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 these 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, outline its implementation, and present a performance evaluation.**

I. INTRODUCTION

A *protocol development environment (PDE)* aids the creation of correct, efficient, scaleable, and robust protocols. This controlled environment allows developers to implement, visualize, and verify their work before deployment in the field. Research into protocol development environments is particularly timely because of the proliferation of new services in the Internet. 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 [1], under the auspices of an IEEE subcommittee [2], 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 [3, 4]. 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 [5] and MCI's Vault [6] 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.

II. REQUIREMENTS

A. Ease of use

An ideal PDE 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 to monitor protocol state.

B. 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.

C. 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

To appear, Proceedings of IEEE INFOCOM'99, March 1999.

packet losses, packet corruption, line failures, and routing protocol corruption, to stress the protocol under test.

D. Visualization

Protocols tend to be hard to design, and inexperienced developers have difficulty understanding them, and especially their interactions. Good visualization can play a key role in developing correct and efficient protocols.

E. Extensibility

Good developers tend to customize their development environment to quickly solve routine tasks. Besides allowing customization of the user interface, an ideal PDE should allow developers to add protocol components as required. 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.

F. 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.

G. 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 [7, 8].

III. STATE OF THE ART

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

A. 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 [9], wireless snoop protocols [10], 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 [11]!). For such protocols, the Internet protocol development environment offers exact emulation and little else.

B. 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 [12], the USC TCP-Vegas testbed [13], the Harvard simulator [14], and the NIST emulator [15]. 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, researchers at Torrent Networks and Harvard have independently built extensions to FreeBSD that provide exact emulation, extensibility, controllability, and scalability [16, 14]. 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.

C. 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 [17]). Examples of widely used network simulators include, in the public domain: ns [17], VINT [18], and REAL [19], and commercially: OPNET [20] and BONeS [21].

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.

	Direct	*Kernel extensions*	*General-purpose simulators*
Examples		NIST emulator, Torrent, Harvard simulator	ns, REAL, MARS, OPNET, BONeS
Ease of use		*	*
Exact emulation	*	*	
Controllability		* (Torrent)	*
Extensibility	*	*	*
Visualization			*
Scalability		* (Torrent)	*
Verification			

Table 1: Comparison of current PDE approaches.

The relative merits of the three approaches are compared in Table 1. Note that no single approach satisfies all the criteria for an ideal protocol development environment.

IV. DESIGN

The ENTRAPID protocol development environment combines the best features of the multi-kernel approach and general-purpose network simulation. Figure 1 outlines the architecture of the system. At the top level, ENTRAPID is 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 V). 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 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 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 hundred 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.

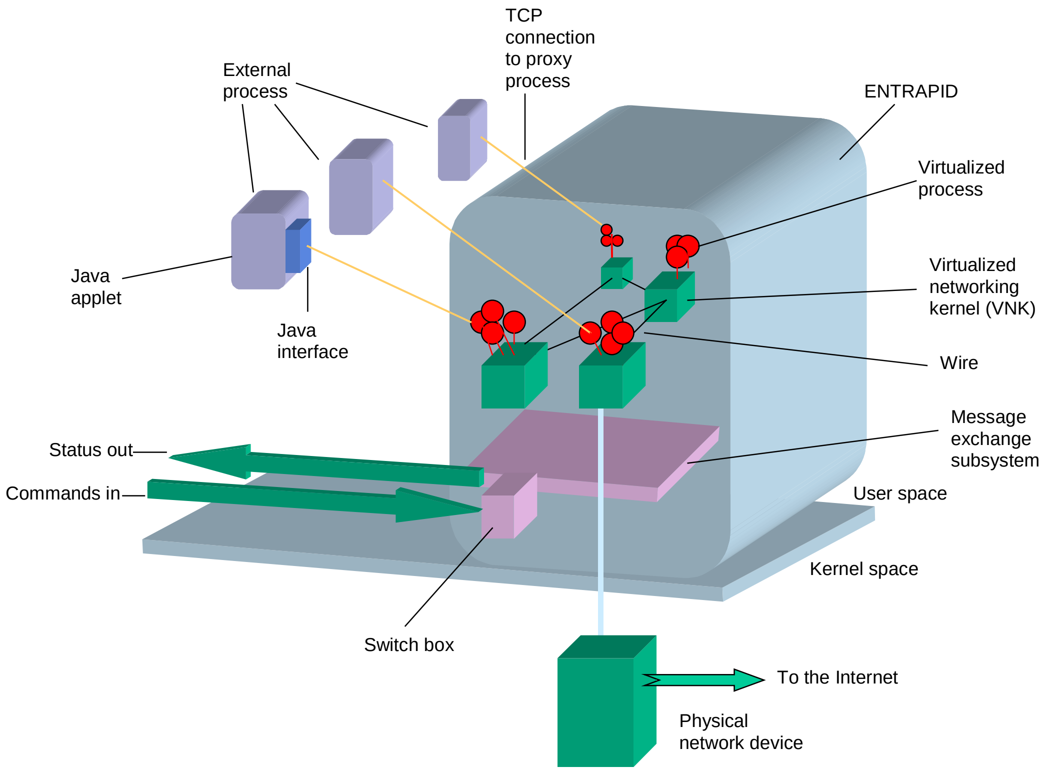

Figure 1: ENTRAPID architecture

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.

V. 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.

A. Kernel virtualization

Virtualization is the combination of multiplexing and indirection that allows a physical resource to be shared among multiple entities without their knowing it. 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.

