1999 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'99), June 28 - July 1, 1999, Las Vegas, Nevada.

Towards Realistic Million-Node Internet Simulations

James Cowie Cooperating Systems Corp.
Hongbo Liu Rutgers University
Jason Liu Dartmouth College
David Nicol Dartmouth College
Andy Ogielski DIMACS Center, Rutgers University

Abstract

This paper describes the Scalable Simulation Framework (SSF)*, a discrete event modeling API designed for the construction and simulation of very large networks.* SSF *can execute detailed simulations of complex topology networks with a million or more concurrent TCP/IP flows. We describe the overall architecture of* SSF*, the architecture of the network modeling layers above the basic API structure, and discuss preliminary performance figures for up to 100,000 concurrent TCP flows in a test topology.*

1 Introduction and Rationale

After several years of research, we now stand on the threshold of being able to model the Internet at realistic scales and levels of detail. We have already described our success in modeling networks with complex multi-domain topologies and large diameter, consisting of over 80,000 hosts and routers transporting Pareto distributed UDP/IP packet flows [1]. These studies have given us confidence that our techniques for modeling networks at the IP packet level are sufficiently scalable to tackle million-node networks and larger.

However, the UDP/IP traffic models we have studied in the past represent a significant simplification over the real Internet: their behavior does not incorporate network feedback on traffic sources. The computational workload of a UDP traffic model lends itself to accurate prediction of its scaling characteristics; this is not the case with models that include TCP traffic flows.

TCP flows interact strongly with each other when network congestion results in packet delays and losses, which in turn trigger flow control and congestion avoidance actions of autonomous TCP sessions. This *adaptive behavior* reduces TCP transmission rates, and may close connections in extreme cases. In this paper, we analyze some of the subtle challenges that adaptive workloads pose for scalable simulation of very large, realistically detailed Internet models, and describe how the Scalable Simulation Framework can be used to meet those challenges.

2 Scalable Simulation Framework

SSF is a specification of classes and methods (with Java and C++ bindings) for the description and simulation of large, complex systems. There are presently two independently developed C++ implementations, and one Java implementation. SSF models are standard-compliant C++ or Java programs; they hide all details of simulator internals (threads, processors, event queues, and synchronization) from the modeler.

The SSF API contains five base classes that form a self-contained design pattern for constructing process-oriented, event-oriented, and hybrid simulations. The **Entity** class is essentially a container of model state, of **Processes** that operate on that state, and of **inChannel**s and **outChannel**s that identify the endpoints of communication channels. **Event**s are communication objects that are passed between entities. SSF processes can be thought of as threads, and the SSF kernel as a thread scheduler in simulation time.

As an object framework, SSF offers significant advantages over custom-designed simulation languages: SSF models can be compiled using existing compilers, debugged using existing debuggers, and extended simply by deriving new classes. Additional component layers can be built atop SSF to model specific domains such as radio propagation, factory automation, game theory, or IP networking. These *derivative frameworks* add their own, more specific metaphors to those offered by SSF. In the next section we describe one such derivative framework for scalable Internet modeling.

3 SSFNET: Scalable Internet Research Tools

SSFNET is the first collection of SSF-based models for simulating Internet protocols and networks. Available in C++ and Java bindings, the SSFNET packages provide classes that can be used directly or extended to construct very large network models. The SSFNET libraries include component models for network elements (hosts, routers, network interface cards, local area networks) and network protocols (currently IP, UDP, TCP, BGP, and OSPF).

SSFNET incorporates several hard-won lessons about supporting extremely large models and extracting high performance from them. First, intelligent support for model configuration is critical to getting results that make sense, regardless of how fast they arrive. Second, large network models are best built and managed using compositional approaches that intelligently glue together large numbers of smaller, simpler models. Finally, achieving higher simulation performance depends intimately on understanding the interaction among simulation components, and interpreting their collective computation and communication requirements in terms of SSF's underlying performance model.

Self-configurable models

SSFNET models are *self-configuring* — that is, each SSFNET class instance can autonomously configure itself by querying a parameter database, which may be either locally resident or available over the Web. Configuration data is hierarchically structured, meaning that parent entities pass a fragment of their own configuration information to their children. The configuration database contains its own hierarchical *schemas* – information about the acceptable values and nested structures that may be stored in and retrieved from the database.

