Canopy: A Controlled Emulation Environment for Network System Experimentation

Dan Ports, Austin Clements, Jeff Arnold
MIT 6.829 Computer Networks Project Report
{drkp,amdragon,jbarnold}@mit.edu

December 15, 2005

Abstract

Network systems are hard to debug because they are inherently parallel and non-deterministic. Canopy *assists with network debugging by putting the entire network system into a* controlled emulation environment *constructed from virtual machines and a simulated network. This puts all variables under the user's control and provides a coherent, omniscient viewpoint of the entire system. To aid the user in observing and manipulating the system,* Canopy *provides tools such as traffic visualization, packet manipulation, rollback and replay.*

1 Introduction

Two of the most important properties of network systems — scale and nondeterminism — make constructing a debugger for these systems particularly difficult. These same properties also suggest how a network system debugger should be designed.

Scale. Traditional debuggers operate on a single process running on a single node. An ideal network system debugger should provide a coherent vantage point over an entire network and be capable of efficiently observing and controlling an arbitrary number of nodes simultaneously. Additionally, single-node commands such as "step this machine forward one instruction" are no longer meaningful when many nodes are involved, so a network system debugger should support system-wide commands such as "roll back the entire system to immediately before this event happened".

Tolerance to varying conditions. Network systems are designed to work under many conditions, but this flexibility can make understanding how a network system will behave more difficult. Constraints help designers understand how a system will behave; conversely, flexibility tends to add complexity and introduce subtle problems. An ideal network debugger should provide the experimenter with control over the conditions of the simulation that can lead to differing behavior. To facilitate this level of control, when a simulation is rolled back and replayed without changes, the simulation should exhibit exactly the same behavior as it did during the first execution.

Canopy, our network system debugger, includes the following key features derived from the properties discussed above:

1. The ability to scale the system to utilize available hardware resources
2. The ability to roll the simulation back to any previous time
3. The ability to replay the simulation from any point *with only specified changes*

Building a scalable system that provides feature (1) requires being able to take advantage of many physical machines. In a user-configured scenario involving n virtual computers, *Canopy* can distribute the computational work across up to n physical machines. Inter-node communication and synchronization uses a master/slave system, as will be discussed in detail in Section 4.

Nondeterminism is a significant concern for features (2) and (3). If any of the individual nodes behaves differently in any capacity during a replay, then the outcome of the replay could change for reasons unrelated to the modifications of the experimenter. Such changes can easily arise because computers generally exhibit at least slightly different behavior between different runs – for example, because the operating system's random number generator seeds itself using hardware event timings that vary across executions or because context switches occur at slightly different times. These unintended variations between executions can confuse the experimenter by adding confounding variables to a replay in which only a specified set of variables are supposed to change.

Thus, to effectively experiment with and debug network systems, we need to reign-in nondeterminism and form an omniscient, controlled viewpoint of the entire network system. This situation calls for a new type of debugger that is not only aware of network behavior, but that controls all "real-world" events in order to ensure perfectly repeatable execution. Building such a debugger naturally requires some mechanism for putting the entire network system into a closed environment that's both controlled and observable. To create this closed environment, *Canopy* virtualizes the *entire* network system.

The non-determinism of an individual node derives entirely from its coupling with the "real world" – specifically, this interaction includes the precise timing and content of asynchronous hardware events such as clock interrupts and input events. The non-determinism of an overall network system follows from the non-determinism of its individual nodes , as well as from events that occur in the network fabric. Thus, to reign-in non-determinism, we strictly control the passage of *virtual time* across the system, as well as the timing and content of "external inputs" at individual nodes, such as network packets.

The rest of this paper is organized as follows. Section 2 discusses related work. Section 3 presents basic information about using *Canopy*, including the *Canopy* interface (Section 3.1) and the *Canopy* network model (Section 3.2). Section 4.1 provides details about how *Canopy* emulates individual nodes. Section 4.2 describes a simple version of *Canopy* that achieves all desired network debugger functionality except scalability. Section 4.3 deals with extending *Canopy* so that it provides good scaling behavior. Sections 5 and 6 suggest future work and conclude.

