Network Virtualisation for Transparent Testing and Experimentation of Distributed Applications

Chris Edwards, Aaron Harwood, and Egemen Tanin
NICTA Victoria Laboratory
Department of Computer Science and Software Engineering
University of Melbourne, Victoria 3010, AUSTRALIA
{caedwa,aharwood,egemen}@cs.mu.OZ.AU

***Abstract*— Popular network simulation tools, such as *ns-2*, are useful for undertaking experiments with emerging networking technologies. As networked applications become distributed at scales comparable to the Internet, such as peer-to-peer applications, testing and experimentation becomes increasingly difficult and important. With this paper, we are introducing an elaborate extension to existing simulation capabilities by allowing realistic highly distributed application prototypes to be attached to a simulator for transparent testing and experimentation. We enable developers to focus on building their applications rather than detailing simulation scripts. Testing can then be performed in a natural setting. PDNS is a parallel and distributed version of the commonly used ns-2 simulation package. We describe our extensions to the PDNS simulator which allow real application prototypes to be run across a simulated network. We describe our use of virtualisation as a means for sending an application's network traffic through the simulator. Our implementation allows for large scale simulations with thousands of real peers and hundereds of thousands of simulated nodes in a network, thus we can test real peer-to-peer software at large scales.**

I. Introduction

As networked applications become decentralised and highly distributed, it becomes more important to predict potential failure or success of these applications before a large scale deployment. Even small glitches can lead to large scale problems and increased costs. Furthermore, it is desirable to test actual system software code rather than basing predictions of behavior on idealised models and parts of algorithms built using scripts. Large testbeds such as PlanetLab (http://www.planet-lab.org) provide a good basis for such testing at medium scales. Beyond this, and when greater control of test environment variables is required, network simulations are still a good choice. In this paper, we discuss the role of virtualisation for testing real system software code on a simulated network at large scales.

The use of virtualisation techniques is increasing across many application domains. Uses of virtualisation range from system partitioning, checkpointing, usermode network filesystems, to program supervision. We use the technique to virtualise a program's networking, redirecting network IO requests across a simulated network. With minimal assumptions on application loading behavior, our virtualisation technique is applied transparently to existing, unmodified applications.

This work was funded in part by the Victorian Partnership for Advanced Computing - VPAC, under project EPPN ME111.2004.

Testing and more specifically, debugging complex distributed applications is widely accepted to be a challenging task [1]. Reasons include difficulties in gaining access to the required number of machines across a wide-area, capturing trace data from experiments, and interpreting this data to make appropriate changes in the system. It is desirable to be able to go step-by-step through the messages that are passed between components of the distributed system. Simulations provide a means to solve these problems, however they introduce their own problems. One such problem is how to run an application in conjunction with a simulator, since the simulator is usually a self-contained program with its own implementation of protocols and scripting procedures. This implementation cannot then be used directly in a real application due to the existence of some unintended assumptions. For highly distributed applications, such as peer-to-peer applications, this is an important problem. In this paper, we address this issue by extending a popular simulator to accommodate real applications transparently.

Finally, network simulations are also computation and communication expensive. From a computation point of view, memory requirements will typically grow quadratically (or at least linearithmically) with network size and/or packet transmissions. Communication requirements are a consequence of the need for causality and the use of a globally accessed time based priority queue of events. Parallel network simulation can be used to increase the scale of simulations. After investigating various simulation packages, including *GTNETS* [2] and *OPNET* [3], we have based our work on *PDNS* [4]. The PDNS simulation is a parallel/distributed version of ns-2 [5]. The authors of PDNS have run simulations with as many as 600,000 nodes in a simulated network [4] that can address the scales that are desired by applications such as peer-to-peer applications.

A. Our contribution

In this paper, we describe an implementation and use of virtualisation for testing, experimenting, and debugging complex distributed applications in conjunction with PDNS. In particular, we provide a modified simulator, along with

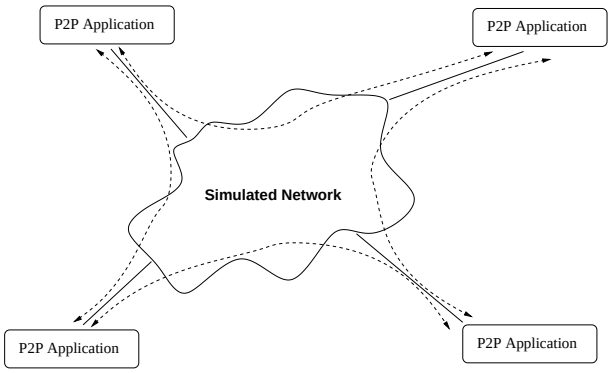

Fig. 1. How applications communicate with each other through the simulator

a library which is linked into applications at runtime. This library redirects appropriate network traffic to the simulated network, transparently to the application (Figure 1).

