# Scalability and Composability Techniques for Network Simulation

A Thesis
Presented to
The Academic Faculty

by

**Donghua Xu**

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy

College of Computing
Georgia Institute of Technology
May 2006

# Scalability and Composability Techniques for Network Simulation

Approved by:

Dr. Mostafa H. Ammar
(Co-Advisor)
College of Computing
*Georgia Institute of Technology*

Dr. Richard M. Fujimoto
(Co-Advisor)
College of Computing
*Georgia Institute of Technology*

Dr. George F. Riley
School of Electronic and Computer Engineering
*Georgia Institute of Technology*

Dr. Jun Xu
College of Computing
*Georgia Institute of Technology*

Dr. Douglas Blough
School of Electronic and Computer Engineering
*Georgia Institute of Technology*

Date Approved: January 13, 2006

*To my parents,*

*for their unconditional love and support.*

# TABLE OF CONTENTS

# LIST OF FIGURES

# SUMMARY

Simulation has become an important way to observe and understand networking phenomena under various conditions. Typically, researchers run a serial network simulator on a workstation for a time period to obtain results. However, as the demand to simulate larger and more complex networks increases, the limited computing capacity of a single workstation and the limited simulation capability of a single network simulator have become significant obstacles. In this research we develop techniques that can scale a simulation to address the limited capacity of a single workstation, as well as techniques that can compose a simulation from different simulator components to address the limited capability of a single network simulator.

We scale a simulation with two different approaches. First, we reduce the resource requirements of a simulation substantially, so that larger simulations can fit into one single workstation. In this thesis, we develop three technique's, Negative Forwarding Table, Multicast Routing Object Aggregation, and NIx-Vector Unicast Routing, to aggregate and compress the amount of superfluous or redundant routing state in large multicast simulations. Each of the three techniques is effective in its own right, and combining them allows us to perform multicast simulations much larger than previously could be completed.

The second approach to scale network simulations is to use parallel processing. We partition a simulation model in a way that makes the best use of a computer cluster, and distribute the simulation onto the different processors of the cluster to maximize performance. We develop a novel empirical methodology called BencHMAP (Benchmark-Based Hardware and Model Aware Partitioning) that runs small sets of benchmark simulations to derive formulas for calculating the weights that are used to partition the simulation on a given computer cluster. With this methodology we are able to derive a good trade-off between various performance factors, and achieve performance better than other well-known partitioning methods.

To address the problem of the limited capability of a network simulator, we develop techniques for building complex network simulations through the composition of independent components. Different network simulators offer different strengths and weakness in modeling different layers/scenarios. It would be more effective for each simulator to execute the layers where it excels, using a simulation backplane to interface different simulator components. With this approach we demonstrate the construction of very complex wired and wireless hybrid network simulations using three network simulators: GloMoSim, ns2 and PDNS.

In this thesis we demonstrate that these techniques enable us to not only scale simulations by orders of magnitude with a good performance, but also compose complex simulations with high fidelity.

# CHAPTER I

# INTRODUCTION

## *1.1 Motivation*

In networking research and engineering, simulation has become an important way to study network operation and performance characteristics under various conditions. A number of packet level network simulation tools have appeared (e.g., see [26, 4, 52]) to help researchers construct and simulate various kinds of network scenarios.

Packet-level network simulations usually model the network as a graph of nodes and edges, with nodes representing hosts, and edges between nodes representing communication links. To simulate packets traveling across the network, the simulator models 1) the network protocol stack at each node to simulate packet processing at each protocol layer, and 2) the link characteristics (bandwidth, propagation delay etc.) of each link, to simulate the packet's behavior in transmission.

Rather than the behavior of a single packet, network researchers are usually more interested in the aggregate behavior of many packets, such as TCP throughput [12]. Therefore, a meaningful simulation usually involves simulating a large number of packets being transferred through the network. With the level of detail being modeled in packet-level network simulation, large, complex network simulations can be very resource intensive. This has often made it difficult and sometimes infeasible to run large-scale network simulations on a single workstation.

For example, Nonnenmacher and Biersack [29] used a mathematical model to analyze their scalable multicast feedback mechanism for large groups of up to one million members. However, they were not able to perform packet-level simulations to validate this analysis due to memory requirements for multicast simulations at this scale. Instead, they had to rely on certain assumptions on the end-to-end delay distribution in their coarse simulation. As another example, Shi and Waldvogel [43] proposed a sender-based congestion control

approach for multicast traffic and used *ns2* to evaluate their approach. However, they were not able to both construct large multicast groups and maintain large number of SRM flows at the same time in the simulation. As a result they had to maintain only two receivers per group in their simulation.

In these examples, the really meaningful validation would be large scale simulations, since conclusions from small scale simulations might not be valid when scaled. Research in large scale networks demands the ability to simulate large network. As more and more networking research involves large networks, the demand for large scale network simulation can be expected to increase.

In addition, as there exist a number of different network simulators, and each simulator excels at modeling different network protocol layers or scenarios, it sometimes becomes difficult to choose the "right" simulator. The ability to combine the strengths of different simulators offers clear advantage.

## *1.2 Challenges and Approaches*

### 1.2.1 Scalability

To address the scalability, there are two general directions we can take:

1. We can partition a large simulation into a number of parts and distribute them onto different processors to run them in parallel to maximize performance.

   A challenge with this approach is to determine how to partition the simulation to achieve the best possible performance. More specifically, the challenge is two-fold:

   (a) There are three well-known factors that affect parallel and distributed simulation performance: load balancing, lookahead, and communication overhead[15]). Ideally we want to partition the simulation to achieve a load distribution that is as even as possible, a lookahead as high as possible, and a communication overhead as low as possible all at the same time. However, the effect of the three factors often offset each other: a partitioning with an even distribution of workload could result in a low lookahead, and a partitioning with a high lookahead

could result in high communication overhead.

(b) The simulation performance is also determined by the specific characteristics of the computer cluster that runs the simulations. Any difference in the hardware aspects such as CPU, cache, memory or interconnect, or the software aspects such as operating system, file system, compiler, or inter-processor communication mechanism could result in different performance. There is so far no theoretical model that captures all these characteristics to guide the simulation partitioning for good performance on a given computer cluster. One partitioning of a large network simulation that performs the best on one computer cluster might not perform the best on a different computer cluster.

Therefore, to achieve a good trade-off between the three factors on a specific given computer cluster, we develop a empirical methodology called BencHMAP (Benchmark-Based Hardware and Model Aware Partitioning) that runs a series of small benchmark parallel simulations on the target computer cluster, and derives the representative performance-factor relations (usually in the form of a set of equations) from the measurements of these benchmark simulations, reducing the network simulation partitioning problem into the well-known *weighted graph partitioning* problem [42], then use a graph partitioning tool [23] to partition the original network simulation with these derived relationships to achieve good performance.

2. On the other hand, we can also reduce the resource requirements of a network simulation so that larger simulations can fit into a single work station. As seen in the examples in Section 1.1, the difficulty to run large-scale packet-level multicast simulations has hampered the validation of much multicast-related research.

   The main challenge here is to reduce the resource requirements without sacrificing the accuracy of the simulation results. It is possible to abstract away certain simulation details to reduce resource requirements, however the validity of the simulation results could become questionable because of the loss of accuracy.

   We observe that the reason multicast simulations are hard to scale is because these

simulations typically maintain a large amount of routing state in memory, and a significant portion of this state is in fact superfluous or redundant. Therefore, we develop techniques to aggregate and compress this state so that memory requirements are significantly reduced. With these techniques we are able to run multicast simulations orders of magnitude larger than were possible previously.

### 1.2.2 Composability

To attack the issue of exploiting particular features of two or more different simulators, we develop techniques to run different network simulators together to simulate complex network simulation scenarios. For example, we want to utilize the rich TCP implementations of the ns2 simulator as well as the high fidelity of wireless MAC layer simulation of the GloMoSim simulator within a single model.

The challenge here is to make different simulators run simultaneously and cooperating with each other. Our solution is to use an underlying simulation backplane [38] to glue the simulators together, so that each simulator runs the portion of simulation model for which it excels, exchanges messages with each other and synchronizes with each other through the simulation backplane and a run-time infrastructure [34]. With this approach, we are able to run complex simulations that involve wireless edge network and backbone wired networks with three types of simulators: GloMoSim, ns2 and PDNS.

## *1.3 Thesis Contributions*

The main contributions of this thesis are:

1. We propose BencHMAP: Benchmark-Based, Hardware and Model-Aware Partitioning for Parallel and Distributed Network Simulation, a systematic methodology to produce a partitioning result of large network simulations to maximize parallel performance on a given computer cluster. It is a novel approach in that it empirically derives performance-factor relationship for a specific computer cluster through a series of benchmark experiments. We provide preliminary evidence that this approach can

effectively partition a large network by observing that it provides twice the performance of simulations using other partitioning methods. We also develop a tool called "AutoPart" to help automatically partition large ns2 network simulations so that it can be run with PDNS, the parallel and distributed version of ns2.

2. We propose three techniques to reduce the memory requirements of large scale multicast simulations by aggregating and compressing routing state in multicast simulations. The three techniques, namely Negative Forwarding Table, Multicast Routing Object Aggregation, and NIx-based Unicast Routing, each can reduce the memory requirements of multicast simulation. We show that by combining all three techniques together, we are able to improve the multicast simulation scale by orders of magnitude. In addition, the Negative Forwarding Table technique can potentially be applied to Application Layer Multicast to reduce multicast routing state memory requirements.

3. We develop a framework to compose a simulation from different simulators which excel at modeling different protocol stack layers and cooperate through an underlying simulation backplane. We show that complex wired and wireless hybrid network simulations can be constructed by running GloMoSim, ns2 and PDNS together on top of the simulation backplane.

## *1.4 Thesis Organization*

The rest of this proposal is organized as follows. Chapter 2 presents the BencHMAP methodology to partition network simulations. Chapter 3 introduces memory saving techniques for large multicast simulations. Chapter 4 presents the split protocol stack method of network simulation composition, and Chapter 5 summarizes the work in this thesis and presents future work. Appendix A presents AutoPart, the automatic partitioning tool we develop for large network simulations.

# CHAPTER II

# *BENCHMAP:* BENCHMARK-BASED, HARDWARE AND MODEL-AWARE PARTITIONING FOR PARALLEL AND DISTRIBUTED NETWORK SIMULATION

Computer simulation of large-scale and complex networks can be resource intensive. Several tools to parallelize and distribute the simulation to a number of different machines have been developed. One of the main challenges facing users of these tools is how to partition the simulation among the computing resources available. The focus of this chapter is on the development of a framework and methodology (ultimately leading to a semi-automated tool) to partition network simulations. The main distinguishing feature of our approach is that the partitioning is performed in a manner that takes into account the specific distributed computation environment as well as the specific details of the network model. To achieve this, we derive the relations between impact factors and the simulation performance from measurements of *benchmark experiments*. We then apply the derived relations to the given network topology and workload model to construct a weighted graph which we then partition using a graph partitioning tool. Experiments on a 120k-node, 100k-stream network simulation show that the full application of this approach improves performance significantly over other partitioning heuristics.

There has been extensive research on general computation and data partitioning for parallel and distributed execution, such as [11, 13, 44]. This work mostly deals with compiler-level optimization for parallel programs, trying to automatically parallelize general computation tasks by optimizing various cost factors.

There is also some work (such as [28, 25, 51]) specifically concerning partitioning network simulation or emulation. The work in [28] focuses on the theoretical analysis of how factors such as load balancing and communication overhead impact simulation performance, with

the presumption that the effect of the simulation model on the performance factors is already known. The work in [25, 51] makes the simplifying assumption that the effect is constant. This work does not consider details of the specifics of the actual architecture of the computer cluster or distributed computing hardware.

The main distinguishing feature of our approach is that the partitioning is performed in a manner that takes into account the specific distributed computation environment available as well as the specific details of the network model including both the network topology and the traffic characteristics. To achieve this, we derive the relations between impact factors and the simulation performance from measurements of *benchmark experiments*. These benchmark experiments are designed to explore the performance of parts of the specific model to be partitioned and run on the same specific computing hardware.

This chapter is organized as follows. In the next section we present an overview of our BencHMAP methodology. In section 2.2 we give a brief background on parallel and distributed simulation, and discuss conceptually how the properties of the simulation model represent factors that affect the performance. Section 2.3 will present more details of the partitioning procedure, particularly on how to derive the factor/performance relations based on the results of the benchmarking experiments. Section 2.4 gives concrete examples of factor/performance relations on our computer cluster, and Section 2.5 presents the performance of simulation partitioned using our BencHMAP heuristic and how this performance compares when other heuristics are used. Then Section 2.6 concludes the chapter.

## *2.1 Overview of BencHMAP*

Given a network simulation model and a computer cluster, our goal is to partition the network simulation model into a number of parts to be run on the cluster, so that the partitioned simulation runs as fast as possible.

Compared to some other computation partitioning tasks, network simulation partitioning has the obvious advantage that a network topology can be naturally mapped into a graph, where each network node is represented by a vertex, and each network link is represented by an edge in the graph. Therefore, we can reduce the network simulation

partitioning task into a graph partitioning task, and employ existing graph partitioning tools that utilize well-researched graph partitioning algorithms that can efficiently produce a high quality partitioning.

Our approach, then, is to reduce the simulation partitioning problem into the well-known *weighted graph partitioning* problem [42]. The weights on the graph are determined through a heuristic procedure that involves benchmarking of simulation tasks derived from the particular model to be partitioned on the computing environment (*computer cluster*) over which we desire to partition.

It is well known that there are three factors, *load balancing, lookahead and communication overhead*, that can affect parallel simulation performance [15]. We discuss these three factors in detail in the next section. The main challenge here is how to obtain a good trade-off among these three factors while taking into account the given simulation model and the computer cluster hardware configuration in order to make the best use of the given resources. This is difficult because there is no known analytical model to predict the effect of the simulation model and hardware configuration on these three factors in a particular simulation. In our BencHMAP methodology we employ a benchmark-based scheme to empirically determine how the performance of a given simulation model running on a specific hardware configuration would be affected by these three factors.

The overall partitioning procedure is shown in Figure 1. With the given computer cluster configuration and a simulation model (including network topology and workload or traffic model), first we need to construct and run a set of benchmark experiments to derive the factor/performance relations. We then apply the derived relations to the given network topology and workload model to construct a weighted graph. The graph is then input into a graph partitioning tool to produce a partitioned simulation model.

## *2.2 Factors Affecting Parallel Simulation Performance*

### 2.2.1 Load Balancing

Partitioning a simulation results in a division of the resource requirements (including computation, I/O, and memory requirement) of the simulation into a number of parts. Each

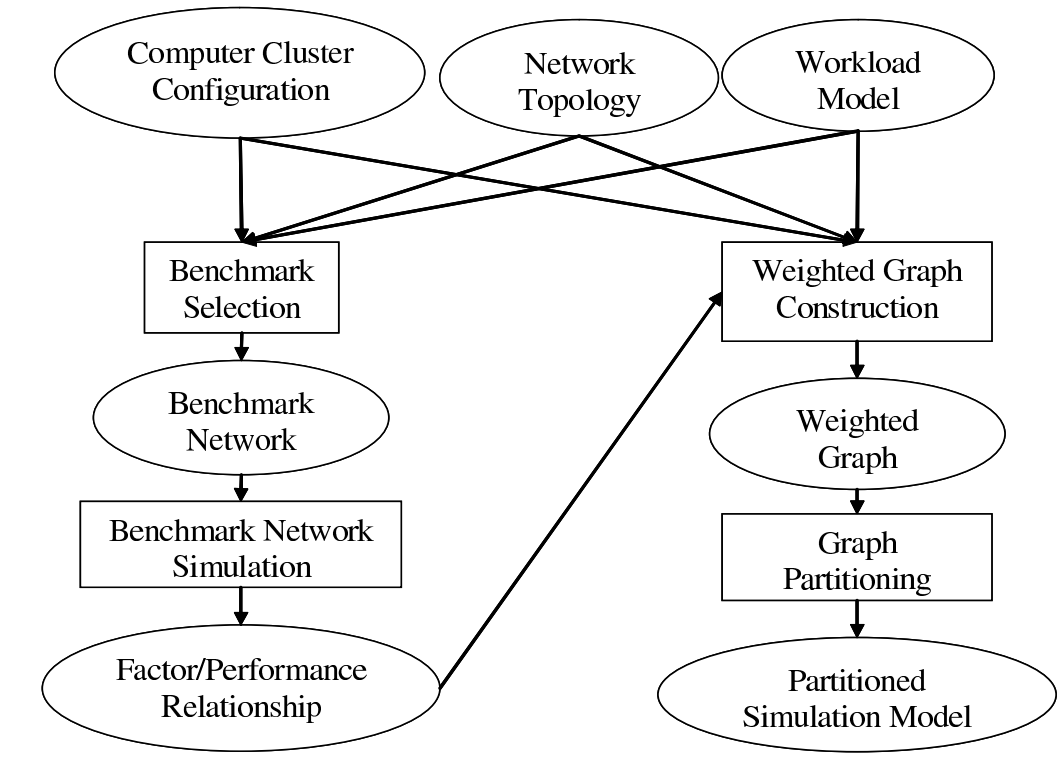


**Figure 1:** Procedure of Benchmark-Based, Hardware and Model Aware Partitioning

part is then allocated to a computing resource (processor and memory). Observe that a partitioned simulation is complete only when the last partition completes processing its last event. Ignoring communication and synchronization overhead, minimizing this time means that simulation should be partitioned such that the total event execution time (= number of events to be executed / rate of event processing at a machine) for each machine should be made equal. Note that this total event execution time is a complex function of CPU speed, memory speed, memory sharing architecture and other hardware and operating system factors. It is in general not possible to determine this number *a priori* for any hardware.

### 2.2.2 Lookahead

In the field of PDES (Parallel Discrete Event Simulation) research, the term *lookahead* is essentially a measure of the a logical process's ability to predict the future. There have been a number of definitions of lookahead (see e.g., [6], [18], [19], [47], [35], [22]). Here we take

the definition from [15] and say that if a logical process (LP) at simulation time $T$ has a lookahead of $L$, any event scheduled by the LP (at time $T$) must have a time stamp greater than or equal to $T + L$. In our parallel network simulation running on the conservative synchronization library RTIKIT [16], lookahead is the simulation time window in which a process can process its events without the fear that it will receive past events from other systems. In this work we based our PDES performance discussion and observation on the conservative synchronization mechanism of RTI [14].

A larger lookahead value leads to more parallelism and more speedup. This is because with large lookahead, there is usually less synchronization overhead necessary to complete the execution. When there are a large number of systems participating in the PDES, the cost of each step of synchronization (such as LBTS computation [15]) sometimes cannot be neglected. Since the number of LBTS computations in RTIKIT is roughly the total simulation time over the lookahead value, larger lookahead value means less number of times LBTS is computed, hence less total synchronization overhead.

When partitioning a packet-level network simulation, we can either split the simulation between protocol layers (*split-stack*) [50], or split at the links between network nodes (*split-link*). The split-stack approach is good for composing simulations from different network simulators that excel at specific protocol layers. However, it does not lead to much parallelism because it provides zero or near-zero lookahead, since network simulation models usually assume a packet takes no time or very little time to travel between protocol layers. The split-link approach, on the other hand, has the potential of providing much better parallelism, since in this case lookahead between two partitions takes the value of the lowest link propagation delay among the links that are split to provide the partition. We focus on the split-link approach in this chapter.

However, as we will see in later sections, there is sometimes a lookahead value threshold beyond which larger lookahead does not bring significant performance improvement. This is because this lookahead value already provides a large enough window, and keeps all systems executing partitions busy all the time without exhausting safe events before the next LBTS computation. In this case, there is no more room to improve on aspect 1) discussed above,

and the effect of aspect 2) can be neglected when the frequency at which LBTS is computed is small enough.

