

MODELING THE GLOBAL INTERNET

A new scalable modeling framework and scalable parallel simulations make it possible to analyze the detailed behavior of large, multidomain, multiprotocol Internet models.

The last decade has seen a radical shift in thinking about dynamic behavior of data networks. Classical teletraffic theory has yielded to empiricism, as detailed traffic measurements reveal self-similar correlations of aggregate traffic and multifractal scaling behavior.¹⁻⁴ Empirical routing analyses have found ubiquitous route instabilities across many time scales that subvert static network topology concepts.⁵ This research is very recent and still incomplete—practical network engineers are only beginning to appreciate it.

Moreover, the lack of a good model of the Internet makes it difficult to analyze empirical data. For example, so-called congestion storms have yet to be properly characterized; these non-local phenomena are anecdotally associated with sudden routing changes, random flow aggregations, and user's behaviors that might be impossible to correlate using limited, localized measurements.

The greatest challenge, however, is that the Internet is an “immense moving target.”⁶ It not only grows at an exponential rate, it undergoes dramatic qualitative changes over time as well. These changes include paradigm shifts in use

(recently, the Web) and gross topology (recently, peering patterns among domains). In such a situation, we want to make predictions based on detailed, alternative-future scenarios.

Constructing appropriate global Internet models that can meet these challenges has proven extremely difficult, and so far, no project has gone beyond small models focused on limited problems. The principal difficulties include the large number of heterogeneous network elements (100,000 or more nodes) to be modeled, the long time scales required to model phenomena with long-range correlations, and the required level of detail, including the use of standard protocols and empirical traffic sources. Whether such models can be significantly simplified is currently the subject of a vigorous debate, which only cross-verification of detailed and simplified models can resolve.

This article focuses on simulation research. It describes the software designs that let us construct and run appropriately large models. After several years of research, we have developed a scalable network-modeling framework, a scalable simulation framework (SSF), and scalable parallel discrete-event simulators capable of modeling the Internet at unprecedented scales.

Modeling requirements

What is the global Internet? In July 1998, it was a collection of about 4,000 interconnected routing domains (autonomous systems, or ASs): node groups under common administration and intradomain routing confined within a domain. These domains can be roughly classified into network providers and end users. Interdomain

1521-9615/99/\$10.00 © 1999 IEEE

JAMES H. COWIE
Cooperating Systems Corp.

DAVID M. NICOL
Dartmouth College

ANDY T. OGIELSKI
DIMACS and WINLAB, Rutgers

routing reflects the peering arrangements negotiated among domain owners. Last July, about 37 million named hosts, such as `dimacs.rutgers.edu`, existed (<http://www.nw.com/zone/WWW/top.html>). The exponential growth extrapolates to 100 million hosts in the year 2000—a conservative estimate that misses machines hidden in private networks.

Clearly, a major downscaling is inevitable in any approach to Internet modeling. The details of acceptable simplification are an open question that comparisons of simulations with empirical data will help answer. A reasonable guess is that a model should describe a set of domains, characterize their interconnection topology, and simulate user behavior or packet traffic at a large-enough scale to accommodate spatial correlations that might arise in the real network. A large-enough model has a chance to exhibit “rare” critical fluctuations that seem to emerge regularly in the real Internet.

This resembles scaling conditions in statistical physics. In short, interesting phenomena arise only in sufficiently large networks, with a sufficiently large number of traffic flows. To find adequate model sizes, investigators should evaluate simulation behavior across three to four orders of magnitude, noticing the sizes where qualitatively new behaviors take place. This strategy requires a simulation model range extending to several hundred thousand hosts and routers, distributed across 100 or more domains.

Computing requirements

Network simulations of 100,000 multiprotocol hosts and routers are unprecedented in scale and pose two immediate challenges: the magnitude of the computing resources required and the invention of modeling techniques for managing the complexity of hundreds of thousands of simulated network elements. The first challenge is actually the easy part, thanks to Moore’s law and commodity multiprocessor technology. As we show, the computing requirements are already within reach.

