Virtual IP Machines: A System Framework for Emulating Multiple IP Hosts

Jesus Arango
University of Arizona
Gould-Simpson 721
Tucson, AZ 85721
+1(520)888-1816
jarango@cs.arizona.edu

Xxx
University of Arizona
Gould-Simpson 721
Tucson, AZ 85721
+1(999) 999-9999
xxx@cs.arizona.edu

ABSTRACT

This paper proposes a framework for emulating multiple IP hosts (virtual IP machines) inside a single system kernel. By augmenting and restructuring certain system components, the framework can provide an emulation and testing environment where a single system is capable of transparently representing multiple IP hosts comprising a virtual network.

The main advantage of the framework is that existing protocols, applications and network configuration utilities may execute under multiple virtual IP machines without being modified nor recompiled. The framework significantly reduces the equipment and spatial resources required by test beds and laboratory environments to appropriately conduct experiments and emulations.

This paper describes the architecture of the framework. We also describe our implementation in the Linux kernel and analyze the performance and scalability of the framework based on the results obtained from experiments conducted on our implementation.

Keywords

Virtual IP machine, VIPM, network, simulation, emulation, test bed, lab.

1. INTRODUCTION

Distributed systems, protocols and applications continue to emerge as the Internet keeps growing in size and complexity. The Internet's large-scale nature and global reach makes it even more important that we focus on correctness, performance and scalability. Therefore, developers and researchers must have at their disposal tools and resources that allow them to test their protocols and applications for these various types of performance metrics. These tools can be generally categorized as three different types: simulators, emulators and test beds. The latter we will also be referred as *testing environments* or *laboratory environments*.

Simulation tools construct a synthetic representation of the network topology, protocols and applications. The simulated protocols and applications are usually simplified versions of the real systems, perhaps removing details irrelevant to simulations. Time is generally also synthetic because a difference is made between simulation time and real time.

Simulators are economical and flexible because they can represent arbitrary topologies and diverse node and link characteristics. The notion of simulation time has also a positive effect on the scaling characteristics. Namely, there is no spatial or economical constraints on the size of network scenarios that can be simulated. Larger network scenarios may be simulated at the expense of longer execution time. Simulation time is also advantageous because the efficiency of the code has no impact on performance. For example: no optimizations need to be done and floating point operations may be used with no cost in performance. Another nice property of simulators is that experiments can be replicated. Unless random processes are intentionally introduced, the same network scenario[1] will always yield the same results.

Simulators unfortunately do have their drawbacks. Applications must be modeled in a way the simulator can understand by using the application interface (API) exported by the simulator. This often means that protocols have to be re-written entirely. Real systems introduce unpredictability not easily replicated by simulators. For example, multiprogramming in real systems affects the order and timing of network events.

There is no unique consensus of what network *emulation* involves. The ns [7] simulator defines emulation [10] as the ability to introduce the simulator into a live network where special objects within the simulator are capable of introducing live traffic into the simulator and injecting traffic from the simulator into the live network. The authors of [11] define emulation as environment where the applications to be analyzed can be run on separate, real machines not inside a simulator. In either case, it is assumed that an emulator has both synthetic and real hardware and software components. The advantages and drawbacks of emulations will depend on which components are real and which are synthetic.

Test beds represent an environment where all objects are real, including hardware, software and the notion of time. These environments obviously provide the most accurate representation but they exhibit serious flexibility problems. Different network scenarios require extra equipment and/or hardware rearrangement as well as software reconfiguration. Test beds are also very restricted in terms of scalability. Only

[1] "Network scenario" in this context represents the entire configuration of the simulation.

small network scenarios can be represented with limited hardware and space.

This paper proposes a tool that has two goals with respect to the discussion above: First, it provides novel approach and a new tool to test and analyze protocols and applications for correctness, efficiency and scalability. Second, it addresses the scalability issues of test beds or laboratory environments by having each hardware system effectively represent multiple IP hosts.

