

Real-time On-line Network Simulation*

Boleslaw K. Szymanski, Yu Liu, Anand Sastry, and Kiran Madnani
Department of Computer Science, RPI, Troy, NY 12180, USA

Abstract

The complexity and dynamics of the Internet is driving the demand for scalable and efficient network simulation. Yet, parallelizing network simulation at packet level does not work efficiently and therefore do not scale to large number of processors because of tight synchronization between network components. To overcome this problem we designed a method in which a large network is decomposed into parts and each part is simulated independently and concurrently with the others. These parts exchange information periodically about the packet delays and drop rates along the paths within each part. Each part iterates over the selected simulated time interval until the exchanged information changes less than the prescribed tolerance.

Each decomposed part may represent a subnet or a subdomain of the entire network, thereby mirroring the network structure in the simulation design. The proposed method is independent of the specific simulator technique employed to run simulators of the parts of the decomposed network. Hence, it is a general method for efficient parallelization of network simulation based on convergence to the fixed point solution of inter-part traffic. The method can be used in all applications in which the speed of the simulation is of essence, such as: on-line network simulation, network management, ad-hoc network design, emergency network planning, large network simulation or network protocol verification under extreme conditions (large flows).

The paper describes the method, its implementation based on ns simulator, and its performance for sample communication networks.

1 Introduction

The major difficulty in simulating large networks at the packet level is the enormous computational power needed to

*This work was supported by the DARPA Contract F30602-00-2-0537. The content of this paper does not necessarily reflect the position or policy of the U.S. Government—no official endorsement should be inferred or implied.

execute all events that packets undergo traversing the network [5]. The usual approach to providing required vast computational resources relies on parallelization of an application to take advantage of a large number of processors running concurrently. Such parallelization does not work efficiently for network simulations at packet level because of tight synchronization between network components [3]. To overcome this difficulty, we designed a method described in this paper, in which a large network is decomposed into parts and each part is simulated independently and simultaneously with the others. Each part represents a subnet or a subdomain of the entire network. These parts are connected to each other through edges that represent communication links existing in the simulated network. In addition, we partition the total simulation time into separate simulation time intervals selected in such a way that the traffic characteristics change little during each time interval.

In the initial (zero) iteration of the simulation process, each part assumes on its external in-links either no traffic, if this the the first simulated interval (alternatively, the initial external traffic may be defined by the real-time measurements of the simulated network), or the traffic defined by the packet delays and drop rate defined in the previous simulation time interval for external domains. Then, each part simulates its internal traffic, and computes the resulting out-flow of packets through its out-links.

In the subsequent $k > 0$ iteration, the inflow into each part from the other parts will be generated based on the out-flows measured by each part in the iteration $k - 1$. Once the inflows to each part in iteration k are close enough to their counterparts in the iteration $k - 1$, the iteration stops and the simulation either progresses to the next simulation time interval or completes execution and produces the final results (see Figure 1).

More formally, consider a network $\Gamma = (N, L)$, where N is a set of nodes and L (a subset of Cartesian product $N \times N$), is a set of unidirectional links connecting them (bidirectional links are simply represented as a pair of unidirectional links). Let (N_1, \dots, N_q) be a disjoint partitioning of the nodes, each partition modeled by a simulator. For

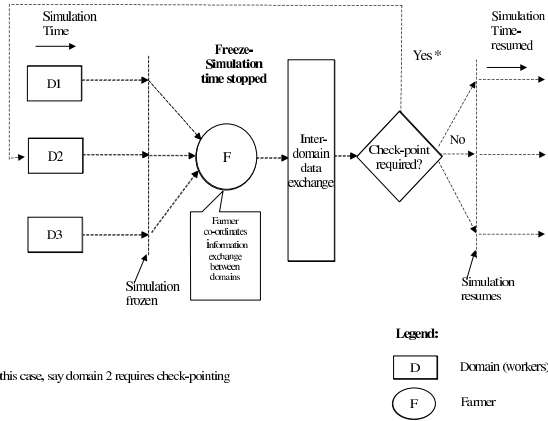


Figure 1. Progress of the Simulation Execution

each subset N_i , we can define a set of external out-links as $O_i = L \& N_i \times (N - N_i)$, in-links as $I_i = L \& (N - N_i) \times N_i$, and local links as $L_i = L \& N_i \times N_i$.