By running multiple copies of applications, each connected to different simulated hosts, we can experiment with their performance on a large network. Such experiments are particularly useful for peer-to-peer protocols and applications, where network traffic is not directed to and from a single, central server. Traffic patterns of this kind require simulations of large numbers of hosts and nodes in order to be realistic, as there are many potential bottlenecks.

B. Related work

Other approaches have been taken for the simulation of networks, running real applications. Techniques range from routers performing the simulation [6], to capturing traffic from virtualised operating systems [7]. However these techniques are not appropriate for the scale of simulations we would like to perform, or the computer power available to many enterprises.

A similar approach to ours has been used in the past, to redirect sockets via a SOCKS [8] proxy [9]. However our work differs in scale. We wish to simulate hundreds if not thousands of application processes, each with numerous sockets. It is important therefore to multiplex all sockets from an application into a single socket connecting to the simulator. If we used the SOCKS model, the simulator would have as many real socket connections as it does simulated sockets. This would be likely to test operating system limits that the simulator runs on, i.e., particularly on shared cluster machines used for experimentation.

In [7] and [10], User Mode Linux (UML) was used as an environment for each simulated host. This involves running a special Linux kernel which runs entirely in the user space of another. A virtual network device can then be used to send Ethernet frames to and from the simulation. This approach has the advantage of using a real TCP/IP stack, the overhead of UML is considerable, as hardware is not accessed directly by the kernel. In addition, each simulated host requires its own operating system kernel and file system.

An alternative approach to running simulations of real software is to port a networking framework to a simulator. One example is the porting [11] of the Click Modular Router [12] to ns-2. This framework is built on a software architecture for building routers which can then run in the Linux and FreeBSD kernels. While an efficient technique for such code, it is limited to modules for a particular framework and architecture. This makes comparisons with existing protocols difficult in simulations. It also places restrictions on the software outside of the simulation.

II. Virtualisation technique

Our virtualisation relies on the ability provided by operating systems to preload a library (Figure 2). This feature allows a library to provide functions which would otherwise be provided by standard system libraries. It is done without the knowledge of the application, however since it relies on dynamic linking, it will not work if the application has been statically linked against the functions to be replaced. The advantage of simply replacing user-space functions, is that the approach is relatively portable, compared with trapping system calls. It can also be done without special privileges, and is suitable for use on generic clusters.

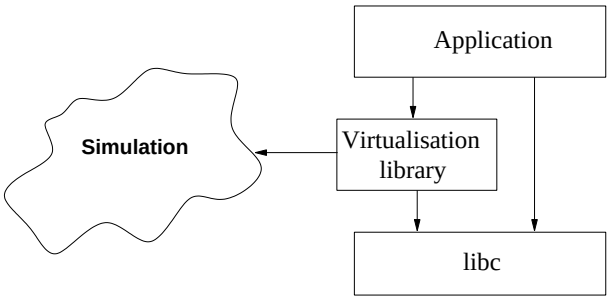

Fig. 2. Shows how the virtualisation library interacts with the simulator and application

In contrast to techniques such as [13], where a modified router is used to simulate packets travelling through a simulated network, ours can be run as a standard user on either a single machine or a cluster. It does not require special access to any machines or any specific hardware.

A. Socket trapping

Rather than trapping the system calls directly, our library simply replaces the standard C library functions used for networking. Alternatives exist, such as using the `ptrace` debugging facility [14], [15], often used by debuggers.

Our method of trapping a program's networking calls uses the `LD_PRELOAD` facility provided by most POSIX operating systems, such as Linux. It works by loading our library which will provide the application with the appropriate symbols, rather than the C library versions. Thus when an application makes a call to a function such as `socket`, the function in our library is called. A list of the functions we provide is given in Table I, along with a brief description of the more involved functions.