Therefore, our second simulation partitioning objective is to find the partitioning result whose lookahead (i.e. smallest split link propagation delay) is larger than the lookahead of other partitionings, while treating all partitioning results with a lookahead larger than the threshold as optimal in the sense of lookahead optimization.

### 2.2.3 Communication Overhead

Other than synchronization, another kind of interaction between different systems in a PDES is the remote scheduling of certain events. In particular, in packet-level network simulation, when a network node delivers a packet to a next hop through a link, it needs to schedule a receiving event at the next hop. But if the next hop is simulated by another system, i.e., the link is a split link, then information about this packet has to be actually transmitted from the current system to the next hop's system, and the receiving event at the next hop has to be remotely scheduled. The cost associated with this transmission sometimes cannot be neglected when different systems are connected through physical network links (e.g., an Ethernet) and not through shared memory, and there is a large number of such transmissions going on in the simulation. We call the packets transmitted between systems the *cross-border traffic*, or simply *CB Traffic*.

To reduce the total communication overhead, we want to find a partitioning that can reduce inter-processor traffic as much as possible. This means we need to estimate how much traffic each link would have to carry during the simulation, then try to split at the links with the least traffic.

## *2.3 The BencHMAP Methodology for Network Simulation Partitioning*

As shown in the flowchart in Figure 1, the overall framework of network simulation partitioning can be roughly divided into three segments: 1) the input of computer cluster configuration, network topology and workload model, 2) the determination of factor/performance

relations through benchmark experiments, and 3) partitioning the given network simulation model with the derived relations. In this section we will discuss these three segments in more detail, focusing on the three performance factors (load-balancing, lookahead, and communication overhead).

### 2.3.1 Input

**Network topology:** Our ultimate goal is to partition this input network topology into a number of sub-topologies, with each sub-topology containing a number of nodes and links of the original topology, with a number of *split links* representing the links crossing between sub-topologies.

Current large network simulation models usually model the network topology by assuming a number of end-nodes and a number of routers, each end-node connected to only one router to represent the last hop link from the network to the end host, and the routers are interconnected to represent the backbone network. If the simulation is approximating a realistic Internet scenario, the links closer to the core (i.e., farther away from the end hosts) would tend to carry a larger volume of aggregate traffic. Therefore, if we only consider the cross-border traffic factor in the partitioning process, the links to split would primarily be the last hop links. But adding in the lookahead and load-balancing factors would split the topology at links closer to the core, producing better overall performance. We resolve this trade-off through the benchmarking process.

**Workload Model:** The workload model represents the traffic flows being simulated. A rough indicator of the amount of processing required in a simulation is the number packet-hops [1] that the workload flows generate during the simulation [17]. This is because each packet-hop typically results in two events, one for a packet's transmission and one for its reception. To provide load balancing in the partitioning it is desirable to balance the packet-hops resulting from the workload within each partition.

It is usually difficult to precisely predict packet-hops before actually running the simulation, because some packets could get dropped at congested links, and it is hard to predict

[1] A *packet-hop* is one packet traversing a single hop.

when and where congestion would occur beforehand. However, if congestion is not severe or does not last long, one may be able to estimate the number of packet-hops processed in the simulation as a whole.

**Computer Cluster Configuration**: A common platform for running a PDES is a computer cluster that consists of a number of stand-alone SMP machines interconnected through a high-speed LAN; each SMP machine has a number of CPUs sharing a certain amount of memory. In this chapter we only consider homogeneous computer cluster, where every machine in the cluster has the same number of the same type of CPU, and the same amount of memory. Such clusters are commonly used for such simulations. Extension of this work to heterogeneous cluster is an area of future study.

Since the primary goal of the partitioning task is to make the parallel network simulation run as fast as possible on this platform, we need to take a more careful look at the characteristics of this platform. When each CPU runs a part of the simulation, the interaction [2] cost between two processes can vary greatly depending on whether the two CPUs running the two parts belong to the same machine or different machines. If the two CPUs belong to the same machine, then the interaction is done through fast shared memory communication; but if the CPUs are on different machines, then the interaction would have to go through the much slower physical network links. Therefore, the impact of interaction is very different between these two cases. We should take this discrepancy into account when devising the partitioning approach.

### 2.3.2 Determining Factor/Performance Relations by Benchmark Experiments

**Benchmark Selection → Benchmark Network:** Given a computer cluster configuration and a class of large network simulation models, ideally we should be able to abstract the essential features from the detailed models, and construct a set of much smaller benchmark simulation models. The relationships between the performance factors and the performance that we derive from these benchmark experiments should be applicable to the original simulation model with similar effectiveness.

[2] As described before, the interaction includes synchronization and packet transmission.

For example, a basic set of benchmark networks can consist of two symmetric groups of nodes and links, each group simulated by a partition, and the two partitions are connected through a split link. Then we could vary the link delay of the split link to measure the effect of lookahead, vary the cross-border traffic ratio to measure the effect of communication overhead, and vary the traffic on each side to measure the effect of load-balancing between two partitions on the overall performance. Section 5 will give a concrete example of this type of benchmarking.

**Benchmark Network Simulation → Factor/Performance Relations:** After running the set of benchmark experiments, we should have a set of measurements of overall simulation performance for different lookahead values, cross-border traffic ratio, and load-balance. We can then plot these measurements to determine the formulas that approximate these relations, and later use these formulas to estimate the effect of the factors on the overall performance of actual simulations.

Specifically, in this step our goal is to determine the following functions and relative weights, that can be combined together later to calculate the edge weights when partitioning actual simulation models.

- $F_{LA}(Delay)$: This function approximates how the variation of lookahead value (i.e., split link delay) affects simulation performance. To determine this function, we can run on the given computer cluster a set of benchmark partitioned simulations that keep other factors constant but vary the split link delay value, and measure the performance of these simulations. Then we can plot the performance against the lookahead value to extrapolate a specific formula of $F_{LA}(Delay)$.

- $F_{CT}(CBTraffic)$: This function approximates how the variation of the cross-border traffic amount over a split link would affect simulation performance. To determine this function, we can run on the given computer cluster a set of partitioned benchmark simulations that keep other factors constant but vary the cross-border traffic volume, and measure the performance of these simulations. Then we can plot the performance against cross-border traffic to extrapolate the formula of $F_{CT}(CBTraffic)$.

- $W_{LA}$ and $W_{CT}$: These are two relative weights representing the comparison of the effects of lookahead and cross-border traffic on the simulation performance. Selecting the correct weights would result in a good trade-off between these two factors when partitioning the actual simulation. To determine the two weights, we can run on the given computer cluster a set of benchmark partitioned simulations that vary both the lookahead and cross-border traffic volume, and measure their performance. Then we can plot the performance against cross-border traffic and link delay in a 3-D plot to deduce the relative weights between cross-border traffic and link delay.

### 2.3.3 Partitioning the Network Simulation Model

**Weighted Graph Construction → Weighted Graph:** The weighted graph consists of both weighted edges and weighted vertices. The network nodes and links can be naturally mapped to graph vertices and edges, so we only need to determine the weights of the vertices and the edges.

- *Edge Weight:* Since the weight of an edge should be determined by combining the lookahead factor and cross-border traffic factor, after the functions $F_{LA}(Delay)$ and $F_{CT}(CBTraffic)$ and the weights $W_{LA}$ and $W_{CT}$ have been determined as described above, the weight of an edge can be calculated from:

$$W_{total} = F_{LA} * W_{LA} + F_{CT} * W_{CT}$$

- *Vertex Weight:* For the load balancing objective described in Section 2.2.1, vertex weights can be converted from the load associated with each node. A rough indicator of the load is the number of packets a node sends/receives in the simulation. One way to estimate beforehand the total number of packets each node sends/receives in the simulation is to traverse the path of every traffic stream from one end node to the other end node, and add the estimated number of packets in this traffic stream to a variable associated with each node on the path. However, this is just rough estimation, because the actual number of packets transferred in a stream can be different in the actual simulation.

**Graph Partitioning → Partitioned Simulation Model:** After constructing the weighted graph, we can use an existing graph partitioning tool to partition the graph into a number of parts, and create the partitioned simulation model to be run in parallel by different machines. The graph partitioning tool we use in this work is the Metis package developed by University of Minnesota [23]. Metis is a fast graph Partitioning tool. This tool takes a graph with vertex weights and edge weights as input, tries to partition the graph with balanced vertex weights and minimized edge-cut weights, and then outputs the part to which each vertex belongs.

To address the communication overhead difference between communicating across shared memory and communicating across a network link discussed before, we devise a simple *two-level partitioning* approach in which we use the graph partitioning tool twice. Suppose there are $n$ machines each with $m$ CPU's, we first partition the whole simulation at the cluster level into $n$ parts, with a large weight associated with each link; then partition each part further into $m$ sub-parts with a small weight assigned to each link. When starting the simulation, we start the $n * m$ sub-parts at the same time on the $n$ machines, with the rule that sub-parts of the same part must be started on the same machine, then the operating system would automatically assign each sub-part to a CPU of the machine.

The first level of partitioning tries to reduce interaction between the $n$ parts as much as possible, since the interaction across physical network links is comparatively much more expensive. The second level of partitioning on the other hand focuses much more on load balancing; because the interaction between CPU's through shared memory is relatively fast.

One point to note is that, in the second level partitioning, though the weight associated with link traffic should be reduced significantly, the weight associated with link delay should not be reduced. This is because when all the $n * m$ sub-parts are running, for each sub-part the simulation engine always picks the smallest split link delay between this sub part and the other $n * m - 1$ sub-parts as the lookahead. Therefore, partitioning at the second level should also result in split links with large enough delays to avoid reducing the overall lookahead in the whole simulation.

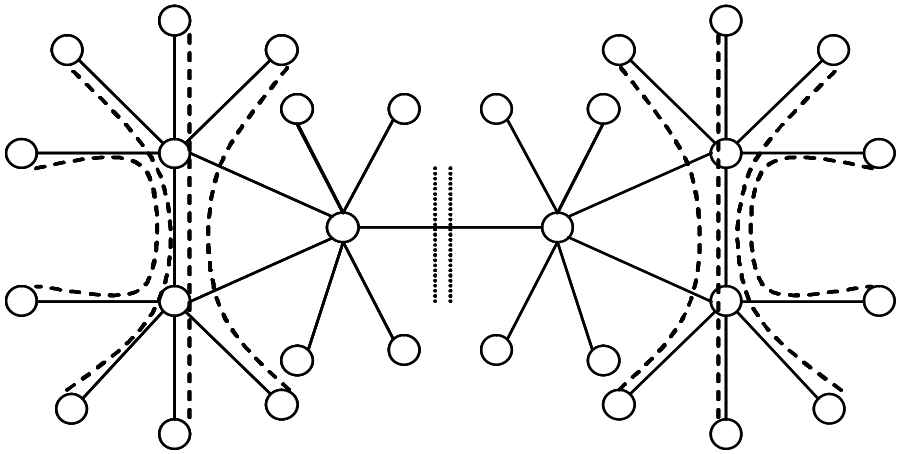

**Figure 2:** Benchmark simulation with zero cross-border traffic stream

## *2.4 A Benchmarking Case Study*

This section describes a set of benchmarking experiments in order to obtain appropriate graph weights. The computer cluster used for these benchmark simulations, the Ferrari cluster of School of Electronics and Computer Engineering, Georgia Institute of Technology, consists of eight machines, interconnected through a Gigabit LAN, with each machine having two 3GHz P4 CPUs and 2GB memory. The simulator we use is the Parallel and Distributed ns (PDNS) [40] developed within our group. PDNS is a parallel and distributed version of the ns2 simulator, relying on the Libsynk/RTI library [34] as the underlying synchronization and communication infrastructure between different simulator instances.

The following subsections will describe simple examples of the benchmark simulations, how they were designed, and how the measured performance was used to derive the factor/performance relations and weights.

To determine the effect of one performance factor in the partitioned simulation, we need a set of benchmark simulations that vary the value of this factor while keeping other factors unchanged. As shown in Figures 2 and 3, the simple benchmark simulation model is designed so that each packet traverses exactly three links (three hops), because the outer nodes send packets to vertical counterparts, and internal nodes send packets to horizontal counterparts. We split the link in the middle to partition this topology into two parts, each

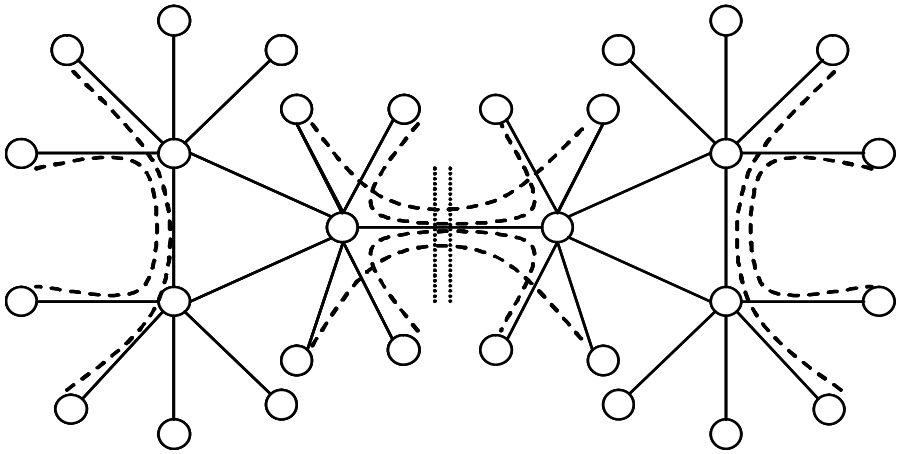

**Figure 3:** Benchmark simulation with four cross-border traffic streams

part being simulated by one PDNS instance.

In this set of benchmark simulations we maintain the total number of traffic streams at eight, and since each traffic stream traverses three hops, the total is 24 hops. The load between the two partitions remains evenly divided. We vary the ratio of (cross-border traffic packet-hops/total traffic Packet-hops) by increasing the number of horizontal traffic streams while decreasing vertical traffic streams by the same number, hence keeping the total packet-hops unchanged.

Figure 2 shows zero cross-border traffic streams, and since the total number of hops is 24 for all streams, the cross-border traffic ratio in this case is 0/24. Figure 3 shows four cross-border traffic streams, i.e., four cross-border traffic hops, and since the total traffic is still eight streams, i.e., 24 hops, the cross-border traffic ratio in this case is 4/24.

### 2.4.1 Communication Overhead: Varying Ratio of Cross-Border Traffic

Varying the cross-border traffic ratio from 0/24 to 8/24, we get the performance plot shown in Figure 4. The two lines represent respectively the running time of the split simulation running on two machines connected via LAN (SPLIT), and the split simulation running on two CPUs on the same machine communicating via shared memory (SPLIT-SHM).

We can see that as the cross-border traffic ratio increases, the running time of both SPLIT and SPLIT-SHM increase linearly. Starting from about the same point, SPLIT

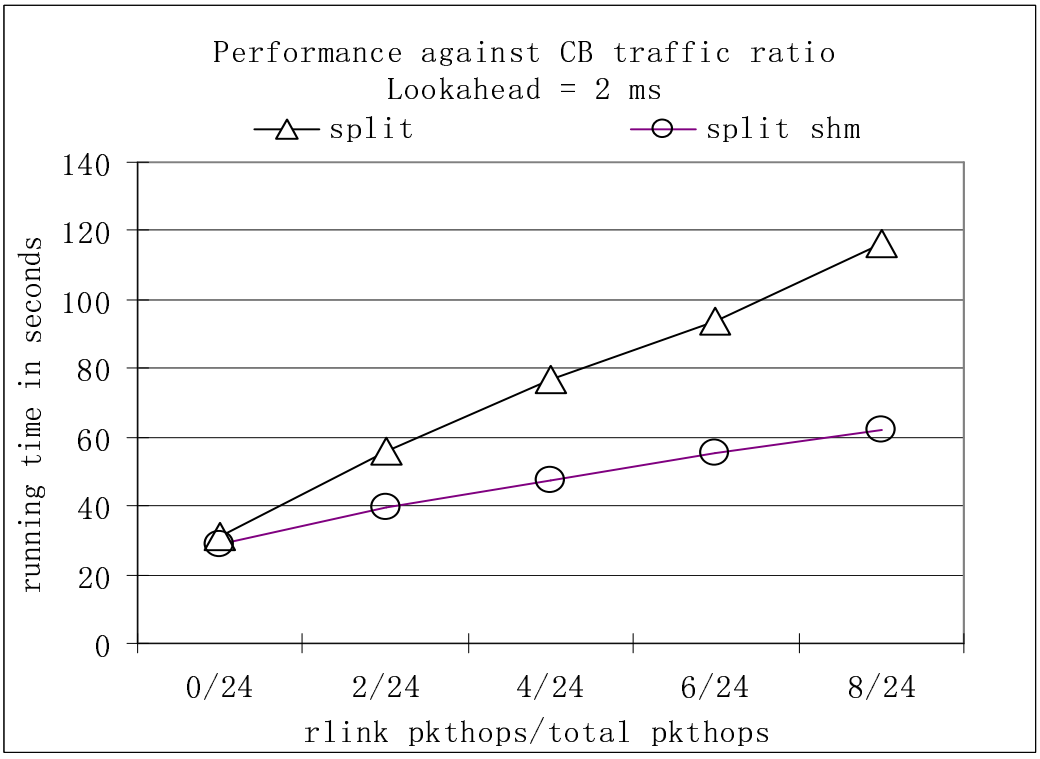


**Figure 4:** Varying the ratio of cross-border traffic for the split simulation on two machines, and the split simulation on a shared memory machine

increases faster than SPLIT-SHM, as expected, because the packet transmission over the LAN is much slower than over shared memory.

Therefore, we can obtain $F_{CT}$ of a link from the traffic amount on this link divided by the total traffic amount:

$$F_{CT}(CBTraffic) = \frac{CBTraffic}{TotalTraffic} * 24 + C$$

### 2.4.2 Lookahead: Varying Split Link Delay

To measure the effect of lookahead on the performance of split simulations, we use the same simulation scenario in Figure 2, i.e., no cross-border traffic between two partitions. Then we vary the propagation delay of the split link to create different lookahead values between the two partitions. The result can be seen in Figure 5. Notice that when lookahead is smaller than 20ms, both SPLIT and SPLIT-SHM running times drop logarithmically as lookahead increases. But when lookahead is greater than 20ms, the lines become flat, which means

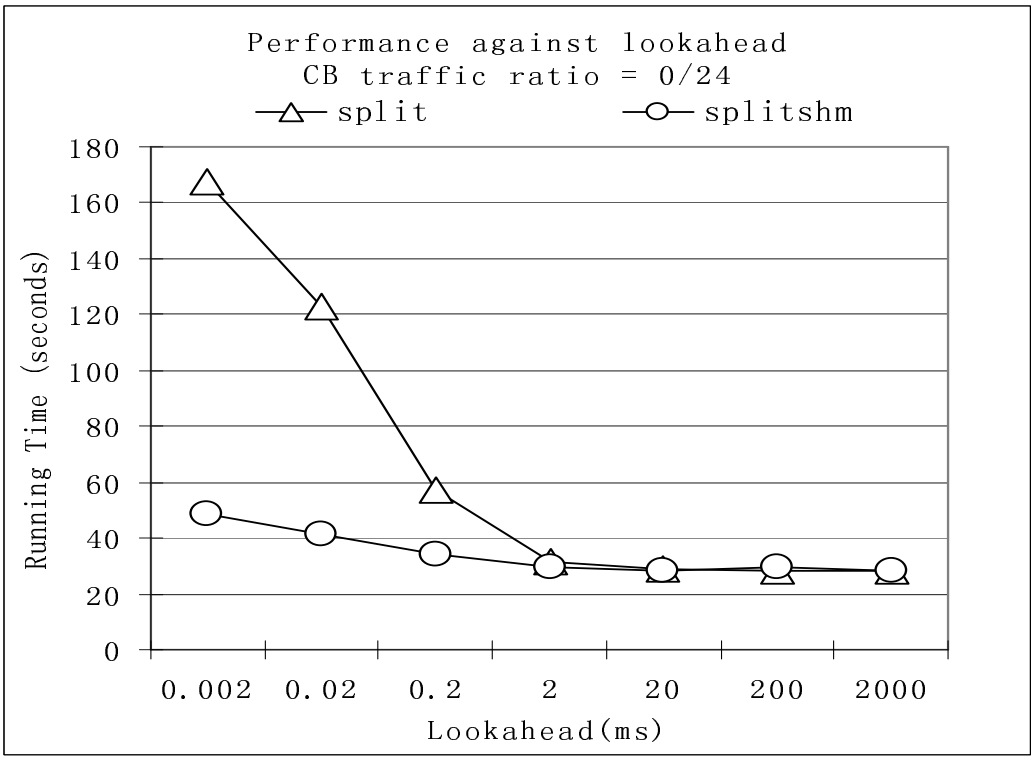


**Figure 5:** Varying the split link delay for the split simulation on two machines, and the split simulation on a shared memory machine

that lookahead larger than 20ms does not improve the performance. This is because, as we explained in section 2.2.2, lookahead values beyond the threshold often result in few synchronization operations.