The purpose of a simulator S_i , that models partition N_i of the network, is to characterize traffic on the links in its partition in terms of a few parameters changing slowly compared to the simulation time interval. In the implementation presented in this paper, we characterize each traffic as an aggregation of the flows, and each flow is represented by the activity of its source and the packet delays and drop rates on the path from its source to the boundary of that part. Since the dynamics of the source can be faithfully represented by the copy of the source replicated to the boundary, the traffic is characterized by the packet delays and drop rates on the relevant paths. Thanks to queuing at the routers and the aggregated effect of many flows on the size of the queues, the path delays and packet drop rates change more slowly than the traffic itself.

It should be noted that we are also experimenting with the direct method of representing the traffic on the external links as a self-similar traffic defined by a few parameters. These parameters can be used to generate the equivalent traffic using on-line traffic generator described in [10]. No matter which characterization is chosen, based on such characterization, the simulator can find the overall characterization of the traffic through the nodes of its subnet. Let $\xi_k(M)$ be a vector of traffic characterization of the links in set M in k -th iteration. Each simulator can be thought of as defining a pair of functions:

$$\xi_k(O_i) = f_i(\xi_{k-1}(I_i)), \quad \xi_k(L_i) = g_i(\xi_{k-1}(I_i))$$

(or, symmetrically, $\xi_k(I_i)$, $\xi_k(L_i)$ can be defined in terms of $\xi_{k-1}(O_i)$).

Each simulator can then be run independently of others, using the measured or predicted values of $\xi_k(I_i)$ to compute its traffic. However, when the simulators are linked together, then of course $\bigcup_{i=1}^q \xi_k(I_i) = \bigcup_{i=1}^q \xi_k(O_i) = \bigcup_{i=1}^q f_i(\xi_{k-1}(I_i))$, so the global traffic characterization and its flow is defined by the fixed point solution of the equation.

$$\bigcup_{i=1}^q \xi_k(I_i) = F\left(\bigcup_{i=1}^q (\xi_{k-1}(I_i))\right), \quad (1)$$

where $F\left(\bigcup_{i=1}^q (\xi_{k-1}(I_i))\right)$ is defined as $\bigcup_{i=1}^q f_i(\xi_{k-1}(I_i))$. The solution can be found iteratively starting with some initial vector $\xi_0(I_i)$, which can be found by measuring the current traffic in the network.

We believe that communication networks simulated that way will converge thanks to monotonicity of the path delay and packet drop probabilities as the function of the traffic intensity (congestion). For example, if in an iteration k a part N_i of the network receives more packets than the fixed point solution would deliver, then this part will produce fewer packets than the fixed point solution would. These packets will create inflows in the iteration $k + 1$. Clearly then, the fixed point solution will deliver the number of packets that is bounded from above and below by the numbers of packets generated in two subsequent iterations I_k and I_{k+1} . Hence, in general, iterations will produce alternately too few and too many packets in the inflows providing the bounds for the number of packets in the fixed point solution. By selecting the middle of each bound, the number of steps needed to convergence can be limited to the order of logarithm of the needed accuracy, so convergence is expected to be fast. In the initial implementations of the method, the convergence for UDP traffic and small networks was achieved in 2 to 3 iterations.

It should be noted that the similar method has been used for implementation of the flow of imports-exports between countries in the project Link [4] led by the economics Noble Laureate, Lawrence Klein. The implementation [7] included distributed network of processors located in each simulated country and it used global convergence criteria for termination [8].

One issue of great importance for efficiency of the described method is frequency of synchronization between simulators of parts of the decompose network. Shorter synchronization time limits parallelism but decreases also the number of iterations necessary for convergence to the solution because changes to the path delays are smaller. Variance of the path delay of each flow can be used to adaptively define the time of the synchronization for the subsequent iteration or the simulation step.

The efficiency of our approach is based on the following property of network simulation:

The simulation time of a network grows faster than linearly with the size of the network.