To estimate the minimum computing requirements of a model with 100,000 multiprotocol hosts and routers, assume a simple, semirealistic scenario in which the majority of nodes are Web clients receiving Internet protocol (IP) packets at the average rate of 100 packets/s for 1% of total time. If packets traverse 10 routers on average, about 10^{10} network events (such as packet generation, routing, and receipt) are generated to produce traffic statistics for one hour of sim-

ulated network operation. One simulated hour is not overkill—too-short runs can generate misleading results in the presence of long-lasting transients and long-range correlations.⁷

We attack this computational challenge by employing parallel simulator kernels that can configure and simulate 100,000 nodes at rates of 10^6 network events per second. This puts simulations that generate 10^{10} events well within practical reach today; by Moore’s law, our software will simulate the same network in real time by the year 2002. By comparison, the simulation tools used today in engineering practice cannot handle models of this size at all. They are actually performance-limited to a few tens of thousands of network events per second on much smaller models.

Modeling software requirements

The breakthrough was finding the right software design. A usable parallel simulation engine needs a lean, self-configurable modeling API, and a framework of scalable design patterns for network modeling above it. Modeling scalability is critically important. It is needed to solve the known (if not widely discussed) difficulty of actually configuring and managing heterogeneous network models with 10,000 to 100,000 nodes or more, each with its own protocol graph. Such a task cannot be easily performed manually (as is done in existing modeling and simulation tools) because network graphs generally lack regularity. For small or structurally simple models, practically any network-modeling method might be adequate. However, as model size and complexity increase by orders of magnitude, most commonly used modeling methods rapidly begin to break down. Having crawled out of the wreckage several times ourselves, we feel qualified to comment on a few of the good ideas that simply haven’t worked in practice.

Models hand-tuned to utilize multiple processors become instantly useless outside of their original design context because of inattention to standard API design, lack of component configurability, or both. The abstraction level clearly has to increase—but this leads to other problems.

The temptation to create one more simulation language is intense. While new languages offer

The abstraction level clearly has to increase—but this leads to other problems.

the hope of increased modeling power by virtue of a raised abstraction level, more often than not they impose a modeling view that gets in the way of constructing scalable model elements.

Object orientation—universally applied, but improperly understood—has not fared much better. Modelers’ first instincts tend toward deep inheritance chains, which defeat component

reuse and model reconfiguration. What works instead is hierarchical composability from simple to complex objects, according to well-tested design patterns.

Scalable simulation framework

An object-oriented framework (a collection of codesigned class interfaces with prototype implementations) offers an appropriate alternative to building either a new programming language or a library of inheritable components. By constructing the smallest, most generic framework interface possible, we could sidestep the language pitfall of imposing a restrictive worldview on all derivative models. Because the framework is embedded in a standard object-oriented language, implementing new protocols or network elements is straightforward, as is simple black-box submodel manipulation.

The scalable simulation framework is such a modeling framework. The SSF application programming interface (SSFAPI) is the next-generation core-modeling interface; it provides a compact, high-level target for simulator implementers, hiding all details of simulator internals (threads, processors, event queues, and synchronization) from the modeler.

SSFAPI defines just five base classes: Entity, inChannel, outChannel, Process, and Event (see the “SSF at a glance” sidebar). These five classes form a self-contained design pattern for constructing process-oriented, event-oriented, and hybrid simulations.

The SSF standard has both C++ and Java bindings, which resemble each other as closely as possible in both syntax and semantics so that model components might be ported from one to the other. An SSF model is a standard C++ or Java program that derives new component classes that extend Entity, Event, and Process and uses the framework to establish channel mappings and deliver events.

Heuristic parallelization

The modeler might guide the framework in realizing available concurrency by specifying which entities are coaligned (tightly coupled) in model time and the minimal transmission latencies associated with channels connecting loosely coupled component collections. These heuristics are sufficient to allow a clever SSF implementation to partition a given model over available processors and perform appropriate model-time event-exchange synchronization

SSF at a glance

The scalable simulation framework provides a maximally compact interface for building discrete-event simulations; it contains just five core classes, with just a couple dozen methods altogether.

Entity is the base class for all simulation components; it serves primarily as a container mechanism for defining alignment relations among a model’s pieces. Entities that the modeler has coaligned will presumably interact at close quarters through event exchange on channels with low or zero intrinsic minimal delay. The underlying simulator might take this presumption into account when mapping entities to processors.

Event is the base class for the quantum of information exchange.

InChannel and *OutChannel* are communication endpoints for event exchange; each instance of *InChannel* and *OutChannel* belongs to a specific Entity. SSF supports multicast in-channels (many to one communication) as well as multicast out-channels (one to many) and bus-style channel mappings (many to many). Each *OutChannel* might have associated with it an intrinsic minimal transmission delay (ascrivable, for example, to device latencies or transmission delay on a simulated link), which is automatically added to the per-write delays of individual events sent on it.

