

Stateless Routing in Network Simulations *

*George F. Riley
Mostafa H. Ammar
Richard Fujimoto*

College of Computing
Georgia Institute of Technology
Atlanta, GA 30332
{riley,ammar,fujimoto}@cc.gatech.edu
(404)894-6855, Fax: (404)894-0272

Abstract

The memory resources required by network simulations can grow quadratically with size of the simulated network. In simulations that use routing tables at each node to perform per-hop packet forwarding, the storage required for the routing tables is $O(N^2)$, where N is the number of simulated network nodes in the topology. Additionally, the CPU time required in the simulation environment to compute and populate these routing tables can be excessive and can dominate the overall simulation time.

We propose a new routing technique, known as Neighbor-Index Vectors, or NIX-Vectors, which eliminates both the storage required for the routing tables and the CPU time required to compute them. We show experimental results using NIX-Vector routing in the popular network simulator ns. With our technique, we achieve a near order of magnitude increase in the maximum size of a simulated network running ns on a single workstation. Further, we demonstrate an increase of two orders of magnitude in topology size (networks as large as 250,000 nodes) by using this technique and running the simulation in parallel on a network of workstations.

*This work is supported in part by NSF under contract number ANI-9977544

1 Introduction

The use of simulation is becoming increasingly prevalent in the networking research community. The popular ns[5] network simulator is used in many areas of networking research, including transport protocols, queue management, scheduling policies, mobility, and multicast[3]. However, limits on the resources available within the simulation environment place severe constraints on the size and complexity of the networks being simulated. In particular, the physical memory required to simulate a network of N nodes grows proportionally to N^2 , for reasons discussed below.

One approach to simulating larger networks is to utilize distributed simulation techniques, and run the simulation on several loosely coupled systems in a network of workstations, or on a tightly coupled symmetric multiprocessor system. Several researchers, including Nicol[7, 6], Ogielski[2], Perumalla[9, 8], Bagrodia[16, 1], Rao[10], and Riley[11] have discussed various methods for implementing distributed network simulations, each with varying degrees of success. Another approach, proposed by Huang[4], is to reduce the level of detail in the simulation by using abstractions which reduce the amount of state and the number of events needed to model network components.

We chose a two pronged approach, reducing the memory requirements in a single system environ-

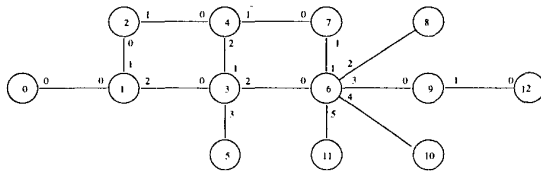


Figure 1. Simple Simulated Network

ment, and distributing the application on several systems. By identifying and eliminating the main cause of excessive memory requirements within a single system simulation, we can increase the overall size of the simulated network proportionally. Then by distributing the simulation on several systems, we again can proportionally increase the overall topology size.

As mentioned previously, the memory required by a network simulation grows quadratically with the number of simulated nodes. This is due to the presence of simulated *routing tables* in the simulation environment. Consider the simple topology shown in figure 1, consisting of 13 nodes and 14 links. Further consider that a simulated packet has arrived at node three with the ultimate destination of node twelve. In this example, node three must select from four possibilities (nodes 1, 4, 5, and 6) for the next hop to the final destination. In general, any node must be able to choose a next hop neighbor to route packets to any potential destination. The simplest solution is to create routing tables consisting of N entries at each of the N simulated nodes. Each table is indexed by the final destination, and returns the next hop neighbor. This method results in total memory proportional to N^2 . It is easy to see that as N grows, the routing tables will start to account for the majority of the memory used by the network simulation. In addition, the CPU time required to compute the routing table information can become excessive and can dominate the overall time required to run a simulation. Techniques for compression of the routing tables have been discussed[13], showing some promising results. However, even with compression, the memory required is kN^2 , $0 < k \leq 1$.