Theoretical analysis indicates that for the network size of order $O(n)$, the simulation time contains terms which are of order $O(n * \log(n))$, that correspond to sorting event queue, of order $O(n^2)$, that result from packet routing, and even of order $O(n^3)$, that are incurred while building routing tables. Some of our simulation performance measurements [9] indicate that the dominant term is of order $O(n^2)$ even for small networks. Using the least squared method to fit the measured execution time for the different network sizes, we got the following approximate formula for star-interconnected networks:

$$T(n) = 3.49 + 0.8174 \times n + 0.0046 \times n^2 \quad (2)$$

where T is the execution time of the simulation, and n is the number of nodes in the simulation. From the above, we can see that the execution time of a network simulation may hold a quadratic relationship with the network size. Therefore, it is possible to speed up the network simulation more than linearly by splitting a large simulation into smaller pieces and parallelizing the execution of these pieces.

As we demonstrate later in the performance section, a network decomposed into 16 parts will require less than 1/16 of the time of the entire sequential network simulation (so also less computational power, because there are 16 parts each needing less than 1/16 of the computational power of the sequential simulator), despite the overhead introduced by external network traffic sources added to each part (as explained below) and synchronization and exchange of data between parts. Hence, with modest number of iterations the total execution time can be cut an order of magnitude or more.

Another advantage of the proposed method is that it is independent of the specific simulator technique employed to run simulators of the parts of the decomposed network. Rather, it is a scheme for efficient parallelization based on convergence to the fixed point solution of inter-part traffic which is measured by a set of parameters necessary to characterize this traffic rather than flow of packets. Our primary application is the use of the on-line simulation for network management [9] to which the presented method fits very well and can be combined with the on-line network monitoring. The simulations in this application predicts changes in the network performance caused by tuning network parameters. Hence, the fixed point solution found by our method is with high probability the point into which the real network will evolve. However, this is a still an open issue under what conditions we can guarantee that the fixed point solution is unique, and if it is not, when the solution found by the method is the same as the point that the real network reaches.

The method can be used in all applications in which the speed of the simulation is of essence, such as:

- on-line network simulation,

- ad-hoc network design,
- emergency network planning,
- large network simulation,
- network protocol verification under extreme conditions (large flows).

2 Implementation

Our current system design is based on *ns* network simulator [6]. In *ns*, a simulation is defined by Tcl scripts which can also be used to interface the core of the simulator. The kernel of the simulation system is written in C++. The ease of adding extensions and rich suite of the network protocols made *ns* a popular and common, albeit not too efficient, platform for research in networking. Hence, we believe that implementing our method within *ns* will enable others to experiment with our system.

Our extensions to *ns* enable collaboration among individual parts into which the simulated network is divided. Since network domains are convenient granules for such partitioning, we will refer to these parts as *simulation domains* or *domains* in short. Each domain is simulated by a separate copy of *ns* running on a unique processor. The implementation specifics are described in the sections below.

2.1 New Features Added to ns

To accomplish per processor based domain simulation the following extension were added to *ns*.

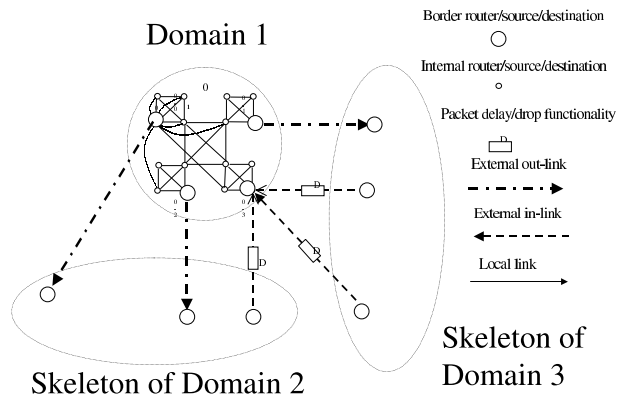


Figure 2. Active Domain with Connections to Other Domains

- The ability to suspend the simulation to enable exchange of data on path delays using message passing between processors simulating individual domains. During the simulation freeze, each individual simulation domain exchanges information on packets generated and dropped along links leaving the domain (cf. Figure 2).