Process is the base class for describing Entity behavior. Each instance of *Process* is normally associated with a specific Entity; it might wait for input to arrive on the channels of that Entity, wait for some amount of simulation time to elapse, or do both in turn. The simplest *Process* waits for an event to arrive on a channel, responds to it, and then goes back to sleep. The binding of *Process* to Entity is not tight; a *Process* might wait on channels or access methods of all Entities that are coaligned with the *Process*’ nominal owner.

These five base classes are truly generic—sufficient to model not only telecommunications networks, but also any other domain that can be described as a collection of objects that communicate via event exchange. As described in this article, additional component layers (SSF packages) are positioned atop SSF to model specific domains. These can be regarded as derivative frameworks that provide their own, more specific metaphors to extend those offered by SSF. This strategy promotes independence of models from the simulation fabric, and of the simulation fabric itself from the specifics of parallel discrete-event simulation engines. This is a prerequisite for building models of the size and complexity required to support Internet research.

without further modeler guidance.

Specifically, an SSF model might be compiled against sequential, shared-memory parallel, and distributed-memory clustered SSF implementations without rewriting. SSF models are standard C++ or Java programs that use or derive from the five core classes. To move an SSF model from the desktop to a parallel enterprise server, therefore, the modeler simply recompiles the model using g++ or javac, and links with a high-performance SSF library such as DaSSF (the Dartmouth SSF implementation), which we describe later.

Modeling layers

Minimizing the amount of time needed to start on a new modeling project requires that the programming interface be as small and simple as possible. Even more important for portability and scalability, a small API encourages construction of multiple simulator implementations and facilitates the process of verifying their conformity to the SSF specification.

Because SSF is general, and the interfaces are simple (in class count and method count), we tailor it to more specific modeling domains such as network analysis by augmenting it with a layer of domain-specific, open-source standard components known as SSF packages. Using the know-how acquired with scalable modeling in earlier projects, these packages abstract the software patterns with which modelers actually tend to approach new network-entity and protocol design.⁸ Not coincidentally, they incorporate model-design techniques that have been proven to give good parallel performance.

One package, **SSF.DBMS**, provides standard configuration-database support for building very large models. Another, **SSF.Net**, provides standard models for familiar network components: hosts, routers, LANs, and links. A third, **SSF.OS**, provides a standard class basis for building protocol graphs in the spirit of the x-kernel,⁹ together with implementations of standard protocol modules.

SSF components can be distributed as standard component libraries without releasing their source code. Because SSF models can be linked with any compliant SSF library after recompilation, developers are insulated from dependence on any single simulation software supplier. The net effect is to give modelers high confidence that their models will run unmodified on multiple platforms with predictable performance and reproducible results.

Implementations

Java SSF, developed at Cooperating Systems, supports prototype development of moderately large network models. What Java lacks so far in mature performance it makes up for in portability, ease of use, and standard library support for threads, networking, and graphics. Using native threads to implement multiple simulation timelines, Java SSF has demonstrated scalability under Solaris on multiprocessor Sun servers. CSSF, also developed at Cooperating Systems, is a sequential C++ implementation of SSF. It currently runs on Linux/x86 and Sun/Solaris platforms, and additional Unix ports are planned.

For high-performance simulation, users can compile their models against DaSSF, a C++ implementation of the SSF API developed at Dartmouth College.

DaSSF parallel simulator— implementation techniques

DaSSF was designed from the first to provide scalable high performance on very large models using parallel processing. Recent demonstrations show incredible performance on multiprotocol big Internet models with tens of thousands of complex network entities: over one million packet events per second. In other experiments designed to test the DaSSF framework, over three million simple network entities have been simulated in parallel, yielding event processing rates that scale linearly with the number of processors and remain constant with increasing problem size, up to the available memory limit.

DaSSF achieves its scalability and high performance with a few techniques. To conserve memory, it implements the threads called for by SSF at the source-code level, meaning that SSF models are transformed into C++ programs that do not use standard user-thread packages. Instead they implement threaded functionality within the context of a single-threaded process. This puts memory use under our control; we allocate only what is needed to save thread state, when it is needed, and no more. A simple DaSSF thread might consume only a few tens of bytes for state, whereas standard threads packages require orders of magnitude more. Self-threading also gives us complete control over scheduling mechanisms and overhead. We don't have to build a temporal scheduler on top of some other package's thread scheduler, nor do we pay the price for thread-scheduling features we don't use. For random number generation, in addition to a stan-

standard Lehmer generator, DaSSF provides a high-quality random number generator—the Mersenne Twister by Takuji Nishimura—with a period of $2^{19,937} - 1$.