At scales approaching millions of simulated components, the use of object database technology for configuration management becomes essential. Without such a strategy, it is extremely difficult to consistently and efficiently supply parameters to a large model over the course of an experimental series. A structured database approach also supports limited automatic verification of the structure and typing of configuration data, eliminating a primary source of modeling errors.

Model composition from components

SSF models use a simple hierarchical attribute tree notation (DML, or Domain Modeling Language) to specify a tree of configuration parameters for each of the components that makes up a large model.

Through references to a database of standard definitions, modelers can rapidly compose arbitrary network configurations. Each network configuration database becomes a valuable product in its own

```
Net [
  router [ id 1 graph [ProtocolSession [name ip use SSF.OS.IP]]
           interface [ num 1 ip "10.0.1.1/30" ptp [ router 2 interface 1]]
  ]

  LAN [  id 1  router [id 1 interface 2]
    hosts [range [from "10.0.0.1/27" to "10.0.0.9/27"]
      graph [
        ProtocolSession [name server use SSF.OS.TCP.simServer port 10 interval 0.1]
        ProtocolSession [name socket use SSF.OS.TCP.tcpSocket]
        ProtocolSession [name tcp use SSF.OS.TCP.tcpSessionMaster]
        ProtocolSession [name ip use SSF.OS.IP]]
    ]
  ]
  # ... client side of broom network and protocol parameters omitted for brevity
]
```

Figure 1: Partial DML configuration database specification for a broom network

right, distinct from the executable C++ or Java code that implements each component. Configuration databases can be shared, maintained, and extended independent of a particular simulation kernel or parallel hardware environment.

To understand how self-configuration and model composition interact, consider the small extract from a DML configuration database shown in Figure 1. It describes a simple topology of a "broom network" (see Figure 2) that has been often used in simulation studies of TCP [2]. The DML specification for the broom starts with a description of the protocol graphs used for the routers, TCP clients, and TCP servers that make up the network, as shown in Figure 1.

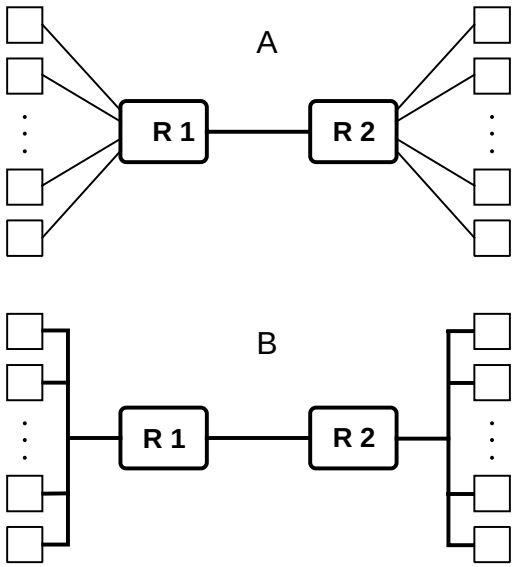

Figure 2: Topology of "broom" networks. A: with point-to-point links, B: with LANs.

Self-configurable models

A DML specification like this one consists of a list of key/value pairs; a value may consist of a list of subattributes, delineated by brackets. The top-level node, in this case `Net`, corresponds to a matching `Net` class from the SSFNET library. `Net` is self-configuring, meaning that it implements a particular library interface (`Configurable`) that allows it to perform database queries to obtain its own parameters.

The `Net` loads the database, and learns from its attribute list that it must in turn create instances of `Router` and a `LAN`, both of which are standard self-configurable SSFNET classes, and connect their network interface cards appropriately. Once constructed, each instance receives a reference to its particular part of the configuration database, so that it can self-configure in turn. This continues with sub-entities configuring sub-sub-entities, until the entire model has been recursively constructed and configured.

Network Protocol Graphs and Network Interface Models

`Routers` and `Hosts` are parameterized by particular network protocol graphs, using a design pattern reminiscent of the x-kernel [3]. Each host contains a protocol graph, made up of individual protocol sessions which push messages up and down in response to application activities or packet arrivals on the host's network interfaces. Figure 3 illustrates the logical structure of the protocol stack in the broom network example.