TABLE I

VIRTUALISED FUNCTIONS PROVIDED BY OUR VIRTUALISATION LIBRARY

Function	Comments
`socket`	Creates a dummy socket locally, to keep the file descriptors in-sync with the kernel, then sends a message to the simulator informing it of the new socket.
`connect`	Instructs the simulator to create a connection, either blocking or causing a timeout if the connection cannot be made.
`bind`	Sends the bind request to the simulator, which checks its own table of sockets for the host. This is needed because multiple applications may be running on the same simulated host, so the library cannot check if the bind will succeed by itself.
`accept`	Creates a new socket locally, and sends this file descriptor value to the simulator (so it can store this value itself), then waits for the simulator to either return with a new socket, or fail.
`listen`	Tells the simulator to begin listening on the socket.
`read`	Requests the next block of data from the simulator, and waits until it is received.
`write`	Sends a block of data to the simulator, and waits for an acknowledgement.
`close`	Tells the simulator the socket has been closed.
`recv`	As above for `read`
`recvfrom`	Similar to `read`
`recvmsg`	Similar to `read`
`send`	As above for `write`
`sendto`	Similar to `write`
`sendmsg`	Similar to `write`
`select`	Sorts the fd sets into virtualised sockets and local file descriptors. Sends a select message to the simulator for the virtualised sockets, then calls select on the remaining file descriptors, as well as a newly created pipe. The pipe is written to by the message receiving thread when the select reply arrives. This way the call can be awaken by the simulator or another file descriptor. The simulator will send a reply if the timeout value is reached, waking the call. This way the timeout works in simulated time.
`pselect`	Currently the same as select
`poll`	Transforms the call into a select call and runs as above.
`fcntl`	Passes options through to the simulator.
`getpeername`	Asks the simulator for the IP address of the remote host.
`getsockname`	Returns the IP address of the simulated host.
`getsockopt`	Retrieves options which have been successfully set.
`setsockopt`	Passes some options through to the simulator, others are handled locally, and some are ignored.
`ioctl`	Passes some options through to the simulator.
`sigaction`	Needed to simulate `SIGPIPE` for sockets.
`sigprocmask`	Needed to allow the library access to `SIGPIPE`.
`sigpending`	Needed to allow the library access to `SIGPIPE`.
`sigsuspend`	Needed to allow the library access to `SIGPIPE`.
`signal`	Needed to simulate `SIGPIPE` for sockets.
`sleep`	Tells the simulator to reply after the required elapsed time. This causes the sleep to occur in simulated time.
`gettimeofday`	Queries the simulator for the current simulated time.

From an application's point of a view, a socket is represented by a file descriptor, which is simply an integer. Our virtualisation library must provide the same interface. To do this, it needs to ensure that real file descriptors do not clash with file descriptors given to virtualised sockets. We considered two approaches for this. The first was to map file descriptors seen by the application into the file descriptors used by the kernel. The second approach was to create a real socket for each virtualised socket, so that the kernel's file descriptors matched up with those seen by the application. In this second method, the only information we need to maintain is whether a particular file descriptor is a virtualised socket or not. Note that the socket created for virtualised sockets is actually never used, its sole purpose is to reserve a file descriptor for our own use.

When a virtualised function is called, it first checks to see if the file descriptor is virtualised. If it is not, the original call is immediately initiated. Otherwise, the function interacts with the simulator.

All state is kept in the simulator, so these functions simply pass a message to the simulator and wait until they receive a reply, giving them the return and `errno` values, and any other data needed. It is necessary to refer to the simulator's state because sockets can be shared between processes (consider the case where a web-server forks off multiple processes to handle requests). There may also be multiple applications running on the same simulated host, so the allocation of TCP/UDP ports becomes an issue.

B. Handling auxiliary services

In order to create a realistic environment for applications, we must virtualise other functions as well as those for networking. We virtualise the functions related to time, such as `gettimeofday` and `sleep`.

By virtualising time related functions, the application is unaware that the simulation time may not be the same as real time, and everything appears to happen in simulation time. The only exception to this is the amount of CPU time the application receives. Currently we simply ensure the simulation runs slow enough that applications, which are assumed to generally be IO-bound, receive enough CPU time. The simulation of network elements can be slowed if applications are not receiving enough CPU time. How to automatically adjust the rate of the simulation is an area for further work.

C. PDNS interface

Our virtualisation implementation interfaces with a modified version of the PDNS simulator. However, the protocol used for communication between the library and simulator is generic enough for the implementation to be possible in other simulators. Indeed, using federation of a simulation [16], it would be possible to create a simulation running on more than one simulation package concurrently. This could be useful where a specialised simulation exists for a particular type of network. For example, by having a specialised wireless

TABLE II

INITIALISATION PARAMETERS SENT FROM THE LIBRARY TO THE SIMULATOR

Parameter	Comments
IP Address	The IP address of the host the application will run on within the simulation.
Start Time	The simulated time in seconds at which the application should begin running.
Identifier	An optional unique identifier, so that a simulation created process can be identified and located by the simulator. This means the simulator knows when the application it just created has connected.

network simulation package running in conjunction with a generic network simulator such as PDNS.