DaSSF synchronizes processors conservatively, using a technique that has been mathematically proven to scale.¹⁰ The method is globally synchronous, meaning that all processors synchronize periodically to exchange events. Within a synchronization period, a processor might execute without any additional synchronization with any other processor. All overhead associated with synchronization is limited to establishing the synchronization window. This cost depends only on the frequency of synchronization; being independent of the model size or model behavior, it allows a fixed cost to amortize over the considerable amount of computation one typically finds in a big Internet simulation. For instance, suppose the processors synchronize with each other once per 10 simulated milliseconds. This can be accomplished if the model is partitioned and assigned to processors so that the only links the partition cuts are those at the scale of long-distance trunk lines. Consider a model with 10^6 hosts, each of which offers one packet per 10 ms; each packet requires 10 events to be executed. This yields 10^7 events to be executed, in parallel, every synchronization step. Even if the event granularity is as low as one microsecond per event, it is an ample workload for more than 100 processors on either a shared-memory or message-passing parallel computer.

The DaSSF simulator is highly portable. The current implementations encompass SGI IRIX, Sun Solaris, DEC/OSF, Linux, and Windows.

A case study

Choosing an appropriate Internet model depends very much on objectives. To study the dynamics of IP traffic over wired networks, we can work at the logical IP-network level. The model then consists of realistic IP hosts and routers, abstracted LANs, and wide-area links, but ignores the details of link-level transmission beyond gross characterizations of the bandwidth and transmission delays on long-distance links. Consequently, the simulation quantum is an IP packet.

With the exception of certain major service providers, real network maps are conspicuous in their absence from the literature, either for reasons of security or embarrassment. We rely on network-generation tools and intuition about

the nature of networks to produce subject network topologies. We start by augmenting the output from the *gt-itm* graph-generation tool with autonomous system (AS) numbers for each node, attach LANs to selected nodes, assign a set of hosts to each LAN, and perform assignment of CIDR-compliant IP addresses to each network and host interface.¹¹ We also augment network graph edges with bandwidths and delays appropriate to the links they represent. Most of these operations have been automated.

Simulation entities are then constructed (routers, hosts, and LANs) and configured with traffic sources and Internet protocols for routing and transport. The intradomain (OSFPPv2) and interdomain (BGP-4) routing protocols are implemented according to Internet standards documents. Finally, the connections between hosts and LANs, LANs and routers, and between routers are established and initialized. For the SSF performance evaluation, we configured a network model family as follows: Each non-transit AS contains 100 routers plus 700 hosts (10 per LAN). Every AS-internal router is on some OSPF source-destination path, and each AS has one designated BGP router, which has exactly one link to a single router in the transit AS. A transit domain models connectivity between autonomous systems: a random graph of 100 routers, with strong connectivity (individual link probabilities of 0.75). At most, one AS is attached to each transit domain router; the SSF entities modeling a single AS are coaligned, with the intent that each will be allocated to a single processor if resources permit. Figure 1 shows a graph of a 100-router AS.

Links within an AS model fast Ethernet (100 Mbps, nominally 9,000 IP packets per second at full capacity), and interdomain links model OC-48 speeds: 2,488 Mbps, or 207,000 IP packets per second. We assume transmission delays of 1 ms between the OSPF routers within each AS, and 10 ms between BGP routers (on links into and within the transit domain).

We consider two traffic scenarios, one for a lightly loaded network, and one for a heavily loaded network. Each host runs a stochastic *on/off* process generating IP packet trains, with *on* and *off* periods alternating with durations given by independent random variables. In the *off* period, the host sleeps for an exponentially distributed interval with a two-second mean for the light load, and a 0.25-second mean for the heavy load. In the *on* period, the host emits a train of packets at 1-ms intervals; the number of

packets in the train is Pareto-distributed, with an average of 101 packets. We used the inverse-transform method on the complementary cumulative distribution function $Prob\{X > t\} = 1/(t^a)$. In this case $a = 1.01$, and the formula used to sample packet-train length is

$$(int) (1.0/pow(U(), 1.0/1.01) ,$$

where $U()$ samples a uniform $(0, 1)$ random number stream.