Therefore, we can obtain $F_{LA}$ of a link by:

$$F_{LA}(Delay) = \begin{cases} \log(LA_{th}/Delay) + C, & Delay < LA_{th} \\ C, & Delay \geq LA_{th} \end{cases}$$

Where $Delay$ is the actual propagation delay of this link, $LA_{th}$ is the threshold value, and in this case $LA_{th} = 20ms$. Here the log calculation is to convert the logarithmic decrease to linear decrease for the convenience of linear weighted sum calculation of $W_{total}$.

### 2.4.3 Varying Both Lookahead and Cross-Border Traffic

Now that we have estimated how lookahead and cross-border traffic each affects performance in isolation, the remaining problem is to determine the relative weights between these two

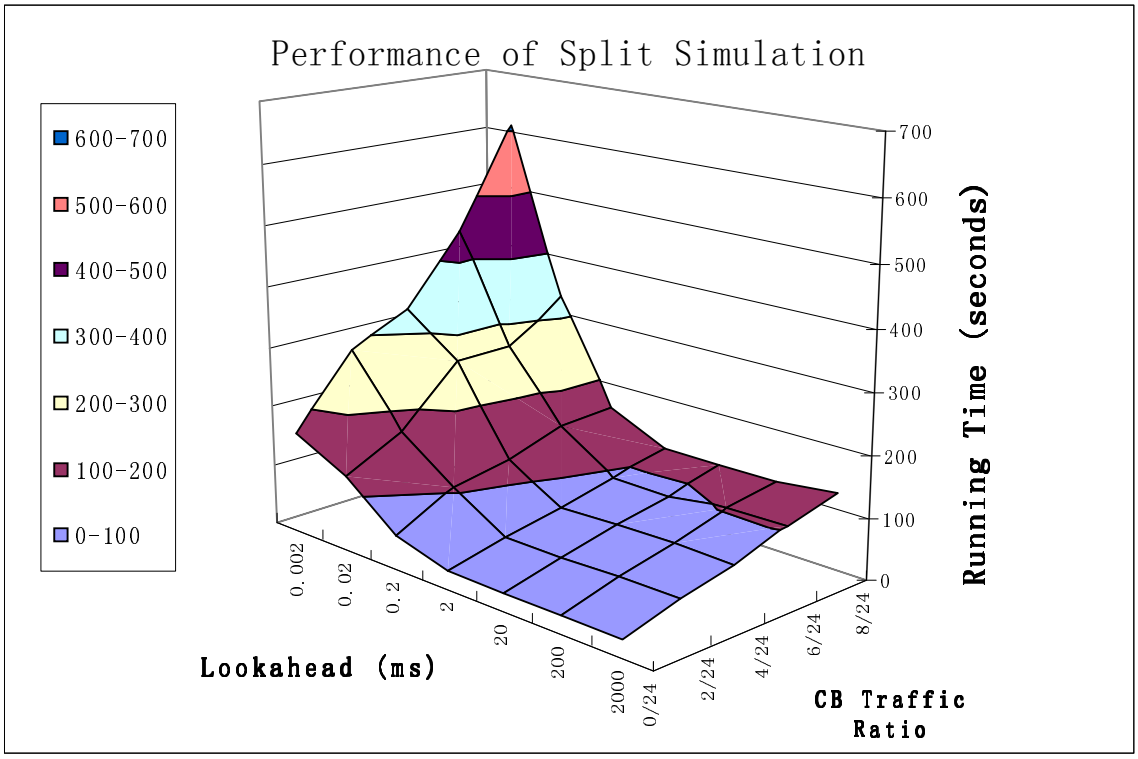


**Figure 6:** Varying both lookahead and cross-border border traffic ratio for split simulation on two machines

factors. Using the same topology in Figure 2, we vary both the lookahead and the cross-border traffic ratio for SPLIT, and obtained the performance measurements in Figure 6

From these measurements, we observe the trend that, on the plotted surface, moving two steps along the axis of cross-border traffic ratio is about equivalent to moving one step along the axis of lookahead. I.e., increasing cross-border traffic ratio by 4/24 produces about the same performance as decreasing the lookahead to 1/10. Therefore, the weights are roughly:

$$W_{CT} \approx 1, \;\; W_{LA} \approx 2$$

Figure 7 shows the more fine-grained performance measurement on the same Ferrari cluster, and we can see that the overall trend is consistent.

Figures 8, 9 and 10 show the coarse measurements on three different clusters with different configurations. Figure 8 is the measurement on the Photon cluster of School of Electronics and Computer Engineering, Georgia Tech, which consists of 16 machines connected

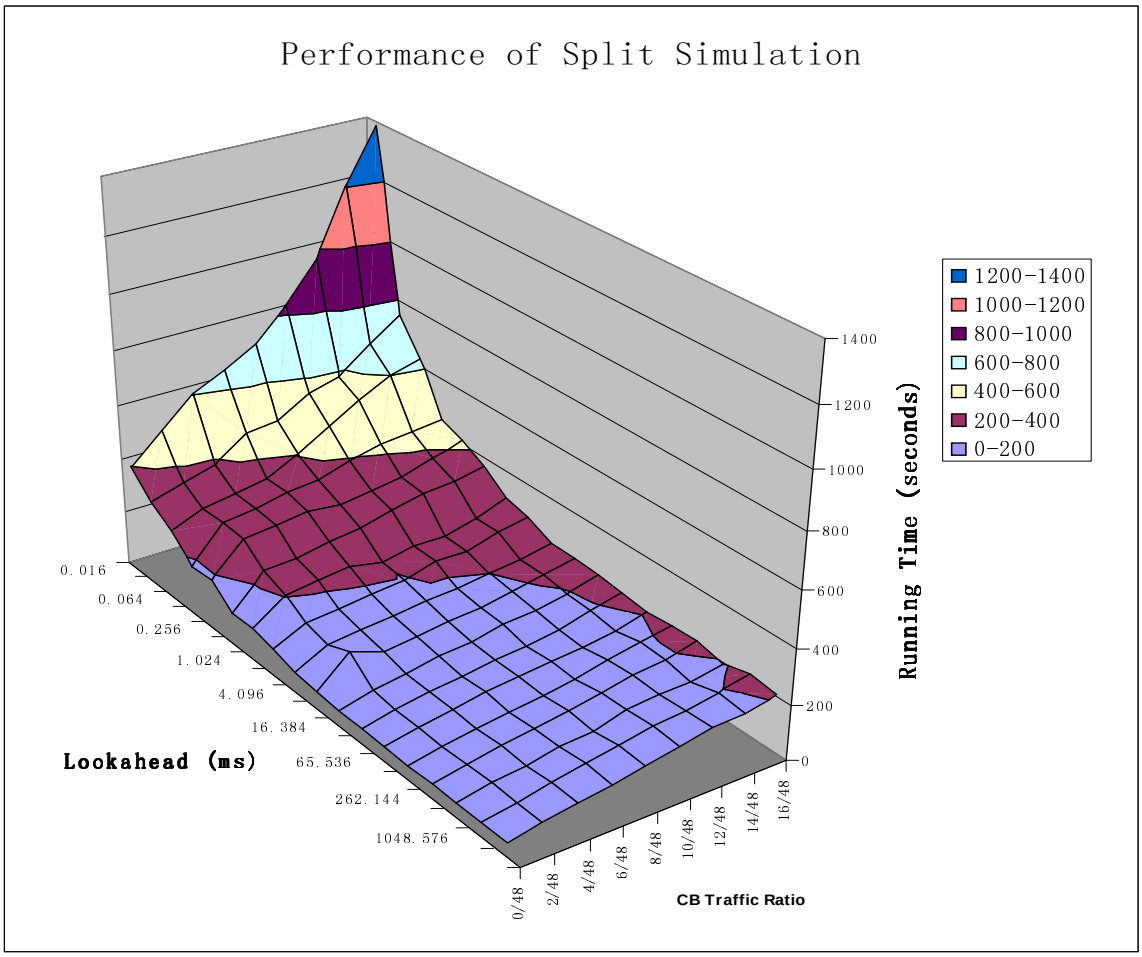


**Figure 7:** More fine-grained variation both lookahead and cross-border border traffic ratio for split simulation on two machines

via Gigabit LAN, each machine having two P4 2.8HZ CPU's and 2GB memory. Figure 9 is the measurement on the Jedi cluster of College of Computing, Georgia Tech, which consists of 16 machines connected via Gigabit LAN, each machine having eight P3 550MHZ CPU's and 4GB memory. Figure 10 is the measurement on two standalone machines Jawks and Preep of College of Computing, Georgia Tech, connected via normal LAN, each machine having two P4 4GHZ CPU's and 3GB memory.

We can see that even though the absolute running times are different, they all exhibit a trend similar to Figure 6. Which means we can derive similar factor/relation formulas for those clusters as well.

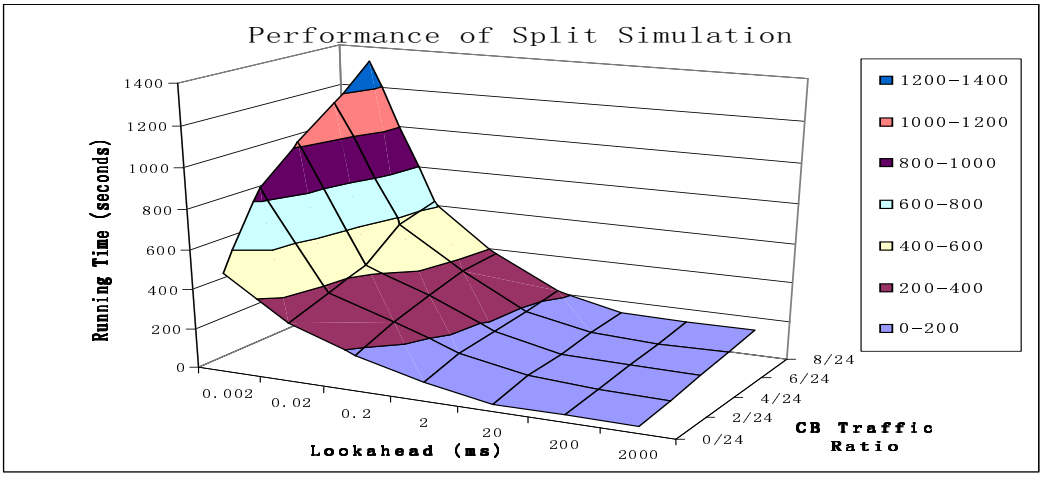


**Figure 8:** Photon Cluster of ECE: Dual CPU P4 2.8GHz, Giga-bit Ethernet

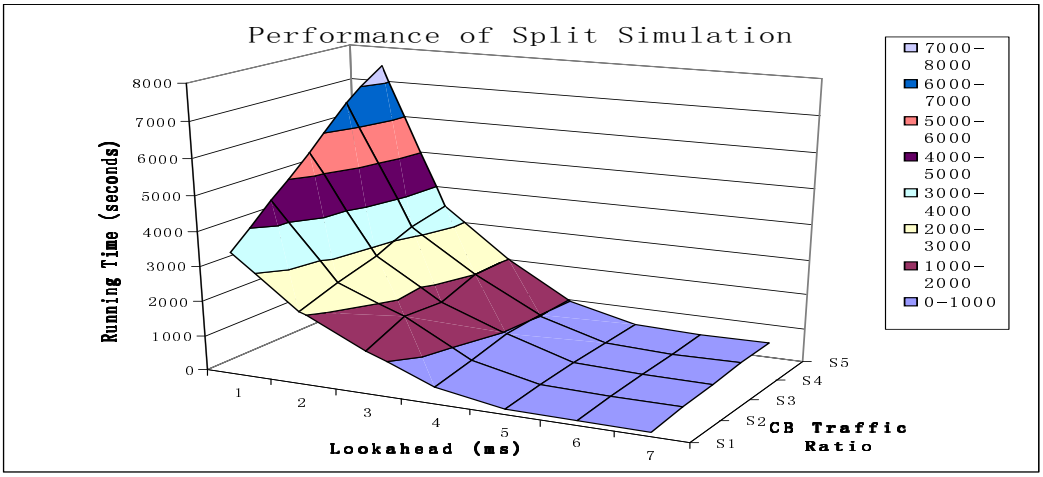


**Figure 9:** Jedi Cluster of CoC: 8-CPU P3 550MHz, Giga-bit Ethernet

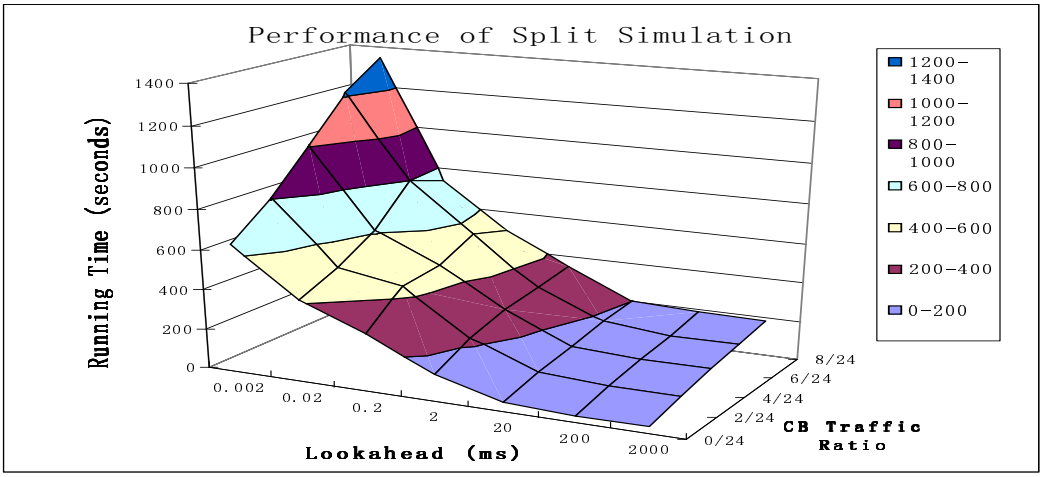


**Figure 10:** Jawks and Preep of CoC: Dual CPU P4 3GHz, usual Ethernet

## 2.5 Partitioning Experimental Results

In this section we use the factor/performance relationships that we obtained from the benchmark simulation above to compute a weighted graph derived from a large-scale network model. For the network topology we use MILNET, an Internet topology developed in [24]. We convert the MILNET topology from the original XML format to ns2 format, and create random traffic streams for the test. More specifically, the topology we test in this research consists of 123,536 nodes and 132,341 links. Among the nodes, 5,775 are routers and 117,761 are end hosts. We create 100,000 FTP streams, for each stream selecting two random end hosts to be the source and the sink. The simulation stops at the 40th second of simulation time. This scale of simulation is not achievable with ns2 on a single workstation. We use four machines (each with two CPUs) from the cluster described in the previous section. Our goal is to partition the simulation into eight parts, and run the eight parts in pdns in parallel.

As mentioned before, we partition the simulation in two steps. The first step is at the cluster level where we partition the whole simulation into four parts, each part to be run on a machine in the cluster, and the four parts communicate through the LAN. The second step is at the machine level where we further partition each of the four parts into two sub-parts, so that each sub-part can be run on a different CPU, and these two sub-parts communicate through shared memory. A partitioned simulation is then run with pdns on four machines in the cluster to measure the performance resulting from a partitioning strategy.

Here we compare the BencHMAP with several other partitioning strategies:

- *RNL*: We assign a random weight to each node and each link before using Metis to partition the simulation. With this strategy the result of partitioning is eight random partitions, each partition with a random size.

- *CNL*: We assign a constant node weight to each node, and a constant link weight to each link. With this strategy the partitioning results in eight partitions with about the same number of nodes, since Metis tries to balance the load between different partitions.

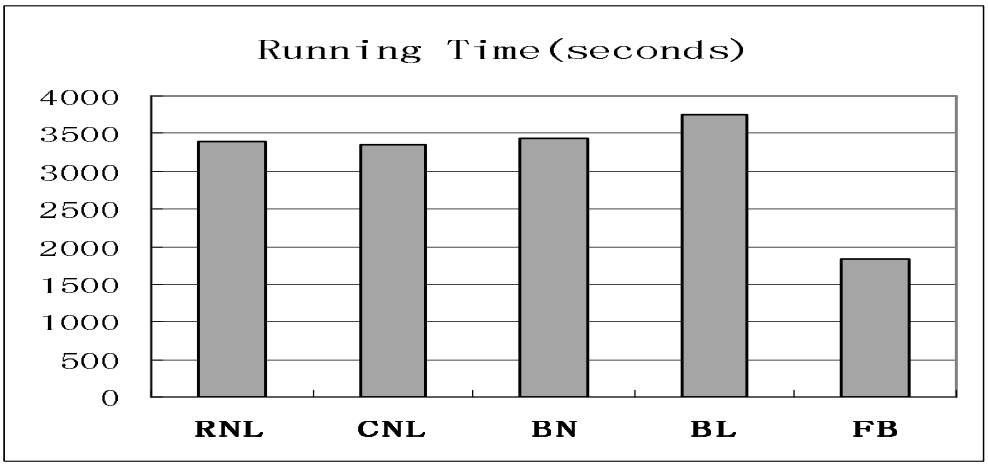


**Figure 11:** Simulations' performance

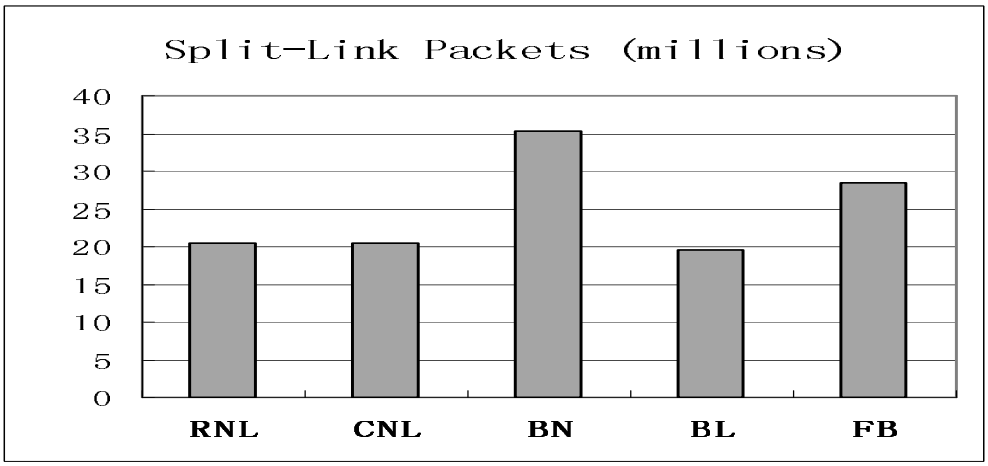


**Figure 12:** Total packets transmitted on split-links

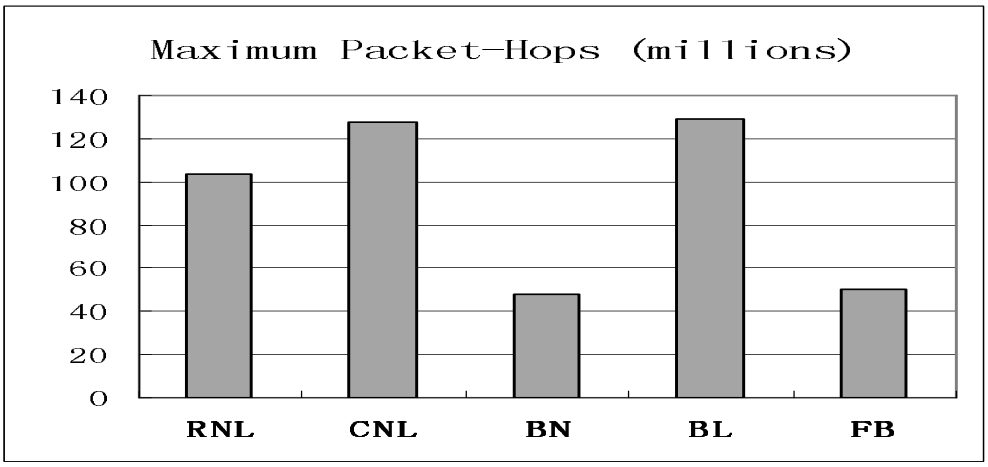


**Figure 13:** Maximum partition packet hops among eight partitions

- *BencHMAP-Link(BL)*: We assign a constant weight to each node, and calculate link weights with the traffic and link delay information using the benchmark results from the previous section. This strategy tests how link weight alone affects the partitioned simulation performance.