2 Related Work

2.1 Network Simulators and Emulators

The ns2 network simulator [15] is commonly used for evaluating network protocols because it supports a vast number of network elements and traffic models. Typically it is used for packet-level simulation, in which packets are generated by synthetic sources. However, it also includes emulation extensions [6] which make it possible to "tap" live networks, introducing their packets into the simulated network, and to inject packets from the simulated network into a live network. This sort of emulation makes it practical to test real systems under esoteric network topologies, but does not provide the corresponding level of control over source behavior that *Canopy* provides by virtualizing not just the network but also the individual nodes.

Emulab [18] overcomes the lack of realism in network simulators by configuring physical computers and network switches to deploy the desired network topology on actual hardware. Emulab is fairly realistic because it runs real code on a physical topology instead of simulation code on a simulated topology. However, this realism comes at the cost of control and reproducibility because the experiment is essentially running "in the wild", albeit a wild tailored to the experiment configuration.

2.2 Replay

Replaying an execution history has been used as a tool for debugging both distributed and single-node systems. Nondeterminism is a common concern not only in distributed, networked systems of the type *Canopy* targets, but also in single-node multiprocessor systems, in multithreaded single-CPU systems (since the order in which tasks are scheduled can affect the outcome), or even in sequential single-CPU programs if they access external state. Because of this nondeterminism, replay is a useful tool in debugging these systems.

2.2.1 Replay of single-node sequential programs

Though it is especially useful in networked or parallel systems where execution is highly nondeterministic, replay is useful even in sequential execution environments with little or no nondeterminism: it can be used simply as a tool for visualizing the execution of a program. The EXDAMS debugger [1], dating from the late 1960s, recorded the history of program execution, and allowed a user to walk through the log to inspect the execution and debug the system. However, since it is not actually replaying execution, merely browsing a pre-generated trace, it cannot rollback and make changes; it is simply a visualization tool.

One of the earliest systems to apply rollback and replay to debugging was the IGOR debugger for the DUNE distributed operating system [7]. IGOR uses periodic incremental snapshots of a process's memory image to rollback a process to a previous state. It was designed for debugging sequential, single-process programs, and does not support parallel or distributed systems.

2.2.2 Replay of single-node parallel programs

Recap [14] was one of the first systems to apply this replay approach to *parallel* systems. To achieve deterministic replay, it logs of system call results, signals, and other asynchronous events (all external sources of nondeterminism) and periodically checkpoints the system's state by forking and suspending each process. Though it was designed for debugging parallel systems, it only allows rolling back of a single process, not the entire system state.

Instant Replay [13] also uses replay for debugging parallel programs. This system records the order in which threads acquire locks, which makes it possible to replay an execution history provided that all interaction between threads occurs using shared memory and is correctly synchronized with locks.

Tai et al. [16] considered the challenges inherent in debugging a parallel (but single-node) Ada system. They address the nondeterminism introduced by concurrent execution by adding synchronization sequences that allow an execution history to be deterministically replayed during debugging.

2.2.3 Replay of distributed/networked systems

Bugnet [3, 19, 20] supports the debugging of distributed systems via replay, using an approach quite similar to *Canopy*'s. Replay is achieved by logging inter-process messages and performing periodic global checkpoints. The architecture is also similar to *Canopy*: execution is distributed across multiple nodes using a Global Control Module analogous to *Canopy*'s master daemon and a Local Control Module analogous to *Canopy*'s slave daemons. Unlike *Canopy*, it runs processes directly with trapping of system calls rather than in full emulation; this is less computationally intensive but does not provide the same level of control. For example, it would not be possible to test changes to the kernel scheduler or network drivers. Unlike *Canopy*, it also does not appear to perform any network emulation, merely passing messages directly. This means it is not possible to test systems in the face of complex, non-deterministic network behavior such as packet delays, loss, and reordering.

