Space–Parallel Network Simulations using Ghosts *

George F. Riley1
Talal M. Jaaffar1
Richard M. Fujimoto2
Mostafa H. Ammar2

1Department of Electrical and Computer Engineering
Georgia Institute of Technology
Atlanta, GA 30332-0250
{riley,jaaffar}@ece.gatech.edu

2College of Computing
Georgia Institute of Technology
Atlanta, GA 30332-0280
{fujimoto,ammar}@cc.gatech.edu

March 4, 2004

Abstract

We discuss an approach for creating a federated network simulation that eases the burdens on the simulator user that typically arise from more traditional methods for defining space–parallel simulations. Previous approaches have difficulties that arise from the need for global topology knowledge when forwarding simulated packets between the federates. In all but the simplest cases, proper packet forwarding decisions between federates requires routing tables of size \(O(mn)\) (\(m\) is the number of nodes modeled in a particular simulator instance, and \(n\) is the total number of network nodes in the entire topology) in order to determine how packets should be routed between federates. Further, the benefits of the well–known NIx–Vector routing approach cannot be fully achieved without global knowledge of the overall topology. We seek to overcome these difficulties by utilizing a topology partitioning methodology that uses Ghost Nodes. A ghost node is a simulator object in a federate that represents a simulated network node that is spatially assigned to some other federate, and thus that other federate is responsible for maintaining all state associated with the node. However, ghost nodes do retain topology connectivity information with other nodes, allowing all federate in a space–parallel simulation to obtain a global picture of the network topology. We show with experimental results that the memory overhead associated with the ghosts is minimal relative to the overall memory footprint of the simulation.

1 Introduction

One approach to creating simulation models for large–scale topologies in network simulations is to use a space–parallel partitioning methodology, coupled with distributed simulation methods. In a space–parallel network simulation, the model for the entire simulated network is divided logically into \(k\) sub–models, where \(k\) is the number of federates in the distributed simulation. With this approach, each federate is responsible for approximately \(1/k^{th}\) of the entire topology model, and instantiates simulation objects to represent its own portion of the network elements in the complete topology. Since a given federate has no responsibility for the remaining \((k−1)/k\) portion of network elements, no simulation objects are created and thus the federate has no knowledge about the remaining topology. This approach is fairly easy to implement, and is the method used by existing space–parallel distributed network simulators such as Parallel/Distributed ns (pdns) and the Georgia Tech Network Simulator (GTNetS) [1, 2]. Further, this method has very good scalability, since each federate need only be concerned with its own network elements, and thus only allocates simulator memory for a fraction of the entire set of network elements. However, as we shall show this approach introduces a number of difficulties that must be addressed in order to insure correct packet forwarding between the federates.

Our solution to these difficulties is to introduce the notion of a Ghost Node. A ghost node is a simulator object that acts as a placeholder for nodes that are assigned to other federates. The ghost node object has none of the complex and memory intensive state needed for real nodes (such as queues, routing tables, port maps, and applications). Rather, a ghost node simply contains topology connectivity information about links and neighbors. Thus, using ghosts, a federate is afforded a global picture of the simulated topology, without the memory overhead of maintaining unneeded state for the ghosts.

The remainder of this paper is organized as follows. Section 2 discusses the space–parallel approach for distributed network simulation, and shows some of the difficulties associated with these traditional approaches. Section 3 gives the basic design of our Georgia Tech Network Simulator with emphasis on the ghost node implementation. Section 4 discusses related work. Section 5 presents memory usage statistics comparing our ghost node approach to more traditional routing table approaches. Finally, section 6 describes conclusions from this work.

*This work is supported in part by NSF under contract numbers ANI-9977544, ANI-0136969, ANI-0240477, ECS-0225417, and DARPA under contract number N66002-00-1-8934.
2 Space–Parallel Simulations

To illustrate the concepts and issues regarding space–parallel network simulations, we present two simple examples. Consider the simple topology shown in figure 1, consisting of four subnetworks. Each subnetwork has four hosts, two intermediate routers, and one gateway router. Each of the four gateway routers is connected to each of the other three gateway routers, forming a fully connected mesh. All of the simulated nodes for a subnetwork have a common 24 bit network address prefix, such as 192.168.0.x for subnetwork 0 as shown.