- *BencHMAP-Node(BN)*: We assign a constant weight to each link, and calculate node weight with the work load information. This strategy tests how node weight alone affects the partitioned simulation performance.

- *Full-BencHMAP(FB)*: This is the full application of our methodology in which both node weights and link weights are calculated from the estimated work load of each node as well as the benchmark results.

Figure 11 shows the performance of the partitioned simulation of these partitioning strategies. Figure 12 shows the total number of packets transmitted on the split links in the simulation in each case. Since the split-link packets transmitted over LAN constitute the main interaction cost in the parallel simulation, more split link packets would result in slower simulation. Figure 13 shows the maximum number of processed packet-hops among the eight partitions in each case. This is an indicator of the load-balance factor, and since the performance is to a large degree determined by the slowest partition, i.e., the partition that processes the most packet-hops, larger maximum packet-hops would result in slower simulation.

As we can see in the figures, *RNL* and *CNL* run more slowly primarily because the load is imbalanced: the maximum packet-hops is too high for these two cases. *BN* balances the load well and results in a low maximum packet-hops, but since it does not take into account link weights, too many split-link packets make the simulation run slowly. *BL* considers link weight, therefore reducing split-link packets, but ignoring load balancing factor results in a high maximum packet-hops, slowing down the whole simulation. This shows how important it is to obtain a good trade-off between the different factors for a good performance. *FB* takes all factors into account, producing a moderate number of split-link packets and better load-balancing, resulting in the fastest simulation.

## *2.6 Concluding Remarks*

In this chapter we propose BencHMAP: Benchmark-Based Hardware and Model Aware Partitioning, a methodology to partition large scale network simulations to best exploit the hardware resources and create the best simulation performance. We also discuss the factors that affect the performance of parallel and distributed network simulation, and how we can obtain a good trade-off between the factors.

We give examples of how to use simple and fast benchmark simulations to obtain the formulas that approximate the relations between the factors and the performance of parallel simulation. These formulas are then applied to the partitioning of a large scale network simulation with 120k nodes 100k traffic streams. Comparing various partitioning techniques, we show that the full application of BencHMAP results in the best trade-off between various factors, therefore producing the best performance for our parallel network simulation. This is a preliminary evidence that the BencHMAP methodology can improve parallel simulation performance in certain cases. However, further experiments and study are still required to understand various issues related this methodology, e.g., benchmark selection, effect of workload migration etc.

We have developed AutoPart, an automatic partitioning tool to partition ns2 scripts into a set of pdns scripts which can then be run on a cluster of machines. See Appendix A for more details.

# CHAPTER III

# ENABLING LARGE-SCALE MULTICAST SIMULATIONS BY REDUCING MEMORY REQUIREMENTS

The simulation of large–scale multicast networks often requires a significant amount of memory that can easily exceed the capacity of current computers, both because of the inherently large amount of state necessary to simulate message routing and because of design oversights in the multicast portion of existing simulators.

In this chapter we describe three approaches to substantially reduce the memory required by multicast simulations: 1) We introduce a novel technique called the "negative forwarding table" to compress multicast routing state. 2) We aggregate the routing state objects from one replicator per router per group per source to one replicator per router. 3) We employ the *NIx–Vector* technique to replace the original unicast IP routing table. We implemented these techniques in the ns2 simulator to demonstrate their effectiveness. Our experiments show that these techniques enable packet level multicast simulations on a scale that was previously unachievable on modern workstations using ns2.

This chapter is organized as follows. In Section 3.1 we give an overview on related work on reducing multicast memory requirements on actual routers. In Section 3.2 we briefly introduce multicast routing and discuss the memory requirements for storing routing state in a simulator. In Section 3.3 we discuss three memory saving techniques for multicast simulation, namely, the "Negative Forwarding Table", the replicator aggregation, and the *NIx–Vector* techniques. Section 3.4 presents experimental results demonstrating the memory savings of these three techniques. And in section 3.5 we give conclusions of this chapter.

## *3.1 Related Work*

There is a significant body of work dealing with the problem of reducing the memory required to store multicast[10] state in routers. This body of work is constrained by the

concern with developing techniques that can be deployed in real networks. In contrast, we are concerned with memory saving techniques to be used in simulations. Nevertheless, we survey this body of work below since it does provide some insight into the problem at hand.

An observation that can be made from multicast trees is that, on a sparse tree, the "branching points" are rare and the majority of the tree branches are long, unbranched paths. Storing multicast routing state on the intermediate routers between branching points is therefore inefficient. There have been some schemes[45, 21] that attempted to store the multicast state only at the branching points. This can drastically reduce the memory required for routing state, as most multicast trees in the Internet are expected to be sparse trees.

Tang and Neufeld [45] proposed to establish an IP tunnel between two branching points and remove the multicast routing state on the intermediate routers. This can significantly reduce the memory required for routing state, as most multicast trees are expected to be sparse trees. The cost is the overhead of establishing and tearing down the IP tunnels. Stoica et al. [21] introduced the REUNITE protocol to replace the original multicast protocols. One of the features of REUNITE is that multicast routing state is inherently only kept on the branching points (in a fashion similar to the IP tunneling approach above), thus significantly reducing the multicast routing state for sparse trees. However, this is a new set of protocols that requires further study.

On the other hand, some other researchers also worked on aggregating multicast routing state without the sparse tree assumption. Briscoe and Tatham [5] proposed a completely new multicast address naming scheme that explicitly provides methods for routers to aggregate multicast routing state. However, this method would require all multicast protocols to be upgraded to accommodate this new naming scheme. Radoslavovet et al. [30] introduced a leaky aggregation method to aggregate multicast routing state that are similar but not exactly the same, sacrificing some bandwidth (for excessive packet forwarding) to save router memory. The main disadvantage of this method is that sacrificing bandwidth to save memory does not seem very appealing.

Thaler and Handley [46] suggested that multicast routing state can be aggregated even

without changing the addressing scheme or sacrificing bandwidth: we can always transform the original multicast routing table into "interface-centric" routing tables, where each interface has a forwarding table that maintains the set of multicast addresses whose packets must be forwarded onto this interface. This method then tries to aggregate every region of continuous addresses into one single entry. It was shown that as long as there is a sufficiently large number of multicast addresses on an interface, the effect of this aggregation would be very significant. However, the effectiveness of this method completely relies on a large percentage of multicast addresses maintained on the interface in the entire multicast address space. On a large network it is difficult to imagine any router covering a significant percentage of the entire multicast address space.

In addition, Huang et al.[20] proposed to abstract away certain details of the network simulation to reduce memory consumption, however this approach sacrifices the fidelity of the simulation. Riley et al.[41] proposed NIx-Vectors to reduce unicast routing memory usage in network simulations, nevertheless, this work did not take multicast simulation into consideration.

## *3.2 Multicast Routing Memory Requirements*

### 3.2.1 Background on Multicast Routing

Multicast data delivery services usually operate at the network layer. A multicast group uses an *IP* address that represents a set of hosts on the Internet. The sending application on a host perceives that it is only sending data to this single address, while the underlying network services are responsible for actually delivering the data to multiple target hosts represented by this group address.

In order to achieve multicast data delivery, the multicast routers need to first construct multicast forwarding trees, then forward the multicast data packets along these trees to all receivers. The routers construct the trees based on existing *unicast* routing tables. More specifically, a router usually performs an "RPF" (Reverse Path Forwarding) check in the *unicast* routing table when the router needs to determine from which interface a source's multicast packet will arrive.

There are in general two scenarios where an "RPF" check is necessary: 1) when a router needs to determine to which neighbor it should send join/grafting or leave/pruning requests; 2) in dense mode multicast protocols such as "DVMRP", the first broadcast packets can arrive at a network node from any neighbor, so this network node needs to determine which packet arrives via the shortest path from the source, and discard other packets.

If there is only one source (i.e., sending host) in a multicast group, then the multicast routing tree is a shortest path tree from the source to all receivers of this group. If there are multiple sources in a group, the Source-Based Tree (SBT) approach[10, 36] constructs a different tree for every different source, so that packets from each source are delivered along this source's tree. Compared to the Core-Based Tree (CBT) approach[3] that designates a core and constructs a tree from this core to all receivers, SBT has the advantage of packet delivery efficiency, but suffers when the size of a group scales up. This is because in SBT, each router would maintain a separate multicast routing state entry for each tree that passes through this router. Here, we address this scalability problem, hence we will assume the SBT approach in multicast routing for the remainder of this chapter.

There are two common ways to construct the multicast routing trees. The first, called Source-Based Trees (SBT), is to construct a different tree for every source, so that packets from each source are delivered along this source's tree [10, 36]. The second, called Core-Based Trees (CBT), is to designate a single router to be the core (or Rendezvous Point) of the group, and construct a tree from the core to all the receivers. With this method, all sources must first send the multicast data to the core, then the core delivers the data to all the receivers along this tree [3]. The PIM architecture [9] tries to combine the two approaches by starting the tree as CBT and changing parts of the tree to SBT when the multicast traffic could be delivered more efficiently along the SBT.

The CBT approach has the inherent problems of 1) traffic concentration around the core and 2) the core becoming a single point of failure for a group. The SBT approach does not have these problems, but it does not scale as the number of senders in a group increases, since each router would maintain a separate multicast routing state entry for each tree that passes through this router. The total amount of state maintained by a router would depend

on the number of groups and sources. We address precisely this scalability problem, hence we will assume the SBT approach in multicast routing for the remainder of this chapter.

A multicast router maintains the multicast routing tables that contain the routing entries such as:

$\langle$ *grp*, *src*, *iif*, *oifset* $\rangle$

Where *grp* and *src* are the group address and sender address, respectively; *iif* is the input interface of this $\langle grp, src \rangle$ pair; and *oifset* is the set of output interfaces of this $\langle grp, src \rangle$ pair. When a multicast packet of $\langle grp, src \rangle$ arrives at a router, the router looks up the $\langle grp, src \rangle$ in its multicast routing table: if there is no entry for this $\langle grp, src \rangle$ pair, or if there is such an entry but the input interface is wrong, then this is a stray packet and should be discarded by the router. Otherwise, the router forwards this packet onto the set of output interfaces indicated by *oifset*.

The above multicast routing table entry indicates that, in a multicast network simulation, the memory required by multicast routing states is:

$$N * G * S * (R + I * Q)$$

Where $N$ is the number of nodes in the simulation; $G$ is the average number of groups whose trees pass through each node; $S$ is the average number of senders in each group whose trees pass through each node; $I$ is the average number of output interfaces of a $\langle grp, src \rangle$pair; $R$ is the overhead memory required per group per source on the router regardless of the number of output interfaces; and $Q$ is the amount of memory required to represent each output interface.

For example, suppose we are simulating 2000 nodes, with each node carrying an average of 500 groups' trees, each group on a node having an average of 100 sources, each $\langle grp, src \rangle$ pair having on average four output interfaces on each router, the memory overhead per group per source on each node is 40 bytes, and the memory requirements of each output interface is four bytes, then the total memory requirements of multicast routing states is:

2000*500*100*(40 + 4*4) bytes = 5.6G bytes

The multicast routing state alone has exceeded the capacity of many contemporary computers, preventing multicast simulations of this modest scale to be completed.

## *3.3 Techniques to Reduce Memory Consumption in Multicast Simulation*

In this section, we present three techniques to reduce the memory required by the multicast routing state, multicast-related objects, and unicast routing state. These constitute the principle state information required by a multicast simulation.

### 3.3.1 Negative Forwarding Table

First, we propose a novel "Negative Forwarding Table" approach to compress the multicast routing state. This approach is based on an interesting observation regarding multicast trees: trees of the different sources of the same group often largely overlap each other. This is due to the fact that different sources all need to deliver data to the same set of group members. Therefore, for overlapping trees, it is preferable to maintain the difference between routing trees of the same group, so long as representing the difference requires less memory than simply replicating the trees.

To do this, we also take the view point of interface-centric routing: each interface has a forwarding table that maintains the $\langle grp, src \rangle$ pairs whose packets need to be delivered onto this interface. The table consists of entries of the following form:

$$\langle grp1 : src11, src12, ... \rangle$$

$$\langle grp2 : src21, src22, ... \rangle$$

Where each entry represents a group and the sources of this group. For example, if the router receives a multicast packet $\langle grp1, src11 \rangle$, then this packet needs to be delivered onto this interface, since $\langle grp1, src11 \rangle$ is found in this forwarding table of this interface.

Now call this table the Positive Forwarding Table (PFT), and introduce a Negative Forwarding Table (NFT) for the same interface. The NFT entries have the same format as the PFT, but has the inverse meaning: the multicast packet of the $\langle grp, src \rangle$ pair in the NFT should not be forwarded onto this interface. For example, if an entry in the NFT for an interface is:

$$\langle grp1 : src11 \rangle$$

Then the multicast packet of $\langle grp1, src11 \rangle$ should *not* be forwarded on this interface,

since it can be found in the NFT of this interface. On the other hand, a multicast packet $\langle grp1, src12\rangle$ should be forwarded onto this interface, since it is not in the NFT of this interface.

On each interface, a group is either in the PFT, the NFT, or neither, but never both. The router maintains an overall *group_source* table that records all the $\langle grp, src\rangle$ pairs that have to be maintained by this router. (This *group_source* table also takes the same format as the PFT and NFT.) Group entries can move between PFT and NFT when join/leave operations happen on an interface. When an interface finds that an entry of *grp* in its PFT has more than half of the sources of *grp* in the overall *group_source* table, then this entry is moved to the NFT with its complemented content. More specifically, a new entry for *grp* is created in the NFT that only contains the sources that are in the overall *group_source* table but not in the PFT for *grp*, and the old entry for *grp* in PFT is deleted. Similarly an entry in the NFT can also be moved back to the PFT when it has more than half of the sources of the same group in the *group_source* table.

An example illustrates how the PFT and NFT work. Assume that in the overall *group_source* table of a router, there are four sources for *grp1*:

$$\langle grp1 : src11, src12, src13, src14\rangle$$

and assume an interface only needs to deliver the packets of $\langle grp1, src13\rangle$. Then the PFT would have the entry $\langle grp1, src13\rangle$, and the NFT would not have an entry for *grp1*. On the other hand, assume the interface needs to deliver the packets of $\langle grp1, src11\rangle$, $\langle grp1, src13\rangle$ and $\langle grp1, src14\rangle$ of *grp1*, then the NFT would have the entry $\langle grp1, src12\rangle$ and PFT would not have an entry for *grp1*.

In other words, each entry in the NFT is in effect an "exception list": it tells the interface to deliver packets from all sources of this group, except for the sources in this list.

There are two ideal cases where the NFT approach works: for a group *grp*, the interface needs to deliver packets that come from either none or all of the sources of *grp*:

1. If the interface needs to deliver none, then there is no entry for *grp* in either PFT or NFT.

2. If the interface needs to deliver packets from all sources of *grp*, then there is an empty entry in the NFT:

   $\langle grp : \rangle$

   Which indicates that there is no "exception", i.e., packets from all sources of *grp* must be forwarded onto this interface. Notice that even though the list of output interfaces is empty, we cannot remove this entry from the NFT, because removing it from NFT would mean it is in neither NFT nor PFT, which represents case 1 above.

The worst case is when an entry of *grp* in PFT or NFT contains half of the sources of *grp* in the *group_source* table. In this case there is basically no saving over the original PFT-only approach.

The effectiveness of the NFT technique (i.e., how much saving the NFT approach could achieve over the PFT-only approach) can be measured by the ratio $M_{PFTonly}/M_{NFT}$ where $M_{PFTonly}$ is the memory required by the routing state of the PFT-only approach (PFT's), and $M_{NFT}$ is the memory required by the routing state of the NFT approach (overall *group_source* table + PFT's + NFT's). This ratio is largely determined by the ratio $C_{PFTonly}/C_{NFT}$, where and $C_{PFTonly}$ is the number of times the $\langle grp, src \rangle$pairs have to be stored (in the PFT's) for the PFT-only approach, and $C_{PFTonly}$ (in *group_source* table or PFT's or NFT's) for the NFT approach. The second ratio depends on three factors: the number of sources of a group, the receiver *denseness* of a group, and the *connectivity* of the network.

The first factor is the number of sources of a group whose trees pass through a router. If the router only carries one source of a group, then the NFT approach actually may require more memory than the PFT-only approach for this group on this router, because with the PFT-only approach, this $\langle grp, src \rangle$ pair is only stored in the PFT's of all its output interfaces, while with the NFT approach this $\langle grp, src \rangle$ pair has to also be stored in the overall *group_source* table.

However, when there are more than one source of a group on this router, with the PFT-only approach, since each *src* has an independent set of output interfaces, each $\langle grp, src \rangle$

pair has to be stored in all the PFT's of all output interfaces to which this *src* corresponds. Assume the number of output interfaces is $F$. Then the PFT-only approach would need to store this $\langle grp, src \rangle$ pair $F$ times. On the other hand, with the NFT approach, since the *oif* sets of different sources of the same group largely overlap with each other, it is likely that on most output interfaces this *src* would overlap the majority of the other sources of this group. Therefore, this *src* does not appear in the "exception lists" on most of its output interfaces, which means this $\langle grp, src \rangle$ pair does not have to be stored on all its output interfaces' PFT's or NFT's. The only place that every $\langle grp, src \rangle$ pair has to be stored is the overall *group_source* table. So the number of times this $\langle grp, src \rangle$ pair must be stored is likely less than $F$. Therefore, the more sources a group has on a router, the larger the value of $C_{PFTonly}/C_{NFT}$.

Since trees of different sources of the same group often overlap each other, a reasonable expectation is that when a router maintains multicast routing state for more than one source of one group, then the sets of output interfaces of different sources of this group are likely to be very similar. Thus for each interface, the number of sources whose packets need to be delivered onto this interface is likely either close to none or close to all of the sources of this group. As a result, we are more likely to have the most saving when there are more sources for a group on a router.

The second factor is the receiver *denseness* of the group. This denseness can be defined as the ratio of the number of receivers in a group divided by the total number of nodes in the network. It determines the number of output interfaces that a $\langle grp, src \rangle$ pair can have on a router. In a sparse group, most routers on a multicast tree are non-branching points, which means there is only one output interface for this $\langle grp, src \rangle$ pair on these routers. On a non-branching node, the NFT approach does not save memory over the PFT-only approach for this $\langle grp, src \rangle$ pair, since with the PFT-only approach this pair is only stored once in the PFT of the only output interface, while with the NFT approach this pair has also to be stored once in the overall *group_source* table.