Thane and Hannson [17] devised a scheme for debugging real-time distributed systems using a modified kernel to provide replay. Their system logs events such as context switches and data such as network messages and system call results in order to provide a history to replay. The most interesting idea is their notion of time synchronization, where time is quantized into units of time Π and synchronized between every node with a specified precision $\delta < \Pi$. With this, it becomes possible to decrease the amount of information that must be logged to enable rollback to a specific point: the global time orderings make it possible to rollback to an earlier point and deterministically replay to the target point. *Canopy* uses this approach of deterministically replaying from available checkpoints when necessary, although *Canopy* does not need to log context switches or other information directly related to playback because of how *Canopy* achieves deterministic replay at the hardware level.

The work most similar to ours is PDB [9, 10]. PDB makes it easier to debug distributed systems by running the entire system in a virtualization layer. It uses Xen [4] virtualization for each of the nodes; we considered using this approach, which would give higher performance, in *Canopy*, but decided to use Qemu instead for ease of implementation. Like *Canopy*, PDB allows simulating bandwidth limitations, latency, and packet loss on network links. PDB also includes a more comprehensive GDB [8]-like debugger for examining the state of processes running within each individual node in the system. We considered implementing a similar debugger, but relegated this part of the system to future work because of time constraints. Our principal improvement over PDB is our scheme for distributing the execution over multiple physical hosts; PDB is designed to run under a single host.

3 Using *Canopy*

3.1 Interface

Canopy is controlled through a Python interface – it can be fully manipulated using the Python interpreter shell. Both user-friendly and programmatic commands are available, which means that the full expressive power of Python is available when specifying configurations or debugger operations. In the future, a graphical interface may be added.

The interface allows emulation progress to be controlled: the user can start or suspend execution, step the system forward by a specified amount of virtual time, or roll the system back to a previous state. It also allows the user to view the network state via various mechanisms. The most basic such mechanism is a simple list of packets currently in transit. The user can use this list to select packets to drop or delay as desired. *Canopy* also allows network traffic to be visualized in timing diagrams. Figure 1 shows a timing diagram for a simple experiment with three nodes, in which one node sends ICMP pings to another node, with a network latency of 1.3 seconds. Each of the three black horizontal lines within a group represents a node, and virtual time progresses from left to right, top to bottom. Each packet is represented by a colored line between nodes; the color indicates the type of the packet. This method of visualization provides a high-level overview of network traffic. For more detailed inspection, it is possible to obtain a `PCAP`-format dump of packet contents on a given node. This data can then be viewed with a standard tool such as Ethereal [5], parsing the packet contents and giving essentially the view of a packet sniffer attached to that virtual node.

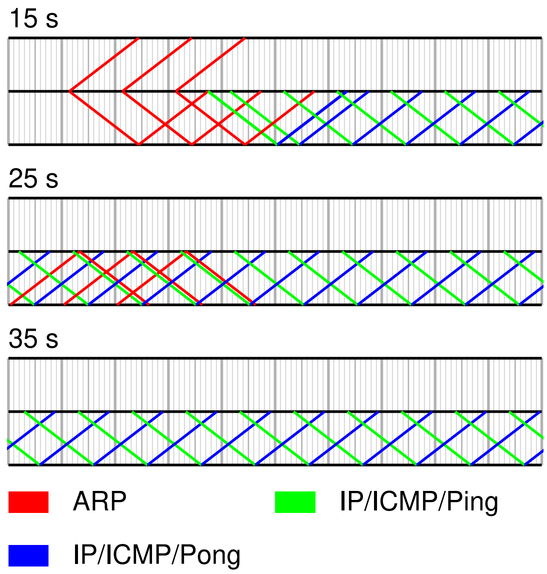

Figure 1: Ping trace

A *Canopy* experiment is set up by specifying the configuration for each of the virtual nodes in the system. This generally consists of the MAC address/IP address of each virtual node in the system, and an ISO image for it to boot. For convenience, we provide a base disk image which contains a minimalistic Debian installation with a kernel configured to run in Qemu. Users of *Canopy* have several options for creating a customized software environment: a user may create their own disk image from scratch, modify the provided disk image, or simply create an ISO image to be mounted in the emulated CD-ROM drive of each virtual machine. The provided base disk image automatically configures the network and runs an initialization script contained on the ISO image attached to the machine, if one exists. These options make setting up small experiments easy without restricting the user's ability to create a completely customized software environment when necessary.

