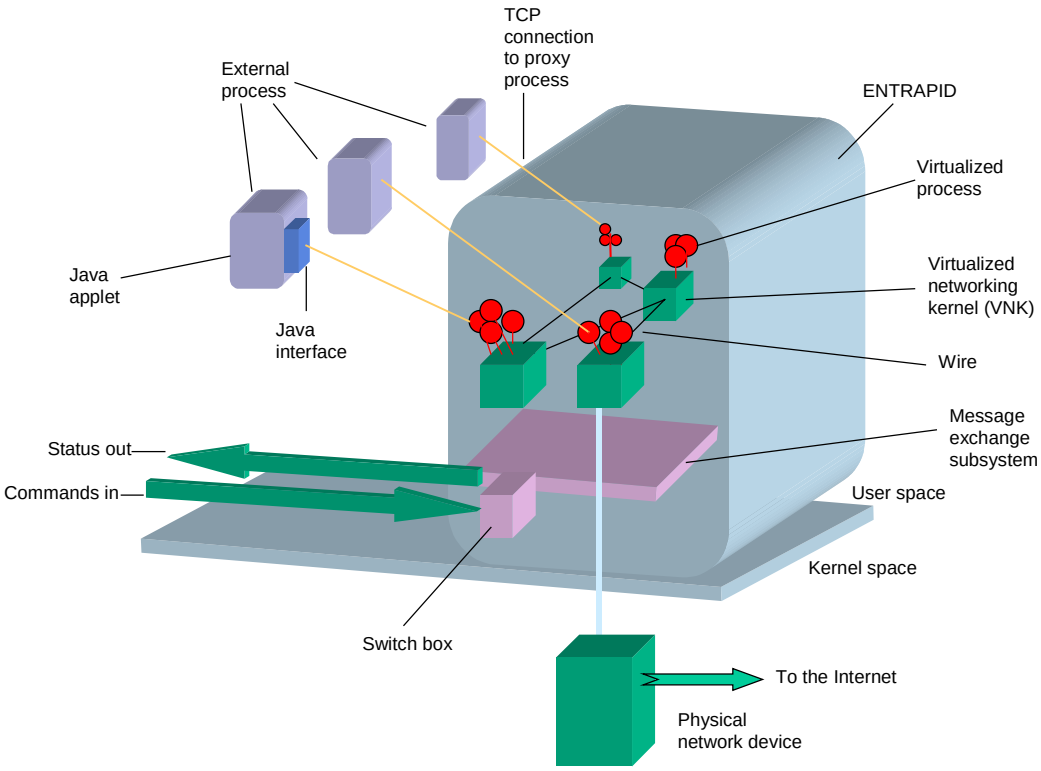

Figure 1: ENTRAPID architecture

is that of an *external process.* An external process is not virtualized, but is able to communicate both with virtualized processes and with other external processes by means of a proxy process.

From a developer's perspective, ENTRAPID provides the abstraction of a 'network in a box'. Each VNK corresponds to a machine on the Internet, and each virtualized process corresponds to a process running on that machine. Since ENTRAPID can support several thousand VNKs, developers can work with large topologies when developing and testing protocols. A developer can instantiate new protocols either directly on a VNK, or as an external process, and test its behavior when interacting with other network protocols already implemented within ENTRAPID. Note that because ENTRAPID is entirely in user space, a developer with access to the source code can monitor or modify *any* aspect of the entire protocol stack without having to make any changes to the kernel.

Much of ENTRAPID's power comes from our design of a VNK. As stated earlier, the VNK is derived from 4.4 BSD networking code. Applications built using the BSD socket API can be ported immediately to a VNK. More importantly, even applications that need to use non-standard lower-level APIs such as the `kvm_read` interface can be ported to ENTRAPID with no modification. This allows us to directly port common commands and protocols such as `mrouted`, `gated`, `routed`, `ping`, `netstat`, and `ifconfig` to ENTRAPID. Consequently, the set of commands used to configure one of the VNKs is identical to the set of commands used to configure an actual BSD machine.

While ENTRAPID is written in C++, it can interoperate with Java applets. A Java interface component links calls in the net class to a proxy virtualized process. This allows all Java applications to use the ENTRAPID infrastructure with no change.

Finally, ENTRAPID can directly control a physical network device through a VNK. *ENTRAPID therefore interfaces seamlessly with all Internet protocols at layer 3 and above*. For example, if we connect a machine running ENTRAPID to another, unmodified machine using an Ethernet hub, the unmodified machine cannot distinguish between packets forwarded among ENTRAPID VNKs and packets forwarded on the Internet. Thus, a program like `traceroute` can be used to find the path to a destination within a simulated network, and an HTTP server running within ENTRAPID can be used to serve web pages to a browser running on an external machine. We can also use this interface to link multiple ENTRAPID processes together, allowing us to emulate large topologies.