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.

B. 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 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.

C. 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 a TCP/IP connection to the switch box, which forwards it to the appropriate proxy virtualized process. The proxy decodes the message and carries out the read. The results of the read are then returned, 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.

D. 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.

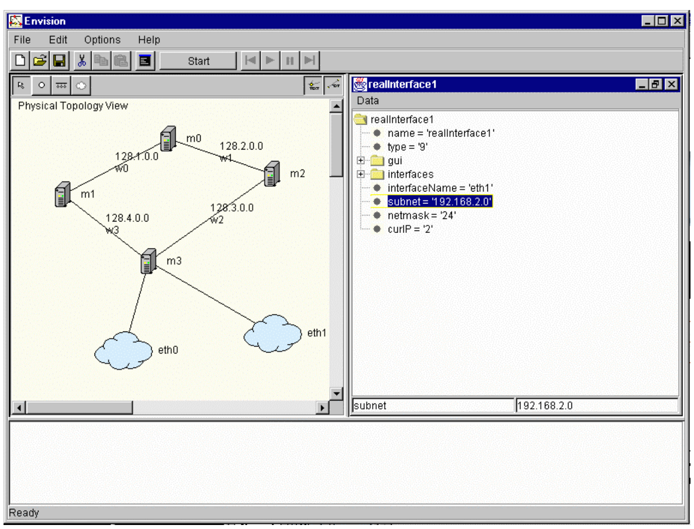

Figure 2: Screenshot of visualization GUI

E. Visualization

The ENTRAPID visualization environment provides:

- Topology creation and network configuration
- Packet-trace generation and animation
- A graphical front end for simulation control

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

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. All configuration parameters are available as drop-down windows keyed off nodes and links.

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 emits a status line in *tcpdump* format on the status channel. External packet filters parse this status line. 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 actions. This general mechanism allows a developer to easily create highly customized event logs and animations.

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. The ENTRAPID visualization environment supports these features.

VI. 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.

A. Overhead

Virtualization necessarily increases the time to make a system call. The table below compares the time taken by a normal process 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, corresponding to two additional context switches and data copies.

	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 system call for an external process 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 main lesson here is that data copying is expensive. We propose to use well-known copy avoidance techniques to deal with this problem [22, 12, 23].

B. 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.

We measured the size of the ENTRAPID process as a function of the number of VNKs when each VNK supports zero virtualized processes (so this represents the most optimistic scaling possible). We found that the 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, the limit on scaling comes not 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. To measure this overhead, we sent ping packets from an external process to each of the VNKs in a linear topology and computed the mean time to ping each VNK. Figure 3 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.

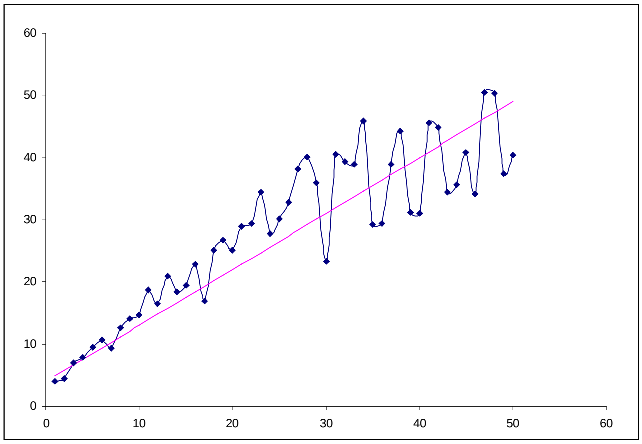

Figure 3: Ping time as a function of hop count

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.)

VII. APPLICATIONS.

Besides the obvious application of protocol development, ENTRAPID serves as the foundation for a variety of applications, including:

- Service creation
- Testing routing protocols
- Configuration and capacity planning
- Testing and benchmarking
- Training and research

VIII. DISCUSSION

The key idea in ENTRAPID is kernel virtualization. While other systems have virtualized device drivers (as in U-Net [12]) and entire operating systems (as in IBM's Virtual Machine from the 1960's [24]), 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 simultaneously provides both exact emulation and 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.

We now discuss the degree to which we believe ENTRAPID has met its goals. As discussed earlier, we think that ENTRAPID meets many of the requirements of an ideal protocol development environment.

- It is easy to use, since ir uses the standard BSD sockets 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..
- Finally, while ENTRAPID does not currently support verification, we hope to add it shortly.

IX. FUTURE WORK

Although ENTRAPID meets many of the requirements for an ideal protocol development environment, there are still two areas that need improvement.

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. We can get rid of these copies using a zero-copy architecture such as the ones described in [22, 12, 23].

Second, ENTRAPID does not scale well when emulating networks with large bandwidth delay products. Such networks require substantial per-node buffers. We hope to exploit techniques to collapse router buffers, such as those described in [25], to reduce this overhead.

X. 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. We are pleased to announce that ENTRAPID is available to academic institutions at no charge. For details, please refer to http://www.ensim.com.

XI. REFERENCES

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

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

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

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

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

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

[7] 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).

[8] 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*, Colorado Springs Co. USA, April 1998.

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

[10] 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.

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

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

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

[14] S.Y. Wang and H.T. Kung, A Simple Methodology for Constructing an Extensible and High-Fidelity TCP/IP Network Simulator, *Proc. Infocom 1999*.

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

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

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

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

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

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

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

[22] 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.

[23] 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.

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

[25] 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.