However, at a branching point where a $\langle grp, src \rangle$ pair has $F$ output interfaces ($F > 1$), the PFT-only approach stores the $\langle grp, src \rangle$ pair $F$ times on the $F$ output interfaces' PFT's,

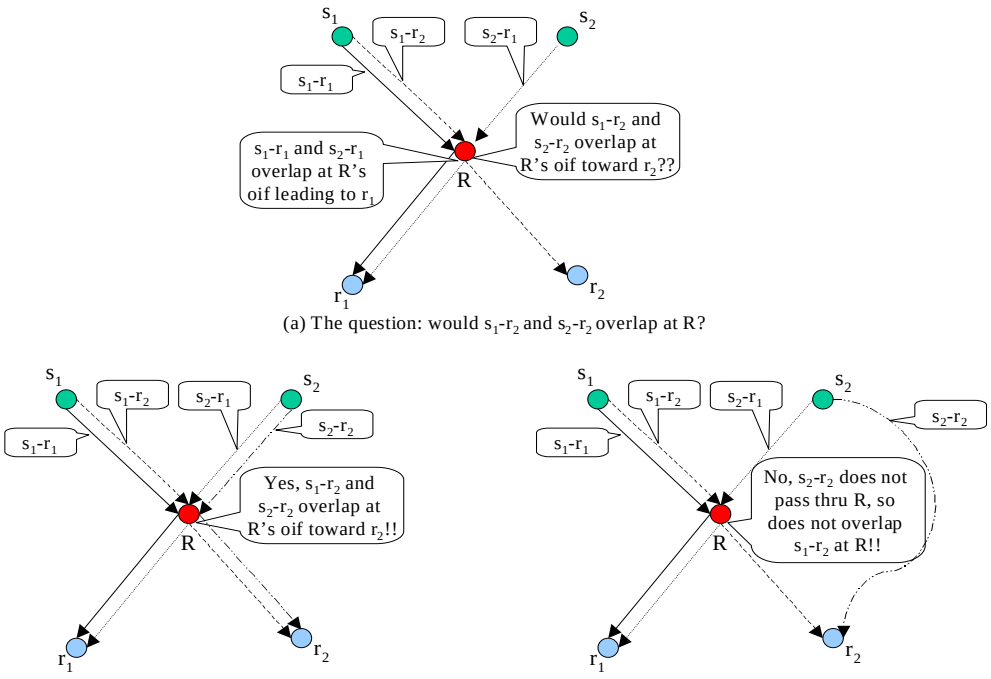


**Figure 14:** The impact of network connectivity

while the NFT approach can store the $\langle grp, src \rangle$ pair less than $F$ times, since on some of its output interfaces, it can overlap with the majority of other sources of the same group and be omitted in the NFT of those interfaces. Therefore, the denser a group, the larger the $C_{PFTonly}/C_{NFT}$ value.

The third factor is the connectivity of the network, i.e., the availability of alternative paths between any two nodes in the network. A simple example is shown in Figure 14 that includes receivers $r_1$ and $r_2$, two sources $s_1$ and $s_2$ of the same group, and an intermediate router $R$ in the network. Figure 14(a) shows that the path from $s_1$ to $r_1$ overlaps with the path from $s_2$ to $r_1$ at $R$, and that the path from $s_1$ to $r_2$ also passes through $R$. We are interested in whether the path from $s_2$ to $r_2$ would also pass through $R$, i.e., whether the path from $s_2$ to $r_2$ overlaps the path from $s_1$ to $r_2$ at $R$.

If the connectivity of the network is so low that there is only one path from $s_2$ to $r_2$, then this path must pass through $R$, because there is a path from $s_2$ to $R$ as well as a path from $R$ to $r_2$, and concatenating them forms the only path from $s_2$ to $r_2$. In this case the tree

of $s_1$ and the tree of $s_2$ completely overlap with each other at $R$, as shown in Figure 14(b), hence the NFT approach does not need to store the $\langle grp, src \rangle$ pairs in the NFT/PFT's on $R$, reducing the number of times the $\langle grp, src \rangle$ pairs must be stored.

However, if the connectivity of the network is higher, so that there is more than one path from $s_2$ to $r_2$, then it might happen that an alternative path from $s_2$ to $r_2$ is chosen as part of the multicast routing tree instead of the path from $s_2$ to $R$ to $r_2$, as shown in Figure 14(c), because the alternative path is shorter, or because of other reasons such as administrative choices. In this case, the NFT approach needs to store $\langle grp, s_1 \rangle$ in the PFT on the interface leading to $r_2$ on $R$, since the path from $s_2$ to $r_2$ does not overlap the path from $s_1$ to $r_2$ at $R$.

Therefore, the more alternative paths a network can supply, the less often different sources' trees would overlap with each other. The extreme case is a completely-connected graph where every pair of nodes has a direct link between them. In this case no trees ever overlap each other. In most cases the network connectivity is still low enough so that the NFT approach could achieve fair saving on memory requirements.

Multiple-source multicast is useful for conferencing applications and multi-user games. In the traditional multicast routing schemes that work on IP layer, the Internet is so large, and typical multicast trees become so sparse that most routers on the trees only carry a small number of all possible sources of the group, and most trees are sparse trees, (whose receiver density is low,) thus limiting the overall effectiveness of savings of this NFT approach. However, as the recent trend of Application Layer Multicast (or End System Multicast) suggests, it may be more reasonable to place the multicast responsibility of the conferencing groups on the end systems, not the Internet routers. The idea is to construct an overlay network that connects end systems through virtual links. Thus the end systems function as multicast routers as well, and this virtual topology is tight enough so that an NFT-like approach would likely result in significant savings in the multicast routing state.

### 3.3.2 Aggregating the Replicator Objects in ns2

The second technique we propose deals with multicast-related objects other than the multicast routing state. In ns2, multicast routing state is carried by the Tcl objects called *replicators*. When using Source Based Trees, for each $\langle grp, src \rangle$ pair in the simulation, ns2 constructs a shortest path tree from the source to all the receivers of this group. On each node through which the tree passes, ns2 creates a new replicator object for this tree. This replicator maintains the set of output interfaces for this $\langle grp, src \rangle$ pair on this node, and is responsible for duplicating and forwarding multicast packets onto the set of output interfaces upon receiving multicast packets of this $\langle grp, src \rangle$ pair.

It is reasonable to assume that a different state should be maintained for each $\langle grp, src \rangle$ pair on each node through which the tree of $\langle grp, src \rangle$ passes. However, in ns2 each replicator requires about 1.5K bytes of memory. Using $R = 1.5K$ in the expression we have seen in Section 3.2.1, the total multicast memory requirements is:

2000*500*100*(1.5K + 4*4) bytes = 150G bytes

Therefore, we want to reduce the impact of replicator object size by aggregating the replicators. More specifically, we feel it is more appropriate to create only one replicator object on one network node for all $\langle grp, src \rangle$ trees that pass through this node. Of course, the replicator object has to be modified to be able to maintain the routing state of more than one $\langle grp, src \rangle$ pair, and be able to choose the corresponding routing state to forward multicast packets of more than one $\langle grp, src \rangle$ pair.

After this modification, the total size of multicast routing state becomes: $N * (G * S * I * Q + R)$. Now that $R$ is no longer multiplied by $G$ and $S$, the total memory requirements is reduced substantially, and as a result the term $G * S * I * Q$ might become dominant in the second factor if there are sufficiently many groups and sources in the simulation. Still using the above example, the total multicast memory is now 1.6G bytes, compared to the previous 150G bytes

The key point here is to avoid unnecessary repetitions of large objects. In multicast simulations, though it seems unavoidable to maintain state for every $\langle grp, src \rangle$ pair, we still want to avoid associating the large objects such as replicators with each $\langle grp, src \rangle$ pair on

every node. The original implementation of the replicators in ns2 has the advantage that the design and program codes are somewhat easier to understand and maintain, since each replicator handles only one $\langle grp, src \rangle$ pair on a network node. However, as the simulations scale, it may be more desirable to sacrifice this advantage in order to reduce the excessive memory requirements caused by this design. This is the reason we propose to aggregate the replicator objects, from one replicator per sender per group per network node, to one replicator per network node.

Of course, it is also possible to further aggregate the replicators, from one replicator per node, to only one replicator for all nodes in the simulation, since there is little node-specific information maintained in the replicator. Thus the multicast routing state would be further reduced to: $N * G * S * I + R$, i.e., $R$ no longer multiplies $N$. Nevertheless, every network node in ns2 comes natively with some fundamental ns2 objects such as "node objects", "link objects" etc., hence the size of the state and related objects associated with each node is usually on the order of 30-40KB. And since each replicator itself only requires 1.5KB, aggregating all replicators to one replicator for all nodes would provide the memory saving of at most 1.5/30, i.e., 5%, which would not be a worthwhile effort in our estimation.

### 3.3.3 Running Multicast Simulations without Unicast Routing Tables

The third technique deals with the large unicast routing memory. As we have seen in Section 3.2.1, when constructing multicast routing trees, the routers need to perform "RPF" checks based on existing *unicast* routing tables. Each "RPF" check involves a search in the unicast routing table to determine which interface leads to the next hop to the source, based on the assumption that the paths in the network are symmetric, i.e., the reverse of the shortest path from a node to the source is the shortest path from the source to this node. Therefore, the unicast routing states must be maintained by the simulator to support the construction of the multicast routing states.

However, the complete unicast routing tables are one of the major factors that inhibit large scale network simulations, because of the quadratic relationship between the memory requirements and the number of network nodes in the simulation. For example, if there

are $n$ nodes in the simulation, each node would maintain $n-1$ entries in its routing table, each entry recording the output interface to one of the other $n-1$ nodes. Hence the total number of unicast routing entries is $n*(n-1)$, i.e., the total required memory is on the order of $O(n^2)$. When the number of nodes is large, the simulator would often run out of memory when trying to construct the complete unicast routing tables, before the simulation could even start. Therefore, to realize large scale multicast simulation, the first step would be to remove *unicast* routing tables. We use the *NIx–Vector* [37] technique in place of *unicast* routing tables to provide *unicast* routing support to the *multicast* tree construction. The *NIx–Vector* routing method is a form of source routing that allows the complete path between a pair of nodes to be stored in a compact representation. Complete routes between pairs of nodes are calculated only on–demand, when it is known that such a route is needed by the simulation. After the route is calculated, it is cached at the source node, again using the compact *NIx–Vector* format, and subsequently used as required. When a node needs to perform an RPF check toward a source for the first time, it computes the *NIx–Vector* from this node to this source, then uses this *NIx–Vector* to determine the interface leading to the source, and at the same time caches this *NIx–Vector* for future reuse. Using the *NIx–Vector* method instead of *unicast* routing tables allows larger topologies to be used in multicast simulations.

Notice that this approach saves memory because of the fact that in most simulations, only a small fraction of the $n^2$ possible routing states need to be maintained. On the other hand, some extreme cases do require a majority of the $n^2$ possible routing states. For example, when nearly every pair of nodes communicate with each other in a *unicast* simulation, or in a *multicast* simulation when every node is a receiver that needs to receive data from all the other nodes, it becomes necessary to calculate and store $n^2$ *NIx–Vectors*. In extreme cases such as this, skipping *unicast* routing table computation does not provide much benefit, so the simulationist should choose to use the default *unicast* routing tables instead of the *NIx–Vector* technique for routings in these cases.

However, in most cases, skipping the *unicast* routing table computation does save substantial memory and CPU time for large-scale topologies, allowing much larger multicast

simulations to be performed than would be possible otherwise.

## *3.4 Experimental Results*

We carried out a series of experiments to test the effectiveness of the memory saving techniques that we introduced in the previous sections. The platform where our experiments were run is a Pentium-III 866MHZ system running Red Hat Linux 7.1, with each system having 2GB memory. We modified ns2.1b7 according to the three different techniques described in the previous section, and ran various experiments to compare the memory usage with and without each technique.

Since we are only interested in the memory requirements of multicast routing states and related objects, not the protocol details between routers, we performed all experiments using CtrMcast mode of multicast in *ns*. In the simulation, CtrMcast directly computes the shortest path trees from the senders to all receivers of the same group without the join/leave protocol overhead.

Also, since we are interested in multiple-source multicast simulations, the experiments used source–based trees where a different tree has to be constructed for each sender instead of the core-based-trees where only one tree needs to be constructed for the Rendezvous Point.

We constructed the network topology in the simulation as follows: first we constructed a tree topology with a fanout parameter and a depth parameter, and then we added random links between non-leaf nodes. The leaf nodes represent the end systems, and the non-leaf nodes represent the Internet routers. Only the end systems (leaf nodes) can be senders or receivers of the multicast groups. The intermediate routers (non-leaf nodes) will carry the traffic that passes through them, but they never become senders or receivers of any multicast group. The purpose of creating a tree first is to ensure that the graph is connected. However, in a tree topology, there is only one path between any two leaf nodes, which would have been the ideal case for our NFT approach, because having no optional path implies that multicast trees stemming from different senders of the same group become completely overlapped with each other once they converge at any router. But in the real

Internet, there are almost always redundant paths, and the ideal case would seldom occur. That is the reason we added random links between non-leaf nodes, ensuring that there are alternative paths between non-leaf nodes. This makes topology construction very fast, easy to understand, and the scale of experiments easy to adjust.

The number of random links between non-leaf nodes is controlled by another parameter which we call $CrossLinkRate$. Suppose there are $L$ links in the original tree, then the number of random links to be added is $L * CrossLinkRate/100$. This parameter reflects the connectivity of the network. Large $CrossLinkRate$ values result in more alternative paths between nodes.

Each experiment first creates $G$ multicast groups using CtrMcast mode. Then for each group, $R$ leaf nodes are selected from a uniform distribution as the receivers, and $S$ leaf nodes are similarly selected as sources, where $G$, $S$, $R$ are all parameters used to control the scale of simulation. Each source continually multicasts UDP CBR traffic to all receivers of its group.

Figure 15 shows the memory usage of the six different combinations of the three techniques:

1. Original: the original *ns*.
2. RA: only using Replicator Aggregation technique.
3. RA, NFT: using RA and NFT techniques together.
4. Nix: using only Nix technique.
5. Nix, RA: using Nix and RA techniques together.
6. Nix, RA, NFT: using Nix, RA and NFT techniques together.

Each memory usage column is divided into four categories. From the bottom upwards, these categories are:

1. Memory used by unicast routing states(either the unicast routing table in the case without the *NIx–Vector* technique, or the *NIx–Vectors* in the case with the *NIx–Vector*

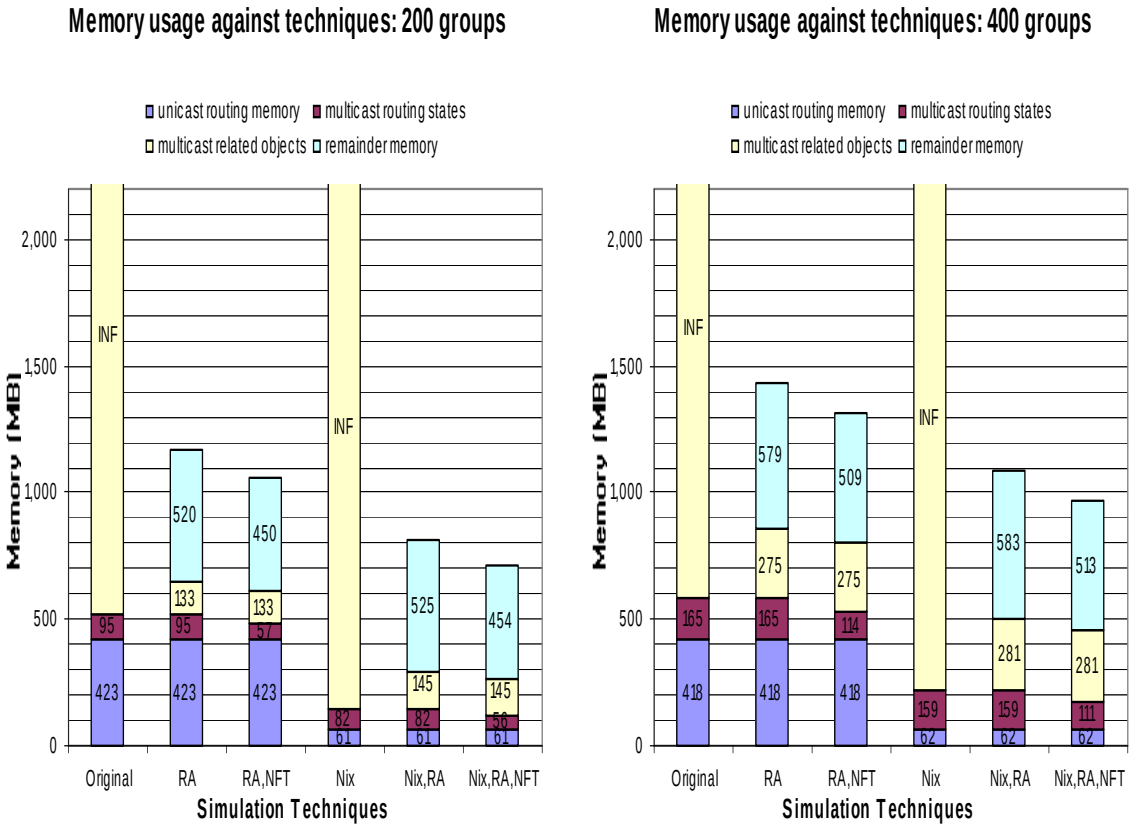


**Figure 15:** Simulation topology: fanout=6, depth=5, CrossLinkRate=100. Each multicast group has 100 senders and 200 receivers.

technique).

2. Memory used by multicast routing states(including the NFT, PFT, group_source table etc. in the case of interface-centric routing, or traditional multicast routing table otherwise).

3. Memory used by the other multicast routing related objects, such as replicator objects.

4. Memory used by the other parts of the simulation.

Since $fanout = 6, depth = 5$, there are in total 1555 nodes, and among them 1296 are leaf nodes. The left chart shows the memory usage when there are 200 multicast groups in the simulation, and the right chart shows the 400 group case.

Let us focus on the left chart first. In the figure, we can see that using *NIx–Vectors* significantly reduces the memory used by unicast routing states (shown by the lowest block of the columns). This network is not very large, and the unicast routing states only require 423MB memory(as in the first three columns). Using *NIx–Vectors* reduces the unicast routing states to about 61MB (as shown in the last three columns).

The chart also shows that the multicast routing states themselves do not require a significant amount of memory in these simulations, as indicated by the second lowest blocks of the columns. Without the NFT technique, the multicast routing states require 95MB (in the case without *NIx–Vectors* as shown in the first and second columns) or 82MB (in the case with *NIx–Vectors*, as shown in the fourth and fifth columns) of memory. With the NFT technique, the multicast routing states require about 57MB and 56MB memory, respectively, as shown in the third and sixth columns.

However, the third lowest blocks of the columns show that there are originally too many multicast related objects such as replicators in the simulation, because the numbers of groups(200) and senders(100) are both large. Without the Replicator Aggregation technique, the simulations exhaust the 2GB of available memory because the replicators and other multicast-related objects require an excessive amount of memory. (The columns labeled *Inf* indicate that the memory requirements for those cases exceeded the available

memory on our systems, and was therefore not measurable. However, it could be estimated that the replicators alone would have required 17GB memory if they are all created.) With the Replicator Aggregation technique, the memory required by the replicators and multicast-related objects is only about 133MB (without *NIx–Vectors*, second and third columns) or 145MB (with Nix, fifth and sixth columns) of memory.

The remainder of the memory required by the simulation is indicated by the uppermost block of the columns. It ranges between 420MB and 525MB. With all the techniques applied, as indicated by the last column, the unicast/multicast routing states and related objects are occupying an insignificant amount of memory compared to the remainder of the memory usage, which means the majority of the memory usage is spent on other parts of the simulations, exactly the goal we wanted to achieve in this research.