Our architecture is totally transparent. By borrowing from the concept of virtual machines [6], we introduce the notion of a virtual IP machine as a representation of an IP host. Under this architecture we can have many protocols and applications executing on the same physical machine but in the context of different virtual IP machines. Processes executing on different virtual IP machines have the illusion that they are executing on different IP hosts with their own network interfaces, IP addresses, and network state. Transparency is also achieved because existing applications, protocols and network configuration utilities can execute under different virtual IP machines without modification or the need to recompile.

The reminder of this paper is organized as follows: Section 2 describes the architecture of our framework and shows how multiple virtual IP machines are supported by the kernel. Section 3 describes our implementation in the Linux kernel. Section 4 presents the performance and scalability studies and describes the results obtained on experiments conducted in our implementation. Section 5 discusses related work, and explains how we have either improve or contributed on previous work. Section 6 elaborates on possible areas of future work. Finally, in section 7 we present the conclusions of our research.

2. VIRTUAL IP MACHINES

The virtual IP machine architecture enables processes in a single system to execute under different network states called virtual IP machines, which are transparent and independent representations of virtual nodes within the network. Figure 1 illustrates this concept. Just as every process is associated with a set of open files, a virtual address space, an execution state and process credentials, it is also be associated with a virtual IP machine. A virtual IP machine identifier is added to every process descriptor[2]. This identifier points to the virtual IP machine under which the process is running.

A virtual IP machine is essentially a data structure containing all the components necessary to represent the state of any IP host or router. Figure 2 shows the components of a virtual machine. The two most salient elements are the routing table and a set of network interfaces, but other components are necessary as well and each one will be explained.

[2] Other commonly used names for this data structure are process control block, task control block, task descriptor, process table entry. In Unix, this structure is usually called proc structure

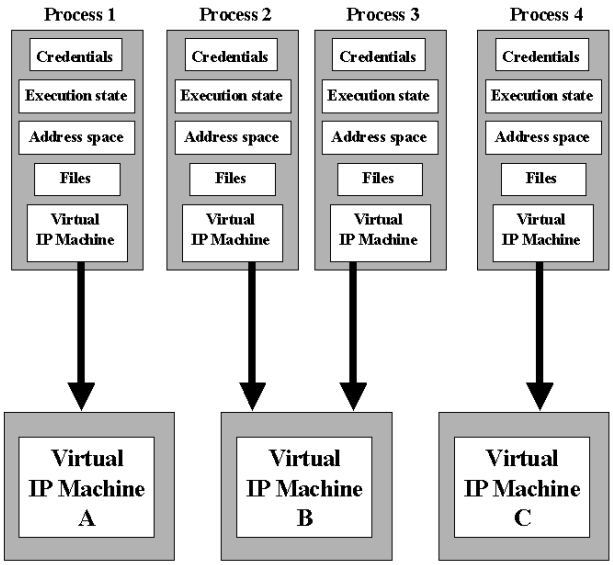

Figure 1: Relation between processes and VIP Machines

A virtual IP machine has a set of interfaces that connect it to other virtual IP machines residing either in the same physical machine or other external system. Each interface could either be real or virtual. A real interface links the virtual IP machine with a physical device attached to a network that may in turn be connected to other legacy systems or other systems capable of supporting multiple virtual IP machines.

A virtual interface is not associated with a physical device. It is used to link two virtual IP machines in the same system. Each interface is configured with a bandwidth and latency value. When a packet arrives at the virtual interface, the bandwidth and latency are used to schedule the arrival of the packet at the other end through the use of software interrupts. Multi-access networks such as Ethernet may be emulated by having the software interrupt deliver the packet to all virtual IP machines connected to the same IP subnet.

In Section 6, we discuss the possibility of having several virtual IP machines share a physical network interface by stacking several virtual interfaces on top a physical interface and defining the appropriate software interface between the two levels of abstraction. However, this requires a more elaborate re-structuring of the system's network stack.

Each virtual machine has a route table and a route cache used to forward packets generated by a local process or received through one its virtual network interfaces. All processes execute in the context of some virtual IP machine. Therefore, their IP packets will be forwarded according to the route table and route cache of corresponding virtual IP machine.