All the packets in a single burst are directed to the same destination host, with 0.95 probability the host is in the same AS. Hosts within the chosen AS are chosen uniformly at random. A packet traverses between five and six hops, on average. Router output is buffered on each outgoing link, with the buffer size equal to the product of the link bandwidth and delay. Under the light-traffic model, the long-time average offered load on intra-AS links is about 500 packets/sec for a capacity of 9,000 packets/sec; in the heavy-traffic model, the average offered load is 4,000 packets/sec.

To evaluate scalability, we constructed a sequence of such models consisting of 10, 20, 50, and 100 ASs and a transit domain; they contain 8,100, 16,100, 40,100, and 80,100 hosts and routers, respectively. We consider two measures of performance that relate to network modelers: the model-time advance rate and the packet-event rate.

Model-time advance rate is defined as the ratio of model-internal time (called logical time in the simulation community) to the time elapsed during simulation execution (commonly called the wallclock time). A rate of one or greater indicates a capability to perform real-time simulation.

A packet event is any packet production, forwarding, or reception simulator action; the packet event rate is the number of packet events executed per second of wallclock time. This indicates the simulator's raw horsepower, in terms understandable to the network modeler. Execution time depends on

implementation overheads hidden in the measured execution time; this metric can reasonably compare different simulators' performance.

We conducted all experiments on a 14-processor Sun Enterprise 4000 with 250-MHz ultra-Sparc CPUs, running Solaris 2.6. In parallel simulations, the network is partitioned in a simple way by dividing the whole ASs among the processors as evenly as the numbers permit.

Figure 2 illustrates the model-time advance rate under light and heavy traffic. For a sufficiently large number of processors, our simulations of 8,000- and 16,000-node models run faster than real time under light traffic conditions; that is, model time advances faster than the wallclock time during simulation.

Figure 3 illustrates the packet-event rate under heavy traffic. Performance approaches one million packet events per second, and the scalability is clearly evident, both with increasing model size and with an increasing number of processors (parallel speedup). Packet-event rate under light traffic is shown in Figure 4.

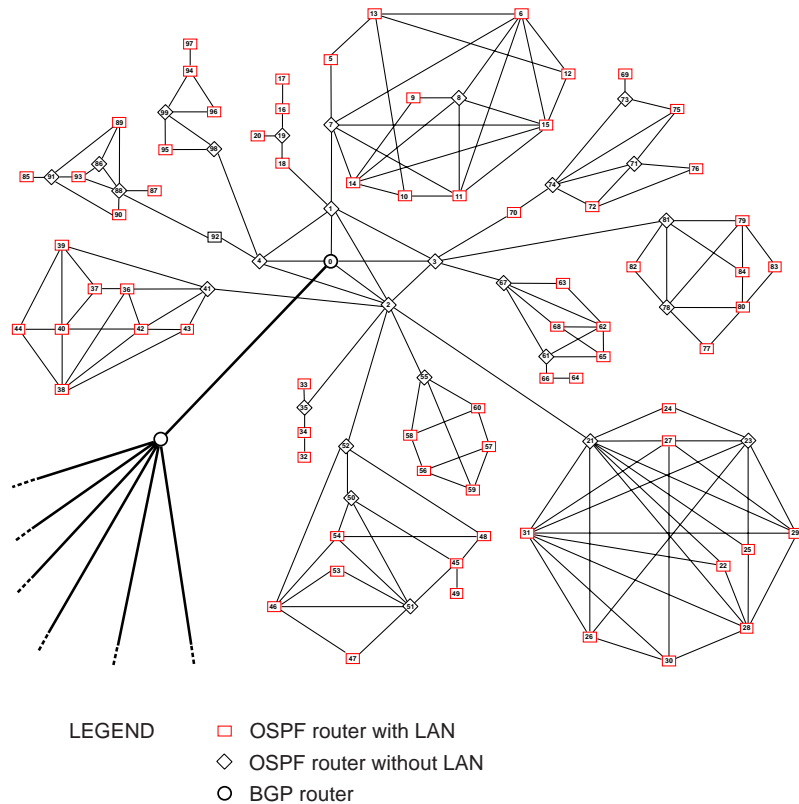


Figure 1: Topology of a single domain (autonomous system) used in the case study, containing 100 routers and 700 hosts. We have simulated networks composed of from 10 to 100 such domains (8,000 to 80,000 hosts and routers), interconnected via a transit domain with 100 BGP routers.