5. Example

Perhaps the best way to understand how ENTRAPID works is to step through a simple example. Suppose we wanted to simulate the exchange of messages between a TCP client and a TCP server that are running as virtualized processes. The client sends a stream of characters to the server, which prints them out on standard output.

The first step is to start ENTRAPID and tell the switch box which port to listen to, say port 15000:

```
ensim% ENTRAPID 15000
```

We now need to create two VNKs and a wire that links them. We first connect to the switch box using a standard telnet connection, then give it the appropriate commands:

```
ensim% telnet ensim 15000
switch>> new machine m1
switch>> new machine m2
switch>> new wire w1
```

To set up a machine, its network interfaces need to be created and configured with IP addresses, and the machine needs to be given a routing table. VNK interfaces are configured using the standard `ifconfig` command, and routes are set up using the standard `route` command, as shown below. We first connect to the VNK through the switch box, as shown next:

```
switch>> connect m1
```

Next, we add interface `vx0` to the machine and configure it:

```
m1>> addif vx0
m1>> ifconfig vx0 inet 128.84.254.1 netmask 255.255.255.0
```

We now connect this interface to the wire we previously created:

```
m1>> wireup m1 vx0 w1
```

Our next step is to start the TCP server application on m1:

```
m1>> echoserver&
m1>> exit
switch>>
```

Machine m2 is symmetrically configured, with IP address 128.84.254.2. Finally, we need to connect to the other machine and start the TCP client application. The client application is given the IP address of the server, and, using the standard socket library, a socket between the client and server is established.

```
switch>> connect m2
m2>> echoclient 128.84.254.1
```

From this point on, every character typed into machine m2 is sent, using TCP, to the echo server, which prints every data packet it receives.

As a second example, let us return to the researcher who wanted to study the effect of TCP modifications and see how ENTRAPID can be used for this purpose. Since ENTRAPID includes TCP source code, the researcher would start with this existing code and modify it, creating a new VNK type. ENTRAPID allows users to instantiate different VNK types using the `new` command and to link them up with the `wireup` command. Thus, arbitrary test topologies can be set up with simple shell scripts. The researcher can now test the features of the modified VNK. If there is a bug in the code, a standard debugging environment (capable of handling multiple threads of execution) will suffice to find and fix the bug. The researcher can also use the visualization features described in Section 6.5 to customize packet traces, animate these traces, and set and remove breakpoints on multiple VNKs. We believe that this approach to studying TCP modifications is far more efficient than traditional ones.

These examples point out several important aspects of ENTRAPID.

- Note that the set of commands needed to configure the network is identical in every way to the ones used in the real Internet. This reflects the fact that the underlying VNK exactly emulates a BSD kernel, down to the data structures accessed by the `ifconfig` and `route` commands.
- The source code for the TCP client and server are identical to what would be written in the Internet. This means that code developed in ENTRAPID can be ported, with no change, to the Internet, and *vice versa*.
- The `new machine` and `wireup` commands allow us to create arbitrary topologies of VNKs, mimicking arbitrary configurations. This makes the environment controllable.
- The switch box can monitor the actions of any VNK and report this to a logging process external to ENTRAPID. By separating the emulation and visualization engines, we can develop arbitrarily complex visualization methods without modifying the emulation engine. The communication link between the visualization GUI and the switch box allow it to make arbitrarily complex changes to a running emulation.

To sum up, ENTRAPID is easy to use (completely in user space), provides exact emulation, is controllable and extensible, and supports visualization.

6. Implementation

The core technologies underlying ENTRAPID are kernel virtualization, process virtualization, direct control of physical network devices, external process support, and visualization. We discuss each of these below.

6.1 Kernel virtualization

Virtualization is the combination of multiplexing and indirection to allow a physical resource to be shared among multiple entities without their knowing it [Keshav 97a, pp106]. For example, with virtual memory, programs share physical memory, but are never aware of the existence of other address spaces: as far as each program is concerned, it is the sole owner of the entire physical memory. The key to virtualization is the ability to trap every reference to a physical resource and map it through an appropriate indirection table to a managed partition. For instance, with virtual memory, every memory reference is mapped by a memory manager to a physical address in the range actually owned by the process.

Virtualizing a kernel or, more precisely, the networking portion of the kernel is accomplished by carefully extracting the networking code from the kernel, then determining every non-local reference. Each such reference is mapped through an indirection table to the appropriate portion of the shared resource. It turns out that the FreeBSD networking subsystem is closely tied together and makes only a few external

references, (primarily to the network device for I/O, to the scheduler for timed sleep events, and to user-level processes to read and write data streams). Thus, with some care, it is possible to virtualize the networking portion of a kernel with no change to its functionality. We caution that the actual process is not for the faint-hearted! It requires a deep understanding of kernel functionality and a series of interlocking decisions about which portions of the kernel should be virtualized, and which should be left alone.