The network in Figure 2 is split into three individual domains, named 1, 2 and 3. Each of the domain simulations runs concurrently with the others and they exchange information about the path delays incurred by packets leaving the domain. The interval for exchange of this information is user configurable (in the Tcl script). For example, each domain may run its individual simulations for one second from n -th to $n + 1$ -st second of the simulation time, and pause thereafter. Then, information about delays of packets leaving the domain during this time interval is passed onto the target domain to which these packets are directed. If these delays differ significantly from what was assumed in the target domain, the simulation of the time interval $(n, n + 1)$ is repeated. Otherwise, the simulation progresses to the time interval $(n + 1, n + 2)$. The threshold value of the difference between the current delays and the previous ones under which the simulation is allowed to progress in time it is set by the user. This threshold impacts the speed of the simulation progress and defines the precision of the simulation results.

A new generic event called *Freeze* has been added to ns. It pauses the simulation at intervals defined by the user. During the event execution, it executes functions provided by the user in Freeze definition. On return, Freeze reactivates the simulation.

- The ability to record information about the delays and drop rate experienced by the packets leaving the domain. Each delay measures the time expired from the instance a packet leaves its source to the time it reaches the domain boundary. Packet drop rates are computed for each flow separately. Also recorded is information about each packet source and its intended destination. Having this information enables us to replicate the source from the original domain to the boundary of the target domain (sources in skeletons of domains 2 and 3 in Figure 2) and postpone an arrival of each packet produced by the replicated source at the domain boundary by the delay measured in the source (and transient, if necessary) domains. Also, with probability defined by packet drop rates, packets are randomly dropped during the passage to the boundary of the destination domain (D boxes in Figure 2).
- The ability to define domain members and identify individual sources within the domain that generate pack-

ets intended for nodes external to the domain. This feature enables us to directly connect a source to the destination domain to which it sends packets. We refer to such replicated source as a *fake source* and to the link that connects it to the domain internal nodes as a *fake link*, as explained below. The domain is defined by the user using a Tcl level command which takes as its parameters the nodes that the user marks as belonging to the domain. Then, the simulation of this domain is created by deactivating all domains external to the selected domain.

2.2 Details of additions and modifications to ns

2.2.1 Domain definition: Domain is a Tcl-level scripting command that is used to define the nodes which are part of the domain for the current simulation. In the first iteration of the simulation the traffic sources outside the domain are inactive. The traffic generated within the domain is recorded and the statistics calculated. In the following iterations, the sources active within other domains with a link to the domain in question are activated.

When a domain declaration is made in the Tcl script, the nodes defined as a parameter to this command are stored in the form of a list. Each time a new domain is defined, the new node list is added to a domain list (a list of lists). The user selected domain is made active. Any link with one end connected to a node in this domain and the other end connected to a node in another domain is defined as a cut-link. All packets sent on these links are collected for their delay and drop rate computation.

Source generators connected to sources outside the active domain are deactivated. This is done by a new Tcl script statement that attaches an inactive status to nodes outside the active domain (cf. 2.2.3. Traffic Generator description below).

2.2.2 Connector: The connector performs the function of receiving, processing and then delivering the packets to the neighboring node or dropping the packets. A modification has been made to this connector class which now has the added functionality of filtering out packets destined for the nodes outside the domain and storing them for statistical data calculation.

A connector object is generally associated with a link. When a link is set up, the simulator checks if this link connects nodes in different domains. If this is the case, this link is classified as a cross-link and the connector associated with this link is modified to record packets flowing across it. Each such packet is either forwarded to the neighboring node or is marked as leaving the domain based on its destination.

2.2.3 Traffic Generator: *TrafficGenerator Class* is used to generate traffic flows according to a timer. This class

is modified, so that for the domain simulation, the traffic sources can be activated or deactivated. Initially, at the start of the simulation, the traffic generator suppresses nodes outside the domain from generating any traffic.

2.2.4 Fake Link: Fake links are used to connect the fake sources to a particular cross-link on the border of the destination domain. When a fake traffic source is connected to a domain by a fake link, the packets generated by this source are sent into the domain via the fake link and not the regular links which are set up by the user network configuration file. The fake link adds a delay and, with certain probability, drops the packet to simulate packet's behavior during passage through the regular route. With the fake traffic sources and fake links, the statistical data from the simulation of another domain are collected, and the traffic to the destination domain is regenerated.

When a fake link is built, the source connector and the destination connector must be specified. A fake link shortens the route between the two connector objects. Each connector is identified by the nodes on both ends of it. Link connectors are managed in the border object as a link list. The flow id to build up a fake link is specified, one fake link is used for one flow.