Our approach is to simply delete completely the need for routing tables. We do this by computing routes *on demand* whenever a new packet is

generated, and then including the routing information in the simulated packet. As a packet arrives at a simulated node, the routing decision can be made by examining the packet, rather than a routing table lookup. We use a compact representation for routing information known as a *Neighbor-Index Vector*, or *Nlx-Vector*. When a *Nlx-Vector* is calculated at a given source for a given destination, it is cached at the source and reused repeatedly as more packets are generated for the same destination. By caching the routing information at each node, the CPU overhead to compute a route is incurred only once per source and destination pair. We show in the next section that the routes can be stored in an extremely small amount of memory, resulting in negligible memory overhead for the route caching. This method gives a substantial memory reduction for the overall simulation.

The remainder of this paper is organized as follows. In section 2 we give an overview of *Nlx-Vector* routing. In section 3 we discuss our implementation of *Nlx-Vector* routing in the popular *ns* network simulator. In section 4 we describe our simulation environment, the large scale topologies used for our experiments, and give memory and CPU usage results for the simulations. In section 5 we give some conclusions and future directions of our research.

2 *Nlx-Vector* Routing

The use of *Nlx-Vector* routing as a technique for efficient packet processing in Internet routers is discussed in detail by Riley et. al. in [14]. We give an overview of *Nlx-Vectors* here to explain how this technique can be used to give a compact representation of routing information.

The basis of *Nlx-Vector* routing is the observation that, at any *routing decision point* (in the simulation environment this is a simulated router or end host), the routing decision consists of simply choosing one element from an ordered set of *routing neighbors*. If there are N routing neighbors at a given node, then the decision can be recorded and retained using only $k = \lceil \lg_2 N \rceil$ bits. A *Nlx-Vector* is therefore the concatenation of the routing decisions at each routing decision point along the path from a source to a destination, where each routing

Table 1. *Nix*-Vector Example

Hop	Node	N	k	<i>Nix</i>	Used	<i>Nix</i> -Vector
0	0	1	1	0	1	0
1	1	3	2	2	3	0 10
2	3	4	2	2	5	0 10 10
3	6	6	3	3	8	0 10 10 011
4	9	2	1	1	9	0 10 10 011 1

decision point uses only the smallest number of bits needed to record the decision. Note that the number of bits needed at each node to record the routing decision will vary from node to node (it is a function of the number of directly connected neighbors at each node), but for a given node it is a constant (as long as routing neighbors are not added or deleted over time during the simulation). To route a packet containing a *Nix*-Vector, each node simply extracts the next $k = \lceil \lg_2 N \rceil$ bits from the vector, and uses this value as an index into an ordered set of directly connected neighbors.

To illustrate, consider again the simple simulated network topology shown in figure 1 and consider a packet generated at node zero with a destination of node twelve. It is easy to see that the correct shortest path route is 0-1-3-6-9-12. Each link to routing neighbors at any node is numbered sequentially, beginning from zero as shown in the figure, which results in the ordered set of directly connected neighbors. Table 1 shows how a *Nix*-Vector can be created for this route using only nine bits. The column labeled *Node* is the node number where this routing decision is recorded. The column labeled *N* is the number of routing neighbors at each hop, column *k* is the number of bits needed to record the route, column *Nix* is the correct neighbor index for this route, column *Used* is the cumulative length in bits for the *Nix*-Vector, and the last column is the actual *Nix*-Vector.

The resulting *Nix*-Vector for packets traveling from node zero to node twelve is 010100111. To use the *Nix*-Vector to route packets, the source of the packet (node zero in our example), stores this value in the packet header, and makes a routing decision. Since node zero has only one routing neigh-

bor, it extracts the first bit from the vector (0 in this example), and routes the packet to neighbor 0 (which is node one). Since node one has three routing neighbors, it extracts the next two bits (10 in this example), and routes the packet to neighbor two (node three). Since node three has four routing neighbors, it extracts the next two bits (10 in this example) and routes the packet to neighbor two (node six). Since node six has six routing neighbors, it extracts the next three bits (011 in this example) and routes the packet to neighbor three (node nine). Finally, since node nine has two routing neighbors, it extracts the next one bit (1 in this example), and routes the packet to neighbor one (node twelve), which is the final destination.