In order to support multiple VNKs within a single process, we make heavy use of threads, which are available in most modern operating systems. The ENTRAPID process is associated with a pool of 'worker' threads that are dynamically assigned to VNKs. Requests for service by a VNK are translated to a task request that is registered with the ENTRAPID thread scheduler. At a future time, an available worker thread handles the request. In order to minimize race conditions, we ensure that only a single thread is within a particular VNK at any given moment. (If it should prove necessary, we can allow multiple threads within a VNK by remapping the `splhi` and `splx` calls in the virtualized code.)

One of the more troublesome aspects in kernel virtualization is dealing with interactions with the file system. BSD sockets and regular files allocate file descriptors from the same space. Since we do not wish to virtualize the entire file system, calls by a process on non-socket file descriptors must be passed to the actual kernel (suitably massaged, as described next), and calls to socket file descriptors should be passed to the associated VNK. We distinguish between socket and non-socket file descriptors using a hash table that is updated appropriately by the `socket` and `close` calls. We also associate each VNK with its own virtual 'root' in the actual file system. All calls to the file system that contain an absolute path are prepended with this root string. This allows multiple VNKs to cleanly share a common file system.

We note that although the current version of ENTRAPID virtualizes the networking portion of the FreeBSD protocol stack, we can use an identical approach to virtualize any kernel, or, more generally, the implementation of *any* API. To virtualize an implementation, we determine, for each call in the API, whether access is made to a shared resource. If it is, then the call is redirected, using a library, to an indirection routine that uses the virtual instance identifier to appropriately remap the call. So, for example, we can extend our approach to support a virtualized Windows NT kernel or a virtualized Solaris kernel. Thus, with our approach, we can leverage network protocol implementations in a variety of existing development environments. We can also emulate heterogeneous protocol development environments.

6.2 Process virtualization

Process virtualization allows us to run multiple copies of a program within a single ENTRAPID process. As with kernel virtualization, it requires modifications of the process source code to remap all accesses to shared resources. We have automated some of these modifications by providing a virtualization library that massages the most common system calls that access shared resources. Within the virtualization library, these system calls are mapped to messages that are relayed to the ENTRAPID task scheduler, which carries them out in due course. For instance, consider a virtualized process that makes the read system call on a socket file descriptor. Since the process is linked with our library, the read system call is remapped to a procedure that serializes the parameters of the system call, encapsulates the serialized stream in a message, and schedules the message for eventual handling by the ENTRAPID task scheduler. The task scheduler, on seeing the message, dispatches a worker thread to execute the read function in the appropriate VNK. If this read accesses a simulated interface, then data arriving on the simulated interface are available to the read with no further intervention. Otherwise, the read call is shepherded by another worker thread to the actual OS kernel for further handling. When the OS call succeeds, the reply is returned to the appropriate VNK, and eventually to the calling process.

6.3 External process support

While process virtualization is easy for some processes, it is much harder for those that make use of advanced system services such as `sysctl`, or those that are written as non-reentrant code (because a VNK cannot simultaneously execute multiple copies of non-reentrant code). In such cases, it turns out to be easier to run the process externally, and, instead, set up a connection between the external process and a proxy virtualized process running within ENTRAPID (see Figure 1). External processes have the added advantage that process state is external to the simulator, so implementation bugs are contained. We create

an external process by linking unmodified source code with a proxy library that converts networking and file system calls to messages sent to a proxy virtualized process (each external process is associated with its own proxy virtualized process). The proxy virtualized process simply decodes the message and executes the appropriate system call in the ENTRAPID context. For concreteness, consider a read system call made by an external process. When linked to the proxy library, the read call is converted to a read message that is sent, via the switch box, to the appropriate proxy virtualized process. The proxy decodes the message and carries out the read. The results of the read are then forwarded, via the switch box, to the external process. As far as the external process is concerned, it cannot distinguish between this read, and a read done on an actual FreeBSD kernel. We have also modified a standard Java virtual machine to run as an external process, so that ENTRAPID can support unmodified native Java bytecode.

6.4 Direct control of physical network devices

The ENTRAPID process can directly control a physical network device. This allows it to capture incoming IP packets on that interface and forward them to virtualized interfaces. VNKs can also create IP packets that are forwarded to the Internet. Thus, the network simulated within ENTRAPID becomes indistinguishable from the actual Internet. We can attach an unmodified machine to a machine running ENTRAPID with an Ethernet cable, then proceed to ping ENTRAPID nodes, or `traceroute` to internal nodes.

We implemented direct control in Windows NT by adding a custom NDIS shim to the operating system. It was relatively simple to add the shim because it did not need to deal with security issues. We also implemented a simple DNS name resolver as an external process. This allows external machines to transparently resolve internal ENTRAPID names to internal IP addresses.

6.5 Visualization

The ENTRAPID visualization environment provides:

- Hierarchical topology creation and network configuration
- Packet-trace generation and animation
- A graphical front end for simulation control
- Shared visualization environments

We now describe these features in more detail. A screen shot of the visualization tool is shown in Figure 2.