Comparing the left chart and the right chart, we can see that doubling the number of groups increases the memory usage, but not by much. Detailed examination between memory usage and group number will be presented in the next figure.

The above topology with 1555 nodes is about the limit that our platform would allow without the *NIx–Vector* technique. When we increase node numbers, the unicast routing table would exceed the 2G capacity at the scale of about 3000 nodes. Above this scale, the simulation can only be carried out with the *NIx–Vector* technique.

Figure 16 shows the memory usage as the number of groups increases, with all three techniques applied. The left chart shows the case of 1555 nodes ($fanout = 6$) where simulation without *NIx–Vector* technique can still be carried out, and it makes sense to distinguish the unicast routing memory and multicast routing memory. The right chart shows the case of 4681 nodes ($fanout = 8$) where simulation can only be carried out with the *NIx–Vector* technique. In both charts, the multicast memory roughly doubles as the number of groups doubles, suggesting that the multicast memory usage with all three techniques applied increase linearly with the number of groups.

Comparing the two charts, the total memory usages on the right chart roughly double the counterparts in the left chart, as the topology size of the right chart triples that of the left chart. This means memory usage increases more slowly than the increase of the

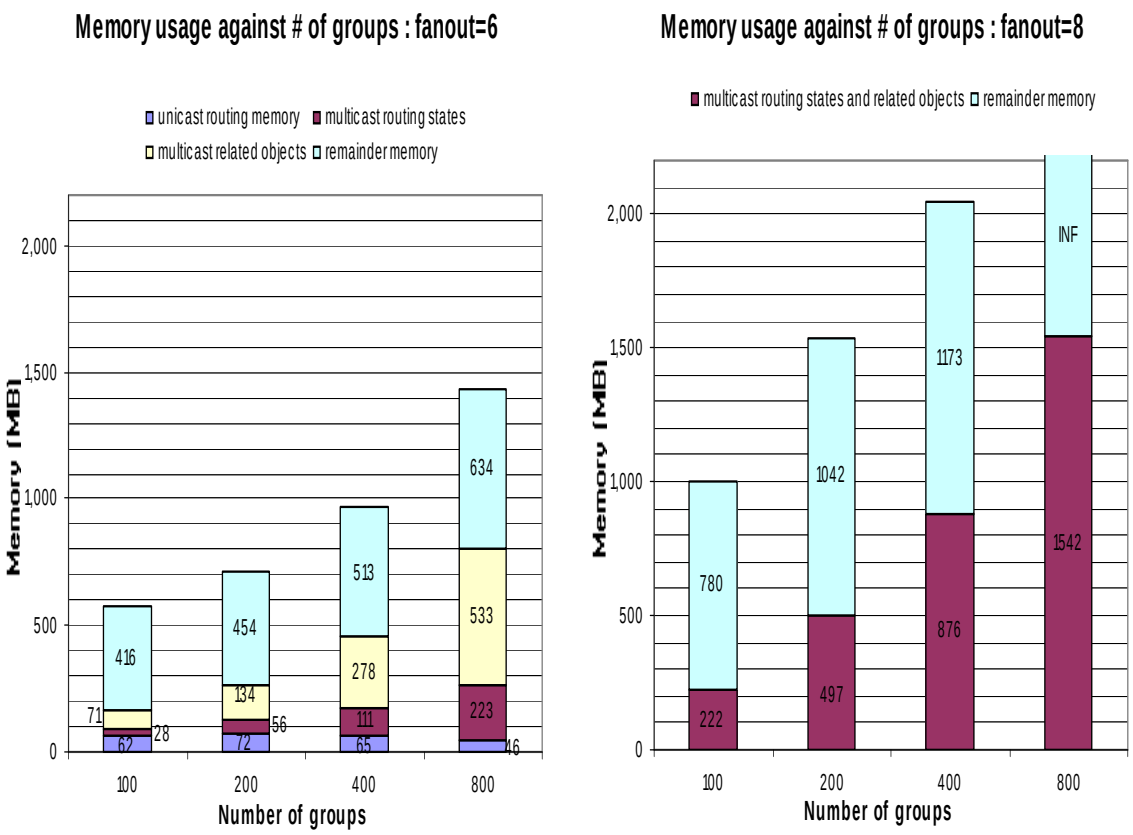


**Figure 16:** Simulation topology: depth=5, CrossLinkRate=100. left chart fanout=6, right chart fanout=8. All three techniques Nix, RA and NFT are applied. Each multicast group has 100 senders and 200 receivers.

topology size, one of our goals for this research.

Figure 17 shows how the network connectivity impacts the effectiveness of the NFT technique. As the *CrossLinkRate* increases, the multicast routing memory also increases. On the left chart that shows the case of 200 groups, with the provided number of senders(100) and density of receivers (200/1555), when the CrossLinkRate is 0%, the multicast routing states require about 33MB memory; when the CrossLinkRate is 100%, the multicast routing states require about 56MB memory. This confirms our expectation that more alternative paths in the network reduces the effectiveness of NFT.

The right chart doubles the number of groups in the left chart. As a result, in the right chart the multicast routing states and multicast related objects require about twice the

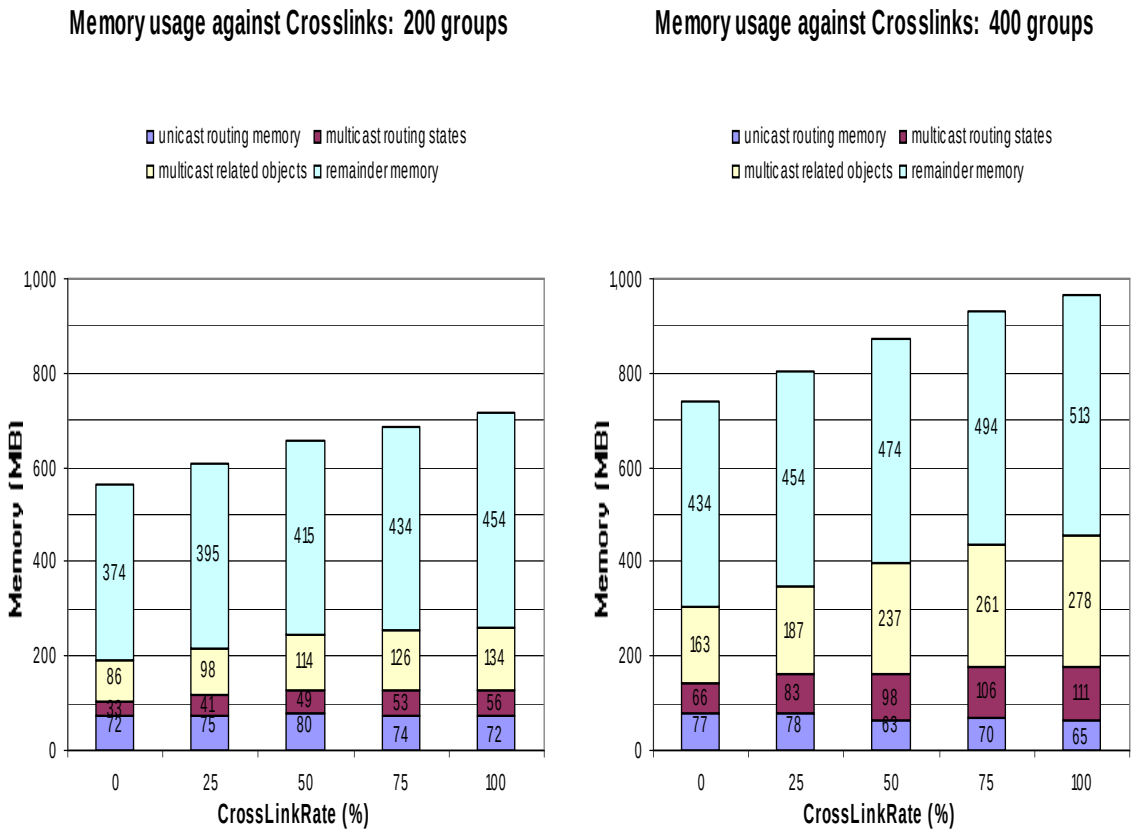


**Figure 17:** Simulation topology: fanout=6, depth=5. All three techniques Nix, RA and NFT are applied. The left chart has 200 groups, and right chart has 400 groups. Each multicast group has 100 senders and 200 receivers.

memory as their counterparts in the left chart.

Figure 18 shows how the density of the receivers impact the effectiveness of the NFT technique. The topology is relatively small, with 587 nodes in total, among them 512 leaf nodes. As the number of receivers increases, the multicast routing state of the PFT-only approach increase more or less linearly. On the other hand, although the multicast routing states of the NFT approach also increases, it increases much more slowly, and in the end, as the density(receivers/total nodes) tends to 1, the memory of the NFT approach tends to a constant number, which is ideal for End-System Multicast where the density is often exactly 1.

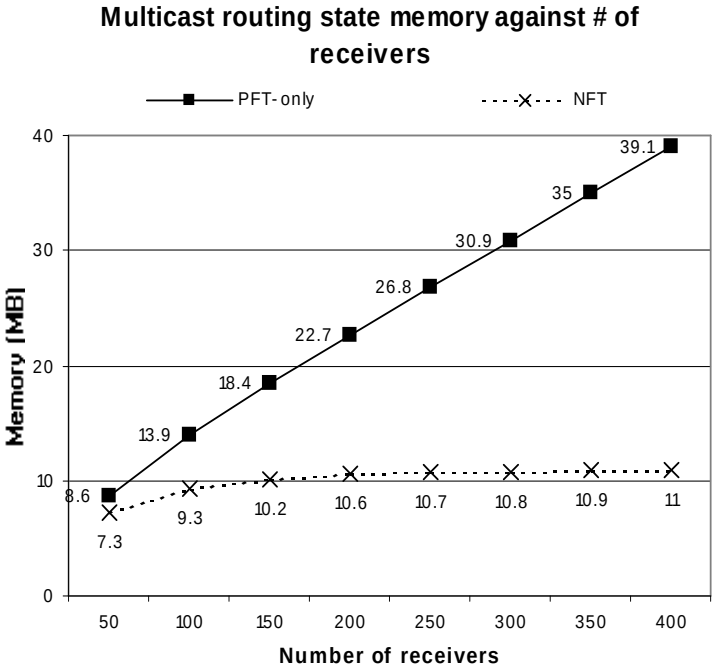


**Figure 18:** Simulation topology: fanout=8, depth=4, CrossLinkRate=100. Nix and RA are both applied. There are 100 multicast groups. Each multicast group has 100 senders.

## *3.5 Conclusions*

In this chapter we analyzed the memory requirements of multicast simulation, and presented three techniques to reduce memory requirements of a simulator in order to achieve large-scale multicast simulations.

1. We introduced a new data representation technique called Negative Forwarding Table based on the notion of interface-centric routing to compress the *multicast* routing states.

2. We aggregate the replicator objects, from one replicator per group per source per node, to one replicator per node, thus drastically reducing the memory requirements by the replicator objects.

3. We remove the *unicast* routing table, so that simulations of large networks do not run out of memory simply because of the $O(n^2)$ memory requirements of the *unicast*

routing tables.

Through experiments, we show that each of our techniques is effective in its own right. Combining all three techniques reduces the size of routing states from a prohibitively large amount to an insignificant amount compared to the total memory size required by the simulation, thus allowing large-scale multicast simulations to be performed without routing state alone exhausting available memory.

# CHAPTER IV

# SPLIT PROTOCOL STACK NETWORK SIMULATIONS USING THE DYNAMIC SIMULATION BACKPLANE

A number of researchers have explored methods and techniques for homogeneous parallel and distributed simulation within one single simulator. Riley et al.[40, 39] have designed and implemented the *Parallel/Distributed ns* (*pdns*) to provide scaling and improved performance for the ns2 simulator. Perumalla et al. [32, 31] created the *Telecommunications Description Language* (*TED*), which allows multi-threaded network simulations on an SMP processor. Nicol et al. [27] propose the *Infrastructure for Distributed Enterprise Simulations* (*IDES*), a Java based simulation engine designed specifically for parallel and distributed simulations (although not necessarily network simulations). Cowie et al.[7, 8] describe the *Scalable Simulation Framework* (*SSF*) as a method for parallel simulation of large scale networks. Bagrodia et al. implemented the GloMoSim [52] simulator previously mentioned. GloMoSim is built on top of the *PARSEC*[2] parallel simulation engine, and is designed to improve performance and scalability when run on a shared-memory symmetric multiprocessor. Additionally, Bagrodia[1] has implemented a version of GloMoSim that includes portions of the ns2 TCP protocol. This effort uses the glue approach with sections of source code copied from ns2, resulting in some difficult software maintenance issues.

In this chapter we introduce and discuss a methodology for heterogeneous simulations of computer networks using the *dynamic simulation backplane*. This methodology allows for the exchange of protocol information between simulators across layers of the protocol stack. For example, the simulationist may wish to construct a simulation using the rich set of TCP models found in the ns2 network simulator, and at the same time use the highly detailed wireless MAC models found in the GloMoSim simulator. The backplane provides an interface between heterogeneous simulators that allows these simulators to exchange meaningful information across layers of the protocol stack, without detailed knowledge of

the internal representation in the foreign simulator. With this method of heterogeneous simulation, new and experimental protocols can be validated and tested in conjunction with existing and accepted simulations of lower protocol layers.

We discuss the particular problems presented by the split protocol stack model, and present our solutions. We give results of our implementation of the split protocol backplane, using the ns2 simulator for the higher protocol stack layers, and the GloMoSim simulator for the lower layers.

## *4.1 Ways to Split Network Simulation*

First, we can partition the network topology, and have each simulator simulates a monolithic protocol stack of a portion of the network. In Figure 19, the graphic to the left depicts the overall network topology, and the right graphic depicts the protocol stack of each node. We only split the network topology, and the protocol stack is not partitioned This is useful when for example one partition is a wireless network and the other partition is a fixed network, and we want to use GloMoSim to simulate the wireless network and ns2 to simulate the fixed network.

A second way to split a simulation is to split it between protocol layers, with each simulator simulating a portion of the protocol stack of the entire network topology, as shown in Figure 20. Here we keep network topology intact, and only partition between protocol layers. This is useful when for example we want to utilize the TCP models in ns2 on the wireless MAC layer of GloMoSim, then we can split the simulation between the TCP layer and the IP layer, then run ns2 to simulate the upper portion of the protocol stack, and run GloMoSim to simulate the lower portion.

A third approach is a combination of the previous two approaches, as shown in Figure 21 We can partition the network topology and at the same time split the protocol stack in the simulation. Each simulator will simulate a portion of the protocol stack of some portion of the network topology.

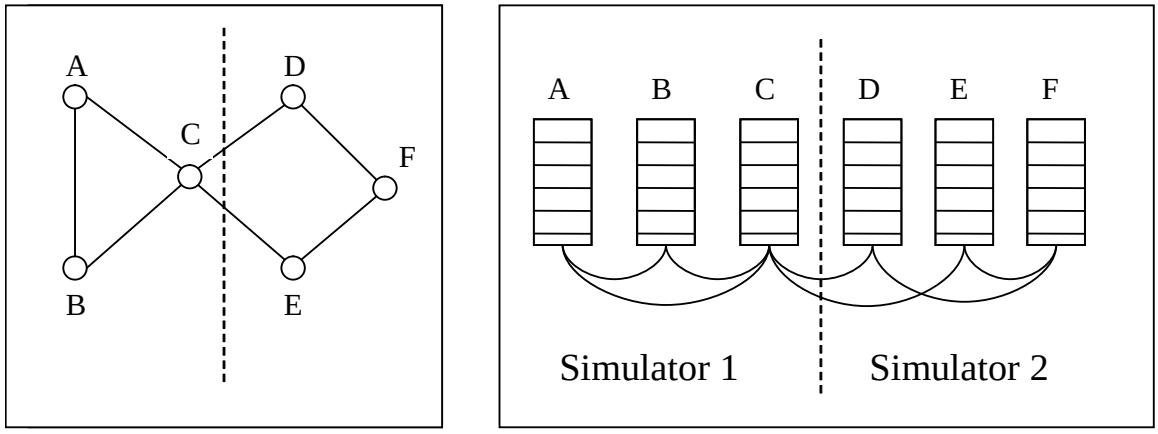


**Figure 19:** Simulation Split Horizontally

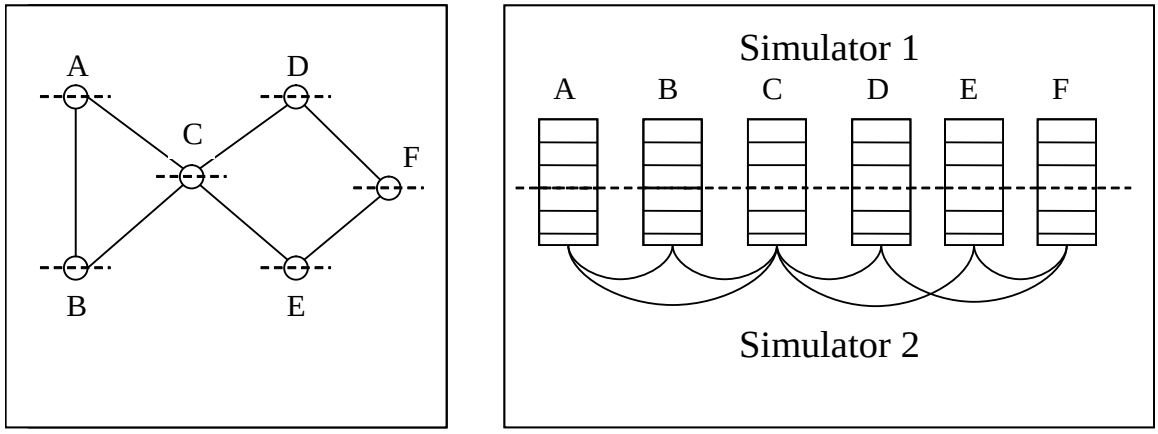


**Figure 20:** Simulation Split Vertically

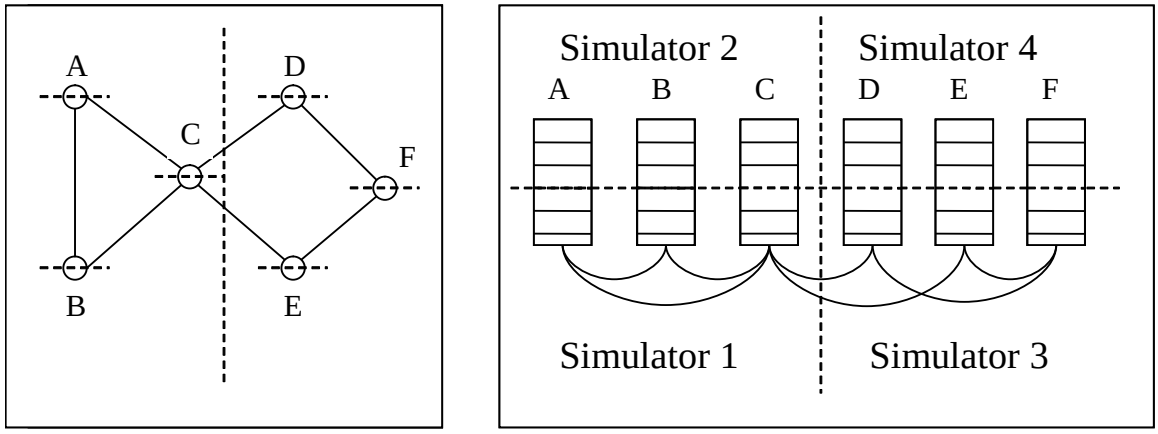


**Figure 21:** Simulation Split Both Horizontally and Vertically

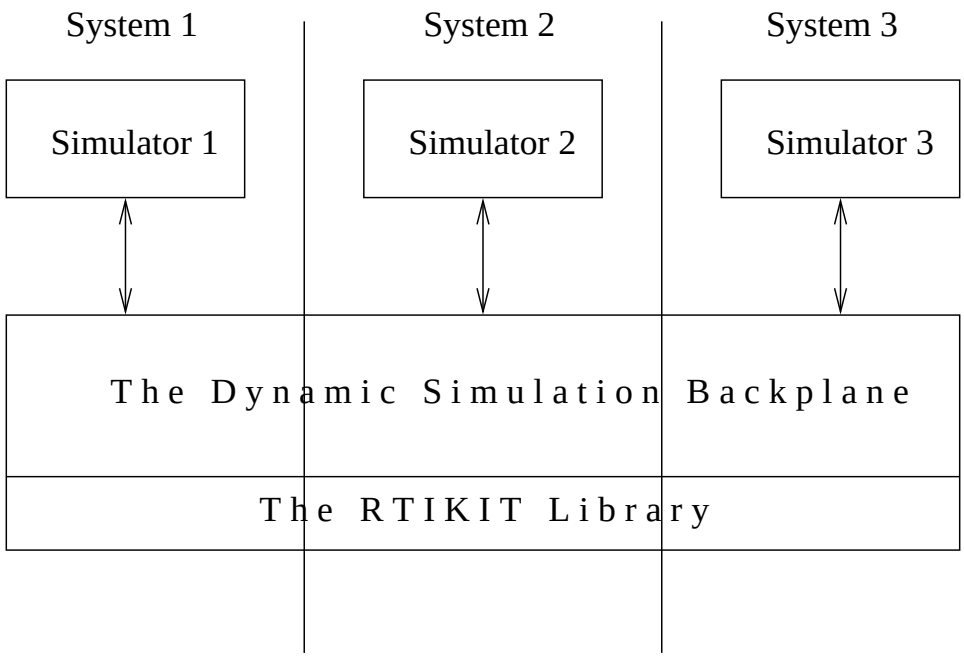


**Figure 22:** Dynamic Simulation Backplane architecture

## *4.2 The Dynamic Simulation Backplane*

The *Dynamic Simulation Backplane* is described in detail in [38]. An overview of the operation of the backplane is given here to assist in understanding the split protocol stack model. Figure 22 shows the overall architecture of a distributed simulation using the Dynamic Simulation Backplane. The figure shows a distributed simulation running on three systems. Each simulator sends and receives event messages from the backplane in *native* format, using the internal representation for events that are specific to that simulator's implementation. The backplane converts the event messages to a common, dynamic format and forwards the events to other simulators. The format of the dynamic message is determined at runtime, on a message–by–message basis. Details of this dynamic conversion process are given later in this section.