Now suppose that, due to resource constraints in our simulation environment, we cannot model more than seven network nodes in a given simulator instance without running out of memory on the computing platform in use. Clearly, such limited resources are not realistic, but are used here for illustrative purposes. Even with these resource constraints, we can still create a simulation of the four subnetwork topology by using space–parallel distributed simulation. We create four different simulator instances, each running on a separate hardware platform. Each of the four simulator instances instantiates models for the seven nodes in a single subnetwork. For example, simulator 0 would model the seven nodes in subnetwork 0, simulator 1 would model the seven nodes in subnetwork 1, etc. Our simulation environment must have some way to simulate links between federates (such as the link from G0 to G1). Links that span federate boundaries are called remote links, or rlinks. Issues such as time management and event distribution between federates can easily be solved using one of several available Runtime Infrastructure packages, such as the Georgia Tech Federated Simulation Developers Kit (FDK)[3], or the DMSO RTI[4]. The end result is that we are able to model twenty–eight network nodes, using four instances of a simulator that can only model seven nodes each, using the space–parallel methodology. The following paragraphs discuss some of the issues that arise when determining correct packet routing in this type of simulation.

Default Routes. In this simple example, the routing of packets between federates is nearly trivial. Suppose host H00 sends a packet to host H23. Simulator 0 can easily determine that the destination node (H23) is not modeled locally. Since in this example the destination node is defined and managed on simulator 2, simulator 0 must make a routing decision based on incomplete knowledge of the overall topology. In this case however, gateway node G0 is the only way that packets can leave or enter subnet 0 (and hence simulator 0), H00 simply forwards the packet to node G0 (through node R00) for further processing. In pdns and GTNetS this is known as a Default Route, and works well when there is a single simulated node responsible for all packets in and out of a simulator’s topology view.

Inter–Federate Route Aggregation. Route aggregation is a method used in Internet routers to reduce routing table size. If all of the routing table entries for a set of destination addresses are identical, and the destination set has a common address prefix, then this entire set of routes can be stored with a single entry.

Using route aggregation, once the packet arrives at gateway node G0, the routing decision is again easy and takes little memory. Although gateway node G0 has three rlinks, the routing decision can easily be made based on the destination IP Address. The rlink from G0 to G1 is the correct path for any IP Address starting with 192.168.1., since all nodes with that prefix are defined in simulator 1. Thus, using route aggregation, only three routing table entries (one for each rlink) are sufficient for simulator 0 to make correct routing decisions in all cases. Both pdns and GTNetS provide commands to specify this type of aggregated routing entries for the remote links. In this simple case, assuming the use of NIx–Vector routing within each federate, we need routing state in each of the gateway nodes of size O(f), where f is the number of federates in the distributed simulation.

\[1\] Details of how this is done are dependent on the implementation, but not important for this discussion
Since $f$ is rarely more than a few hundred, this can be expected to be quite memory efficient.

**A More Complicated Example.** It appears from the above discussion that the problem of inter–federate routing in space–parallel network simulations is easily solved. However, consider the slightly modified topology shown in figure 2. This topology is nearly identical to the previous example, excepting the addition of two more intra–subnet links, connecting certain hosts to hosts in a neighboring subnet, and two extra links from gateway node $G0$ to neighboring interior routers. With this topology, the simplifying assumptions observed for the previous example no longer hold, and inter–federate routing of packets becomes much more difficult to manage.

First, the notion of the default route, indicating that all packets not destined to a local IP Address should be routed to a common gateway, can no longer be used. Thus, each node in each simulator will need a routing table (potentially with $O(n)$ entries, $n$ being the total number of nodes in the simulation) to select which inter–federate gateway node is the best path to remote nodes.

