The Distributed Open Network Emulator: Using Relativistic Time for Distributed Scalable Simulation

Craig Bergstrom Srinidhi Varadarajan Godmar Back
Department of Computer Science
Virginia Tech
Blacksburg, Virginia 24061
{cbergstr—varadarajan—gback}@cs.vt.edu

Abstract— **In this paper, we present the design and implementation of The Distributed Open Network Emulator (dONE), a scalable hybrid network emulation/simulation environment. It has several novel contributions. First, a new model of time called relativistic time that combines the controllability of virtual time with the naturally flowing characteristics of wall-clock time. This enables a hybrid environment in which direct code execution can be mixed with simulation models. Second, dONE uses a new transparent object based framework called Weaves, which enables the composition of unmodified network applications and protocol stacks to create large-scale simulations. Finally, it implements a novel parallelization strategy that minimizes the number of independent timelines and offers an efficient mechanism to progress the event timeline. Our prototype implementation incorporates the complete TCP/IP stack from the Linux 2.4 kernel family and executes any application code written for the BSD sockets interface. The prototype runs on 16 processors and produces super-linear speedup in a simulation of hundred infinite-source to infinite-sink pairs.**

I. Introduction

With the continuing exponential growth of the Internet and the ensuing rapid proliferation of network protocols, there is an urgent need for large-scale protocol development environments that can act as controlled experimental testbeds for protocol and interoperability testing, verification and validation. The last several years have seen the deployment of protocol development environments that allow users to create complex controlled experimental testbeds to verify and validate network protocols. Protocol development environments can be broadly classified into (a) Network simulators and (b) Direct code execution environments. Simulators such as NS [4], OPnet [7], SSFNet [8], MaRS [1], and GloMoSim [21] use an efficient, event-driven execution model, but they require that the protocol under test be expressed in that model. The simulated protocol must then be converted to a real-world code implementation before it can be deployed, raising the problem of how to guarantee the equivalence of simulated protocol and actual code implementation. In addition, such simulated models fail to accurately express the idiosyncrasies of real-world TCP/IP implementations, which can significantly affect performance [2].

By contrast, emulators such as Utah Emulab [20], dummynet [16], ENTRAPID [11], and ModelNet [18] directly execute unmodified real-world code in a network test-bed environment. As such, they cannot easily model virtual networks whose emulation would require resources that exceed the resources of the physical system used to perform the emulation. Additionally, direct code execution environments typically map each network application to a separate OS process, which limits their scalability. Finally, and most importantly, since direct execution environments work in real time as opposed to virtual simulation time, they suffer from an inherent lack of temporal determinism which impacts the controllability of experimental test-beds.

Consequently, we need an approach that combines the predictability and controllability of event-driven simulation with the ability of emulators to run unchanged network applications and their associated protocol code. Although the high degree of parallelism that is inherent to large-scale network lends itself to parallel or distributed implementations, creating efficient frameworks for large-scale, high-fidelity simulations has proved challenging for several reasons.

First, it requires a temporal model that can reconcile the real-time nature of direct code execution with the event-driven nature of simulation models. To reduce synchronization overhead, this temporal model should minimize the number of independent timelines the simulator has to track. Second, it requires a runtime support system that enables the use of unmodified applications within the emulation/simulation system. To be scalable, such a runtime system must accommodate multiple simulation entities within a single OS process.

In this paper, we present the Distributed Open Network Emulator, or dONE, a scalable, distributed emulation/simulation framework. dONE has several novel features. First, it deploys a new model of time called *relativistic time* that combines the controllability of virtual time models with the self-synchronizing, continuous nature of wall-clock time, without violating the model's consistency even for directly executed codes. Second, dONE uses a composition framework called Weaves that provides the ability to emulate multiple instances of an application or protocol stack inside a single OS process.

Proceedings of the 20th Workshop on Principles of Advanced and Distributed Simulation (PADS'06)
0-7695-2587-3/06 $20.00 © 2006 IEEE

Weaves provides this support at the object code level, allowing emulated applications to be written in any programming language for which a compiler is available. Finally, dONE implements a novel parallelization strategy that minimizes the number of independent timelines and which offers a low-overhead mechanism to synchronize their progress with the timeline of simulated events.