Fake link is used to simulate a particular flow, so when the features (packet delay and drop rate) of this flow change, the fake link object needs to be updated. After updating the parameters of the fake link object, the performance of the corresponding fake link changes immediately. Fake links themselves are managed in the border object as a link list.

2.2.5 Connectors with Fake Targets: In the original version of ns, connectors are defined as *an NsObject with only a single neighbor*. But our new ns simulation required this definition to be changed to build fake links to shortcut the routes for different packet flows. These fake links are set up according to the network traffic flows and each flow from the fake sources will need a separate fake link. The flows that go through one source connector may reach different cross-link connectors at the destination border, so there will be fake links connecting this connector to some different connectors. Different flows going into one connector are sent to different fake links, which are defined as fake targets here. Thus, the connector could now be defined as *an NsObject with one neighbor and a list of fake targets*. When the fake connection is enabled in a connector, this connector would have a list of fake links (fake targets), and would classify the incoming packets by flow id and send them to the correct destinations.

The connector class will maintain a list of fake targets. Once a new fake link is set up from this connector, it will be added to this connector's fake target list (this is done by the shortcut method defined in the Border class).

2.2.6 Border: Border is a new class added to the ns. It is the most important class in the domain simulation. A bor-

der object represents the active domain in the current simulation. The main functionality of the border class includes:

- Initializing the current domain: setting up the current domain id, assigning nodes to different domains, setting up the data exchange etc.
- Collecting and maintaining information about the simulation objects, such as a list of traffic source objects, a list of the connector objects and a list of the fake link objects maintained by the border object.
- Implementing and controlling the fake traffic sources: setting up and updating fake links, etc.

The border object is set up first, and its reference is made available to all objects in the simulation. A lot of other ns classes need to refer to the variables and methods in the border object. The border class has an array which for each simulation object stores the domain name to which this object belong. This information is collected from domain description files that are created by the domain object implementation. The names are created for the files assigned to each domain to store some persistent data needed for inter-domain data exchange and restoration of the state from the checkpoint.

All traffic source objects created in the simulation are stored. These traffic sources can be deactivated or activated using the flow id. All the connector objects created in the simulation are stored. These connectors are identified by the two nodes to which they are connected. The connector information is used to create fake links.

The traffic sources outside the current active domain are deactivated while setting up the network and domains. When a fake link is set up for a flow, the traffic source of this flow will be reactivated. The border class searches the traffic source list to find the object, and calls the `reactivate()` method of the matching source object to reactivate this flow.

When the border receives flow information from other domains, it will set up a fake link for this flow, and initialize the parameter of the fake link using the received statistical data. When setting up a fake link, it goes through the connector list to find the source and the destination of the connector objects, and then shortcuts the route between them by adding a fake target into the source connector. All the created fake link objects are stored in the border as a linked list ready for further update.

2.2.7 Checkpointing: This feature has been included in ns to enable the simulation to easily rerun over the same simulation time interval. We use diskless checkpointing, in which each client process creates a child when it leaves a freeze point. The child is suspended, but preserves a state of the parent at the freeze time. The parent proceed to the next freeze point. Once there, the parent decides whether to return to the previous state, in which case it unfreezes the

child and then kills itself, or to continue the simulation to the next time interval, in which case the suspended child is killed. This method is efficient because the process memory is not duplicated initially; later only pages that become different for the parent and child are duplicated during execution of the parent. The only significant cost is the execution of fork statement creating a child, which however is several orders of magnitude smaller than saving state to disk.