Secondly, the clean and simple route aggregation method that worked nicely on the previous example may no longer work. Now, gateway node $G0$ has four rlinks, two to subnet 1 and two to subnet 3. The assignment of the IP Addresses to nodes in subnets 1 and 3 will affect how well the route aggregation will work for the rlink routing entries. In the best case, we can still use a single aggregate entry for each rlink, but in the worst case we need routing entries for every node in subnets 1 and 3 in the routing table for $G0$. The end result of both of these problems together is that we still need routing state of size $O(mn)$, where $m$ is the number of nodes managed in each federate, and $n$ is the total number of nodes in the global topology. We point out that the $O(mn)$ memory requirement is worst case, and in practice we still expect some saving from route aggregation.

**Using NIx-Vector Routing.** An efficient source–routing methodology called NIx-Vector routing is discussed in [5]. With this method, a route between a source and a destination is calculated only when needed, and is cached at the source for later re–use. Further, the calculated path from the source to the destination is stored in the packet using the compact NIx-Vector format, which allows intermediate routing decisions to be made without the use of routing tables. However, this approach requires a global picture of the topology from the source to the destination. Clearly, in the space–parallel methodology, this global topology knowledge is not available. However, we can provide additional routing information at the rlinks which allows a partial NIx-Vector to be calculated within a federate.

Suppose host $H00$ is sending a packet to host $H13$. Since host $H13$ is managed by simulator 1, simulator 0 lacks global knowledge of the topology to calculate a NIx-Vector to $H13$. However, if each rlink in simulator 0 has routing information specifying which addresses are reachable from this link, and how many hops to each, a NIx-Vector that routes the packet to the appropriate gateway can be calculated using a modified Breadth First Search (BFS) algorithm. In our example, the rlink from $G0$ to $G1$ will have a routing entry indicating it can reach $H13$ in three hops, and the rlink from $H03$ to $H10$ will indicate it can reach $H13$ in five hops. The modified BFS algorithm will calculate that the shortest path from $H00$ to $H13$ should use the rlink from $G0$ to $G1$, and calculates a NIx-Vector from $H00$ to $G1$ (the last hop is the rlink from $G0$ to $G1$).

This method has some of the benefits of NIx-Vector routing, in that no routing tables are needed at any node excepting those with rlinks to other federates. The memory requirements are still $O(kn)$ ($k$ is the number of inter–federate rlinks, and $n$ is the total number of nodes in the simulation). Further, in all cases except the simplest topologies, the calculation of these inter–federate routes can be time consuming. For example, we computed inter–federate routes for the million–node MILNET topology defined by Liljenstam et. al[6]. The off–line computation took 4 hours on a 866Mhz desktop processor, resulted in more than 5 million inter–federate routes, and required over 500MB of disk space to store the computed routing information.

**Using Pre–Computed Routes.** An easy approach to inter–federate routing is to use Pre–Computed Routes. In this approach, an off–line program creates a complete picture of the simulated topology, using a simplified and memory efficient representation of the topology. Once this complete topology model is created, a complete set of routing information can be computed for all nodes, giving paths to all other nodes. An advantage of this approach is that the time–consuming route computation step can be performed once, and used repeatedly in the simulation runs. The obvious disadvantage of this method is the $O(n^2)$ memory requirements for the all–pairs routing tables. For example, if the entire topology consists of one million nodes, the resulting pre–computed routing tables would consist of $10^{12}$ entries, consuming unreasonably large amounts of disk space and memory. This approach is used by the distributed memory Dartmouth SSF (DaSSF) simulator[7] described in [8].