6.5.1 Topology creation and configuration

Simulating a network requires a developer to describe the set of machines being simulated, their interfaces, their default routes, and their interconnection. Moreover, each machine may need to be customized with parameters such as the TCP receive window size and the socket buffer size. Our visualization environment allows a developer to view these parameters at a glance. Nodes and links can be created using a graphical topology editor. The environment also allows subnets to be collapsed into a single node, thus presenting a hierarchical view of the topology. All configuration parameters are available as drop-down windows keyed off nodes and links.

6.5.2 Trace generation and animation

Each VNK supports several *trace points*: points in the VNK code where significant events happen. The default set of trace points are packet arrival, packet departure, and packet loss (additional code points can be added by a developer). On reaching a trace point, the VNK executes a *packet filter* on the packet header. If the filter declares a match, then an associated handler is executed. The handler can create a log event (the default) or carry out arbitrary protocol actions. This general mechanism allows a developer to create highly customized event logs, or even intervene in protocol actions to simulate events such as packet loss, packet corruption, or packet reordering. The ENTRAPID animator reads and animates trace logs. By default, the animator shows color-coded packets moving along network links. It also displays buffer contents at each router, color-coded according to user-specified filters. However, this behavior can be customized by a developer to show other events.

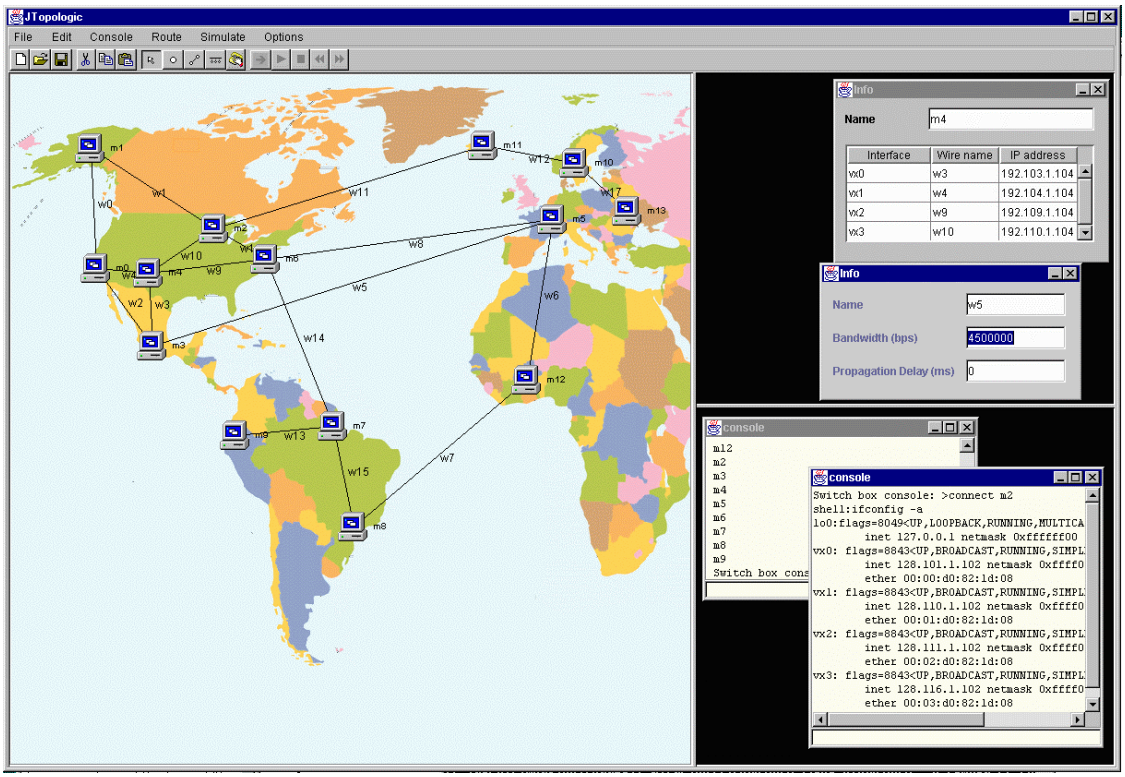

Figure 2: Screenshot of visualization GUI

6.5.3 GUI for simulation control

By *simulation control* we mean the ability for a developer to load and store network topologies; to stop and start simulations based on specific events; to graph a time series of values of a simulation variable; and to step through event sequences in order to debug them. In a sense, this extends the debugging metaphor of a tool like `dbx` to a network of machines. In addition to the capabilities listed above, the ENTRAPID visualization environment allows developers to insert breakpoints at any VNK trace point. A breakpoint can be asynchronous, so that other VNK threads are allowed to execute while a breakpoint is active; or it can be synchronous, so that all VNK threads are stopped when any breakpoint is active. Synchronous and asynchronous breakpoints, in conjunction with a rich set of trace points, make protocol debugging much easier.