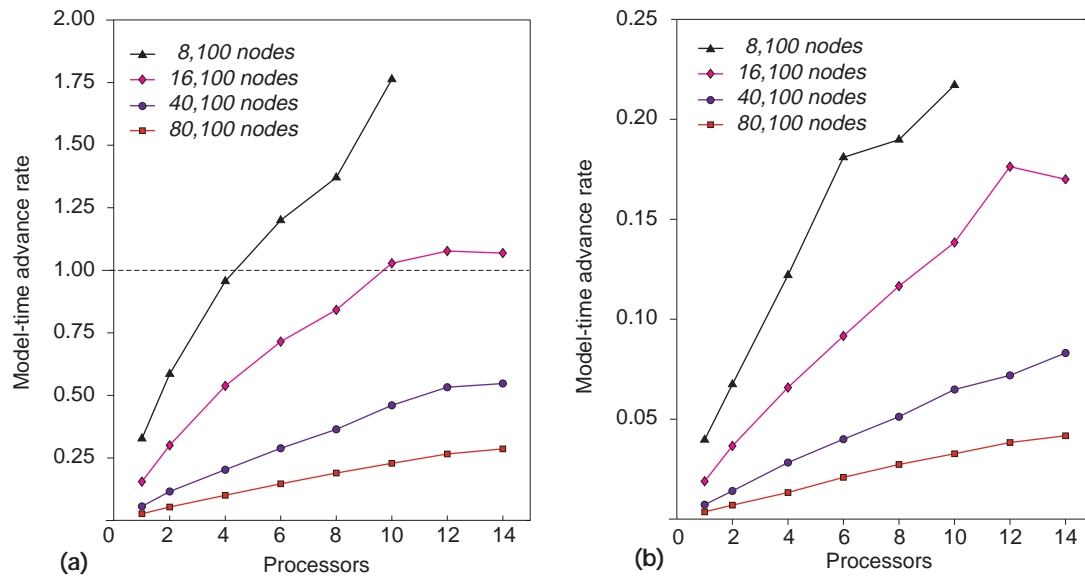


Figure 2. Model-time advance rate, equal to seconds of simulated model time advanced per second of execution (wallclock) time, for various network sizes: (a) light traffic conditions, and (b) heavy traffic conditions. Note that for the rate greater than one, the simulated model runs faster than real time.

Additionally, the performance the packet-event rate measures is comparable for light and heavy traffic conditions, and it actually slightly improves with heavier packet traffic. On problems of this size, the simulator's delivered performance is largely insensitive to network load conditions, in contrast to many other studies of parallel sim-

ulation where performance is quite sensitive to offered load. We are reaching almost one million packet events per wallclock second, using all 14 available CPUs. Even on the largest models, the model-time advance rates are within a manageable factor away from real-time execution. Moore's law, or larger machines, or both together