64-node Network configuration

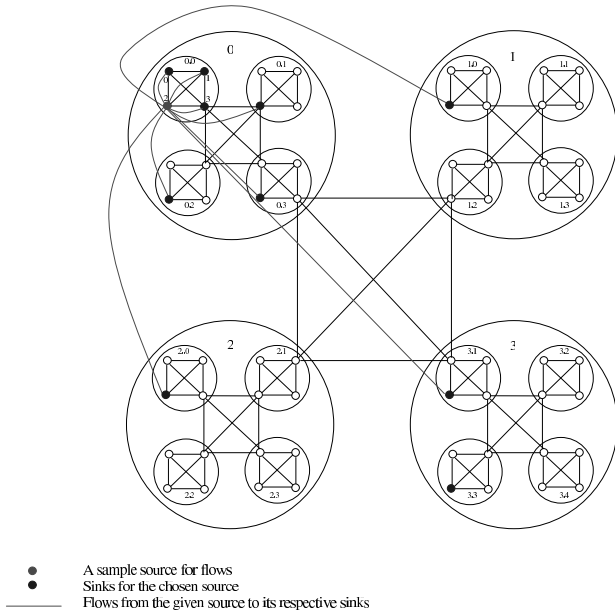


Figure 3. 64-node network showing flows from a sample node

2.2.8 Synchronizing Individual Domain Simulations:

Individual domain simulations are distributed across multiple processors using a client-server architecture. Multiple clients connect to a single server that handles the message passing between them. The server is defined as a single process to avoid the overhead of dealing with multiple threads and/or processes. The server uses two maps (data structures). One map keeps track of the number of clients that have already supplied the delay data for the destination domain. The other map is toggled by clients that need to perform checkpointing. All messages to the server are preceded by *Message Identification Parameters* which identify the state of the client. A decision whether to checkpoint the current state or to restore the saved state is made by the client based on the comparison of packet delays and drop rates in two subsequent iterations.

A client indicates to the server whether it requires checkpointing in the contents of the message itself. A client which has to checkpoint causes all other clients to block until it has resent the data to the server and the server has

delivered it to the destination domain (in other words a domain on another machine). This is achieved by exchanging the maps at the end of each iteration during the simulation freeze.

The steps of collaboration of simulators and the server are shown in Figure 1.

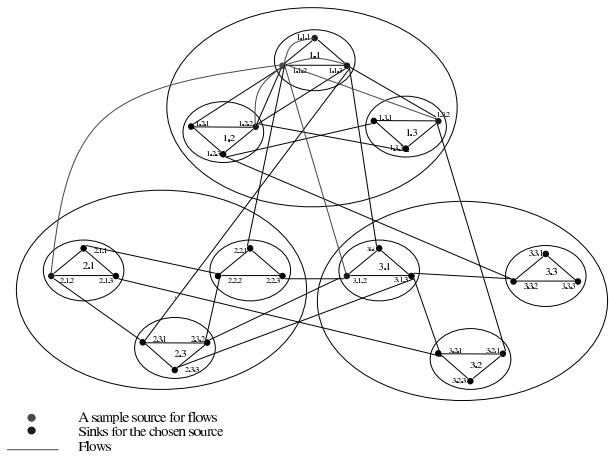


Figure 4. 27-node network showing flows from a sample node

3 Performance

We use two sample network configurations, one with 64 and the other with 27 nodes to measure the performance of our simulation method on two platforms: Sun Solaris and IBM Netfinity FreeBSD workstations. Both networks are divided into a hierarchy of domains. The rate at which sources generate traffic are varied to generate temporal congestion in the network, especially at the nodes at the borders of the domains. All sources produce Constant Bit Rate (CBR) traffic with constant packet size of 64 bytes.

The 64-node network is designed with a great deal of symmetry. The smallest domain size is four nodes; there is full connectivity between these nodes. Four such domains together are considered as a larger domain in which there is full connectivity between the four sub-domains. Finally, four large domains are fully connected and form the entire network configuration (cf. Figure 3).

The 27-node network is a PINNI network [1] with a hierarchical structure. Its smallest domain is composed of three nodes. Three such domains form a larger domain and three large domains form the entire network (cf. Figure 4).

3.1 64-node network

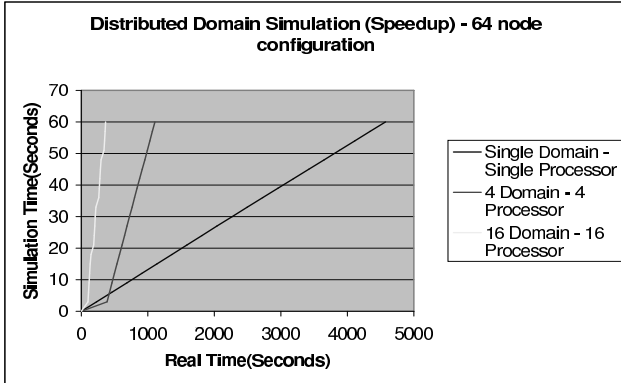


Figure 5. Simulation vs. execution time on Sun Solaris for the 64-node network with different decompositions

Each node in the network is identified by three digits $x.y.z$, where $0 \leq x, y, z \leq 3$, that identify domain, sub-domain and node rank within the subdomain to which the node belongs.