Initial development of the virtualisation library was performed with a simple daemon acting in the place of a simulator. Rather than performing a realistic simulation of a network, it handed data between nodes immediately. This daemon can be used to validate the virtualisation library without the added complexity of a simulator.

Communication between the library and simulator occurs over a TCP socket. While not having the speed of shared memory or other communication methods, it does have the advantage of making distribution of applications across machines straightforward. To avoid hitting socket limits in the simulator process, all virtualised sockets in a process are multiplexed across a single socket between the library and simulator. When a process forks, the new process closes the old socket and opens its own to the simulator. Both data and control messages pass across the same socket.

Upon application startup, the virtualisation library creates a connection to the simulator, and sends an initialisation message, informing the simulator of some parameters, as shown in Table II. Having this information passed by the library on startup allows applications to be started externally from the simulator. This is useful for running a handful of applications on a desktop machine to observe or interact with the simulation. An alternative would be to have all applications started by the simulator, but this removes such flexibility.

Once the library has sent its initialisation message to the simulator, it waits for a reply, which signals that it should begin running the application. This is implemented by having the library initialisation constructor wait for the reply. The application, or indeed other libraries it needs, cannot execute until the virtualisation library's initialisation is complete.

D. Virtualisation Limitations

Our current implementation is not completely transparent to applications. We have not yet implemented the `fork` and `exec` function calls. These two functions interact with the loader and dynamic linker, and would require a little more work than others. They would also require work to allow sharing of file descriptors between processes, which our library cannot currently handle.

```
set monitor [new WrapperDaemon 22334]
$monitor start-thread
```

Fig. 3. A code fragment demonstrating how the connection handling thread of the simulator is initialised.

To avoid hitting operating system limits on file descriptors, it would be useful to avoid creating a dummy socket for every socket the application creates. However, as mentioned, this would require a complex mapping between real and virtualised file descriptors, and may break some applications.

As mentioned earlier, performance could be increased by using a faster method for communication between the wrapper library and the simulator.

III. SIMULATOR ENHANCEMENTS

Our work adds additional functionality to the PDNS simulator, adding facilities for external applications to pass data through the simulated network. PDNS was chosen because it is based on the widely used ns-2 simulator, which many in the field are already familiar with. Creating simulations using our virtualisation technique is in most cases easier than before, as the researcher simply needs to create a network topology, add any extra network traffic, and create instances of applications to attach to nodes. Specifically, they do not need to implement protocols in the simulator, which can be cumbersome due to the different socket programming model used.

A. Connection handler

The simulator acts as the central daemon to which applications connect. This allows for flexibility in running applications by various means, one of which is through methods in the simulator. If the users wish to use our virtualisation, they must create and initialise an instance of the `WrapperDaemon` class we provide. It is up to the users to specify the TCP port it should listen on. The code fragment in Figure 3 demonstrates its use.

When a new connection is received and accepted, the simulator finds the node which has the IP address requested by the application. An agent is then created corresponding to the application, which will handle all the sockets it creates. There can be more than one such agent for a given node, each corresponding to a different application. When an application forks, it will create a new connection to the simulator. Threads of the same application will share the same connection. This architecture is shown in Figure 4.

The flexibility introduced by allowing connections from any external process introduces a complication in knowing when to begin. Because the simulator does not know about all applications which should be connected to it, it does not know when the simulation is ready to run. To work around this, the `WrapperDaemon` class has a method `wait-for` which blocks until a given number of applications have successfully connected. A call to this method then simply needs to be placed in the simulation script, before the command to begin the simulation. Note that having this method does not preclude

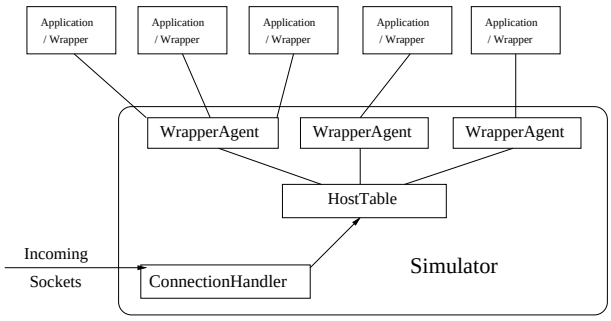

Fig. 4. Extra classes added to the PDNS simulator

applications from coming and going during the simulation. It is simply up to the writer of the simulation script to know how many applications should be connected at the start of the simulation.