In the simulation environment, a *Nix*-Vector from any source to any destination can be easily computed using a simple breadth first search algorithm, which runs in $O(|N| + |E|)$, where *N* is the set of nodes and *E* is the set of edges (links). From a practical standpoint, all *Nix*-Vectors using less than 32 bits are equivalent in storage requirements, since memory is allocated in units of 32 bits. In fact, for our implementation, the minimum size is 64 bits, since we store the maximum length for each vector, as well as the vector itself.

3 *ns* Implementation

We implemented our *Nix*-Vector routing technique in the popular *ns* network simulator, to demonstrate resulting memory and CPU time savings. We started with some basic assumptions about the behavior of the *ns* simulation.

1. The network topology for the simulation is fixed. In other words, the nodes and links are created in advance, and do not change as the simulation progresses in time.
2. The simulation uses *static* routing. Packets from a given source to a given destination will always follow exactly the same path.
3. Routes are computed using constant link weights. In the route computations, no link is "more desirable" than another.
4. In the simulation, the routing tables existed exclusively for the purpose of unicast packet

forwarding, and were used for no other purpose.

Assumption 1 is reasonable, excepting for the simulations of mobile networks. Assumptions 2 and 3 are both the default cases in *ns*, and are reasonable for simulations of end-to-end protocols, queuing disciplines, and packet scheduling mechanisms. These two assumptions may not be valid for simulations involving dynamic routing decisions, routing protocols, or mobility. Assumption 4 is reasonable for any simulation where the routing of packets is not of direct interest to the simulation. This is generally the case, but not always. For example, some multicast simulations may require direct access to routing tables at a given node, and thus may not work properly with our method. However, we believe a large fraction of *ns* simulations fit within the assumptions above.

We started with *ns* version 2.1b6a as the baseline software for our modifications. The implementation of *Nlx-Vector* routing took approximately three weeks, consisting of the addition of eleven new C++ modules, two new TCL modules, modification of three existing C++ modules, and modification of four existing TCL modules. The new C++ modules consisted of the breadth first search implementation for route computation, the creation of a *Nlx-Vector* from the BFS results, and the implementation of a *Nlx-Vector* classifier which routes packets based on a *Nlx-Vector* in the packet header. The modifications to existing code were extremely minor, consisting of a few lines of code to test for *Nlx-Vector* routing and change default behavior when *Nlx-Vector* routing is used. Some other small modifications were implemented to help with the instrumentation of memory usage and CPU usage for our experiments.

The specification of *Nlx-Vector* routing from the perspective of the *ns* user involves inserting a single command in the *ns* script to specify *Nlx-Vector* routing is desired. No other modifications to user code is needed. In our implementation, if *Nlx-Vector* routing is not explicitly enabled by the *ns* user, the normal routing table method is used. When *Nlx-Vector* routing is specified, the *ns* code that calculates and stores the routing tables is skipped completely. Instead, when any new simu-

lated packet is generated at a source node a cache of known *Nlx-Vectors* (cached at each source node) is searched to find an existing route to the desired target node. If it is not found, a new *Nlx-Vector* for the specified target is calculated, and cached. With this technique, the path to any given target from any given source is calculated only once, and saved using a small amount of memory.

In keep the per-packet memory requirements as low as possible, the actual *Nlx-Vector* is not in fact included in the simulated packet. Instead, simulated packets contain a pointer to the cached *Nlx-Vector* and a counter indicating how many of the bits in the *Nlx-Vector* have been extracted and used so far.