SSFNET includes a new "clean room" implementation of TCP that contains all TCP states and all major flow and congestion control mechanisms required in the RFCs, including slow start, congestion control, Jacobson's RTT estimation, the Karn/Partridge algorithm, exponential retransmit timer backoff, adaptive acknowledgment, and fast retransmit. It has been designed to be easy to extend and modify: e.g., send-window, receive-window, and incoming message demultiplexing are all provided in distinct classes.

In SSFNET, as in the real world, TCP operates on top of IP; IP operates on top of one or more pseudoprotocols representing configured network interface cards. Each card (or `NIC`) maintains a pair of buffered SSF channels for exchanging IP packet events with the outside world. A NIC may be connected to another NIC of the same link type, or to a LAN, and supports self-configuration options for physical link characteristics, packet flow buffering, flow interleaving and scheduling.

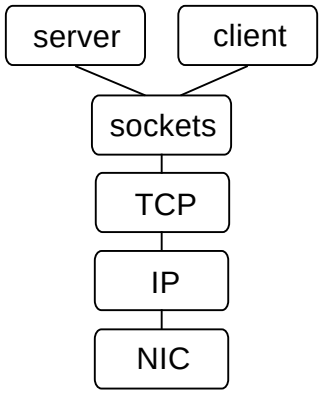

Figure 3: SSFNET protocol graph used in the experiments.

4 Performance

In this section, we review two sets of experiments intended to help understand how to design scalability experiments for TCP network simulation. The tests were performed using two different implementations of the SSF libraries, one in Java and the other in C++, running on very different parallel architectures.

Our earlier studies [1] showed scalable performance on large problems on the order of 100,000 network hosts and routers and complex network topologies with UDP/IP packet traffic. However, TCP is a much more complex protocol, whose session workload strongly depends on network feedback (packet delays and losses). This *adaptive behavior* reduces TCP transmission rates, and may close connections in extreme cases.

Consequently, it becomes much more difficult to evaluate scalability of parallel simulation of many TCP flows, as increasing the computational workload in hosts by opening more TCP connections eventually leads to network congestion, which silently reduces the workload. To illuminate this phenomenon we have chosen to begin by studying a simple network topology, where impact of model parameter variations on simulator performance can be meaningfully analyzed. As we shall see, interaction among multiple TCP flows can produce highly nontrivial simulation performance characteristics, even on small topologies. After we understand the scalability mechanisms at work, we can apply the general principles derived from these experiments to successively larger and more complex networks.

Simulation Performance Metrics The simulation literature uses a number of different ways of describing simulator performance, such as "event-rate". We consider two metrics that are natural for network modelers: the *achieved client-server data throughput*, and the *total IP packet event rate*. The achieved throughput is a measure of actual TCP connection efficiency, and represents the total number of data bytes *received* by their intended destination applications, divided by the total internal (simulated) time of the simulation. The total IP packet rate represents the total number of IP packet events divided by the total execution (wallclock) time. For the broom networks shown in Figure 2 we count packet events as follows: for LAN-based brooms, there are five events for each delivered IP packet; for broom networks with point-to-point links there are three events.

Experiment 1: Java SSFNET

We first consider a simple broom network with two hosts on each side, one client and one server, (Figure 2 A). Each client opens N simultaneous TCP connections to the other side's server (staggered slightly in open time) and then requests a large, fixed-size (512MB) data transfer in each connection.

The network, specified by a DML configuration database, includes optional `timeline` attributes for each host and router. These attributes symbolically partition the network into alignment groups; each alignment group is then aligned by the framework to a distinct physical processor. This allows the modeler to adjust the degree of parallelism by simply editing the model configuration file, so that no recompilation is necessary. The Java simulation was then run on a moderately loaded Sun Enterprise 4000 multiprocessor server at Rutgers, running Solaris 2.7 and Sun's JDK 1.2.

Java SSF uses a window-based conservative synchronization algorithm to control the progress of a given collection of *logical timelines*. Using `timeline` attributes in the configuration database, one can choose to run the broom network serially, with all hosts and routers assigned to the same simulating processor, or in parallel. The simplest parallel partitioning divides the broom into two symmetric partitions, with one router and two hosts in each half. A slightly more elaborate partitioning strategy assigns each client and server host its own processor as well. Because this simple model is bilaterally

symmetric, the primary goal is to understand under which conditions it is reasonable to expect a twofold parallel speedup over the unpartitioned sequential version.