To demonstrate the results of our approach, we present experimental results from a parallel implementation of dONE. The implementation incorporates the complete TCP/IP stack from the Linux 2.4 kernel family and executes unchanged application code written for the BSD sockets interface. We evaluated our prototype on a 16-CPU cluster and found that it scales to thousands of virtual hosts. It has produced super-linear speedup in a simulation of five-hundred infinite-source to infinite-sink pairs.

The rest of this paper is organized as follows. Section II reviews traditional approaches to modeling time in simulation environments. Section III presents the relativistic time model in detail. Section IV describes a distributed implementation of the relativistic time model in dONE. Section V presents experimental results and Section VI concludes.

II. Related Work

Two broad classes of simulation algorithms have been widely cited in the simulation literature. Optimistic protocols such as the time warp protocol [12] allow simulation nodes to process events without bound until a causality error is detected. When this occurs, the simulation is rolled back to the time of the error so that it can be corrected. Although it is possible to rollback directly executed code, doing so would be prohibitive for a system that aims to support the embedding of thousands of unchanged protocol instances.

Conservative algorithms prevent causality errors from occurring in the first place. Algorithms such as Chandy-Misra-Bryant's [5], [6], [14] typically rely on information from other nodes about their simulation state before deciding whether it is safe to process an event without incurring a causality error (input-waiting-rule). This rule requires the nodes to pass NULL messages to explicitly synchronize their clocks and break deadlock conditions if necessary. A number of mechanisms have been proposed to improve the performance of the conservative NULL message algorithm. These include attempts to decrease the number of NULL messages passed between logical processes [17], attempts to use a conservative lookahead window [3], and the recognition that for certain application domains (i.e. Virtual Environments) the precise ordering of events is less important than efficient and highly responsive execution [9]. Despite these improvements, traditional conservative algorithms still incur high communication overhead, because the timelines of the different nodes must be synchronized with a global virtual timeline.

Temporal models provide another axis for comparison. Virtual time models rely on a counter that is forced by the progression of events to represent time. Although this model can provide a very high degree of controllability and precision, its rate of progress is intimately tied to the physics of the simulation. Real-time models prevalent in direct code execution use a naturally flowing clock that progresses independent of the simulation. While this enables real-time models to exploit the parallelism inherent in the simulation, it impacts both precision and controllability.

III. Relativistic Time

The primary challenge in creating a hybrid emulation/simulation environment lies in reconciling two opposing requirements. Direct code execution of network applications runs in real-time or wall-clock time, which flows naturally (not forced by a progression of events), but is uncontrolled. Models of next generation network devices such as ultra high speed networks operate in highly controllable virtual time, but the virtual clock has to be forced by events occurring in the system.

To solve this problem, we developed a system of time called relativistic time, which borrows from the theory of relativity. Relativistic time reconciles real time and virtual time by creating a temporal model that flows naturally, and yet is highly controllable. Three basic properties underlie relativistic time. First, all physical measures of time are non-decreasing and ordinal in nature, rely on a periodic waveform, where the length of each period is controlled by the fundamental nature of space-time. Second, all clocks in a single frame of reference agree on the measure of time. Third, time is defined in terms of change of state.

To see how we use these properties, consider a network with two physical nodes that is used to model a virtual network with two virtual hosts (one per physical node). The physical nodes are connected over a network of bandwidth B. In this system, packets are sent from the virtual hosts to the simulator, which delivers it to the receiver at the appropriate time. If we model this network in a direct code execution environment, we are restricted to real time and hence cannot model a virtual network with bandwidth greater than B. Similarly, the latency of the virtual network we can model is restricted by the minimum latency of the physical network used in the emulation.

To emulate a virtual network that requires resources beyond the resources available to the physical system, we need to control how to stretch the amount of time given to the physical system so that it produces the virtual network's final state. This is similar to how a moving frame of reference's time is scaled relative to an inertial frame of reference.