3.2 Network Model

In order to effectively experiment with and debug networked systems, *Canopy* needs to provide control over the characteristics of the links that connect virtual nodes. *Canopy*'s network model consists of send and receive capacity bottlenecks at each node, fully connected by point-to-point links, as shown in Figure 2. The send and receive bottlenecks are implemented as drop-tail queues with a specified queue size and output rate; these bottlenecks model bandwidth limitations at each node. The links between pairs of nodes have configurable latencies and packet loss; these characteristics approximate the aggregate behavior of links in the rest of the system.

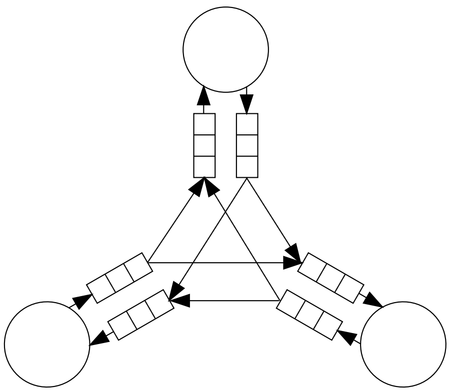

Figure 2: *Canopy*'s network model.

We selected this network model because it captures many of the critical network properties for wide-area distributed systems, and the network emulation can be implemented in a distributed manner. Ideally it would be possible to use a more detailed model that would, for example, take into account the network links and queuing behavior of intermediate routers on the network. We considered using a more comprehensive network simulator such as ns2 to provide this functionality; unfortunately, it remains unclear how to implement such an emulator in a distributed, scalable manner.

The network model provides default behavior whose parameters can be adjusted by the user. In addition, *Canopy* always allows the operator to intervene, manually dropping or delaying particular packets.

3.3 Use Cases

A network system debugger with *Canopy*'s functionality can be useful in many situations. We present some example scenarios where *Canopy* would prove useful for debugging.

Low-level systems Ben Bitdiddle has designed a new congestion control algorithm, and wants to test his Linux implementation to see how it performs in a heterogeneous network of nodes with different operating systems, link loss rates, and latencies. He deploys his implementation in a *Canopy* virtual node, using standard ISO images for the rest of the nodes. Using the timing diagram visualizer, he can see how the system evolves over time. He notices a particular sequence of packet losses triggers a corner case in the implementation, and wonders how it would be different if another packet is dropped. By rolling back the system state, dropping a packet, and replaying, he can see how the implementation behaves differently.

Here, *Canopy* is particularly useful for understanding network protocol implementation details since Canopy performs full system virtualization using the same code as production systems. Network simulators such as ns2 can exhibit different behavior from real-world systems because they simulate a protocol rather than executing real code in a virtual environment. *Canopy*'s abilities to manipulate individual packets are useful for this low-level application for which the details of which packets are lost or reordered is critical.

High-level systems Moving on from his congestion control algorithm, Ben decides to build a peer-to-peer file sharing system. He implements an indexing system atop the Chord DHT, and would like to test its fault-tolerance. To perform this test, he creates a disk image containing his software, and runs it on multiple *Canopy* virtual nodes. Using *Canopy*'s `PCAP` dumps and Ethereal, he monitors RPCs between the nodes, and uses *Canopy* to fail individual nodes at various important steps in the test query execution, verifying that his system correctly detects the failure and responds appropriately.

In high-level systems such as this one, understanding individual packet timings and losses is not especially useful because a higher layer such as TCP masks these effects. Hence, *Canopy*'s ability to manipulate individual packets is no longer critical; higher-level functions such as failing nodes are more useful. The ability to decode the contents of packets is essential to understand how the nodes in the system are interacting.[1]

4 Design

4.1 Node Emulation

At the individual node level, *Canopy* must differ from a typical emulator in a number of respects. First, in order to synchronize with other virtual machines and properly schedule events such as packet deliveries, *Canopy* must precisely control the passage of time in the virtual machine. Second, in order to support deterministic emulation of a whole network system, *Canopy* must, of course, support deterministic emulation at individual nodes. Third, *Canopy* must have an efficient mechanism for rolling back individual nodes to any point in the past.