**Using Routing Protocols.** Another approach to inter–federate and intra–federating routing in network simulations is the use of simulated Routing Protocols within the simulation. For example, we could include a model of the widely–used Border Gateway Protocol (BGP) on each node with inter–subnet connections. In the example in figure 2, this would be nodes $G0$, $G1$, $G2$, $G3$, $H00$, $H03$, $H10$, $R10$, $H13$, $H20$, $H23$, $H30$, $R31$, and $H33$. Further, we could use an Interior Routing Protocol, such as EIGRP[9, 10] or OSPF[11] on interior routers within a subnet (such as nodes $R00$, $R01$, $R11$, $R20$, $R21$, and $R30$ in our example). This approach is used by the SSFNet simulator[12, 13], and results in an easy to use space–parallel simulation. Additionally, this method inherently deals with dynamic topology changes, such as reacting to link or node failures. When creating the simulated topology, the user need not be concerned about routing information, since the routing protocols will compute the best routes using routing message exchanges between federates. Further, these routing protocols use route aggregation techniques to reduce the size of the resulting routing tables as much as possible. However, this approach still requires simulator memory to hold the routing tables calculated by the routing protocol, which in the worst case is still $O(mn)$. 


3 Ghost Nodes in GTNetS

In this section, we discuss the basic design of the space–parallel distributed simulation support in GTNetS, with particular attention to the ghost node approach. A GTNetS network simulation is created by writing a C++ main program, that instantiates objects representing the network topology (nodes, links, queues, etc.), the data flows and applications (web servers, web browsers, FTP clients, etc.). Also each GTNetS simulation instantiates a single Simulator object that controls the simulation (maintains the pending event list and schedules events).

GTNetS supports both sequential, single–process simulations as well as space–parallel distributed simulations. We expect that the majority of GTNetS simulations will use sequential execution, so we wanted to make the distinction between sequential and distributed as simple as possible. To this end, we simply provided two versions of the object constructor for the Simulator object, one with no parameters and one with a single Distributed Simulation Identifier parameter. For sequential simulations, the default constructor without arguments is specified by the user, in which case none of the distributed simulation support functions are called, and the complete topology is assumed present in the single address space. See figure 3 for a simple code snippet. The remainder of this section will focus on the distributed simulation methods.

To specify a distributed simulation, the Simulator object is instantiated with a single integer argument, assigning an instance identifier to this simulator that is unique within the federated simulation. If there are to be \( k \) federates in the distributed simulation, the instance identifier is in the range of \( 0 \ldots (k-1) \). When the Simulator object is constructed in this manner, GTNetS will call the necessary distributed simulation support functions in the RTI, such as initialization functions, data distribution management, and time management requests. Further, the instance identifier is used to determine if node objects are to be real nodes or ghost nodes, as discussed in the following paragraphs.

The next action needed in the distributed simulation script is to identify, for every node in the topology, which simulator instance is responsible for that node object. This is accomplished by providing a node object constructor with a single argument which specifies an instance identifier. If the specified instance identifier matches that specified on the Simulator object constructor, then this simulator is responsible for the node object, and a real node object is created. Otherwise, a ghost object is created.

See figure 4 for a simple code snippet showing a distributed simulation instance. Note that the only differences (other than command line argument processing) are the myId parameter passed to the Simulator constructor, and the single integer arguments passed to the Node object constructors. In this simple example, one simulator process would be initiated with the command line argument “0", and the second would be initiated with the command line argument “1". Notice that when node objects \( n1 \) and \( n2 \) are created, the Node constructor is called with the arguments 0 and 1 respectively, indicating that node \( n1 \) is to be modeled on simulator 0, and \( n2 \) is to be modeled on simulator 1. In simulator 0, node \( n1 \) is a real node and \( n2 \) is a ghost. In simulator 1, node \( n1 \) is a ghost node and \( n2 \) is a real.