To estimate the memory requirements needed for a large scale simulation using *Nlx-Vector* routing, assume the network size is N , the average out degree of each node is d ($d \ll N$), the number of connections between pairs of nodes is k ($k \ll N$), the average path length from a source to a target is $O(N)$, and the number of simulated packets active at any time is $O(N)$. With these assumptions, the memory usage to store and use routing information using *Nlx-Vectors* is $kN \log d$ (cached *Nlx-Vectors*) plus N (packets), which is $O(N)$. Clearly, if the assumption that $k \ll N$ does not hold (for example if every node creates a connection to every other node), then the memory requirements become $O(N^2)$ which is the same as the routing table approach. Our approach of specifying the use of *Nlx-Vector* routing as part of the simulation script gives the *ns* user control over the routing method used. For simulations with smaller numbers of connections and relatively short routing paths, the *Nlx-Vector* approach can achieve a substantial memory savings, as we show in section 4.

3.1 Relaxing the Assumptions

In this subsection, we discuss techniques to enhance our implementation of the *Nlx-Vector* routing in *ns* to relax assumptions 1 through 3 above. We point out that our current implementation does not include these enhancements.

Assumptions 1 and 2 allow the simulation to be sure that, once a *Nlx-Vector* is calculated from some

source to some destination, it will remain valid for the remainder of the simulation. Should more nodes or links be added over time, or existing nodes or links fail during the simulation, previously calculated *Nlx-Vectors* may become invalid and produce incorrect routes.

To relax these assumptions, we introduce the notion of a *topology index* value, represented by a 32 bit value, initially set to zero. When *Nlx-Vectors* are calculated between a pair of nodes, the vector is stamped with the current value of the topology index. The topology index value is incremented whenever the simulation topology changes in any way, either by adding new nodes or links, or by simulating failure of existing nodes or links. Whenever a *Nlx-Vector* is referenced (either when a new simulated packet is generated at a source node, or when making a routing decision at routing nodes), the current topology index value is compared to the topology index value stamped in the *Nlx-Vector* when it was created. If they are different, the *Nlx-Vector* must be validated for correctness.

The validation consists of two steps. First, a new *Nlx-Vector* is calculated from the same source and destination as the *Nlx-Vector* being validated. Then the new *Nlx-Vector* is compared to the one being validated. If they are the same then no further action is necessary. If they are different, then the action depends on the intended use of the invalid vector. Additionally, the new *Nlx-Vector* is cached at the source node and is stamped with the current topology index value.

1. The invalid *Nlx-Vector* is being used for a newly generated packet from a data source. In this case, the new *Nlx-Vector* is used instead of the original, invalid one.
2. The invalid *Nlx-Vector* is being used to make a routing decision for an existing packet. There are three possible sub-cases:
 - (a) The desired destination node can still be reached successfully using the original *Nlx-Vector*. In this case, the original *Nlx-Vector* is used to make the routing decision.
 - (b) The node where the routing decision is being made is on the path from the

source to the destination using the newly calculated *Nlx-Vector*. In this case, the new *Nlx-Vector* is substituted in the packet and the routing decision is made using the new *Nlx-Vector*.

- (c) The node is not on the routing path of the new *Nlx-Vector*. In this case, a new *temporary Nlx-Vector* is calculated, from the routing decision node to the destination, and this temporary *Nlx-Vector* is used for this packet. The temporary *Nlx-Vector* is not cached, since it is only useful for packets that were in transit when the topology change occurred.

Assumption 3 was made for ease of implementation. Assuming constant link weights allows the use of a simple breadth-first-search algorithm for computing routes. Using a slightly more complex routing computation, such as Dijkstra's algorithm, will allow for route computations with weighted links and relax this assumption. Assumption 4 is fundamental to our approach to simulated routing, and cannot be relaxed.

4 Experiments and Results

We ran three basic types of experiments, using two different types of topologies. Two sets of experiments were run in a single system environment, to measure the beneficial effects of *Nlx-Vector* routing in a serial simulation. The third set of experiments were run in a distributed environment, using parallel/distributed *ns*[11] modified to support *Nlx-Vector* routing.