To reuse existing work, *Canopy*'s node emulator is built atop Qemu 0.7.2, an efficient, open-source PC emulator based on a dynamic binary translator. While Qemu provides emulation of the PC hardware, it is intended for interactive use, and, as a result, violates *Canopy*'s requirements.

In a typical emulator, *virtual time* — the wall clock within the guest — progresses at the same rate that physical time progresses on the host. While this makes sense for typical, interactive use, it would make precisely synchronizing multiple virtual machines potentially running on multiple hosts difficult and complicate control of the timing of hardware events such as packet deliveries. Furthermore, such an approach makes the emulated system dependent on conditions at the host, which is undesirable for a system such as *Canopy* in which the debug environment is supposed to be completely controlled.

To solve this problem, *Canopy* completely decouples the passage of virtual time and physical time. The passage of virtual time is made proportional to the number of guest instructions (*ticks*) executed by the virtual CPU, approximating how time would pass on a physical computer. This and a user-specified "ticks per virtual second" value that controls the speed of the virtual CPU are enough to fully specify the passage of time in the guest, independent of passage of time at the host.

Canopy augments Qemu's dynamic translator to insert the appropriate host instructions between each translated guest instruction to increment the virtual tick counter and check a tick counter breakpoint. *Canopy* acts as a discrete event simulator, executing translated guest code until the precise tick of the next scheduled event is reached. When the guest is idle (ie, when it is executing the HLT instruction), *Canopy* immediately skips virtual time forward to the next pending event so the host CPU is fully utilized. Virtual time can progress hundreds of times faster than real time when the guest is idle. Because the timings of events are accurate to the tick, *Canopy* exhibits complete and deterministic control over the virtual clock.

To support *deterministic emulation*, *Canopy* must not be affected by sources of non-determinism in its own execution environment. *Canopy*'s emulator has, essentially, three potential sources of such non-determinism: timing with respect to physical time, user input, and network events. In the *Canopy* vir-

[1] Indeed, while developing *Canopy*'s distributed emulation mechanisms, it would have been extremely useful to have *Canopy* available to debug the distributed system. Unfortunately, circularity prevented this prospect from being realized.

tual machine, all notions of time are based strictly on the virtual clock instead of the host clock, so deterministic timing derives naturally from the deterministic clock. User input typically comes in the form of keyboard and mouse events. By simply using automation and restricting the user's ability to interact directly with the emulator, input non-determinism is easily managed. Finally, events on the emulated network interface (such as a packet arriving from the Ethernet) are also scheduled from the guest clock, so this source of non-determinism is eliminated, given a deterministic algorithm for scheduling packet deliveries.

Finally, in order to support *rollback*, *Canopy* has an efficient mechanism for saving and restoring the state of the entire virtual machine. *Canopy* builds on Qemu's existing snapshot system. While Qemu always saves the entire virtual machine state (a process that can take seconds), *Canopy*'s snapshot system can efficiently save incremental snapshots. *Canopy* maintains a set of dirty bits for the virtual machine RAM (both addressable RAM and miscellaneous RAM managed by virtual PCI cards), which allow it to very efficiently record just the differences since the previous snapshot. The constant overhead of an incremental snapshot is under 4K, and each modified page adds another 4K. To further improve efficiency, these snapshots are passed via shared memory to a separate *album* process, which manages the storage of snapshots, including asynchronously writing the snapshot data to disk while the Qemu process proceeds with emulation.