The backplane uses the services provided by a *Runtime-Infrastructure* library, known as RTIKIT[16]. The RTIKIT assists the backplane by providing the message distribution and simulation time management services required by all distributed simulations. The backplane itself provides services specific to the support for heterogeneous simulations. These services fall into three basic categories: Registration Services, Message Exporting

Services and Message Importing Services.

### 4.2.1 Registration Services

To make use of the dynamic simulation backplane for message exchange between simulators, each simulator first uses a registration process where it describes the information that is defined by event messages within that simulator. Clearly, for heterogeneous simulators to exchange meaningful information, there must be some common object model describing the information to be exchanged. Fortunately, within the networking community, there are well known and widely adopted standards for exchanging data packets between end systems. The *Request For Comments* (RFC's) published by the Internet Engineering Task Force (IETF) clearly define network protocols and required data items to be exchanged by those protocols. During the registration process, each simulator specifies which protocols are known, and which data items within the protocols have meaning. Experimental protocols or new experimental data items within an existing protocol can also be specified, thus insuring the backplane is not limited to only known protocols.

After all participating simulators have completed the registration process, a global consensus protocol is performed which results in a complete picture of all protocols and data items that are registered by any simulator. Each protocol and data item is assigned a unique *item identifier*, which is made known to all participating simulators. This item identifier is later used in the creation of the dynamic format messages exchanged between simulators.

### 4.2.2 Message Exporting Services

At some point during the execution of a heterogeneous simulation, a given event message must be forwarded from one simulator to another, with no guarantee that the two simulators have a common representation of the event format. The event message transfer might be from a given protocol stack layer on one simulator to the same protocol stack layer on the second (for example from the *IP* layer in ns2 to the *IP* layer in GloMoSim). This method is described in [38]. Alternately, the transfer could be between different layers of the same protocols stack (for example from the TCP layer in ns2 to the *IP* layer in GloMoSim).

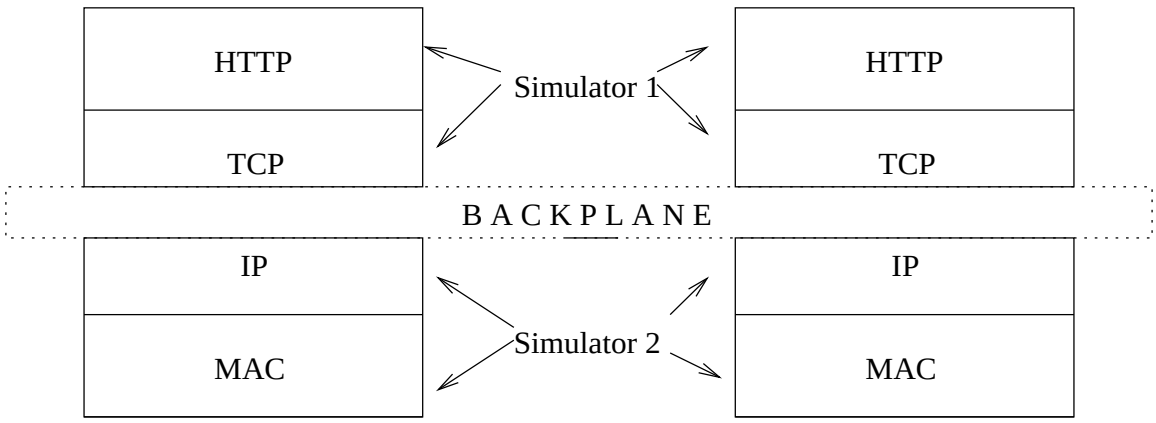


**Figure 23:** Split protocol stack method

To accomplish this heterogeneous message exchange, the backplane uses a message *Exporting* and *Importing* paradigm. A simulator sending an event to a foreign simulator calls the *ExportMessage* service, which creates a common, dynamic format message. The simulator then forwards the dynamic format message to the foreign simulator. The message format is *dynamic* in that only the data items that are meaningful for a given event message are included in the dynamic message. By querying the simulator with callback functions, the backplane can discover which items are meaningful for each message to be exported.

### 4.2.3 Message Importing Services

Once an event message has been exported to the dynamic message format as described above, the message is transfered to another simulator and is ready to be imported by the receiving simulator. The importing process is the inverse of the exporting, and causes a message to be converted from the common dynamic format to the internal event message format of the receiving simulator. This is again accomplished by the backplane using callback functions to inform the simulator of the value of each data item received in the dynamic message.

## *4.3 Splitting the Protocol Stack*

There are two natural places for heterogeneous simulators to exchange event messages. First is between the bottom layers of the protocol stack. Simulator 1 would construct an event message going down the layers of its internal stack, and would export the message after

the lowest layer has processed it. When receiving a message from simulator 1, simulator 2 would import it and then process the message going up the protocol stack starting at the lowest layer. This method of dynamic message exchange (known as the "across protocol stack" method) is the subject of the research described in [38].

An alternative method, and one providing potentially more flexibility, is the "split protocol stack" method. In this method, heterogeneous simulators exchange event messages across layers of a single protocol stack. An example of this method is shown in figure 23. Here, simulator 1 processes event messages for the HTTP and TCP layers of the protocol stack, and then passes those partially processed messages to simulator 2 for the lower layers of the stack. When receiving messages, simulator 2 processes the lower layers (MAC and IP), and then passes the message (using the backplane) to simulator 1 for further processing.

This method provides the flexibility to mix and match simulation functionality in a way that more closely suits the needs of the simulationist. Of course, the two methods described above can be combined, using the split protocol stack model in two or more simulators; connected using the across protocol stack method between other simulators. However, this method introduces a severe limitation on the overall performance of the distributed simulation, namely the presence of a zero–lookahead message exchange, as discussed below.

**Lookahead.** In a conservatively synchronized, distributed discrete event simulation, one of the primary factors affecting the performance of the simulation is the presence (or absence) of *lookahead* between the individual simulators. The lookahead between a pair of simulators is defined as a lower bound on the amount of simulation time that advances as messages are exchanged between the simulators. In a typical distributed network simulation using the across protocol stack method, there is naturally some non–zero (and potentially quite large) lookahead between any two simulators. Since messages are exchanged between simulators as packets are transmitted on some communication medium, the transmission time and propagation delay create a naturally non–zero lookahead value. Unfortunately, there is no corresponding natural delay as messages are exchanged between layers of a single protocol stack. Exchanging messages between simulators modeling different layers of the

same protocol stack results in a zero–lookahead exchange, with resulting poor performance.

Our solution to the zero–lookahead problem is to nominate one of the two simulators as the master, which will represent both simulators in the overall distributed simulation environment. We chose the simulator modeling the lower layers of the protocol stack, but this choice is arbitrary. We implemented a simple shared–memory interface between the master and slave simulators to allow a fast and efficient exchange of information between the two. The master will participate in all of the time management computations of the distributed simulation, and represent both simulators in this computation. The remainder of this section discusses the shared–memory interface and algorithms for time management in this environment. In all of this discussion, the *master* is the simulator modeling the lower layers of the protocol stack, and the *slave* is the simulator modeling the upper layers. The processing model for this split protocol environment is that, assuming the zero–lookahead message passing between the master and the slave, there can be no parallel event processing between the two. Either the master can process an event, or the slave can; but neither can process events simultaneously with the other (ignoring the issues of simultaneous timestamp events). Since we are restricted to serial event processing between the master and the slave, our approach is to minimize the waiting time between the two. Additionally, we propose running the two processes on a dual CPU system, such that one process can be processing events while the other is waiting on permission to process events.

The shared–memory interface consists of:

- Two uni–directional circular message passing queues, one for passing messages from the slave to the master (*S2M*), and a second for passing messages from the master to the slave *M2S*). Uni–directional circular queues are ideal for message passing in this environment because they require no interlocking of shared variables or critical section processing.

- *NERCcount* An integer counter specifying the number of times the slave has requested permission to advance simulation time to a new value.

- *TAGCount* An integer counter specifying the number of times the master has granted

the slave permission to advance simulation time to a new value.

- *NERTime* A floating point value specifying the simulation time advance requested by the slave.

- *TAGTime* A floating point value specifying the simulation time advance granted by the master.

- *SmallestM2S* A floating point value specifying the smallest timestamped event sent by the master to the slave since the last time advance grant to the slave. This is initialized to a value larger than any possible event in the system.

With the above shared variables, our model assumes that the slave has permission to process events if *NERCount* equals *TAGCount*, and the master has permission if it does not. We describe the processing of events at the slave first since it is the simpler of the two, followed by the processing at the master.

**Slave Processing** When the slave has permission to process events (*NERCount* equals *TAGCount*), it simply advances its local simulation time to *TAGTime*, and processes any event with a timestamp less than or equal to the *TAGTime* value. In actuality, with this model there in no possibility that an event with a timestamp less than *TAGTime* exists, because any such events would have been processed on a previous iteration. All events with timestamp equal to *TAGTime* are processed (which may result in new events with timestamp equal to the *TAGTime* being exported and passed to the master via the *M2S* queue). When all such events have been processed, the slave stores the timestamp of the earliest unprocessed event in *NERTime*, and advances *NERCount* by one. At this point, the slave has asked permission to advance time to *NERTime*, and permission to process events has been passed to the master. The slave will wait in a spin–loop until *NERCount* becomes equal to *TAGTime*, indicating permission has been given back to the slave to repeat the process. While spinning, the slave will monitor the *M2S* queue, removing messages (and or course importing to internal format using the backplane importing services), and placing them in the queue of unprocessed events in timestamp order. The processing of the event

importing while spinning gives some amount of parallelism between the master and slave processes.

**Master Processing** The master waits in a spin–loop until *NERCount* is not equal to *TAGCount*, indicating the slave has finished processing for this cycle. The master must participate in a global time management algorithm, such as that discussed in [33] to determine a lower bound on the timestamp of all unprocessed messages that may later be received, not including the slave processes. This value is called the *lower bound on timestamp* (*LBTS*). To determine an *LBTS* value, all simulators report the timestamp of their smallest unprocessed event to a global consensus protocol that computes the global minimum. The value reported by the master to the consensus protocol is determined as follows.

1. Insure the *S2M* queue is empty. If it is not, remove all pending messages from the slave and place them in the queue of unprocessed events (in timestamp order). There is no possibility of a race condition since at this point the slave no longer has permission to process events, and is simply waiting for permission. The *S2M* queue should normally be empty at this point, since the master is monitoring the queue while it is waiting for permission to process events.

2. Report the minimum of the master's own smallest unprocessed event, the *NERTime* requested by the slave, and *SmallestM2S* which represents the smallest timestamp sent by the master to the slave in the master's most recent processing cycle.

Once the *LBTS* value is known, the master can process all pending events with timestamp less than or equal to the minimum among the *LBTS* value, *NERTime*, and *SmallestM2S*. In other words, the *LBTS* value sets an upper bound on the simulation time advancement of the master/slave pair, but the master/slave pair must process events serially between them. Processing these events by the master may cause event messages to be exported and passed to the slave using the *M2S* queue. Each time an event is passed to the slave, the *SmallestM2S* value is set to the minimum of the current *SmallestM2S* value and the timestamp of the message being processed. When the master has processed all eligible

events, the *TAGTime* value is set to the minimum of the *NERTime*, *SmallestM2S*, and the *LBTS* value. The *TAGCount* value is then advanced by one, returning permission to the slave.

The net effect of this shared memory approach and the alternating permission protocol is that the local event queues of the master and slave processes appear to the federation as a single event queue. At any point in time, only the smallest event of the two event queues can safely be processed, which mimics the behavior that would be obtained if the two queues were merged to a single queue.

## *4.4 The ns2 to GloMoSim Split Protocol Stack Simulation*

We experimented with the split protocol stack simulation with GloMoSim and ns. The protocol stack is split between the TCP and the IP layers, with ns2 simulating the upper portion of the protocol stack and GloMoSim simulating the lower portion. Each GloMoSim/ns2 pair simulates a wireless network that includes a number of mobile nodes. These wireless networks are connected to each other through a backbone network, which is simulated by a number of PDNS simulators. Figure 24 shows a simulation configuration that consists of four GloMoSim/ns2 wireless networks and four PDNS backbone networks. Each GloMoSim/ns2 pair connects to exactly one PDNS, and the *pdns*'s are fully connected to each other. There is FTP traffic between wireless nodes in a wireless network, and also FTP traffic between wireless networks that goes through the PDNS backbone.

We ran the simulation on a multi-processor shared-memory system, and each GloMoSim, ns2 and PDNS process was running on a separate processor. One processor was assigned to each PDNS backbone network, and a pair of processors was assigned to each GloMoSim/ns2 pair. The number of processors assigned was increased linearly as the number of wireless networks being modeled was increased.

In the experiments we varied two parameters to measure the time to complete the simulation. The two parameters are, 1) number of wireless networks (i.e. number of GloMoSim/ns2 pairs, which equals to the number of PDNS simulators in between, since each GloMoSim/ns2 pair connects to exactly one PDNS), and 2) the percentage of local traffic

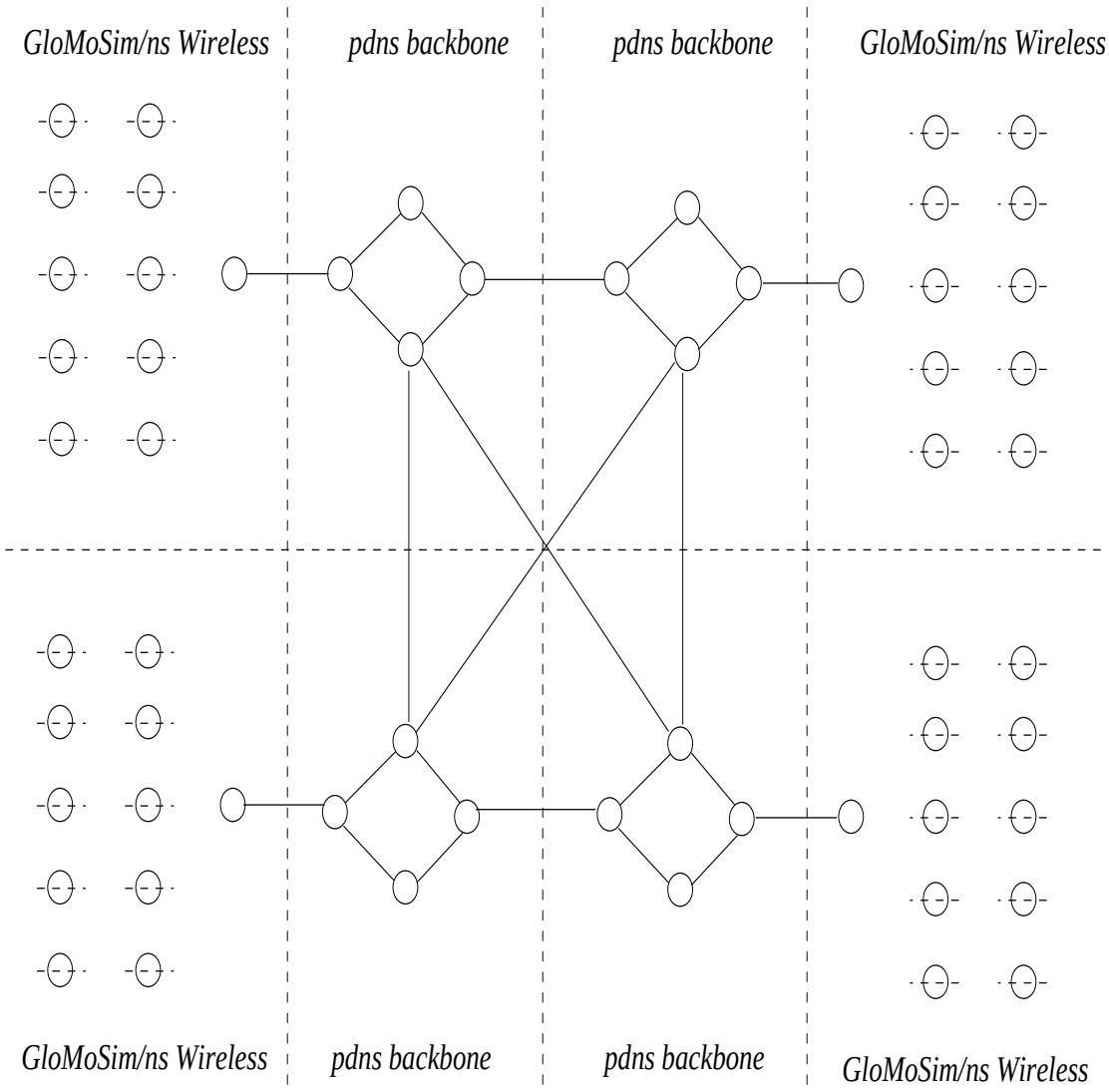


**Figure 24:** Simulation configuration with four GloMoSim/ns2 pairs and four PDNS's

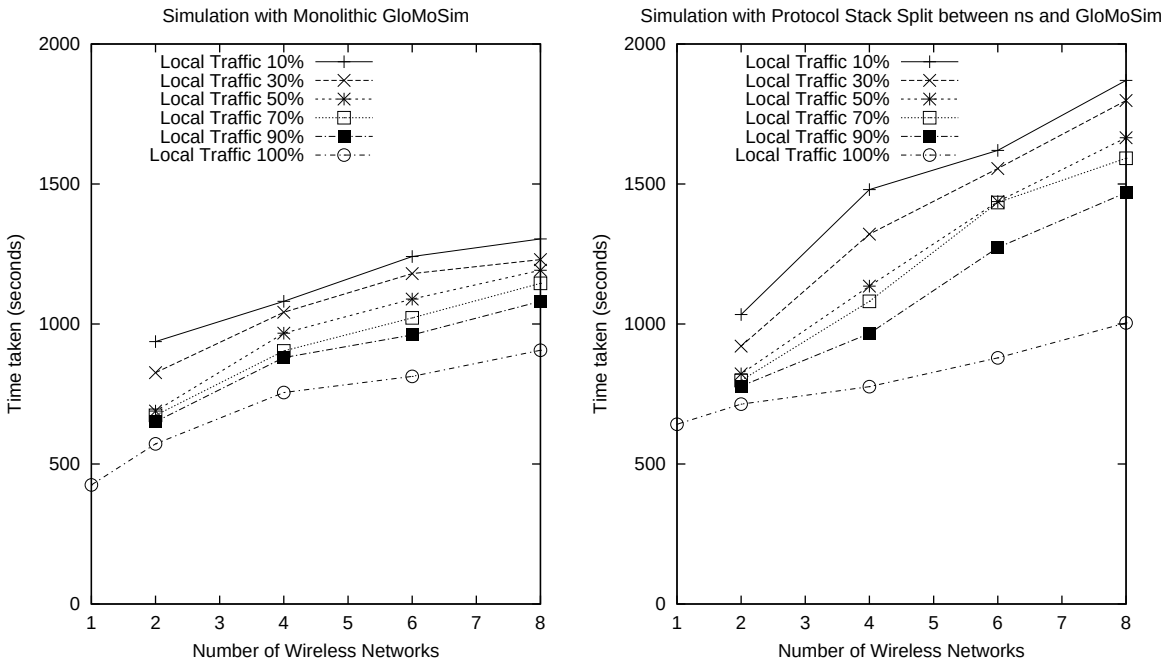


**Figure 25:** Simulation running time with 200 wireless nodes in each wireless network

in the total FTP workload. Note that the total traffic grows linearly with the number of wireless networks modeled. For example, if the total traffic of one wireless networks is 1MB, then the total traffic of eight wireless networks is 8MB, including both the local traffic in the same wireless network and the traffic between wireless networks that goes through the backbone. By growing the traffic linearly with the number of wireless networks being simulated, and by expanding the number of processors in the federation at the same time, a "perfect" speedup ratio would be indicated by identical running times for each of the simulations.