In the above example control is achieved by (a) placing the virtual hosts in a new inertial frame of reference (I) (b) retaining the simulator in the original frame of reference (O), and (c) changing the velocity of I with respect to O. This brings us to the first operator of the relativistic time model—the time dilation operator. The ratio of the velocity of I with respect to O causes time to dilate in I. By judiciously selecting the velocity of I, we can dilate time sufficiently to enable the simulator in frame O to model a network of arbitrary

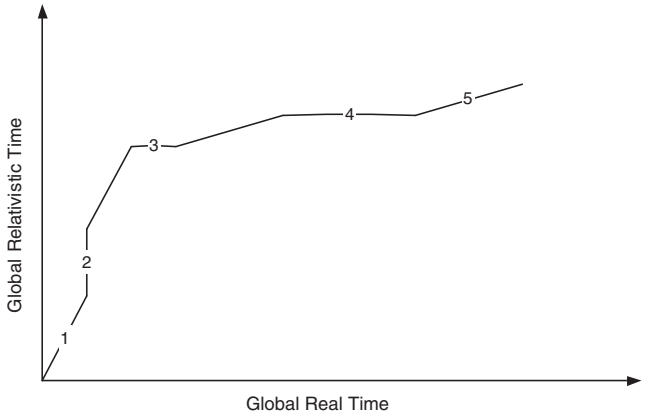

Fig. 1. Mapping of global real time to global relativistic time.

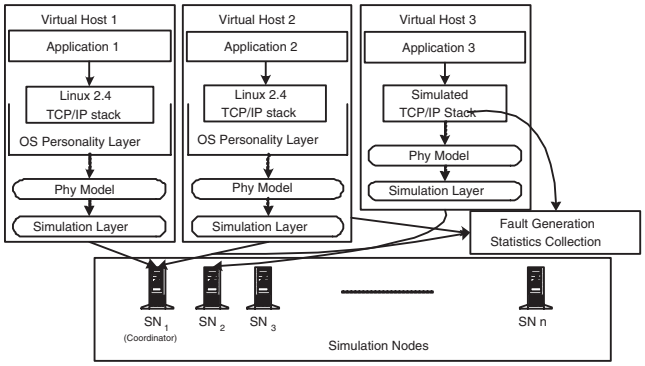

Fig. 2. The architecture of dONE.

bandwidth or latency, thereby achieving controllability. Intuitively, time dilation provides a mechanism to handle resource oversubscription.

While the example presented above handles the case of resource oversubscription, it is desirable to handle resource undersubscription efficiently as well. For instance, if the bandwidth of the virtual network is less than the physical network, it is desirable to run the simulation as fast as possible. This can be achieved in virtual time simulation systems, by warping over quiescent intervals. On the other hand, direct code execution engines use real-time delays to model undersubscription, which causes them to run at real-time or slower. Ideally, we desire a temporal model for direct code execution that can handle resource undersubscription efficiently.

Resource undersubscription can be divided into static undersubscription and dynamic undersubscription. Static undersubscription can be detected by analyzing the virtual network topology and statically setting the appropriate time dilation factor. Dynamic undersubscription occurs when there are global quiescent periods where all virtual hosts are waiting for an event to occur. To exploit quiescence, we use the third property of relativistic time. Since there is no change in the state of the system, time is not physically measurable. This leads to the second operator of the relativistic time model—time warp. If the simulator detects quiescence, it warps relativistic time to the next event in the system.

Relativistic time is thus a functional mapping between two frames of reference—the simulator's frame and the virtual network's frame. For simplicity, the simulator's frame of reference is chosen as real-time reducing relativistic time to a functional mapping of real-time. The functional mapping is kept consistent across all physical nodes in the simulation, thereby creating a single inertial frame of reference for the entire simulation. Since this functional mapping remains constant for the duration of a simulation, keeping timelines synchronized adds only minimal messaging overhead. The novel and advantageous aspect here is that the relativistic time simulator can apply a function to its local real-time clock and derive the current value of relativistic time with the assurance that all other physical nodes are at exactly the same instant of relativistic time. (In a distributed implementation, we assume that the physical real-time clocks of the participating nodes can be sufficiently tightly synchronized over short time intervals.) The result is that because nodes do not need to explicitly communicate to determine global time, messaging overhead is reduced.