6.5.4 Shared visualization

ENTRAPID provides a 'whiteboard' feature for teaching network operations and management. When placed in 'shared' mode, the actions taken by any of a set of simultaneously active front-ends is visible to all other front ends. In this way, an instructor can show students what commands to type to configure a network, and monitor what a student does. The combination of hands-on learning with instantaneous guidance from an instructor can greatly improve training of network managers.

7.Performance

In this section, we present a preliminary performance evaluation of our system. At this time, we have not optimized its performance in any way. These numbers, therefore, serve primarily as a baseline against which we will compare future improvements. They also give a sense for the overheads inherent in exact

emulation and virtualization. The performance results are of two kinds. The first type of results measure the overheads in using the simulator, instead of directly developing a protocol on the Internet. The second type of results examines the limits to scaling the simulator.

7.1 Overhead

Virtualization necessarily increases the time to make a system call. The table below compares the time to make a null system call in the Solaris and Windows NT operating system with the time for a virtualized and an external process to make a similar null system call to a VNK. Note that the external process's system call time includes the overhead in communicating to a virtualized proxy process over a TCP/IP socket.

	Null system call to a kernel by a normal process	**Null system call to VNK from a virtualized process (slowdown ratio)**	**Null system call to VNK from an external process (slowdown ratio)**
Solaris	3.6 us	221 us (61)	1870 us (519)
Windows NT	14 us	134 us (9.6)	1700 us (121)

These measurements were made on two different systems (both high-end PCs but with differing cache architectures and CPU speeds). Thus, the relevant number is the absolute cost of a system call from a virtualized or external process, and its ratio to 'normal' system call on the same operating system. Note that, despite the overhead of socket communication, the cost of an external system call is under 2 ms for both operating systems. Virtualization causes an order of magnitude degradation in the cost of a system call, and external process communication adds another order of magnitude overhead. While we believe that this degradation is an acceptable tradeoff, we intend to investigate techniques to reduce this ratio in future work.

The second test for overhead compares the throughput achieved between a TCP client and a TCP server that are trying to exchange data as fast as possible. The measurements were conducted on a 300 MHz Pentium II PC running Windows NT with 128 Mb main memory and 512 Kb on-chip cache for multiple runs of a 100-million byte transfer. The client sends data as 1024-byte packets and always has a packet to send, regulated only by TCP's window flow control mechanism. The results of this measurement are shown in the table below.

Configuration	**Throughput in Mbps**	**Ratio**
Client and server are regular NT processes on the same NT kernel	2.7	1.0
Client and server are virtual processes on the same VNK	2.1	0.78
Client and server are virtual processes on two VNKs connected by a wire	1.6	0.59
Client and server are virtual processes on two VNKs separated by two wires and a VNK acting as an IP router	1.0	0.37

We see that the degradation in going from two processes on the same NT kernel to two virtualized processes on a VNK is rather small. Although each system call is an order of magnitude costlier, the bulk of the work in the data transfer is in copying data from user space to kernel space, from kernel space to the device and the same process in reverse. Thus, the two virtualized processes achieve nearly 80% of the throughput between to two normal processes. However, as we increase the number of active kernels and start simulating wires and routers, the overall throughput decreases. With a single wire connecting to VNKs, the throughput drops to a little under 60%. This is because we must now simulate not only two entire VNKs, but copy packets from the VNK to the wire and out again. Not suprisingly, adding a router VNK adds two more copies, and brings the throughput down to just about a third of the original rate. The

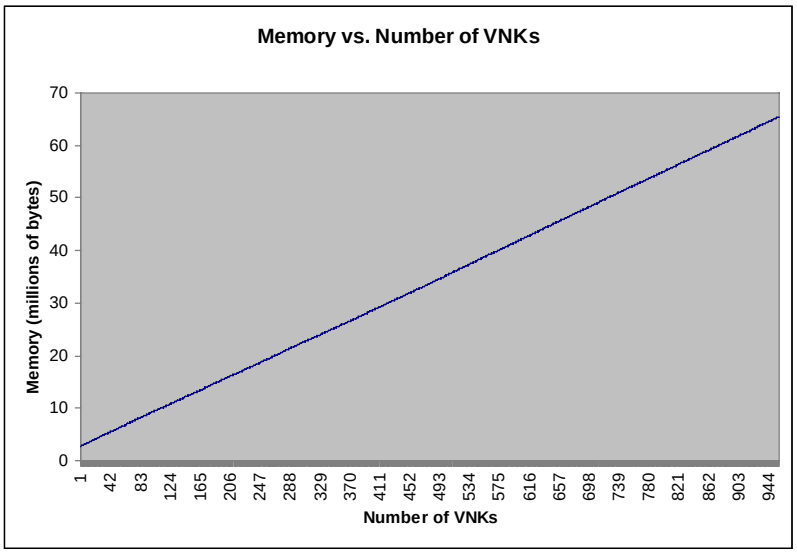

Figure 3: Memory scaling as a function of number of VNKs

main lesson here is that data copying is expensive. We propose to use well-known copy avoidance techniques to deal with this problem [EM 95, VBBV 95, TNML 93].