Evaluating parallel performance Using Java SSF, we see the following results, for three choices of interconnection bandwidth (Table). The first column gives the total number of simultaneous TCP connections originating from the two client nodes in the broom. The final column summarizes the model speed in terms of thousands of IP events per second — dividing the total number of IP packets sent and received by all network interfaces by the total wallclock running time.

TCP conn	Net bw Mb/s	1 Proc seconds	2 Proc seconds	P1/P2	6 Proc seconds	P1/P6	Tput Mb/s	Pkt drops	IP evt/s
16	10/100	26.50	20.47	1.29	–	–	9.7	0	27,600
32	10/100	27.62	20.97	1.32	–	–	9.7	0	26,900
64	10/100	29.75	22.21	1.34	–	–	9.7	0	25,500
128	10/100	30.01	22.38	1.34	–	–	7.6	4,700	20,700
256	10/100	35.30	24.94	1.42	–	–	6.3	7,935	16,000
16	100	66.72	45.92	1.45	35.34	1.89	52	0	31,500
64	100	140.04	89.30	1.57	68.11	2.06	95	0	29,400
256	100	172.59	110.67	1.56	77.43	2.23	95	0	23,600
1024	100	228.67	143.11	1.60	100.80	2.27	72	20,022	15,900
4096	100	506.08	292.05	1.73	238.58	2.12	56	45,795	6,300
64	1,000	125.76	84.76	1.48	64.81	1.94	210	0	30,000
256	1,000	639.11	434.08	1.47	305.03	2.10	814	0	23,500
1024	1,000	1087.68	678.30	1.60	418.88	2.60	966	0	17,500
4096	1,000	2428.81	1321.36	1.84	1059.81	2.29	967	0	4,900
16384	1,000	–	–	–	2345.17	–	584	195,193	3,480

Table 1: Number of concurrent TCP connections, network bandwidth, execution times, parallel speedups, achieved client-server data throughputs (in simulated time), packet drop counts, and IP packet event rates (in wallclock time) for broom network simulation using Java SSF. As congestion appears, care must be taken to increase modeled link bandwidths, so that all simultaneous TCP connections contribute consistently to the computational load.

We start by modeling a broom whose backbone link offers a 100Mb/s symmetric bitrate, and whose client and server links operate at 10Mb/s. Network interface latencies are assumed to be a constant 1 millisecond, and the simulation runs for ten seconds of simulated time. We soon see that the 10Mb/s links to the servers are encountering congestion, and while the overall time to completion stays on the same curve, both the actual effective datarate and the model speed drop because of the overhead of TCP packet retransmission.

In the next phase, we therefore revise the model to use 100Mb/s links uniformly, holding all else constant. We also evaluate a new parallel partition in which clients, servers, and routers are all simulated by individual physical processors. By the time we simulate $N = 4096$ total simultaneous TCP connections, the 100Mb/s links are congesting again; this time, the backbone router link is to blame. TCP's congestion control algorithms are reducing the aggregate throughput between the broom network's clients and servers, and the network event rate suffers correspondingly. Without monitoring the actual workload offered by TCP in this situation, rather than the theoretical maximum workload created by the given number of simultaneous connections, one might be tempted to accept the speedups, which continue to improve, at face value. The overall IP packet rate in the final

column tells the real story, however: the experiment is degrading for lack of bandwidth.

In the third and final model revision, we move to gigabit links throughout the network, and cut the runtime to 5 seconds, holding all else constant. As the link bandwidth-delay product increases, so too does the number of stored events at any given time. When the memory consumed by the simulation event queues grows as a result, Java garbage collection starts to influence the running time.

Stalking the Adaptive Workload. This example shows that Java SSF eventually approaches twofold speedup on the simple broom networks, especially if there are a large number of simultaneous network connections from both sides, and if the links are not saturated to the point that TCP connections begin to back off and, ultimately, close prematurely for lack of network response. (In the largest runs, we use an unrealistically large number of connections over a small number of hosts, to emulate the per-partition workload intensity that one might experience given a smaller number of connections per host within a much larger test network.)