Figure 1 depicts a sample mapping of relativistic time to real time. The inverse of the slopes of the lines represents the time dilation factor. Thus slopes greater than 1 (dilation factor less than 1) such as the line marked 1 imply static resource undersubscription, where relativistic time runs faster than real-time. Similarly the line marked 5 shows a resource oversubscription, where relativistic time runs slower than real-time. The line marked 2 shows the operation of a warp, where the simulator has detected global quiescence and warped relativistic time to coincide with the next event. The line marked 3 depicts a change in the functional mapping between relativistic time and real-time. In this case, the simulator stops the progress of relativistic time, broadcasts the new mapping to all physical nodes and resumes the progress of relativistic time with the new mapping. The line marked 4 shows a terminal case, where progress of relativistic time is stopped for an arbitrary amount of time. This is used to implement a pure simulation model where virtual time (in our case relativistic time) should not progress during an event handler.

IV. Implementing Relativistic Time

To implement the relativistic time model in a distributed environment, we had to design several components. First, we designed a mechanism to load multiple instances of a simulation entity into the simulation environment. Second, we designed a way to multiplex multiple virtual hosts onto a single physical node that kept their timelines in sync relative to each other and relative to the timeline imposed by the discrete event core. Third, we designed a method to separate a virtual host's state such that updates to internal state could be distinguished from updates with potential external effects. Fourth, we designed a coordinator-based mechanism to keep the timelines of virtual hosts executing on different physical nodes in sync. Figure 2 shows an overview of dONE's architecture.

A. Virtualizing the host environment

The Weaves [15] compositional framework provides a multi-threaded environment in which multiple virtual hosts, their network application and protocols, can run as separate threads within a single OS process. Unlike a traditional multi-threading framework, however, Weaves provides each virtual host with a separate namespace for its global and static variables. Weaves uses a binary rewriting process that transparently redirects accesses to those variables to a local copy for the current virtual host. This process is similar to how multiple instances of a shared library can exist in multiple processes, with a similar and small overhead.

B. Virtualizing Time

In addition to virtualizing a physical node's CPU and address space, we must virtualize its notion of time. The relativistic time model requires that a virtual host's time progresses at the rate of real time, or a fraction thereof, while it executes. Consequently, we provide a way for a virtual host to ascertain the fraction of real time during which it used the CPU. This fraction forms the basis for a virtual host's local clock. To accurately measure the virtual time that has elapsed on each virtual host, we have modified the Linux kernel to keep track of the amount of time allocated to each virtual host with nanosecond precision.

To conservatively guarantee the consistency of our simulation and to synchronize the timelines of virtual hosts with the timelines imposed by the discrete event engine, we must control their progress in two ways. First, we must ensure that no virtual host's timeline falls behind the timeline shared among all virtual hosts, as expressed by global relativistic time. If a virtual host fell too far behind global relativistic time, events originating from that virtual host may not be delivered to other virtual hosts on time, leading to temporal inversion. Second, we must ensure that no virtual hosts advance too far ahead of global relativistic time. If a virtual host advanced too far ahead of global relativistic time, its state would not reflect all events, again causing a causality error.

To ensure that no virtual host falls too far behind, we rely on the round-robin behavior and fairness of the OS's scheduler. To ensure that no virtual host advances too far ahead, we use a combination of conservative lookahead window and voluntary yield. Our model estimates the size of the conservative lookahead window based on the minimum lag between events. For instance, when simulating a network with a minimum latency of λ, we must make sure that no virtual host's local clock advances past $GRT + \lambda$.

If we cannot guarantee that a virtual host's local clock stays within λ of GRT, we must make sure that it is *either* preempted from the CPU *or* that its thread does not access state that might depend on external events.

C. State Management

Because current operating systems do not provide the ability to preempt threads at a sufficiently precise granularity, we have resorted to a cooperative approach. Virtual hosts are allowed to progress as long as they access only internal state; they must yield the CPU if they are about to access state that might be influenced by the delivery of external events, or about to modify state whose modification might cause external events.