7.2 Scaling

It is hard to quantitatively measure the scalability of a protocol development environment. The number of VNKs that can be supported is a function not only of the protocols run at each VNK, but also the number of messages exchanged, and the degree to which the working set of pages fits in the processor's memory hierarchy. Here, we present a preliminary attempt to characterize the scaling properties of our system.

Figure 3 plots the size of the ENTRAPID process as a function of the number of VNKs. Each VNK is supporting zero virtualized processes, so this represents the most optimistic scaling possible. Note that the

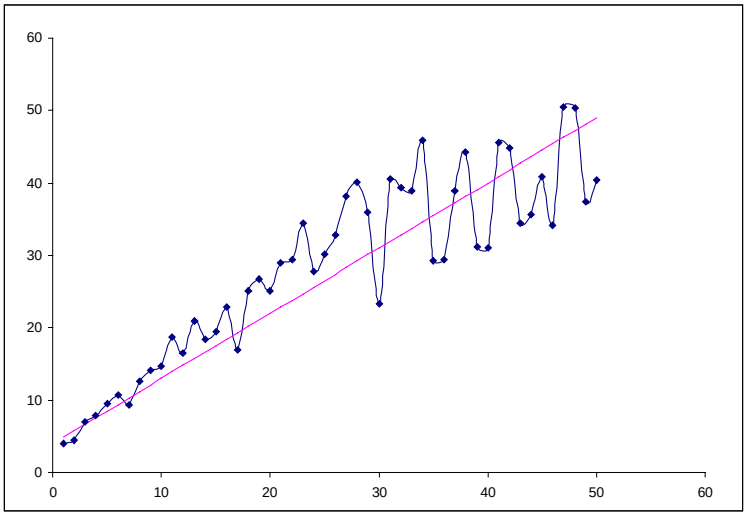

Figure 4: Ping time as a function of hop count

system allows up to nearly a thousand VNKs. The memory required by each additional VNK, after the initial few, is about 60 Kbytes.

In practice, we do not expect the limit on scaling to be from the memory size required for the process, but from the CPU time required to emulate several hundred VNKs and associated virtualized and external processes. We attempt to capture this overhead with the following experiment. We set up a linear network topology, where each VNK has either one or two neighbors, and all the VNKs are in a row. We send ping packets from an external process to each of the VNKs in the topology and compute the mean time to ping each VNK. Figure 4 shows the mean time taken to ping a VNK from an external process as a function of the number of hops needed to reach that VNK. We see that the ping time increases linearly with VNK distance (the straight line in the plot is the trend curve and the variations in the ping time are due to the large context switch times in the Windows NT kernel). Since VNKs not on the path do not consume any CPU, this leads us to believe that the time to simulate a large topology will scale linearly both with the number of nodes and with the number of packets exchanged in the network.

To sum up, we have shown that ENTRAPID creates an order of magnitude increase in the time for a system call, though this increase does not necessarily result in an order of magnitude degradation in protocol performance. Moreover, the system scales to a large number of nodes, and we believe that the scaling is linear both with number of VNKs and with number of packets exchanged. (We will have a more exhaustive analysis of the overhead and scaling performance in the final version of this paper.)

8. Applications

Besides the obvious application of protocol development, ENTRAPID serves as the foundation for a variety of applications, some of which we outline next.

8.1 Service creation

As discussed in Section 3.1 the controllability of a protocol development environment makes it useful even for services that do not explicitly make use of TCP or the layers below it-- it is indispensable for creating the ones that do. The ease of use, exact emulation, scalability, and visualization features in ENTRAPID remove much of the drudgery in creating new Internet services. By adding a telephone network simulation component to ENTRAPID, we believe that telephone service providers can also use ENTRAPID to rapidly develop new telephony services.

8.2 Testing routing protocols

Router vendors currently develop and test their routing software by connecting to a small network testbed comprising of a few physical routers and communication links. This is costly and time consuming. By using ENTRAPID's "network in a box" approach, routing protocol developers can rapidly simulate even a large testbed environment on a single system. Moreover, our emulation approach allows developers to port their code to the real environment with essentially no change. Thus, we believe that this approach has the potential to dramatically increase the reliability of routing software.

8.3 Configuration and capacity planning

At present, it is very difficult to predict the performance or even the behavior of a network component without actually inserting it into the network. In the absence of good tools and a standard network engineering process, network planners rely on a "trial-and-error" approach to network configuration and capacity planning. This often results in non-optimal or incorrect configurations. ENTRAPID allows network managers and planners to test an emulation of their network before deployment. This allows them to scientifically plan their network topology, configure network components, and test their services before rolling them out to their customers.

8.4 Testing and benchmarking

With the proliferation of networking equipment, it is difficult to compare alternative choices without trying them out in a specific network. Equipment benchmarking and testing labs as well as network planners in individual companies can use ENTRAPID to write benchmarks simulating different network

configurations. Since ENTRAPID connects to the Internet at the IP layer and above, equipment can also be tested by connecting it to a machine running ENTRAPID and observing its behavior and performance.