Each node has nine flows originating from it. In addition, each node also acts as a sink to nine flows. The flows from a node $x.y.z$ go to nodes:

$$\begin{array}{lll} x.y.(z+1)\%4 & x.y.(z+2)\%4 & x.y.(z+3)\%4 \\ x.(y+1)\%4.z & x.(y+2)\%4.z & x.(y+3)\%4.z \\ (x+1)\%4.y.z & (x+2)\%4.y.z & (x+3)\%4.y.z \end{array}$$

Thus, this configuration forms a hierarchical and symmetrical structure on which the simulation is tested for scalability and speedup.

In a set of performance measurements, the sources at the borders of domains produce packets at the rate of 20000 packets/sec for half of the simulation time. The bandwidth of the link is 1.5Mbps. Thus, certain links are definitely congested and congestion may spread to some other links as well. For the other half of the simulation time, these sources produce 1000 packets per second. Since such flows require less bandwidth than provided by the links connected to each source, congestion is not an issue. All other sources produce packets at the rate of 100 packets/sec for the entire simulation. For the performance measurements, we defined sources that produced only CBR traffic and the speedup was measured by comparing simulation times of domains to the simulation time of the entire network (excluding synchronization time).

We measured speed up for this configuration over simulation of 60 seconds of traffic. The simulation interval was set at 14.9999 seconds, resulting in five freezes. The simulation speedup with 16 processors (domains with four nodes

each) was superlinear on both Sun Solaris (see Figure 5) and IBM Netfinity FreeBSD platforms (see Table 1), despite repetitive simulations over some of the intervals. The decomposed simulation required at most two iterations to converge to the solution in each simulation time interval. The differences in the total number of packets in each flow, the number of dropped packets and the sizes of the queues at the routers were well below 1% for all three domain sizes.

3.2 27-node configuration

The network configuration shown in Figure 4, the PINNI network adopted from [1], consists of 27 nodes arranged into 3 different levels of domains containing three, nine and 27 nodes, respectively. Each node has six flows to other nodes in the configuration and is receiving six flows from other nodes. The flows from a node $x.y.z$ can be expressed as:

$$\begin{array}{ll} x.y.(z+1)\%3 & x.y.(z+2)\%3 \\ x.(y+1)\%3.z & x.(y+2)\%3.z \\ (x+1)\%3.y.z & (x+2)\%3.y.z \end{array}$$

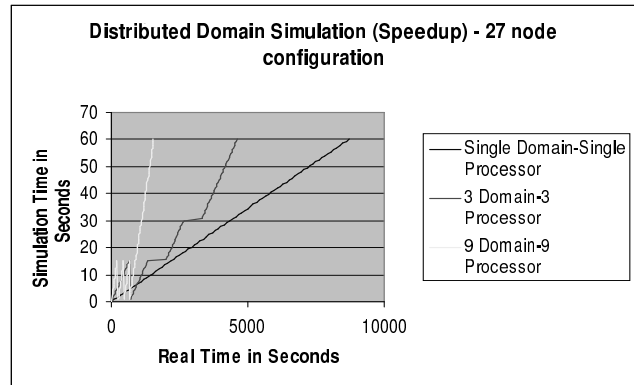


Figure 6. Simulation vs. execution time on Sun Solaris for 27-node network with different decompositions

In these, as in the previous, performance measurements, the sources at the borders of domains produce packets at the rate of 20000 packets/sec for half of the simulation time. The bandwidth of the link is 1.5Mbps. Thus, congestion is definitely produced on certain links shown above and congestion may be produced on certain other links. For the other half of the simulation, these sources produce 1000 packets which is less than the total bandwidth of the links connected to each of them. All other sources produce packets at the rate of 100 packets/sec for the entire simulation. We measured the speed up for this configuration over 60 seconds of simulated traffic. The simulation interval was set at 14.9999 seconds, resulting in five freezes.

size of domain	27-nodes	64-nodes
large = 1 proc/domain	3946.8	1714.5
medium = 3(4) procs/domains	776.0	414.7
small = 9(16) procs/domains	237.3	95.1
speed up for small domain	12.4	18.0

Table 1. Performance measurements on IBM Netfinities (times are in seconds, averaged over five runs)

The simulation with 9 processors achieved superlinear speedup both on Sun Solaris (see Figure 6) and for IBM Netfinity FreeBSD platform (see Table 1).