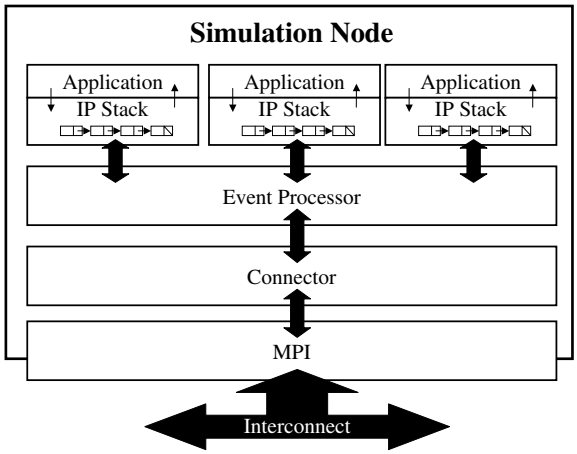

Fig. 3. Layers inside a simulation node in dONE.

We exploit our knowledge of the modular structure of the software we are executing to identify when a virtual host might modify or depend on externally visible state. As an example, consider the interaction between a network application and its protocol stack shown in Figure 3. In this example, external events include the sending or receipt of packets. Because only the protocol code has access to the procedure used to send packets, a packet can only be sent while a thread executes inside the protocol stack. Since the stack exports a well-defined interface to the application, we can interpose on that interface and insert checks that ensure a thread's local time is not running too far ahead of global relativistic time. Should this situation occur, the thread is blocked. Similarly, a thread may enter the protocol code to receive data. Typically, such receive primitives are implemented by polling the state of a packet queue in which arriving packets are enqueued. We make sure that when a thread enters the stack, all packets that should have arrived by that point in time have been delivered, or else the thread will be blocked until such time has arrived. Note that we do not require intimate knowledge of the specific data structures that are used inside the stack to hold received packets, nor do we require changes to the protocol code being run. much more details we can afford here - we could talk about how the stack's about how we control when the

D. Global Relativistic Time & Warp

A physical host's connector layer performs a dual function. First, it is responsible for the distribution of events between the physical nodes in the simulation. Second, it cooperates with other connector instances and a global coordinator in synchronizing global relativistic time among all physical nodes. We use a variant of the two phase commit protocol [10] for this purpose.

TABLE I
OBSERVED TCP THROUGHPUT FOR TRANSMITTING 5MB OF DATA AT DIFFERENT SIMULATED BANDWIDTHS.

Link B/W	Throughput	Run Time	Simulated Time	Speedup	No of Warps
56kbps	54.842kbps	16.01s	764.637s	47.637	83,731
100kbps	95.638kbps	9.98s	438.474s	43.935	51,101
500kbps	478.182kbps	3.32s	87.696s	26.412	15,083
1Mbps	0.956Mbps	2.53s	43.849s	17.331	11,734
10Mbps	9.427Mbps	1.74s	4.448s	2.556	6,727
100Mbps	78.182Mbps	1.66s	0.536s	0.323	6,332

A coordinator starts a new epoch by choosing and announcing a dilation factor for this epoch. During the epoch, the progression of real time on each physical host advances the relativistic timelines of all virtual hosts it contains according to the chosen dilation. A local connector detects an opportunity for a warp if all virtual hosts are idle, or have exhausted their lookahead window and are blocked because they are about to access external state. When a connector detects such an opportunity, it signals its readiness to warp to the coordinator. Once the coordinator detects that all connectors are ready to warp, it decides whether to warp and instructs the participants accordingly.

When announcing a warp opportunity, the coordinator must compute and announce the relativistic time at which the next epoch starts. This value depends on the time stamp of the earliest undelivered event in the entire simulation. The coordinator is able to compute this time as the global minimum of all undelivered events. Each node sends the timestamp of the earliest event known to that node to the coordinator during the first phase of the commit protocol.

To ensure that all events are accounted for, we use a simple acknowledgement protocol. If an event originating at one physical host is destined for another host, the receiving host is required to send an acknowledgement to the originating host. Until the acknowledgement has been received, the originating host will report the event to the coordinator; after the acknowledgement has been received, this responsibility is transferred to the receiving host. While the acknowledgement is in transit, both hosts might report this event's timestamp to the coordinator. Should this occur, the outcome of the coordinator's computation does not change.