8.5 Training and research

ENTRAPID can be used for hands-on training in network configuration and management. It can also be used to learn the behavior of specific components and protocols without having to use expensive real world equipment. We believe that it is an ideal research environment because it allows researchers to model the system to an arbitrary level of detail.

9. Discussion

The key idea in ENTRAPID is kernel virtualization. While other systems have virtualized device drivers (as in U-Net [VBBV 95]) and entire operating systems (as in IBM's Virtual Machine from the 1960's [SG 97]), our choice of virtualizing only the networking component of an operating system gives us almost the same power as a virtual machine, but with far less development overhead. Kernel virtualization gives us exact emulation with ease of use, two of the main requirements for an ideal PDE. The other requirements are met by adding a 'switch box', a clean metalanguage, and an external visualization GUI. These components synergistically interact to enhance the system.

In Section 1, we motivated the need for protocol development environments from two fronts: routing protocol testing and service creation. In Section 2, we presented the requirements of an ideal protocol development environment. We now discuss the degree to which we believe ENTRAPID has met its goals.

We believe that ENTRAPID provides an ideal tool for router testing. By running routing protocols as external processes, we can exactly emulate large routing topologies, testing protocol behavior in nearly arbitrary configurations. While we cannot test packet-forwarding speeds, we believe that packet forwarding is often the easy part in building a router. ENTRAPID does help in the difficult problem of routing protocol design and testing. Similarly, we believe that ENTRAPID is an ideal environment for protocol development. It allows developers to create code that can be directly ported to the Internet. Moreover, any user-level protocols not already present in ENTRAPID can be imported with little work. A developer can test his or her protocol in large topologies, looking for sensitivity to byte- and packet-level errors.

As discussed earlier, we think that ENTRAPID meets many of the requirements of an ideal protocol development environment.

- It is easy to use, since developers have essentially a zero learning curve in learning the packet send/receive API.
- It provides exact emulation of the entire set of kernel services. While it does not model specific Internet impairments, the impairments found in any particular impairment can be easily added to the code base.
- ENTRAPID is controllable. The ENTRAPID topology control language allows developers to set up arbitrary network topologies. Moreover, source code changes in user-level programs allow the behavior of the entire environment, including the behavior of kernel-level components such as device drivers and IP, to be easily modified.
- The ENTRAPID visualization and animation engine allows developers to use a GUI to set up topology and simulation parameters, and to visualize the results of simulation runs.
- The environment can be extended with new protocols either as virtualized processes or as external processes. This allows arbitrary customization of the simulation environment.
- The current version of the simulator scales to several hundred nodes. Scaling is determined essentially by the available memory and CPU. As these increase exponentially over time, so will the size of the simulations. We also hope to investigate some techniques for session-level simulation to increase scalability.
- Finally, while ENTRAPID does not currently support verification, we hope to add it shortly.

10. Future work

Although ENTRAPID meets many of the requirements for an ideal protocol development environment, there are still several areas that need improvement. In this section, we outline these problems and point out areas for future work.

First, we would like to improve overall performance by reducing packet copy costs. At the moment, data from a user is copied to a VNK, from the VNK to the wire, from the wire to the receiving VNK, and from the receiving VNK to the receiving process. Thus, each packet incurs four copies, which is a considerable drain on CPU. We can get rid of these copies using a zero-copy architecture such as the ones described in [EM 95, VBBV 95, TNML 93].

Second, ENTRAPID does not scale well when emulating networks with large bandwidth delay products. Such networks require substantial per-node buffers. Since ENTRAPID exactly emulates routers, its memory usage is the sum of the memory sizes of WAN routers, which limits simulation scaling. We hope to exploit techniques to collapse router buffers, such as those described in Reference [AD 92], to reduce this overhead.

Third, ENTRAPID does not deal well with process failure. In a normal FreeBSD machine, when a process terminates unexpectedly, its state is cleaned up by the kernel, which releases resources such as held locks and open file descriptors. This is hard for a VNK to emulate, because, unlike a BSD kernel, it does not maintain an exhaustive per-process descriptor. Moreover, currently a VNK is not notified on process failure. We plan to deal with process failure by incorporating a garbage collector into ENTRAPID. This would periodically scan all resources and reclaim those held by dead processes. The alternative, which is to explicitly track all resource usage, strikes us as requiring too much bookkeeping overhead.

11. Conclusions

The ENTRAPID protocol development environment satisfies all the requirements of an ideal protocol development environment. It presents programmers with the abstraction of a 'network in a box'. This allows rapid protocol development and testing. We believe that the environment can be used for a wide range of applications that build on this core technology.

12. References

[AD 92] J.S. Ahn, P. B. Danzig, D. Estrin, and B. Timmerman**,** A Hybrid Technique for Simulating High Bandwidth--Delay Product Computer Networks, *USC CS Technical Report 92-528*, 1992.