There are two important points to be seen from this simple example. First, there is little difference from the users’ perspective between the sequential simulation and the distributed simulation. The only differences are the presence of the instance id parameter on the Simulator constructor, and the responsible instance id parameter on the Node constructor. Excepting a few other minor differences discussed later, the remainder of the script is identical. Secondly, each simulation instance runs exactly the same script. Using this method, we don’t need to create a different GTNetS main program for each simulator in the distributed simulation. Each federate runs the same program, differentiated with command line parameters.

```c++
#include "simulator.h"
#include "node.h"
#include "linkp2p.h"

int main()
{
  // Simple sequential simulation
  Simulator s; // Sequential simulation
  Node* n1 = new Node(); // Node 1
  Node* n2 = new Node(); // Node 2
  // Define a link object
  Linkp2p link(Rate("1Mb"),
               Time("10ms"));
  // Add the link from n1 to n2
  n1->AddDuplexLink(n2, link,
                     IPAddr("192.168.1.2"), Mask(32),
                     IPAddr("192.168.1.1"), Mask(32));

  return 0;
}

int main(int argc, char** argv)
{
  // Simple distributed simulation
  Simulator s; // Distributed simulation
  int myId = atol(argv[1]);
  Simulator s(myId); // Distributed sim
  Node* n1 = new Node(0); // Node 1
  Node* n2 = new Node(1); // Node 2
  // Define a link object
  Linkp2p link(Rate("1Mb"),
               Time("10ms"));
  // Add the link from n1 to n2
  n1->AddDuplexLink(n2, link,
                     IPAddr("192.168.1.1"), Mask(32),
                     IPAddr("192.168.1.2"), Mask(32));

  return 0;
}
```
Ghost Node Implementation. From the above discussion, it is clear that in GTNetS the Node objects come in two flavors, real nodes and ghost nodes. Equally clear is that the API for the two node types (i.e., the set of member functions available to object owners) must be identical or nearly identical. If this were not the case, there would be many conditional checks in the simulation script to take different action depending on the real or ghost status of the nodes. Note for example the call to AddDuplexLink for Node object n1 in the above example. While the actions taken in this method are different for real and ghost nodes, the API is the same. In fact, all Node methods are identical for real nodes and ghost nodes. Finally, the ghost node implementation must be memory efficient, utilizing as little state as possible. If this were not the case, the advantage of exploiting multiple processors to simulate larger networks would be lost, since the entire topology is required on every federate.

We achieve these goals by using a simple one-level method indirection. The basic Node object has all the API methods needed by GTNetS to manage nodes, but only has a single Implementation Pointer state variable. This implementation pointer points to an object that is a subclass of class NodeImpl. The NodeImpl class defines the required set of methods needed for nodes, but only has state common to ghost nodes and real nodes. The only common state between ghost nodes and real nodes is the IP Address and a vector of Interfaces. In this context, the word Interface refers to a simulation model of a hardware link interface (such as a NIC card) in a router or end system. Finally, there are two classes that are subclasses of NodeImpl, namely NodeReal and NodeGhost. Objects of class NodeReal have all the state associated with real nodes, such as port maps, routing information, animation size and color, and location information.

When a node is created in a distributed simulation, the Node constructor checks whether the system identifier specified in the constructor argument matches that specified in the Simulator constructor. If so, this node is real, and a new object of class NodeReal is created and pointed to by the implementation pointer. If the system identifiers do not match, the node is a ghost, and a new object of class NodeGhost is created.

We mentioned previously that both real and ghost nodes maintain a list of Interfaces that model the link interfaces in nodes and routers. This seems at first glance to be inefficient in terms of memory usage, since these interfaces have a substantial amount of state (for example a packet queue) that is not necessary for ghost nodes. We solve this problem by defining two Interface subclasses, InterfaceReal (which has the state needed to model an interface), and InterfaceGhost (which does not). When a new Interface object is needed by a node object, real nodes create a real interface, and ghost nodes create a ghost interface. Similarly, we use real and ghost Link objects for the same purpose. The key point is that the API is common across real and ghost objects, such that any owner of these objects can call the defined methods without regard to whether the object is real or a ghost.