A node's connector implements the warp protocol in parallel with the normal processing of events. Warping merely presents an opportunity for "jumping ahead" in the simulation, relativistic time still passes until a warp is announced. Consequently, the coordinator may decide to forgo a warp opportunity should it judge that it would cost more to warp than to simply let time advance to the next event. In this case, the coordinator will poll nodes later to inquire whether they are ready to warp and what their current earliest event time is.

V. EVALUATION

We implemented a prototype of dONE based on a custom version of the Linux 2.6.11 kernel. We used MPI/LAM as the underlying transport layer. All our experiments were performed on a cluster of eight dual-CPU Opteron nodes for a total of sixteen processors with 2 GB of physical memory per node. The nodes were connected via a 10Gbps Infinicon Infiniband interface. Each node was configured based on Redhat Fedora Core 2 in a diskless configuration.

The software environment of the virtual hosts consists of clients written in the C language, linked against a BSD socket API that is implemented over a TCP/IP stack extracted from the Linux 2.4 kernel. Extracting the stack from the kernel was possible with only a small amount of changes to the code, which increases our assurance that our simulation closely models the behavior of an actual TCP/IP implementation [13]. We implemented glue code to link the protocol stack to the simulation environment. For instance, we provided virtual device drivers to link virtual nodes to the discrete event processor. We isolated its top-level interfaces through which applications enter the stack, as well as its bottom-level interfaces, which is entered during interrupt processing. We also redirected the interface a process uses when it is blocked on a wait queue to notify the simulator and yield the CPU.

A. Simulator Overhead

We used a simple file transfer-like application to evaluate the veracity of our simulation. This application opens a TCP connection to a well-known port, then sends a predetermined amount of data across the connection in chunks of 1KB each. After receiving the data, the receiver sends an acknowledgement and both sides close the connection. The receiving virtual host measures the achieved bandwidth using the time API provided to it by the simulator, which reports the virtual host's local time.

Table V shows our results for the nondistributed case, in which the two virtual hosts reside on the same physical host. It measures the overhead of event processing and warping. The throughput numbers match what one would expect from a physical link for each of the bandwidths modeled. The simulation can model links up to 10MBps with a speedup compared to emulating them, while a 100MBps network takes about 3-times as long to simulate as to emulate it.

B. Throughput vs Latency

Sliding window protocols such as TCP/IP can only send a limited amount of packets until the first packet that was sent is acknowledged by the receiver. The maximum amount of data that can be outstanding before an acknowledgement is received is known as the window size. The window size is influenced by the flow and congestion control mechanism used in the protocol. Without extensions that allow for larger windows (e.g., window scaling [19]), TCP's window size is restricted to 64KB. Consequently, achieved throughput drops off as the round trip time grows larger beyond a point. The theoretical maximum, without taking into account packet loss, is known to be $\frac{WMSS}{RTT}$ where W is the size of the window, counted in multiple of the maximum segment size MSS, and RTT is the round-trip time 2λ.

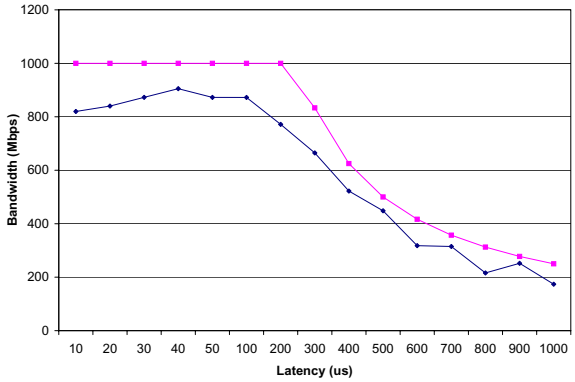

Fig. 4. Measured bandwidth for different simulated latencies. The top curve shows the theoretical maximum, the bottom curve shows the measured throughput. The simulation verifies the expected behavior of a sliding window protocol.