B. Scheduler

Our modifications to PDNS include a wrapper around the existing `RTI` scheduler used for distributed simulations. Because of the additional thread created to accept connections, we require the scheduler to be thread-safe. In addition, to handle IO on existing connections, we must ask the TCL interpreter to check this on our behalf.

We made the wrapper scheduler thread-safe by requiring all calls to the underlying scheduler to acquire a single lock. The only time two threads will both access the scheduler is when a new connection is being handled, which requires a start event to be placed in the event queue. Our implementation handles this safely.

In handling IO, we have used the `IOHandler` class. This class is used by the existing network emulation code, but it normally requires the real-time scheduler to be used. Apart from keeping the simulation clock in time with the wall clock, the real-time scheduler also makes periodic checks for IO. Because we do not wish to restrict our simulations to real-time, we simply make these IO checks in our own scheduler. Our simulations can run at faster than real-time because the virtualisation library fools the application into thinking time is moving at the rate of the simulation.

C. Additional Methods

In addition to the classes added as mentioned earlier, the `Node` class in TCL has been extended. It now contains a `start-wrap-app-at` method, which loads an application using the virtualisation library and connects it to the node. By calling this method, the user does not need to know about the library preloading.

IV. Experiments and examples

We have implemented the techniques described and made our implementation available at `http://p2p.cs.mu.oz.au/software/vpdns`.

A. Cluster specifications

The cluster used for simulations is made up of 97 nodes, each with dual Xeon 2.8 GHz CPUs and between 1 and 2 GB of RAM.

B. PDNS scripts

After selecting PDNS as the simulator to build upon, some experiments were run with simple PDNS scripts. We were able to achieve 391,314 packet hops per second, while simulating over 415,000 nodes and 400,000 traffic streams. This simulation was performed with 10 nodes in the cluster, each using 2 CPUs. 890MB of memory was used across all 10 machines. The simulation run was the 417,200 node script at [17].

The major bottleneck we faced was memory usage. The distributed nature of PDNS means this can be largely overcome by distributing across more cluster nodes. We used the autopart [18] tool to partition our simulations across a cluster. This tool takes an ns-2 script, which must adhere to a special format, and outputs a given number of scripts, each of which is run on a separate machine. It is also capable of producing multiple scripts for machines with more than one CPU.

C. Telnet and server

While providing a relatively trivial interface, `telnet` actually tests numerous features of the virtualisation library. The principal problem we faced was the client's use of `select` to multiplex input and output from both the socket connection and the terminal.

A simple daemon is used in our solution which accepts connections before outputting data sequences of varying lengths, interspersed with calls to `sleep`. This verifies that changes in time are controlled by the simulation. When the simulation was run interactively, the sleep calls were not noticeable, however the simulator's time had indeed progressed by the correct amount.

The daemon also outputs to its own terminal any data received on its end of the socket. Using this tool we were able to verify that keystrokes from the client end were received by the telnet client and sent to the remote daemon.

D. Peer-to-peer

The primary reason for developing the virtualisation tool is to test peer-to-peer software. The peer-to-peer software we used is multithreaded and each peer creates many connections to other peers.

To handle multithreaded applications, the virtualisation library must be thread-safe, meaning that calls to its functions from multiple threads at the same time are handled safely. For the library this requires protecting its data structures and managing access to its socket connection to the simulator.

Our library protects its data structures and write access to the simulation connection using standard mutual exclusion locks. Messages are received from the simulator by a separate thread created by the library for its own use. This thread simply reads messages off the socket and places them in a queue.

It then checks whether a thread is waiting for the message, and wakes it up if so. Other schemes were tried, including having the first thread which waits for a message read from the socket. Complications may arise, because if an application calls `select` it must also be woken by events on other file descriptors. Table I has a more detailed explanation of how this waiting occurs in practice.

Experiments with simple peer-to-peer applications are run. These show that the virtualisation and the simulator can easily work together for highly distributed application testing. We are now in the process of running large scale simulations of a complex peer-to-peer application that we are developing.