Canopy performs an incremental snapshot of the virtual machine every two seconds by default. To benchmark our snapshot system, we booted Linux in the virtual machine, a process which rapidly changes large portions of the machine's memory and is never idle. With a snapshot rate of two seconds, each snapshot during Linux startup requires less than 20 milliseconds of processor time to capture on modern hardware. Each snapshot requires about 3 megabytes of disk due to the significant changes in memory occurring at system startup. Because *Canopy* supports deterministic emulation, it can roll back to any point in time by simply restoring some point before the desired time then playing forward to precisely the desired point in time.

4.2 Local Network Emulation

We first describe how to emulate a networked system on a single physical machine, neglecting scalability issues. This requires emulating every virtual node as well as the network between them. Virtual nodes are emulated using the Qemu-based single-node emulator described above; *Canopy* runs one for each virtual node. In addition, there is controller process for synchronizing the virtual nodes, emulating the network characteristics, and exchanging packets between virtual nodes. For reasons that will become clear in Section 4.3, we will refer to this controller process as the *slave daemon*.

The slave daemon communicates with the node emulator via a simple control protocol called `xctl`, which allows the slave daemon to start and stop execution and set breakpoints. In addition, each virtual node emulator has a virtual NE2000 Ethernet device that the slave daemoncan indirectly manipulate using `xctl`. For example, when the slave daemonwants to have a packet appear on the virtual machine's network interface, it can ask the emulator for this event to occur via `xctl`.

Canopy maintains a constant δ defined to be the minimum latency over all virtual network links, and divides virtual time into *quanta* of length δ. The execution of all virtual nodes is kept synchronized so that their virtual time is always within the same quantum. This means that it is possible to exchange information between virtual nodes only at the end of each quantum: it is not possible for a packet to reach its destination during the same quantum in which it was sent, so each virtual node can be simulated independently for one quantum without having to stop to check whether any other node sent a packet that would affect it.

Network emulation proceeds as follows. The slave daemon maintains a schedule of packets to be delivered at each virtual node. The slave daemon instructs the emulator for each virtual node to proceed until the next packet is scheduled to be delivered — at which point it will be injected into the virtual Ethernet interface — or until the current quantum ends. If a virtual node sends a packet, the slave daemon applies the network model described in Section 3.2 to determine when the packet should be delivered (or if it should be dropped), and adds it to the packet delivery schedule as appropriate. Barrier synchronization is used to ensure that all virtual nodes are always executing the same time quantum: the slave daemon does not allow any virtual node to proceed to the next time quantum until every virtual node has completed emulating the previous time quantum.

This technique allows a networked system to be emulated faithfully by emulating each virtual node and the network that connects them. Global rollback can be achieved by locally rolling back the state of every node using the procedure described in Section 4.1, as well as rolling back the state of the packet delivery schedule. Rollback is deterministic because packets are can be delivered precisely, with nanosecond accuracy. However, with only a single physical node, this scheme does not scale well to many virtual nodes.

4.3 Distribution

The architecture described above accommodates emulation of a networked system, but it does not scale well. Emulating each virtual node is quite CPU-intensive, so it is infeasible to emulate an entire distributed system on a single physical node at a reasonable rate of execution. Hence, we distribute the emulation over multiple physical nodes.

Canopy runs atop one or more networked physical machines, as depicted in Figure 3. One of these machines is designated as the *master physical node*, and the rest of the machines are *slave physical nodes*. The slave physical nodes run slave daemons, essentially as in the non-distributed architecture of Section 4.2, except that they are now overseen by a master daemon on the master physical node, which ensures that the slave daemons are synchronized.

When a *Canopy* experiment begins, the master daemon distributes the virtual nodes that need to be created, assigning zero or more virtual nodes to each slave daemon, depending on the number of physical nodes available. It then transmits configuration information, such as the necessary ISO images and MAC addresses, to the slave daemons.

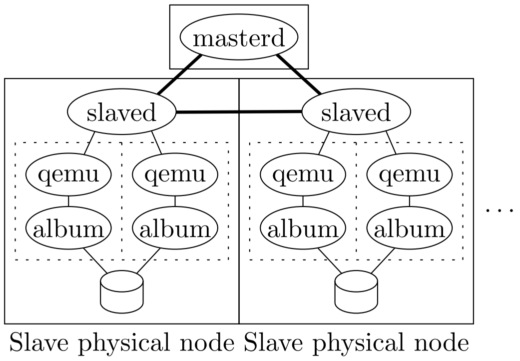

Figure 3: The overall architecture of *Canopy*. Physical nodes are enclosed in solid boxes. Virtual nodes are enclosed in dashed boxes. Network connections are shown in bold.