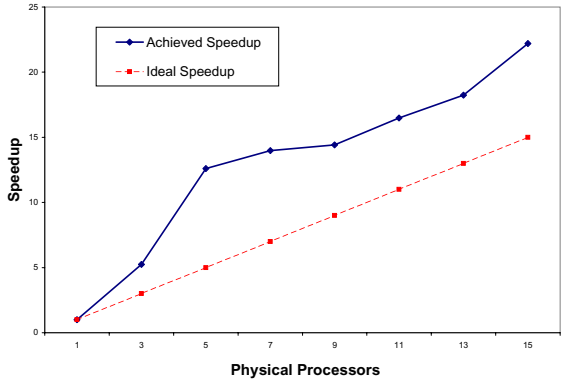

Fig. 5. Parallel speedup for different physical processor counts, assuming an optimal load balancing that minimizes communication. The bottom curve shows linear speedup; dONE achieves a superlinear speedup, shown in the upper curve.

To test if dONE could reproduce this phenomenon, we simulated a single 1Gbps link and varied the simulated latency of the link. The observed throughput when transmitting 10MB of data is shown in Figure 4. The upper line is the theoretical limit (neglecting the per packet header information, and TCP's startup cost). When the round trip time (twice the latency, shown on the X axis) is less than the time necessary to transmit a full window (488.273us), the observed throughput is about 80% of the link bandwidth. As the latency approaches the time to transmit a single window size, the observed bandwidth of the link drops off as expected. This demonstrates that the simulation is correctly modeling the bandwidth-limiting behavior of a sliding window protocol with a finite window size.

C. Scalability Results

To perform an initial evaluation of the scalability of dONE, we ran a simulation of one thousand simulated nodes transmitting 100KB of data over a 1Mbps link with short latencies. The same simulation was repeated for different numbers of physical processors. Each experiment was run four times, and the results were averaged and are plotted in the Figure 5. The same figure shows a theoretical, linear speedup, extrapolated from the single compute node case. All computations in this section ignore the cost of the coordinator node, so the single processor example involves two physical processors, the two compute node number really uses three processors, etc.. The simulation speeds up super-linearly as more physical processors are added, which we attribute to the greater overall cache capacity of the system. It should be noted that this speedup is achieved for a best case in which most virtual links do not cross physical node boundaries.

We also measured the how the accuracy of the simulation is affected as the number of virtual hosts increases. For a scenario of 50 source-sink pairs in which all virtual links crossed physical links, we found the measured throughput to be consistent and the variance negligible.

Although our initial results are encouraging, additional work remains to be done to validate the performance of the simulator on larger processor counts and with different network topologies and workloads.

VI. Conclusion

We have presented relativistic time, a novel model that support the creation of hybrid emulation/simulation environments. We have prototyped a distributed network emulator, dONE, which shows the feasibility of implementing this model in a distributed environment. The core idea behind relativistic time is to rely on the continuous character of real time, but scale it to allow the simulating environment enough time to accomplish the tasks necessary for the emulation or simulation of the environment being modeled. Relativistic time is a conservative approach; careful management of the simulation state allows time to be stopped to avoid causality errors. Time can be fast forwarded in a time warp when it is safe to do so in order to achieve a speed up compared to emulation.

Our current prototype uses a composition framework that allows us to emulate actual network applications and protocol stack implementations rather than relying on abstract models. Initial experimental results indicate that our prototype correctly models the performance behavior of protocols such as TCP, and is able to scale to thousands of virtual hosts on a small cluster of 16 CPUs.

VII. Acknowledgements

This work is supported by National Science Foundation under grant number 0305644. The authors gratefully thank the NSF for all of its' support. Additionally, we thank Joy Mukherjee for his work on Weaves and Chris Knestrick for his work on LUNAR.

REFERENCES

[1] C. Alaettinoğlu, A. U. Shankar, K. Dussa-Zieger, and I. Matta. Design and implementation of mars: A routing testbed. Technical Report UMIACS-TR-92-103, CS-TR-2964, Institute for Advanced Computer Studies and Department of Computer Science, University of Maryland, College Park, MD 20742, August 1992.