Locally generated packets are forwarded somewhat differently when compared to incoming packets arriving at one of the interfaces associated with the virtual IP machine. consider first those packets generated by a local process. Instead of doing a the usual lookup on a global forwarding table or cache, the IP layer must first obtain the virtual IP machine identifier of the current process and perform the route lookup in the table of the corresponding virtual IP machine. Incoming packets are instead forwarded during software interrupts and thus need to be treated

differently. Software interrupts are by definition asynchronous events, therefore they are executed in the context of an arbitrary process which happens to be whatever process is executing at the time of the interrupt. Incoming packets need to be forwarded in the context of the virtual IP machine associated with the incoming interface. But since the current process is arbitrary the current virtual IP machine will also be arbitrary. To circumvent this problem, each network interface data structure includes the identifier of the virtual IP machine it belongs to. When an incoming packet arrives at any given interface, the IP layer first obtains the identifier from the network interface and performs a route lookup in the corresponding virtual IP machine.

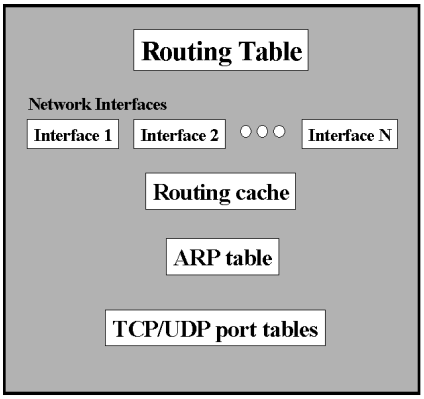

Figure 2: Structure of a Virtual IP Machine

Each virtual IP machine has its own ARP table to achieve better transparency and isolation. The framework can be implemented with a single global ARP table but certain secondary effects are introduced. Certain network utilities such as *netstat* and *arp* can be used to display address resolution information. If a global ARP table is used then the information from all virtual IP machines would be displayed. Any address resolution performed by a given virtual IP machine is also visible on all other virtual IP machines, potentially reducing the number of address resolutions in the network. This could be consider as an undesired change in network behaviour.

Virtual IP machines must also store information regarding currently used TCP and UPD ports and their associated connections. This information is used by TCP and UDP to determine the connection associated with each incoming packet and to deliver the data to the appropriate process. TCP and UDP ports must be unique across all connections in the same IP host because they are used to de-multiplex incoming packets to the appropriate processes. With global port tables, two processes could not bind to the same port even if they execute in the context of different virtual IP machines. Therefore, it is important that each virtual IP machine maintains its own port tables to allow processes to bind to the same port providing they are running on different virtual IP machines. For example, if one wants to run the RIP routing protocol (*routed*) on every virtual IP machine, each process instance of *routed* should be able to bind to port 520 on its corresponding virtual IP machine.

There is no performance penalty for adding support for virtual IP machines. That is, a legacy system will exhibit the same performance as a system running a single virtual IP machine. The only difference is that with virtual IP machines an extra level of indirection has been added when accessing network data structures. This means that virtual IP machine support can be added into main stream kernel distributions to provide extra functionality to the user without incurring in any loss of performance.

3. IMPLEMENTATION

We decided to implement our architecture on Linux. We initially debated between Linux and some open source implementation of BSD Unix. We ended up choosing Linux because we believe it has a broader reach and could be of more use to potential users.

The framework is implemented as a conditionally-compiled module. A dynamic module implementation proved to be difficult because the framework is not sufficiently isolated and independent. Many parts of the system and the network stack had to be modified. Since most of the modifications we made are located on the INET socket layer, we decided to implement the virtual IP machine as a conditionally-compiled component of the INET layer[3].

Our initial version is limited to IPv4. Multicast is not yet supported and only point-to-point virtual links have been implemented. However, this is not a limitation of the architecture. We hope our implementation will be expanded in the near future to support IPv6, multicast, multi-access links, and quality-of-service routing through the use of multiple routing tables, all of which are already supported by Linux.