We ran our experiments on a cluster Intel systems running the RedHat Linux operating system. Each system has eight 500Mhz CPUs and 4Gb of physical memory (although the Linux kernel limits the memory assigned to a single process to 2Gb). Even though each system has eight CPUs, only one CPU per system is used by our applications. All experiments were run on dedicated systems with no other user jobs running. All of the normal operating system overhead and daemons were present.

Our simulation topologies were built using two different methods for creating large topologies.

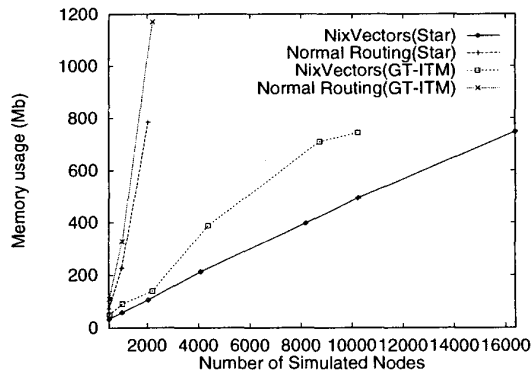


Figure 3. Memory Usage (Mb)

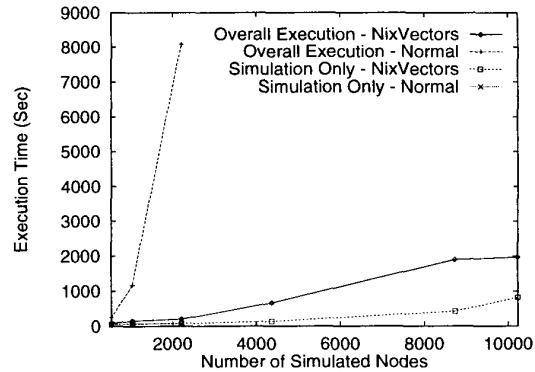


Figure 5. Execution Time, GT-ITM (Sec)

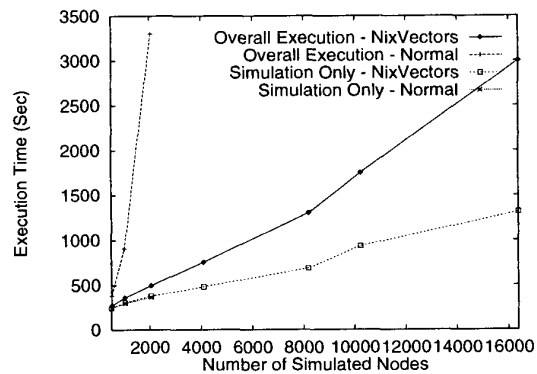


Figure 4. Execution Time, Star Topologies(Sec)

is lengthy, the overall effects of this improvement would be reduced. The elapsed CPU time for *Nix-Vector* routing simulations increases approximately linearly with the number of nodes (we increased the number of connections modeled proportionally with the number of nodes). The CPU usage graphs show no discernible difference in the simulation portion of the run.

We then created a version of parallel/distributed *ns*[12] which uses the *Nix-Vector* routing method, and ran simulations distributed on 2, 4, 8, and 16 systems. Each system used the star topology previously discussed, and created a topology of 16,383 nodes per system. We create 1,000 connections per systems, with 90% of the connections to local nodes, and the remaining to remote nodes. The lookahead value used was 10ms. With this method, the overall topology size and number of connections increased

Table 2. Topology Sizes for Distributed Simulations

Number Systems	Connections	Nodes
2	2,000	32,766
4	4,000	65,532
8	8,000	131,064
16	16,000	262,128

linearly as the number of systems increased.

Table 2 shows the overall topology size achieved with the distributed simulations. Note that with our methods, we were able to simulate a topology consisting of 262,128 nodes and 16,000 connections. In the same size simulation environment, the standard *ns* was only able to achieve a size of about 2,500 nodes.