Execution proceeds in the same manner as the non-distributed case: the emulation of virtual nodes is synchronized to the same quantum, even for nodes on different physical nodes. When virtual nodes send network packets, the packets are exchanged directly between the slave daemons; in addition, they are reported to the master daemon so the user can monitor network traffic and for synchronization purposes. Barrier synchronization is performed on each slave daemon to ensure the virtual nodes are executing the same virtual time quantum; now, however, the slave daemon reports to the master daemon when all of its virtual nodes have reached the barrier, and the master daemon does not allow the slave daemons to proceed until each slave daemon (and hence *every* virtual node) has reached the barrier. In addition, the master daemon transmits to each slave daemon the number of packets it should receive from other slave daemons, to ensure that no slave daemon begins executing the next quantum before it has received all packets from the previous quantum.

5 Future Work

Extended network emulation. *Canopy* support for an ns2-like network simulator would be useful for situations in which highly realistic network behavior is essential. Unfortunately, using ns2 itself with *Canopy* would severely limit the scalability of the system because ns2's design essentially requires that all packets be delivered through some central node running the simulation. This conflict suggests the need for a *distributed* network simulator. Unfortunately, creating a "distributed ns2" is beyond the scope of this work.

Barrier-free synchronization. The barrier synchronization algorithm described in Sections 4.2 and 4.3 guarantees that no packet will ever arrive at a virtual node's emulator after the emulator has already simulated beyond the packet's arrival time. However, it introduces a performance overhead since emulators must periodically remain idle waiting for other emulators to reach the current barrier. By relaxing the virtual time synchronization requirement, we can improve performance by eliminating barrier synchronization. We briefly sketch an algorithm, which we have yet to implement, for ensuring correct emulation using speculative execution [11] and rollback instead of barrier synchronization.

Rather than trying to avoid the case where a packet arrives at a virtual node emulator too late, we can instead accommodate this case after it occurs. If this situation occurs, we can roll back the virtual node's execution to the time before the packet was scheduled to arrive, deliver it, and resume execution. With this procedure, we can eliminate the barrier synchronization, and indeed even the notion of quantized time becomes necessary. The master daemon is only necessary to report packets to the user and allow control.

This barrier-free synchronization algorithm can provide greater performance. Barrier synchronization constantly assumes the worst-case for packet delivery: that every packet will be delivered with the minimum latency possible. This leads to frequent synchronizations and unnecessary blocking; rollback will be more efficient unless the number of out-of-order arrivals is high enough that the cost of rollback exceeds the cost of synchronization. In these cases, a coarse-grained synchronization combined with speculative execution may be useful.

Intra-node debugging. *Canopy* is intended for holistically debugging networked systems, providing operations that affect the entire state of the system. However, it would be useful to be able to examine the internal state of each node in order to understand what is happening. For example, it would be useful to run a GDB-like debugger *inside* each node. However, it is not straightforward to apply a standard debugger directly, since it will lose its state during rollback; what is actually required is a debugger aware of replay and capable of stepping forward and backward in time.

6 Conclusion

Network system debugging presents unique challenges because of the scale and adaptability of network systems. *Canopy* solves some of the problems related to network system debugging by providing scalable centralized control, rollback, and deterministic replay. A user of *Canopy* can leverage the system to examine almost any piece of real-world network software since *Canopy* performs full machine emulation and does not require special software customizations. With *Canopy*, the entire network is placed under the control of the experimenter, making it possible to experiment with network parameters and observe the system's resulting behavior. *Canopy* makes it possible to debug a networked system as a whole, unlike traditional debuggers which only provide limited control over part of the system.