Using this technique of real and ghost objects, each simulator in the distributed simulation has a complete picture of the simulated topology, and can compute NLx-Vector routing information from any source to any destination. We show in the next section that the overhead incurred by ghost objects is small compared to the overall memory footprint of the simulation.

int main(int argc, char** argv) {
    // Simple distributed simulation
    // Get instance id from arguments
    int myId = atol(argv[1]);
    Simulator s(myId); // Distributed sim
    // n1 is managed by simulator 0
    Node* n1 = new Node(0); // Node 1
    // n2 is managed by simulator 1
    Node* n2 = new Node(1); // Node 2
    // Add WebServer application to n1
    WebServer* svr = n1->AddApplication(
        TCPServer());
    if (svr) {
        // Application added
        svr->EnableGCache();
    }
    // Add WebBrowser to n2
    WebBrowser* br = n2->AddApplication(
        WebBrowser(...));
    if (br) {
        // Added, configure and start
        br->ConcurrentConnections(4);
        br->Start(0.1);
    }
}

Figure 5: Adding Applications

There is one case where the behavior of a ghost node and a real node can require the simulator user to be aware of whether the node is real or not. All of the previous discussion has focused exclusively on the topology generation part of the simulation script. In a network simulation, we also need to define the flow of data between the nodes in the topology. In GTNetS this data flow is defined using Application objects. GTNetS presently has defined application models for thirteen different application behaviors, including web browsers, web servers, Gnutella clients, and others. However, we do not use the concept of ghost applications. Since applications are added to nodes using the AddApplication method for node objects, a simpler method is to design ghost nodes to ignore any request to add an application. Since the semantics of the AddApplication method are that it returns a pointer to the newly added application object, the design is that ghost nodes simply return a NULL pointer instead. The user simulation scripts simply check for a NULL return from the AddApplication call, and if so skips further application initialization. See figure 5 for a code snippet illustrating this point. While the script does not directly determine whether an application is being added to a ghost node or a real node, it detects the NULL return to differentiate between the actions performed by the two node types.

Consistent Topology View. It is apparent that, for the ghost node approach to work properly, all federates must have a consistent view of the global topology being modeled. While this seems easy to achieve, there are two instances that can cause problems with this requirement.

First is the use of randomly generated topologies, using
a tool such as the Georgia Tech Internet Topology Modeler (GTITM)[14]. In our GTNetS simulator, a single C++ object can represent an arbitrarily large network, generated randomly based on the GTITM technique. Thus, different federates could randomly generate different topologies, thus violating our consistent view constraint. In this case, care must be taken to insure the random number generators are seeded in a deterministic way, to insure each federate generates identical random topologies.

The second issue is the modeling of link or node failures in a network. If a given federate has a real node representation of a given network node, and generates a random node failure event, other federates must be made aware of this failure. Although not implemented in our simulator, it is straightforward to use state update events between federates to achieve these notifications. Further, these state update events need not be sent between federates with zero simulation time advance, since node and link failures cannot be detected in a network any faster than packets can flow through the network. It is well known that zero time update events between federates leads to poor performance.

The design of GTNetS leads to an easy to use, and low overhead way to manage a space–parallel network simulation. The ghost nodes give the simulation the necessary topology information to calculate NlX-Vector routing information, while at the same time use little memory.

4 Related Work

There are several network simulation tools available that use a space–parallel approach to distributed simulation. Parallel/Distributed ns (pdns) by Riley[15, 16] (based on the venerable ns2[17] simulator), and has used the space–parallel approach from its inception. The SSFNet simulator was initially designed for parallel simulation in a multi-threaded shared–memory environment, but has since been adapted by Liu and Nicol[8] to the support distributed memory platforms. The Dartmouth SSF (DASSF) simulator[7] also has been adapted for a distributed simulation with a space–parallel methodology. Wu et. al[18] report some limited success in adapting the commercial OPNet simulator[19] to operate in a distributed environment, using a space–parallel approach.

The concept of using limited state objects as place–holders for remote objects is not new. In the Distributed Interactive Simulation (DIS) community, tools such as SIMNet[20] often use dead reckoning or other approximation methods to estimate the state of objects that are managed in remote federates. In a battlefield simulation for example, a federate may report the position and velocity of a tank object at a particular point in time. Other federates will maintain the tank’s location by assuming a constant velocity until informed otherwise. Ferenci[21] discusses the use of Proxy Objects in distributed simulations, which are conceptually similar to our ghosts. However, Ferenci’s proxy objects exist primarily to facilitate inter–federate message routing, and do not in fact represent any global state. Additionally, Ferenci discusses his approach in the context of optimistic simulations, where we deal exclusively in the conservative environment.