5 Conclusions and Future Work

The overall memory requirements for a large scale network simulation are dominated by the space used for simulated routing tables. By computing routes on demand, and storing the computed route in simulated packet headers in *Nix-Vector* format, the space used by the routing tables can be eliminated. Our experiments show a near order of magnitude increase in the maximum size of a simulated network within a given physical memory limit. By distributing the application on a network of workstations, we achieve an overall topology size increase of two orders of magnitude, from about 2500 nodes

on the standard *ns*, to over 250,000 nodes when using our methods. In addition, the CPU time required to populate the routing tables can dominate the overall running time of a simulation in the case where the topology is large and the simulation time is short. The *Nlx-Vector* method of routing packets does not populate routing tables and thus this computationally expensive operation can be eliminated.

Our implementation of *Nlx-Vector* routing in *ns* was somewhat quick and dirty, as evident by the short three week implementation effort. We kept the effort minimal by making the restrictive assumptions given in section 3. For future work, we will implement the enhancements discussed in section 3.1 to relax assumptions 1 through 3.

References

- [1] R. Bagrodia, R. Meyer, M. Takai, Y. Chen, X. Zeng, J. Martin, B. Park, and H. Song. Parsec: A parallel simulation environment for complex systems. *IEEE Computer*, 31(10):77-85, October 1998.
- [2] J. H. Cowie, D. M. Nicol, and A. T. Ogielski. Modeling the global internet. *Computing in Science and Engineering*, January 1999.
- [3] D. Estrin et al. Virtual internetwork testbed: Status and research agenda, July 1998. USC Computer Science Dept, Technical Report 98-678.
- [4] P. Huang, D. Estrin, and J. Heideman. Enabling large-scale simulations: selective abstraction approach to the study of multicast protocols. In *Proceedings of the International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, July 1998.
- [5] S. McCanne and S. Floyd. The LBNL network simulator. Software on-line: <http://www-mash.cs.berkeley.edu/ns>, 1997. Lawrence Berkeley Laboratory.
- [6] D. Nicol, M. Johnson, A. Yoshimura, and M. Goldsby. Ides: A java-based distributed simulation engine. In *Proceedings of the International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, July 1998.
- [7] David Nicol and Philip Heidelberger. Parallel execution for serial simulators. *ACM Transactions on Modeling and Computer Simulation*, 6(3):210-242, July 1996.
- [8] K. Perumalla, R. Fujimoto, and A. Ogielski. Ted - a language for modeling telecommunications networks. *Performance Evaluation Review*, 25(4), March 1998.
- [9] K. S. Perumalla and R. M. Fujimoto. Efficient large-scale process-oriented parallel simulations. In *Proceedings of the Winter Simulation Conference*, December 1998.
- [10] D. M. Rao and P. A. Wilsey. Simulation of ultra-large communication networks. In *Proceedings of Seventh International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, October 1999.
- [11] G. F. Riley, R. M. Fujimoto, and M. A. Ammar. A generic framework for parallelization of network simulations. In *Proceedings of Seventh International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, October 1999.
- [12] G. F. Riley, R. M. Fujimoto, and M. A. Ammar. Parallel/Distributed ns. Software on-line: www.cc.gatech.edu/computing/compass/pdns/index.html, 2000. Georgia Institute of Technology.
- [13] G. F. Riley, R. M. Fujimoto, and M. A. Ammar. Reduced-state models for distributed network simulations. In *Submitted for publication.*, 2000.
- [14] G. F. Riley, E. W. Zegura, and M. A. Ammar. Efficient routing using nix-vectors, Mar 2000. Technical Report GIT-CC-00-13.
- [15] E. W. Zegura, K. Calvert, and S. Bhattacharjee. How to model an internetwork. In *Proceedings of IEEE Infocom 96*, 1996.
- [16] X Zeng, R. Bagrodia, and M. Gerla. GloMoSim: a library for parallel simulation of large-scale wireless networks. In *Proceedings of the 12th Workshop on Parallel and Distributed Simulations*, May 1998.