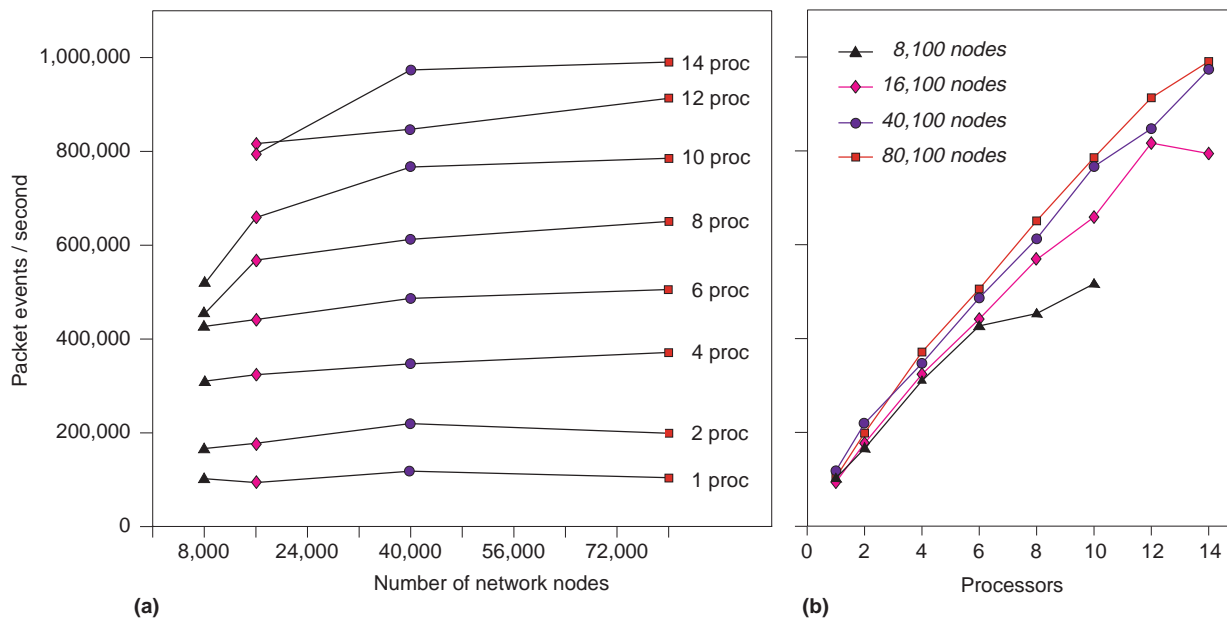



Figure 3. Two complementary views on scalability with model size and with the number of processors under heavy traffic conditions: (a) Simulated packet events per wallclock second plotted versus the number of network nodes, for different numbers of processors used; note the stability of execution rates with increasing model size. (b) The same data plotted versus the number of processors, for different model sizes; note the linear speedup for larger models.

will soon bring real-time execution within reach even for heavy traffic and larger models.

While our simulator can deliver the computational output we need, a great deal of work remains. We prototyped the framework for database-assisted self-configuring model classes and are beginning to explore its potential to increase heterogeneity in models. We are working to improve resource-management automation within DaSSF itself, and are looking toward the challenges posed by executing in a distributed-memory environment. We hope to stress DaSSF by running it on the very-large-scale parallel machines that exist under the ASCI project (Blue Mountain and Blue Pacific).

We are just beginning to exploit the capabilities of our tool in pure networking research. Our early targets include developing models in comparison and conjunction with experimental data to evaluate traffic models. The empirical model of the HTTP/TCP Web traffic, Surge, has been ported to SSF.¹² We will use our simulation capabilities to look for congestion phenomena and its causes, looking first at correlated TCP losses. We are interested in the problem of convergence of various traffic statistics in the presence of long-range dependent traffic.⁹ This is a critical problem—to evaluate simulation output we have to understand what it represents mathematically.

Furthermore, because we have the computational capacity to simulate them at the finest levels of detail (direct-execution simulation of the protocol code itself), we are in a position to develop simpler models and study the trade-offs between model fidelity and model behavior on a large scale.

This is an exciting time: synergy of research on simulation techniques, protocol modeling (such as the work of our colleagues in project VINT (<http://netweb.usc.edu/vint>)), and empirical traffic modeling is opening a new era in network research. With scalable tools to build realistic-scale Internet models in hand, we can ride Moore's law as far as it will take us, opening up new vistas of exploration in networking. 

Acknowledgments

We particularly wish to acknowledge the effort put into this project by our students Xiaowen Liu, Anna Poplawski, and Brian Premore at Dartmouth, and Phillip Kwok and Bryan Cooley at DIMACS.

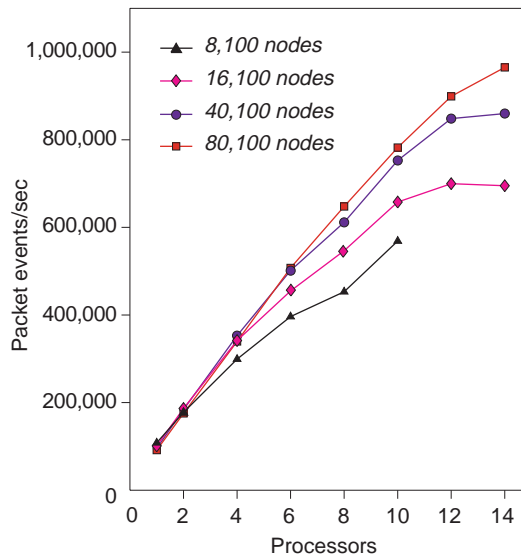


Figure 4. Simulated packet events per wallclock second plotted versus the number of processors under light traffic conditions for different model sizes. Note the linear speedup for larger models, leveling off at the larger number of processors when the work per processor diminishes.

We acknowledge Mark Crovella from Boston University for valuable discussions on defining the SSF big Internet models. Richard Fujimoto and Kalyan Perumalla from Georgia Tech contributed many helpful discussions in the early stages of defining the SSF API. SSF owes much of its strength to our earlier experience with the modeling framework TeD, developed by Kalyan Perumalla and others, and to feedback from dozens of network modelers using TeD and other simulators.