The initialization code has been modified to create a default virtual IP machine with an ID of zero. All physical interfaces detected by the system at startup are assigned to the default virtual IP machine, and the virtual machine identifier of the ***init*** process is set to point to the default virtual IP machine. All processes created thereafter are descendants of the ***init*** process and therefore will inherit the default virtual IP machine. After initial start-up, the system will behave a single virtual IP machine and will behave just like legacy system.

A new Unix command-line utility called ***viprun*** has been created to spawn new processes under a virtual IP machine different that of its parent. This utility is crucial in providing full transparency to existing protocols and applications. The syntax of the *viprun* command is as follows

```
viprun id name [arg1 … argn]
```

the first argument, **id**, should be the id of the virtual IP machine, **name** is the name of the executable for the new process, and **arg1 – argn** are command line parameters passed to the new process. A new system call *setvip* has also been added and is invoked by *viprun* to change its virtual IP machine. After changing to a different virtual IP machine, *viprun* invokes *exec* and transforms itself into the specified process. Similar techniques are used by other well-known Unix utilities to transparently start a process under different conditions unbeknown to the process, therefore requiring no modification. For example, *nohup* is used to execute a process with the SIGHIP signal disabled and *nice* executes a process with a different scheduling priority.

[3] Note that the INET socket layer in Linux is itself a conditionally-compiled module.

A few system calls were added and the INET socket *ioctl* interface was expanded so processes can interface with the kernel to create and configure virtual IP machines and links. The two system calls are *getvip* and *setvip*. They enable processes to determine or change their virtual IP machine. The *ioctl* interface allows processes to create and configure the topology by adding or modifying virtual IP machines and network links. The *setvip* will usually be invoked by applications to change to another virtual IP machine while *ioctl* is generally invoked by network utilities or topology compilers to create and manipulate virtual IP machines and links. Existing applications make use of the *viprun* utility as described above.

We intend to remove the *getvip* and *setvip* system calls and add their functionality to the *ioctl* interface. These two system calls are easier to use and remember when compared to *ioctl* parameters, but adding system calls to Linux can be problematic because the system call numbers might collide with later versions of the Linux kernel. Linux developers recommend using *ioctl* when possible. We plan to create a user-level library of simple stub routines that invoke *ioctl* with the appropriate parameters. To use the *ioctl* interface directly, the application or network utility must first create a socket just for the purpose of interfacing with the INET socket layer inside the kernel. The socket descriptor is used as a parameter to *ioctl*. This is in fact the same technique used by network utilities such as *netstat*, *route*, *ifconfig* and *ip*.

TCP and UDP socket hash tables are created on each virtual IP machine. Linux uses hash tables to keep track of currently used ports. Hash values are generated from port numbers. Each table entry contains a pointer to the socket associated with the given port. Linux uses a single table for UDP. TCP uses two tables. One table keeps track of sockets that have been binded but whose connection has not been established. A similar table is maintained for sockets whose connection has been established.

4. PERFORMANCE AND SCALABILITY

5. RELATED WORK

Previous work may be divided into those that represent the traditional network simulation approach and those that make use of the kernel's network implementation. Our discussion will focus on the latter since it is most related to our work. However, for the sake of completeness and because of their undisputable importance, we mention some of the relevant work in the traditional simulation approach. These include ns [7], REAL [8], and OPNET [9].

S.Y. Wang and H.T. Kung have constructed a tool called the Harvard TCP/IP simulator [1] [2] using a novel approach that resembles a few similarities with our work and uses the real-life UNIX TCP/IP stack. The simulator uses a single routing table in the kernel and relies on tunnels to implement virtual links as user level processes. Figure 3 illustrates the architecture.