To our knowledge, we are the first to apply the limited–state object method to represent the global topology information in space–parallel network simulations.

5 Experimental Results

To demonstrate the effectiveness of the ghost node approach, we ran three sets of experiments to measure the memory usage of the space–parallel network simulations, using both the traditional approach with manually specified inter–federate routing and our new ghost node approach. The first set of experiments uses a simple topology similar to that shown in figure 6. This topology consists of $k$ subnetworks ($k$ is eight in the figure shown), each with $n$ nodes arranged in a star topology ($n$ is sixteen in the example). Each of the subnetworks is connected to its neighbors, forming a ring. This topology was chosen since it is the best possible case for the traditional approach. Each of the leaf nodes in the star subnetwork can use the simple default route method to reach the single gateway node. At each gateway, the route aggregation method can easily and efficiently specify which addresses are reachable on each of the rlinks.

The simple topology was run on eight federates, with varying numbers of leaf nodes per subnetwork. The experiments were performed using both the traditional approach and the ghost node approach. One hundred and fifty TCP flow endpoints were assigned to leaf nodes, and 1MB transfers we simulated. The memory usage and execution time of each simulation is shown in table 1. Since in this experiment, all federates model an identical subnetwork, results are only shown for federate zero. As can be seen, the memory footprint for the ghost node approach is only slightly larger than the traditional method. For the 120,008 node case (the largest we performed), there were 15,001 real nodes and 105,007 ghosts. The ghosts required a total of 19 MB of memory, representing 16 percent of the total memory footprint.

Interestingly, the overall execution time for the ghost node approach is less than the traditional approach. Using ghosts, we pay a one–time cost for the calculation of the NlX-Vector, but gain an $O(1)$ routing decision at each hop in the path. Without NlX-Vectors, the routing at gateway nodes and hubs requires an $O(k)$ computation ($k$ is the number of IP Addresses assigned to the node) to determine if the packet has arrived at the destina-


### Table 1: Star Topology Memory Usage

<table>
<thead>
<tr>
<th>Star Size</th>
<th>Total Nodes</th>
<th>Ghost Nodes</th>
<th>Memory MB</th>
<th>Execution Time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,501</td>
<td>12,008</td>
<td>10,507</td>
<td>No Ghosts</td>
<td>45 156</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Ghosts</td>
<td>47 150</td>
</tr>
<tr>
<td>3,501</td>
<td>28,008</td>
<td>24,507</td>
<td>No Ghosts</td>
<td>53 188</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Ghosts</td>
<td>57 152</td>
</tr>
<tr>
<td>7,501</td>
<td>60,008</td>
<td>53,507</td>
<td>No Ghosts</td>
<td>68 271</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Ghosts</td>
<td>77 150</td>
</tr>
<tr>
<td>15,001</td>
<td>120,008</td>
<td>105,007</td>
<td>No Ghosts</td>
<td>95 331</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Ghosts</td>
<td>114 228</td>
</tr>
</tbody>
</table>

![Figure 7: Milnet Topology](image)

We are looking into a less burdensome way to make this decision, to reduce this to $O(\lg k)$.

The second set of experiments are nearly identical to the previous one, excepting the use of a star size of 7,500 and the addition of more data flows. The data flows were composed of three hundred TCP clients (10MB of data each), six TCP servers, two thousand web browsers (requesting random web objects), and two hundred webservers. The simulation time was five hundred seconds. On average, the memory used by each federate was 71MB using the traditional approach, comparing to 80MB using the ghost node approach. The execution time was higher using the traditional approach as expected; 1,215sec compared to 1,111sec using the ghost node approach.