V. Conclusion

This paper describes the implementation of a network simulator using virtualisation to test highly distributed applications. We have made modifications to a well known network simulator, PDNS, allowing an external library to connect to it, directing an application's network traffic through the simulator. The virtualisation library allows a wide variety of existing applications to be run unmodified while using the simulator. This then removes the need to create a separate protocol or application scripting for simulation purposes. It has been engineered to allow it to scale to large numbers of application processes, including distribution across a cluster.

Such simulations are an important tool for testing and debugging complex highly distributed networking applications. One such example is peer-to-peer applications, which are often utilised with thousands of peers across a variety of different networks and connection types.

References

[1] M. S. Meier, K. L. Miller, D. P. Pazel, J. R. Rao, and J. R. Russell, "Experiences with building distributed debuggers," in *SPDT '96: Proceedings of the SIGMETRICS symposium on parallel and distributed tools*. New York, NY: ACM Press, 1996, pp. 70–79.

[2] G. F. Riley, "The Georgia Tech network simulator," in *MoMeTools '03: Proceedings of the ACM SIGCOMM workshop on models, methods and tools for reproducible network research*. New York, NY: ACM Press, 2003, pp. 5–12.

[3] X. Chang, "Network simulations with OPNET," in *WSC '99: Proceedings of the 31st conference on Winter simulation*. New York, NY: ACM Press, 1999, pp. 307–314.

[4] G. Riley, "PDNS," 2005. [Online]. Available: http://www.cc.gatech.edu/computing/compass/pdns/

[5] L. Breslau, D. Estrin, K. Fall, S. Floyd, J. Heidemann, A. Helmy, P. Huang, S. McCanne, K. Varadhan, Y. Xu, and H. Yu, "Advances in network simulation," *Computer*, vol. 33, no. 5, pp. 59–67, 2000.

[6] K. Yocum, E. Eade, J. Degesys, D. Becker, J. Chase, and A. Vahdat, "Toward scaling network emulation using topology partitioning," in *Modeling, Analysis and Simulation of Computer Telecommunications Systems, MASCOTS*, 2003, pp. 242–245.

[7] D. Mahrenholz and S. Ivanov, "Real-time network emulation with ns-2," in *Distributed Simulation and Real-Time Applications, DS-RT*, 2004, pp. 29–36.

[8] M. Leech, M. Ganis, Y. Lee, R. Kuris, D. Koblas, and L. Jones, "Socks protocol version 5," 1996. [Online]. Available: http://archive.socks.permeo.com/rfc/rfc1928.txt

[9] "Tsocks," 2005. [Online]. Available: http://tsocks.sourceforge.net/

[10] U. Hatnik and S. Altmann, "Using ModelSim, Matlab/Simulink and NS for simulation of distributed systems," in *Parallel Computing in Electrical Engineering*, 2004, pp. 114–119.

[11] M. Neufeld, A. Jain, and D. Grunwald, "Nsclick: bridging network simulation and deployment," in *MSWiM '02: Proceedings of the 5th ACM international workshop on Modeling analysis and simulation of wireless and mobile systems*. New York, NY: ACM Press, 2002, pp. 74–81.

[12] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek, "The click modular router," *ACM Trans. Comput. Syst.*, vol. 18, no. 3, pp. 263–297, 2000.

[13] A. Vahdat, K. Yocum, K. Walsh, P. Mahadevan, D. Kostic, J. Chase, and D. Becker, "Scalability and accuracy in a large-scale network emulator," *SIGOPS Oper. Syst. Rev.*, vol. 36, no. SI, pp. 271–284, 2002.

[14] Z. Liang, V. Venkatakrishnan, and R. Sekar, "Isolated program execution: an application transparent approach for executing untrusted programs," in *Computer Security Applications Conference*, 2003, pp. 182–191.

[15] K. Jain and R. Sekar, "User-level infrastructure for system call interposition: A platform for intrusion detection and confinement," in *ISOC Network and Distributed Systems Symposium*, 2000.

[16] G. F. Riley, M. H. Ammar, R. M. Fujimoto, A. Park, K. Perumalla, and D. Xu, "A federated approach to distributed network simulation," *ACM Trans. Model. Comput. Simul.*, vol. 14, no. 2, pp. 116–148, 2004.

[17] D. Xu, "Autopart website," 2005. [Online]. Available: http://www.cc.gatech.edu/grads/x/Donghua.Xu/autopart/

[18] D. Xu and M. Ammar, "Benchmap: benchmark-based, hardware and model-aware partitioning for parallel and distributed network simulation," in *Modeling, Analysis, and Simulation of Computer and Telecommunications Systems, MASCOTS*, 2004, pp. 455–463.