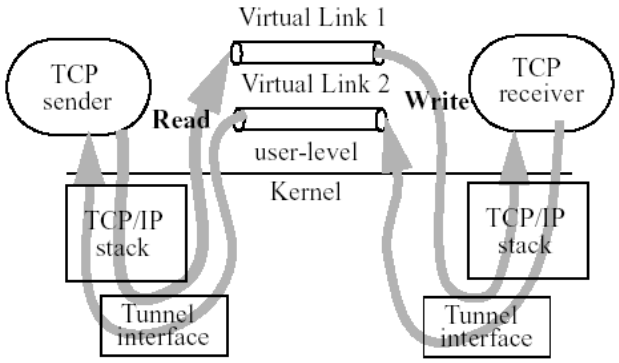

Figure 3: Simulator based on tunnel network interfaces

Each tunnel interface has a corresponding device special file in the /dev directory. Virtual links are implemented as user level processes that receive and send packets by reading and writing, respectively, on the device special files for the tunnel interfaces.

Routers are simulated by having IP packets re-enter the kernel for each router along the path. The Harvard simulator uses a single routing table inside the kernel. This requires that the destination address of each packet be re-mapped before re-entering the kernel. If re-mapping is not done, the IP packet will always be forwarded with the same routing entry and through the same tunnel interface, therefore looping on a single node until the TTL becomes zero.

This scheme fails to provide processes with the abstraction that they are executing under different IP hosts because the routing table and other network-related data structures are shared between all virtual nodes. A process that looks up the routing table will see the routing entries for all network hosts. This architecture is inappropriate for implementing network software such as routing protocols.

The authors of the Harvard simulator argue that by using a different routing mechanism, existing network utilities such as "route" could no longer be used. We have proved that by using the virtual IP machine architecture all existing software, including network utilities will continue to work.

An advantage of the Harvard simulator is that it uses a simulated virtual time instead of real time. This means that it does not have the scalability bounds of the virtual IP machine approach. This is something we are looking into and a possibility for future work. However, we would like to have the user choose what type of simulation time they wish to use. Our framework is meant to work efficiently under normal, non-testing environments and we'd like to maintain that benefit.

Dummynet [3] uses a similar approach to that of the Harvard TCP/IP simulator. However, there are some fundamental differences. Dummynet uses real time similar to our virtual IP machine approach. Routing tables in Dummynet are associated with incoming links rather than nodes. Thus the simulator does not know how to route packets generated by a router because such packets do not come from any links. The concept of virtual IP machines avoids this downside because every application or protocol executes in the context of an independent and well-define network state.

The ENTRAPID development environment [4] adapts the concept of a virtual machine [6] to that of a Virtualized

Networking Kernel (VNK). Each VNK runs as thread inside the ENTREPID User Space Process and represents a single network node. User processes also run inside the ENTREPID address space. VNKs are based on a virtualized version of the BSD Unix network that redirects kernel references to resources implemented inside ENTREPID. User processes are virtualized by redirecting BSD socket calls to routines inside the VNKs. Both VNKs and User processes are virtualized by modifying their source code to re-map all accesses to shared resources.

The approach has several downsides. It duplicates code and functionality already available in the kernel. Application source code needs to be modified to virtualize references to shared resources. Because many applications use other subsystems besides the network and the BSD sockets interface, ENTREPID must also deal with references to file systems, inter-process communications, and any other exported interfaces.

The EMPOWER research group at Michigan State University has designed and implemented of a network emulator [5] which can be used to simulate a variety network conditions inside a controlled laboratory environment. However, it is not intended to be a complete emulation or testing environment. It is used to extend the flexibility of existing laboratory environments to be able to emulate many types of link conditions such as delay, bandwidth and loss models. As shown in Figure 4, the tool is implemented as a Linux module between the IP layer and the network device driver. A configuration utility uses *ioctl()* to configure different link behaviors.

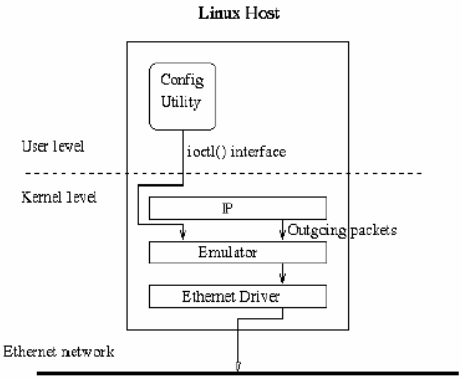

Figure 4: The EMPOWER emulator

6. FUTURE WORK

Further research and implementation is needed to expand and improve the virtual IP machine framework. We mention a few ideas here.