In general, getting parallel speedup from a conservatively synchronized discrete-event kernel requires that the aggregate workload assigned to each processor (here, the generation, routing, and processing of packets on each end of the broom) should easily outweigh the cost of conducting the necessary periodic synchronization and event exchange between processors. This will be the case if the model contains a sufficient number of TCP connections whose endpoints are uniformly distributed throughout the subject network, provided that the network links have adequate capacity to support the specified amount of traffic without severe congestion.

The broom network example amply illustrates a potential pitfall in evaluating parallel performance of a TCP simulation. TCP offers an adaptive workload — by design, if the workload gets too heavy, TCP will back off and wait to retransmit to avoid making a congested situation worse. This makes it inherently difficult to force simulated TCP to scale. Just when the number of simultaneous connections has resulted in enough exchanged packets per synchronization window to make parallel execution worthwhile, TCP may scale back its transmission rate, link utilization drops, and parallel speedup declines.

Fortunately, SSF and SSFNET support low-overhead instrumentation strategies that allow the modeler to instrument network components. When choosing an network partitioning strategy, it's critically important to have some way of systematically identifying and "modeling around" the effects of rate adaptation (even in simple TCP networks) in the presence of congestion.

Experiment 2: C++ SSFNET

The second performance experiment asked the question: how many concurrent TCP connections can SSF simulators execute on today's machines? For this set of tests, we used Dartmouth's SSF simulator (DaSSF) to run the C++ version of SSFNET on Dartmouth's SGI Origin 2000. DaSSF's computational core has previously been shown to be scalable [4], and to yield high performance. There, scalability in problem size was demonstrated by observing that throughput remains constant as the problem size is increased, showing that all the internal mechanisms of DaSSF resist performance degradation as a function of workload.

We now examine simulator performance on the LAN-based broom networks, B in Figure 2, by varying the number of hosts of the broom and the bandwidths of the LANs and the point-to-point link between the routers. LANs are modeled as switched, not shared, and offer symmetric bandwidth to all clients.

The traffic generated for these experiments had each host transmitting (on average) one IP packet per 65 milliseconds of simulation time. The network bandwidth is sized so that packet losses are

TCP conn	Net bw Mb/s	TCP pkt/s delivered	IP evt/s
1	10	18,612	93,060
10	10	22,383	111,915
100	10	20,903	104,515
1	100	18,877	94,385
10	100	22,439	112,195
100	100	21,167	105,835
1,000	100	14,410	72,050
10,000	1,000	9,774	48,870
100,000	10,000	7,210	36,050

Table 2: TCP packet receipt rates and IP packet event rates (in wallclock time) for broom network simulation using Dartmouth's C++ SSF.

minimal. Table 2 summarizes the results; both TCP and IP event rates are measured against execution (wallclock) time.

While there is some variation, the important thing to note is that the packet rates are sustained through two orders of magnitude of the smaller problem sizes. The packet rate drops off on the large networks, because the size of the event list is increasing, and the cost of accessing the event list is proportional to the log of the size. The very large "broom" models are unnatural in any case, given here just to illustrate that one hundred thousand concurrent TCP sessions can be simulated. An important element of SSF models is control over aggregation, which can serve to limit the size of the event list by distributing over more lists.

Conclusions

The Scalable Simulation Framework, and its derivative framework, SSFNET, make possible high performance detailed simulations of complex multi–protocol networks. We have described our general approach to this problem, and demonstrated the capability of simulating 100,000 concurrent TCP sessions. With parallel processing, large memory, and help from Moore's Law, simulation of 1,000,000 host networks will soon be a reality.

Acknowledgments

We wish to acknowledge the help of Myongsu Choe in performance testing.
This work is partially supported by DARPA Contract N66001-96-C-8530, and by NSF Grants NCR-9527163 and ANI-9808964.

References

[1] James Cowie, David Nicol, and Andy T. Ogielski. Modeling the Global Internet. *Computing in Science and Enginering*, 1(1):42–50, January 1999.

[2] K. Park, G. Kim, and M. E. Crovella. On the effect of traffic self-similarity on network performance. In *Proceedings of the SPIE International Conference on Performance and Control of Network Systems*, November 1997.

[3] N. C. Hutchinson and L. L. Peterson. The x-Kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, January 1991.

[4] D.M. Nicol. Scalability, Locality, Partitioning, and Synchronization in PDES. In *Proc. Workshop Parallel and Distributed Simulation*, pages 4–11, Los Alamitos, CA, 1998. IEEE Computer Society Press.