[2] Mark Allman and Vern Paxson. On estimating end-to-end network path properties. In *SIGCOMM '99: Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communication*, pages 263–274, New York, NY, USA, 1999. ACM Press.

[3] Rassul Ayani and Hassan Rajaei. Parallel simulation using conservative time windows. In *Proceedings of the 1992 Winter Simulation Conference*, pages 709–717, 1992.

[4] Lee Breslau, Deborah Estrin, Kevin Fall, Sally Floyd, John Heidemann, Ahmed Helmy, Polly Huang, Steven McCanne, Kannan Varadhan, Ya Xu, and Haobo Yu. Advances in network simulation. *Computer*, 33(5):59–67, 2000.

[5] R. E. Bryant. Simulation of packet communication architecture computer systems. Technical report, Massachusetts Institute of Technology, Cambridge, MA, USA, 1977.

[6] K. Mani Chandy and Jayadev Misra. Asynchronous distributed simulation via a sequence of parallel computations. *Communications of the ACM*, 24(4):198–205, April 1981.

[7] Xinjie Chang. Network simulations with opnet. In *WSC '99: Proceedings of the 31st conference on Winter simulation*, pages 307–314, New York, NY, USA, 1999. ACM Press.

[8] Jim Cowie, Andy Ogielski, and David Nicol. The SSFNet network simulator, 2002. http://www.ssfnet.org/homePage.html.

[9] Richard M. Fujimoto. Parallel simulation: parallel and distributed simulation systems. In *WSC '01: Proceedings of the 33nd conference on Winter simulation*, pages 147–157, Washington, DC, USA, 2001. IEEE Computer Society.

[10] Jim Gray. *Notes on Database Operating Systems*, volume 60 of *Lecture Notes in Computer Science*, chapter Operating Systems: An Advanced Course, pages 393–481. Springer-Verlag, 1978.

[11] X. W. Huang, R. Sharma, and Srinivasan Keshav. The ENTRAPID protocol development environment. In *INFOCOM (3)*, pages 1107–1115, 1999.

[12] David R. Jefferson. Virtual time. *ACM Trans. Program. Lang. Syst.*, 7(3):404–425, 1985.

[13] Christopher C Knestrick. Lunar: A user-level stack library for network emulation. Master's thesis, Virginia Polytechnic Institute and State University, 2004.

[14] Jayadev Misra. Distributed discrete-event simulation. *Computing Surveys*, 18(1):39–65, March 1986.

[15] Joy Muhkerjee and Srinidhi Varadarajan. Weaves a framework for reconfigurable programming. *International Journal of Parallel Programming*, 33:279–305, 200.

[16] Luigi Rizzo. Dummynet: a simple approach to the evaluation of network protocols. *SIGCOMM Comput. Commun. Rev.*, 27(1):31–41, 1997.

[17] Wen-King Su and Charles L Seitz. Variants of the Chandy-Mistra-Bryant Distributed Discrete-Event Simulation Algorithm. In *Proceedings of the 1989 Distributed Simulation Conference*, December 1989.

[18] Amin Vahdat, Ken Yocum, Kevin Walsh, Priya Mahadevan, Dejan Kostic, Jeff Chase, and David Becker. Scalability and accuracy in a large-scale network emulator. In *Proceedings of the 5th ACM/USENIX Symposium on Operating System Design and Implementation (OSDI)*, Boston, MA, USA, December 2002.

[19] Mark West and Stephen McCann. Improved TCP Performance over Long-Delay and Error Prone Links. In *Proceedings of the IEEE Seminar on Satellite Services and Internet*, February 2000.

[20] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. An integrated experimental environment for distributed systems and networks. In *Proc. of the Fifth Symposium on Operating Systems Design and Implementation*, pages 255–270, Boston, MA, December 2002. USENIX Association.

[21] Xiang Zeng, Rajive Bagrodia, and Mario Gerla. Glomosim: a library for parallel simulation of large-scale wireless networks. In *PADS '98: Proceedings of the twelfth workshop on Parallel and distributed simulation*, pages 154–161, Washington, DC, USA, 1998. IEEE Computer Society.