An important topic of research concerns decoupling real time from emulation (virtual) time as done by S.Y. Wang in the Harvard simulator to achieve grater scalability. We are looking for a somewhat different approach that meets the goals of our framework. The virtual IP machine architecture is designed to be incorporated into mainstream system distributions without decreasing system performance. A single virtual IP machine is created during normal startup and the system will behave as any other legacy system. Users may take advantage of virtual IP machine support at any time by creating additional virtual IP machines. To maintain this goal, the default virtual IP machine must be able to execute the network stack using real time. One approach is to use real time by default and provide a system call that processes may use to change to virtual time for network communication purposes.

We are also planning developing a topology compiler which automates the creation and configuration of virtual IP machines and virtual links, as well as process creation in the context of different virtual IP machine. Topologies and network scenarios are specified in a high-level language and the interpreter automatically creates and configures the virtual topology and processes.

Support should also be added for multi-access links. The current implementation supports only point to point links. Virtual IP machines should also be extended to support multicast and other common services that are already supported by the Linux kernel but that have not yet been ported to the virtual IP machine implementation.

7. CONCLUSIONS

We have proposed a system framework and architecture that expands the concept of virtual machine into the networking domain to transparently support multiple virtual IP hosts on the same physical system. We have shown that our framework is fully transparent to user-level processes and supports existing applications, protocols and network utilities without requiring modification or re-compilation.

Our performance results show that our framework exhibits limited but reasonable scalability. Our scalability analysis shows that our framework can significantly increase the size of network scenarios that can be emulated by test beds and laboratory environments. A typical system by today's standards is able to emulate ___ virtual IP machines without incurring in significant performance degradation.

8. ACKNOWLEDGMENTS

Our thanks to Dr. S. Y. Wang for permitting use to use his figures on the Harvard TCP/IP simulator.

9. REFERENCES

[1] S.Y. Wang and H.T. Kung, "A Simple Methodology for Constructing Extensible and High-Fidelity TCP/IP Network Simulators", IEEE INFOCOM'99, March 21-25, 1999, New York, USA

[2] S.Y. Wang and H.T. Kung, "A New Methodology for Easily Constructing Extensible and High-Fidelity TCP/IP Network Simulators", to appear in Computer Networks Journal

[3] L. Rizzo, "Dummynet: a simple approach to the evaluation of network protocols", Computer Communication Review, Vol. 27, No. 1, p.31-41, January 1997

[4] X. W. Huang, R. Sharma, and S. Keshav, "The ENTRAPID Protocol Development Environment", IEEE INFOCOM '99, March 21-25, 1999, New York, USA

[5] Kaushik K. Dam and Lionel M. Ni, "Design and Implementation of a Network Emulator", Michigan State University Technical Report: MSU-CPS-ACS-98-16, May 30, 1998.

[6] A. Meyer and L. H. Seawright, “A Virtual Machine Time Sharing system”, IBM Systems Journal, Vol 9. No. 3, 1970, pp. 199-218

[7] The ns network simulator, http://www.isi.edu/nsnam/ns/

[8] S. Keshav, “REAL: A Network Simulator”, Technical Report 88/472, Dept. of computer Science, UC Berkeley, constructing a TCP/IP network simulator with minimal time 1988.

[9] MIL3 Inc. home page, http://www.mil3.com/products

[10] The VINT project, “The ns manual”, A collaboration between researchers at UC Berkeley, LBL, USC/ISI, and Xerox PARC.

[11] D. Herrscher, A. Leonhardi, and K. Rotherme, “Modeling Computer Networks for Emulation”, Proceedings of the 2002 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'02), Las Vegas, June 2002