References

[1] R. M. Balzer. EXDAMS: Extensible debugging and monitoring system. In *Proc. ACM Spring Joint Computer Conference*, pages 567–580, 1969.

[2] W. H. Cheung, J. P. Black, and E. Manning. A framework for distributed debugging. *IEEE Transactions on Computers*, 7(1):106–115, Jan. 1990.

[3] R. Curtis and L. Wittie. Bugnet: A debugging system for parallel programming environments. In *Proc. IEEE International Conference on Distributed Computer Systems*, pages 394–399, Oct. 1982.

[4] B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, I. Pratt, A. Warfield, P. Barham, and R. Neugebauer. Xen and the art of virtualization. In *Proc. 19th ACM Symposium on Operating System Principles (SOSP '03)*, Bolton Landing, New York, Oct. 2003.

[5] Ethereal: A network protocol analyzer. Available from `http://www.ethereal.com/`.

[6] K. Fall. Network emulation in the Vint/NS simulator. In *Proc. IEEE Symposium on Computers and Communications (ISCC '99)*, Sharm El Sheikh, Egypt, July 1999.

[7] S. I. Feldman and C. B. Brown. IGOR: a system for program debugging via reversible execution. In *Proc. 1988 ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging*, volume 24 of *ACM SIGPLAN Notices*, pages 112 – 123, Madison, Wisconsin, 1988.

[8] J. Gilmore and S. Shebs. *GDB Internals*. Free Software Foundation, Boston, MA, second edition, 2003.

[9] T. L. Harris. Dependable computing needs pervasive debugging. In *Proc. 2002 ACM SIGOPS European Workshop*, Saint-Emilion, France, Sept. 2002.

[10] A. Ho, S. Hand, and T. Harris. PDB: Pervasive debugging with Xen. In *Proc. Fifth ACM/IEEE International Workshop on Grid Computing (GRID '04)*, 2004.

[11] D. Jefferson, B. Beckman, F. Wieland, L. Blume, M. DiLoretto, P. Hontalas, P. Laroche, K. Sturdivant, J. Tupman, V. Warren, J. Wedel, H. Younger, and S. Bellenot. Distributed simulation and the Time Warp operating system. In *Proc. 11th ACM Symposium on Operating System Principles (SOSP '87)*, pages 77–93, Austin, TX, 1987.

[12] M. Kim. On distributed and parallel debugging: Nondeterminism and deterministic replay. Technical report, University of Washington, 2002.

[13] T. J. LeBlanc and J. M. Mellor-Crummey. Debugging parallel programs with instant replay. *IEEE Transactions on Computers*, 36(4):471–482, Apr. 1987.

[14] D. Z. Pan and M. A. Linton. Supporting reverse execution for parallel programs. In *Proc. 1988 ACM SIGPLAN and SIGOPS workshop on parallel and distributed debugging*, volume 24 of *ACM SIGPLAN Notices*, pages 124–129, Madison, Wisconsin, 1988.

[15] The ns2 network simulator. Software available at `http://www.isi.edu/nsnam/ns/`.

[16] K.-C. Tai, R. H. Carver, and E. E. Obaid. Debugging concurrent Ada programs by deterministic replay. *IEEE Transactions on Software Engineering*, 17(1):45–63, Jan. 1991.

[17] H. Thane and H. Hansson. Using deterministic replay for debugging of distributed real-time systems. In *Proc. 12th EUROMICRO Conference on real-time systems*, pages 265–272, Stockholm, Sweden, June 2000.

[18] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An integrated experimental environment for distributed systems and networks. In *Proc. Fifth Symposium on Operating Systems Design and Implementation (OSDI '02)*, pages 255–270, Dec. 2002.

[19] L. Wittie. The Bugnet distributed debugging system. In *Proc. 2nd ACM/SIGOPS European Workshop on Making Distributed Systems Work*, pages 1–3, Amsterdam, Netherlands, 1986.

[20] L. D. Wittie. Debugging distributed C programs by real time replay. In *Proc. 1988 ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging*, volume 24 of *ACM SIGPLAN Notices*, pages 57–67, Madison, Wisconsin, 1988.