The last set of experiments used a large and complex topology known as the MILNet defined by Liljenstam et. al[6]. This topology consists of a backbone network containing over three thousand routers and eleven thousand links, that roughly models the backbone network for United States military bases. Connected to the backbone are 163 subnetworks of various sizes from five hundred to nine thousand nodes each. The entire network exceeds one million nodes. A graphical representation of the MILNet backbone is shown in figure 7.

The results from the MILNet experiments are shown in table 2. The MILNet topology was divided between 8 federates, with federate zero modeling the high-speed backbone and the other federates modeling approximately equal parts of the remaining nodes. A graphical representation of the MILNet backbone is shown in figure 7.

![Figure 7: Milnet Topology](image)

The results show that the memory used for ghosts considerable, but in most cases a small fraction of the total memory usage. The exception is for federate zero, which models only 3,070 of the high-speed backbone routers of MILNet. This federate has more than a million ghosts, using 143MB of memory, which is 66 percent of the total. In all other federates, the ghosts take between 100MB and 150MB, accounting for around 16 percent of the total.

Interestingly, the initialization time for the ghost node simulation is less than half of that of the traditional method. In this experiment, the entire topology is specified in a large XML file, which must be read in its entirety by both approaches. However, the traditional approach also requires the population of the inter-federate routing information. As mentioned, this information consists of over 5 million individual routes, which take considerable time to read and process, as evident by the larger initialization times.

## 6 Conclusions and Future Work

We have shown that the ghost node approach is a viable method to achieve efficient and easy-to-use space-parallel network simulations. The memory required for the ghosts is non-zero, but small relative to the overall memory footprint of a large-scale network simulation. Using ghost nodes, no precomputation of routing information is needed and the memory-efficient Nix-Vector routing method can be used. The implementation of ghost nodes in GTNetS allows the same simulation script to be used for all federates, with simple command line parameters identifying node mapping.

Even with the ghost node approach, simulation user must still specify the mapping of node objects to federates. In all but the simplest cases, determining a suitable and efficient mapping is challenging and requires considerable analysis of the traffic patterns between the simulated network elements. Liu and Chien[22] describe an automated method to partition networks which they used in their MicroGrid[23] emulation tool. These results seem promising, and we are investigating their applicability to our ghost node approach for space-parallel network simulation.

### References

Table 2: MILNet Memory Usage

<table>
<thead>
<tr>
<th>Federate</th>
<th>Memory</th>
<th>Initialization</th>
<th>Nodes</th>
<th>Nodes</th>
<th>Memory</th>
<th>Initialization</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Total</td>
<td>Total</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>3,070</td>
<td>73</td>
<td>1,095,558</td>
<td>1,098,628</td>
<td>692</td>
<td>299</td>
</tr>
<tr>
<td>1</td>
<td>181,268</td>
<td>704</td>
<td>917,360</td>
<td>1,098,628</td>
<td>699</td>
<td>299</td>
</tr>
<tr>
<td>2</td>
<td>144,769</td>
<td>568</td>
<td>953,859</td>
<td>1,098,628</td>
<td>696</td>
<td>299</td>
</tr>
<tr>
<td>3</td>
<td>141,421</td>
<td>557</td>
<td>957,207</td>
<td>1,098,628</td>
<td>695</td>
<td>299</td>
</tr>
<tr>
<td>4</td>
<td>150,060</td>
<td>588</td>
<td>948,568</td>
<td>1,098,628</td>
<td>699</td>
<td>299</td>
</tr>
<tr>
<td>5</td>
<td>151,465</td>
<td>593</td>
<td>947,163</td>
<td>1,098,628</td>
<td>697</td>
<td>299</td>
</tr>
<tr>
<td>6</td>
<td>171,224</td>
<td>671</td>
<td>927,404</td>
<td>1,098,628</td>
<td>705</td>
<td>299</td>
</tr>
<tr>
<td>7</td>
<td>155,351</td>
<td>606</td>
<td>943,277</td>
<td>1,098,628</td>
<td>698</td>
<td>299</td>
</tr>
</tbody>
</table>