[ADLY 95] J.S. Ahn, P.B. Danzig, Z. Liu, and L. Yan, Experience with TCP Vegas: Emulation and Experiment, *Proceedings of ACM SIGCOMM '95*, Boston, August 1995.

[ASDM 94] C. Alaettinoglu, A. U. Shankar, K. Dussa-Zieger, and I. Matta. Design and Implementation of MaRS: A Routing Testbed, *Journal of Internetworking: Research & Experience*, vol. 5, no.1, 17-41 (1994).

[AT&T 98] AT&T Geoplex, AT&T Labs Internet Platforms, `http://www.geoplex.com`

[BSK 95] H. Balakrishnan, S. Seshan, and R.H. Katz.**,** Improving Reliable Transport and Handoff Performance in Cellular Wireless Networks, *ACM Wireless Networks*, 1(4), December 1995.

[BFHR 97] K. Birman, R. Friedman, M. Hayden, and I. Rhee. Middleware Support for Distributed Multimedia and Collaborative Computing. To appear in *Proc. MMCN 1998*, 1998.

[Cadence 98] Cadence Inc., BONeS simulator, `http://www.cadence.com/alta/products/bonesdat.html`

[Cisco 98] Cisco Inc. NetSys performance tools, `http://www.cisco.com/warp/public/734/`

toolkit/performance/

[EM 95] A. Edwards and S. Muir, Experiences Implementing a High-Performance TCP in User-Space, *Proceedings of ACM SIGCOMM '95*, Cambridge, September 1995, pp. 196-205.

[Holzmann 98] G.J. Holzmann, *The Model Checker Spin*, IEEE Trans. on Software Engineering, Vol. 23, 5, pp. 279-295, May 1997, (Special issue on Formal Methods in Software Practice).

[IEEE-PIN 98] IEEE P1520 Proposed IEEE Standard for Application Programming Interfaces for Networks, http://www.ieee-pin.org/

[Keshav 97a] S. Keshav, An Engineering Approach to Computer Networking, *Addison-Wesley*, 1997.

[Keshav 97b] S. Keshav, REAL 5.0 Network Simulator, http://www.cs.cornell.edu/skeshav/real/overview.html

[LS 98] T.V. Lakshman and D. Stiliadis, High Speed Policy-based Packet Forwarding Using Efficient Multi-dimensional Range Matching, *Proc. ACM SIGCOMM '98,* 1998.

[Merit 98] The Multi-threaded Routing Toolkit, http://www.merit.edu/research.and.development/mrt/html/

[MIL3 98] Opnet Network Simulator, http://www.mil3.com

[MCI 98] MCI Corp, Vault press release, http://www.mci.com/mcisearch/aboutyou/interests/technology/ontech/vault.shtml

[NIST 98] NIST Network emulator, http://www.antd.nist.gov/itg/nistnet/

[ns 98] ns network simulator, http://www-mash.cs.berkeley.edu/ns/

[Opensig 98] Open Signaling Initiative, http://comet.ctr.columbia.edu/opensig/documentation/

[Paxson 97] V. Paxson, , Automated Packet Trace Analysis of TCP Implementations, ACM SIGCOMM '97, September 1997, Cannes, France.

[Perlman 83] R. Perlman, Fault-Tolerant Broadcast of Routing Information, *Computer Networks*, Vol. 7, 1983, pp. 395-405.

[SECH 98] F. Schneider S.M. Easterbrook J.R. Callahan and G.J. Holzmann, *Validating Requirements for Fault Tolerant Systems using Model Checking*, Proc. International Conference on Requirements Engineering ICRE, IEEE, Colorado Springs Co. USA, April 1998.

[SG 97] A. Silberschatz and P. Galvin, Operating Systems Concepts, *Addison-Wesley*, November 1997.

[SVSW 98] V. Srinivasan, G. Varghese, S. Suri, and M. Waldvogel, Fast Scalable Algorithms for Level Four Switching, *Proc. ACM SIGCOMM '98,* 1998.

[TNML 93] C.A. Thekkath, T.D. Nguyen, E. Moy, and E.D. Lazowska, Implementing Network Protocols at User Level, *Proceedings of ACM SIGCOMM'93*, San Francisco, September 1993.

[Torrent 97] Torrent Networks, Multi-kernel Network Emulator, *Personal Communication*, 1997.

[VBBV 95] T. von Eicken, A. Basu, V. Buch, W. Vogels, U-Net: A User-Level Network Interface for Parallel and Distributed Computing, *Proceedings of ACM Symposium on Operating Systems Principles*, December 1995.

[VINT 98] VINT home page, http://netweb.usc.edu/vint/

[Zimmerman 80] H. Zimmerman, OSI Reference Model—The ISO Model of Architecture for Open Systems Interconnection, *IEEE Transactions on Communications*, Vol. 28, No. 4, April 1980, pp. 425-432.