This work is partially supported by DARPA Contract N66001-96-C-8530, NSF Grant NCR-9527163, and NSF Grant ANI-9808964.

References

1. W.E. Leland et al., "On the Self-Similar Nature of Ethernet Traffic" (extended version), *IEEE/ACM Trans. Networking* 2, Vol. 2, No. 1, 1994, pp. 1–15.
2. V. Paxson and S. Floyd, "Wide Area Traffic: The Failure of Poisson Modeling," *IEEE/ACM Trans. Networking*, Vol.3, No. 3, June 1995, pp. 226–244.
3. W. Willinger and V. Paxson, "Where Mathematics Meets the Internet," *Notices of the Amer. Mathematical Soc.*, Sept 1998, pp. 961–970.
4. A. Feldman, A.C. Gilbert, and W. Willinger, "Data Networks as Cascades: Investigating the Multifractal Nature of Internet WAN Traffic," *Proc. ACM/Sigcomm*, ACM Press, New York, 1998, pp. 25–38.
5. C. Labovitz, G.R. Malan, and F. Jahanian, "Internet Routing Instability," *IEEE/ACM Trans. Networking*, Vol. 6, No. 5, Oct. 1998, pp. 515–528.
6. V. Paxson and S. Floyd, "Why We Don't Know How to Simulate the Internet", *Proc. Winter Simulation Conf.*, IEEE Press, Piscataway, N.J., 1997, pp. 1037–1044.

7. M.E. Crovella and L. Lipsky, "Long-Lasting Transient Conditions in Simulations with Heavy-Tailed Workloads," *Proc. Winter Simulation Conf.*, 1997, pp. 1005–1012.
8. K. Perumalla, R. Fujimoto, and A. Ogielski, "TeD—A Language for Modeling Telecommunications Networks," *Performance Evaluation Review*, Vol. 25, No. 4, 1998, pp. 4–11.
9. S.W. O'Malley and L.L. Peterson, "A Dynamic Network Architecture," *ACM Trans. Computer Systems*, Vol. 10, No. 2, May 1992, pp. 110–143.
10. D.M. Nicol, "Scalability, Locality, Partitioning, and Synchronization in PDES," *Proc. Workshop Parallel and Distributed Simulation*, IEEE Computer Society Press, Los Alamitos, Calif., 1998. pp. 4–11.
11. E. Zegura, K. Calvert, and M. Donahoo, "A Quantitative Comparison of Graph-Based Models for Internet Topology," *ACM/IEEE Trans. Networking*, Vol. 5, No. 6, Dec. 1997, pp. 770–783.
12. P. Barford and M.E. Crovella, "Generating Representative Web Workloads for Network and Server Performance Evaluation," *Proc. Performance 98/ACM Sigmetrics 98*, ACM Press, 1998, pp. 151–160.

James H. Cowie is the director of technology development at Cooperating Systems Corporation, Deering, New Hampshire, where he has led software development projects for high-performance compilers and runtime systems, distributed factorization, and parallel discrete-event simulation. He received a BS in computer science from Yale University, and cofounded Cooperating Systems. Contact him at Cooperating Systems Corp., RR1 Box 201-B, Deering, NH 03244; cowie@cooperate.com.

David M. Nicol is a professor of computer science at Dartmouth College. He received a BA in mathematics from Carleton College and a PhD in computer science from the University of Virginia. He serves as Editor-in-Chief of the *ACM Transactions on Modeling and Computer Simulation*, and is a Senior Member of the IEEE. He has published extensively on topics in performance analysis and parallel processing. Contact him at the Dept. of Computer Science, 6211 Sudikoff Laboratory, Dartmouth College, Hanover, NH 03755; nicol@cs.dartmouth.edu.

Andy T. Ogielski is a research professor with a joint appointment at DIMACS and WINLAB, Rutgers University. He was a member of the technical staff in physics research and in mathematics research at Bell Laboratories, then joined Bellcore, where he was a director of parallel computing and algorithms research, and later of Internet communications research. He received his MSc in physics and his PhD in theoretical physics, both from the University of Wroclaw, Poland. He has conducted research, written software, and published in networking, applied mathematics, scientific computing, and physics. His current research focuses on modeling, analysis, and simulations of networks. Contact him at DIMACS Center, Rutgers Univ., 96 Frelinghuysen Rd., Piscataway, NJ 08854-8018; ato@winlab.rutgers.edu.