This configuration is less regular than the 64-node configuration and as a result, the number of iterations needed for convergences varied from two to four. The differences in the total number of packets in each flow, the number of dropped packets and the sizes of the queues at the routers were well below 1% for all three different domain sizes.

4 Conclusions and Future Work

The need for scalable and efficient network simulators increases with the rapidly growing complexity and dynamics of the Internet. In this paper we introduced a collaborative on-line simulation scheme to support real-time on-line collaborative simulators.

Traditional parallelization of simulation partitions network topology into Logical Processors (LPs) [3], but the simulation is still executed as a whole. Therefore, the partitioned parts have to exchange a lot of information to keep them synchronized with each other. In contrast, our approach decomposes the network into the separate part. First, each part of this decomposition is simulated separately from a node concurrently with the others over the simulation interval. Then, the simulations are repeated using the output of the other parts as their input until there is no significant difference between the results of two consecutive iterations. This approach greatly simplifies the synchronization between parallel parts and it decreases its frequency, thus it can significantly speed up the simulation of large networks. Our results indicate that the superlinear speedup for the single iteration step is possible and is the result of the non-linear complexity of the network simulation.

In addition to the speedup, the advantages of the presented method include fault tolerance, ability to integrate simulations and models in one run and support for truly distributed execution. When one of the participating processes fails, the rest can use the old packet delay and drop rate data to continue a simulation. When the only information available about a domain are delays across the domain and

its outflows, the simulation of the other parts of the networks can directly use these data to perform the simulation. Finally, the scheme can be implemented in the fully distributed fashion, in which a domain is simulated using computational resources within itself.

Future work will focus on providing online data collection, to increase the benefit of the real-time simulation supported by this scheme. It should be noted that the benefits of the method are multiplicative in regards to the benefits of any simulator that is employed to simulate individual domains. Hence, the choice of the basic simulation tool is important. In the future, we plan to replace ns with the simulators that are at least an order magnitude faster than the sequential ns, such as ssfnet simulator [2].

Finally, while this paper demonstrates that our approach fits the simulation of non-feedback based traffic (such as UDP-based traffic), we plan to verify our implementation on TCP-based traffic as well.

References

- [1] S. Bhatt, R. Fujimoto, A. Ogielski, and K. Perumalla. Parallel simulation techniques for large-scale networks. *IEEE Communications Magazine*, 36, 1998.
- [2] J. Cowie and A. Ogielski. SSFNET tutorial. <http://www.ssfent.org>.
- [3] R. Fujimoto. Parallel discrete event simulation. *Communications of the ACM*, 33:31–53, October 1990.
- [4] L. R. Klein. *Quantitative Studies of International Economic Relations*, chapter The LINK Model of World Trade with Application to 1972-1973. North Holland, Amsterdam, 1975.
- [5] L. A. Law and M. G. McComas. Simulation software for communication networks: the state of the art. *IEEE Communication Magazine*, 32:44–50, 1994.
- [6] NS network simulator. <http://www-mash.cs.berkeley.edu/ns>.
- [7] Y. Shi, N. Prywes, B. Szymanski, and A. Pnueli. Very high level concurrent programming. *IEEE Trans. Software Engineering*, SE-13:1038–1046, September 1989.
- [8] B. Szymanski, Y. Shi, and N. Prywes. Synchronized distributed termination. *IEEE Trans. Software Engineering*, SE-11:1136–1140, September 1987.
- [9] T. Ye, D. Harrison, B. Mo, S. Kalyanaraman, B. Szymanski, K. Vastola, B. Sikdar, and H. Kaur. Traffic management and network control using collaborative on-line simulation. In *Proc. International Conference on Communication, ICC2001*, 2001.
- [10] M. Yuksel, B. Sikdar, K. S. Vastola, and B. Szymanski. Workload generation for ns simulations of wide area networks and the internet. In *Proc. Communication Networks and Distributed Systems Modeling and Simulation Conference*, pages 93–98. SCS Press, 2000.