For comparison, we also ran the simulations with monolithic GloMoSim with the same configurations. Figure 25 shows the performance when the number of wireless nodes in each wireless network is fixed at 200. The left chart is the performance of a monolithic GloMoSim simulation, and the right chart is the performance of GloMoSim/ns2 split protocol stack simulation.

The baseline case is one wireless network where 100 percent of the traffic is local. In the right figure, we can see that as the number of wireless networks increases, the time it takes to complete the simulation does increase, but the increase is reasonably small. Generally

speaking, larger local traffic percentages lead to better speedup. This is expected, because a large amount of local traffic increases the number of local events at a given simulator that can be processed in a single lookahead window. At the other extreme, even when only 10 percent of the traffic is local traffic, running eight wireless networks plus eight PDNS backbones still only takes about twice as the time to run two wireless networks plus two PDNS backbones.

This shows that the parallel simulation speeds up the simulation significantly: it takes 2-3 times more time to simulate eight times larger network and eight times more traffic. Also, splitting the protocol stack(the right chart) results in 1-2 times slow-down compared to running the monolithic simulator(the left chart).

## *4.5 Conclusions*

different network simulators have different strengths and weakness. To combine the strengths of different simulators, we previously introduced the *Dynamic Simulation Backplane*[38] to allow different simulators to work together. In this chapter we discussed the split protocol stack methodology for network simulation that allows networking researchers to run different simulators together, and take advantage of the strong implementation at different protocol layers of different simulators. Using this methodology, we are able to construct complex parallel heterogeneous network simulation scenarios where GloMoSim simulates the MAC layer of wireless networks, ns2 simulates the TCP layer of the same wireless networks and PDNS simulates the wired backbone between wireless networks. Our experiments show that the parallel execution of the GloMoSim and PDNS simulators speeds up the simulation as the simulated network scales.

# CHAPTER V

# CONCLUSIONS AND FUTURE DIRECTIONS

This thesis has presented our research work on scalable and composable network simulation. For scalability, on the one hand we have developed a systematic methodology called BencHMAP to produce a partition a network simulation for efficient parallel execution; on the other hand we have developed aggregation and compression techniques to substantially reduce memory requirements of routing state information in multicast simulations. For composability, we have developed a framework to compose a simulation from different simulators modeling different protocol stack layers and cooperating through an underlying simulation backplane.

We can foresee a number of future directions to continue the three pieces of work in this thesis.

1. For the BencHMAP work, this thesis has only provided a preliminary evidence that BencHMAP can help improve parallel simulation performance in certain cases. More extensive studies are still required to understand the related issues and the effectiveness of this methodology.

   (a) We have only used a simple network topology and traffic model to complete the benchmark experiments, not considering the actual network being modeled. Future research should examine how the large network models themselves should be abstracted to construct more representative benchmark experiments, so that the performance-factor relation derived from the benchmark experiments more accurately reflect the characteristics of the large network models themselves.

   (b) To best exploit the resources of heterogeneous computer clusters which consist of computers with different capacities, it would be necessary to design a method to benchmark each machine's capacity first, and allocate partitions proportionally

based on each machine's capacity.

(c) A set of tools are needed to automate the benchmark experiments on a given computer cluster and derive the correct performance-factor relationships for use by the partitioning tool.

(d) The empirical approach to derive performance-factor relation for large simulations might also be applied to other parallel scientific computation tasks (such as physical simulation, biological simulation etc.) to improve their performance.

(e) In addition to simulation model, workload model, the computer cluster configuration, and the parallel execution performance, the accuracy or level of detail of the simulation may also be taken into consideration in this methodology. It remains to be seen how to construct benchmarks to achieve a good trade-off between accuracy and performance.

2. For the work of reducing memory requirements, this thesis has only attempted to aggregate routing state in multicast simulations. Other types of network simulations, or more generally, other types of computation tasks, may also have scalability issues caused by large amounts of superfluous or redundant state. Aggregating or compressing the state could be an effective approach to scale those simulations.

   (a) When the execution of a simulation occupies a large amount of memory, it might be possible to take a snapshot of the memory occupied by simulation state, and if compressing this data image reduced its size significantly, it would be an indication that some simulation state might be aggregatable. It will be very useful if one could embed instrumental procedures into the simulators to help taking the snapshot and identifying which state is substantially aggregatable.

   (b) Compressing simulation state implies sacrificing CPU cycles for memory space; therefore, modeling and benchmarking the trade-off between CPU, memory, I/O etc. required by a certain representation of simulation state might provide more insights into how a certain computing task could achieve the best performance at a certain scale on a certain hardware configuration.

(c) It may also be possible to distribute multicast simulations onto multiple processors, which is in itself a challenge because the current distributed simulation software such as PDNS still does not support multicast simulation across processors.

(d) One could also explore the potential of the Negative Forwarding Table Technique in the context of Application-Layer Multicast to reduce the memory requirements of multicast routing state.

3. For the split protocol stack simulation work, we have only attempted to compose complex simulation with exploiting any parallelism.

   (a) We believe there is potential for more parallelism between the ns2 and GloMoSim pairs, since each NS/GloMoSim pair typically simulates more than one protocol stack.

   (b) In many network simulation scenarios, a simulator knows a priori when packets of a given flow are going to another simulator, and we may be able to exploit this knowledge to achieve more parallelism in the simulation. We can exploit these potential methods for more parallelism, and further improve the performance of complex parallel network simulation scenarios.

With the ability to scale and compose network simulations, our ultimate goal is to be able to construct and run very large and very complex network simulations with a good performance. In this scenario, a large and complex simulation will be partitioned so that when a portion of simulation is executed by a simulator on a processor, this portion of simulation will best utilizes the capability of this simulator, and this simulator will best utilize its memory resources by aggregating and compressing excessive simulation state, and the overall partitioning will best utilize the resources of the underlying computer cluster to achieve a good performance. Working towards this goal requires more research into how the different pieces of work introduced in this thesis can be combined. For example, heterogeneous capabilities and performance of different simulators would bring new complexities

to the benchmark selection and node weight calculation of the BencHMAP methodology. Bringing all different approaches together would hopefully result in a flexible and powerful framework for large and complex network simulations.

# APPENDIX A

# AUTOPART: A SIMULATION PARTITIONING TOOL FOR PDNS

## *A.1 Introduction*

AutoPart[48] is a tool to automatically partition a large network simulation into smaller simulation instances, so that they can be run on a number of machines with PDNS, a parallel and distributed network simulation tool based on ns2. By distributing a simulation onto a number of machines, we can achieve the simulation scale that the original ns2 cannot achieve on a single workstation.

If the user have a network topology beforehand and wants to do simulation with this topology using PDNS, it would be a very tedious and error-prone task to partition the simulation manually, replacing some links with rlinks, adding all those add-route statements, etc.. When the topology is relatively large (say, ¿1,000 nodes), then it is virtually impossible to partition by hand. In addition, even if the user can partition by hand, he/she would not know for sure how well the performance of the partitioned simulation will be, since this performance depends on a complicated combination of lookahead, load balancing and communication overhead. Therefore, we developed this tool to help the user automate the partitioning process. This tool takes a ns2 script and creates a number of pdns scripts that are ready to run in parallel on a number of machines, attempting to make the best trade-off between lookahead, load balancing and communication overhead in the partitioning process, resulting in the best performance when being run by PDNS.

## *A.2 Prerequisites*

1. This simulation partitioning tool requires the graph partitioning package METIS[23] to be installed beforehand. After installing METIS, make sure the metis/pmetis/kmetis

programs are in the path.

2. The PDNS the user is running must be version 2.27v1b or above. This is because this tool generates PDNS scripts that make use of an expanded "add-route" syntax that only works for PDNS2.27v1b, which is not officially released yet but can be downloaded here:

   $http://www.cc.gatech.edu/grads/x/Donghua.Xu/autopart/pdns_2.27_v1b.gz$

   The installation of pdns2.27v1b is similar to the "Building ns-2 and PDNS" section of pdns2.27v1a:

   $http://www.cc.gatech.edu/computing/compass/pdns/$

## A.3 Download and Installation

1. Download the C++ source code at:

   $http://www.cc.gatech.edu/grads/x/Donghua.Xu/autopart/autopart.cc$

2. The user can use g++ 2.95.2 or above to compile this code, such as:

```
g++ -o autopart autopart.cc
```

## A.4 Assumptions

1. The input ns2 script takes a nam-editor -like format. More specifically, in this format, the node, agent and traffic objects must be defined in the form node(n), agent(n), traffic_source(n), where n is a non-negative integer number. Following is a simple example script of a topology with six nodes, illustrated in Figure26

```
# Create a new simulator object.
set ns [new Simulator]
# Create wired nodes.
set node(1) [$ns node]
set node(2) [$ns node]
```

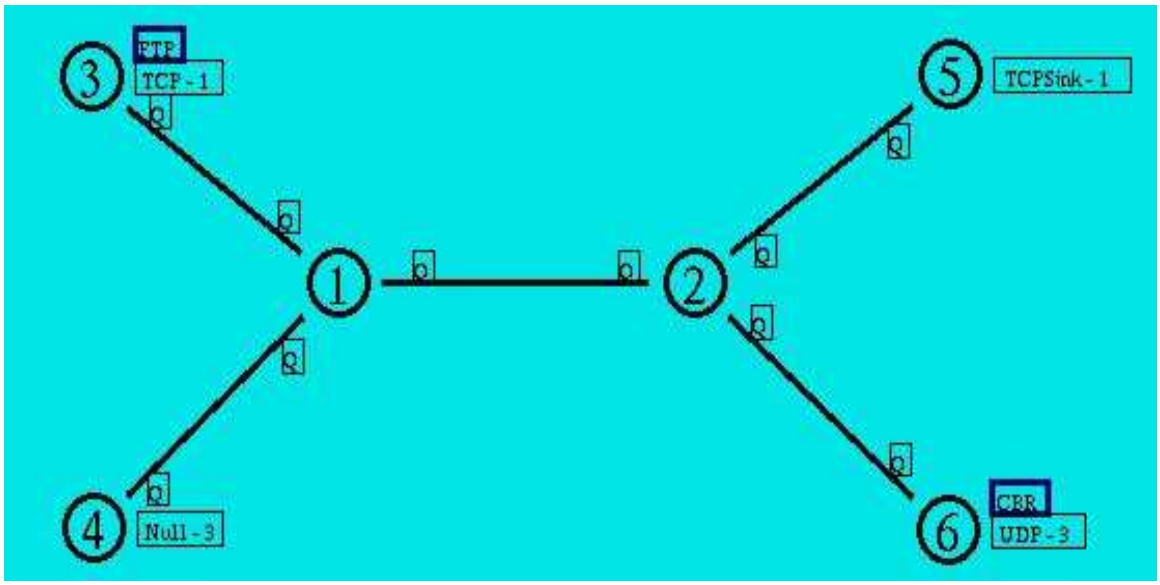


**Figure 26:** Topology of six nodes

```
set node(3) [$ns node]
set node(4) [$ns node]
set node(5) [$ns node]
set node(6) [$ns node]
# Create links between nodes.
$ns duplex-link $node(2) $node(1) 100.000000Mb 2.000000ms DropTail
$ns duplex-link $node(3) $node(1) 100.000000Mb 2.000000ms DropTail
$ns duplex-link $node(4) $node(1) 100.000000Mb 2.000000ms DropTail
$ns duplex-link $node(5) $node(2) 100.000000Mb 2.000000ms DropTail
$ns duplex-link $node(6) $node(2) 100.000000Mb 2.000000ms DropTail


# Create agents.
set agent(1) [new Agent/TCP]
$ns attach-agent $node(3) $agent(1)
set agent(2) [new Agent/TCPSink]
$ns attach-agent $node(5) $agent(2)
set agent(3) [new Agent/UDP]
```

```
$ns attach-agent $node(6) $agent(3)
set agent(4) [new Agent/Null]
$ns attach-agent $node(4) $agent(4)

# Create traffic sources and add them to the agent.
set traffic_source(1) [new Application/FTP]
$traffic_source(1) attach-agent $agent(1)
set traffic_source(2) [new Application/Traffic/CBR]
$traffic_source(2) attach-agent $agent(3)
# Connect agents.
$ns connect $agent(1) $agent(2)
$ns connect $agent(3) $agent(4)
# Traffic Source actions.
$ns at 0.000000 "$traffic_source(1) start"
$ns at 6000.000000 "$traffic_source(1) stop"
$ns at 0.000000 "$traffic_source(2) start"
$ns at 6000.000000 "$traffic_source(2) stop"

# Run the simulation
proc finish {} {
  global ns trfile
  $ns halt
}
$ns at 100.0 "finish"
$ns run
```

If the original ns2 script is not in this format, it is generally not difficult to write a perl or tcl script to convert it into this format. Some larger scales of ns2 simulations

in this format can also be downloaded here:

(a) 538 nodes:

$http://www.cc.gatech.edu/grads/x/Donghua.Xu/autopart/campus538.tcl$

(b) 3,886 nodes:

$http://www.cc.gatech.edu/grads/x/Donghua.Xu/autopart/dartmouth3886.tcl$

(c) 21,424 nodes:

$http://www.cc.gatech.edu/grads/x/Donghua.Xu/autopart/medium.tcl.gz$

(d) 123,536 nodes:

$http://www.cc.gatech.edu/grads/x/Donghua.Xu/autopart/m32.tcl.gz$

(e) and 417,200 nodes:

$http://www.cc.gatech.edu/grads/x/Donghua.Xu/autopart/m64.tcl.gz$

2. The partitioning tool assumes two types of nodes in a simulation topology: endhost and router. Endhosts are the nodes that connect to only one node (i.e., its router) and carry the agents that send or receive application traffic. Routers are the nodes that connect to two or more nodes, and only foward the traffic from other nodes. In Figure 26, nodes 1 and 2 are routers, while nodes 3, 4, 5 and 6 are endhosts. Making this distinction is very important. The user must attach agents and applications to endhosts. If the user attached an agent and application to a router such as node 1 or 2 in the above figure, our tool could produce unpredictable (and maybe un-runnable) pdns scripts.

3. This tool requires a pre-determined set of relations between the three parallel simulation performance factors(look ahead, load balancing and communication overhead) and the simulation model as well as the computer cluster hardware configuration being used, in order to assign reasonable edge and vertex weights to the links and nodes of the topology, so that it can produce a partitioned simulation that runs the fastest. This set of relations should be obtained through a series of benchmark experiments (see [49] and Chapter 2 for details). For now this tool hardcodes the set of relations

that we obtained through benchmark experiments on our Ferrari cluster computer which consists of eight machines connected via a Gigabit LAN, each machine having two 3GHz CPU's, sharing 2G memory, and should be applicable to similar homogeneous platforms directly.

## *A.5 Usage*

The usage of this tool is shown below:

```
autopart -n num[.num] [-W routefile] [-R routefile] ns2script
```

Explanation:

1. The ns2script is the name of the original ns2 script that the user wants to partition.

2. The -n option is to specify how many parts the user wants to partition the original script into. The user can specificy it to be either one level or two level parts. If the user has a computer cluster with x machines, each machine having y CPU's, and suppose z=x*y, then he/she can either specify -n z or specify -n x.y, and in most cases the latter would generate PDNS scripts with a better performance, since it takes into account the discrepancy between the communication overhead over shared-memory and over LAN.

3. The -W and -R option is to ask autopart to write/read the calculated routes into/from a route file. Route calculation might take up quite some time in the partitioning process, especially for large and complex network topologies with a large number of traffic streams. With this -W option, the first time the user partitions a simulation, he/she can store the calculated routes in a file, and next time when he/she partitions the same simulation (e.g., partitioning into a different number of parts), he/she can use the -R option to ask autopart to read the calculated routes and skip the route calculation completely.

## *A.6 Usage Examples*

1. To partition 6nodes.tcl into two parts to be run on two different machines:

```
autopart -n2 6nodes.tcl
```

2. To partition 6nodes.tcl into two parts to be run by two different CPU's on one machine:

```
autopart -n1.2 6nodes.tcl
```

3. To partition medium.tcl into eight parts to be run on four different machines, each machine having two CPU's:

```
autopart -n4.2 medium.tcl
```

4. To partition m32.tcl into 12 parts to be run on six different machines, each machine having two CPU's, and wite the routes into rfile.txt:

```
autopart -n6.2 -Wrfile.txt m32.tcl
```

5. To partition m32.tcl into 16 parts to be run on eight different machines, each machine having two CPU's, but read the routes from rfile.txt instead of recalculating the routes:

```
autopart -n8.2 -Rrfile.txt m32.tcl
```

## *A.7 Running the Partitioned Simulation*

The user can also download runsim.pl at:

*http* : *//www.cc.gatech.edu/grads/x/Donghua.Xu/autopart/runsim.pl*

which is a simple Perl script that helps the user run the partitioned simulation with pdns on a computer cluster. The original runsim.pl runs well on our ferrari cluster which consists of 8 machines sharing the same file system, each machine having two CPU's. The machines are named ferrari001-ferrari008, as can be seen in the runsim.pl. The user can modify the

ferrari part of the script to change the machine names to what the cluster machines are named. Suppose pdns is already in the path, after the user partitioned the m32.tcl into 16 parts as in the last example above, the user can run the partitioned simulation as follows:

```
runsim.pl -m8 -c2 -nm32
```

## *A.8 Performance of Autopart Itself*

We ran autopart on a 3GHz P4 machine with 2GB memory to partition the 417,200-node 400,000-stream ns2 simulation into 8*2 parts. It took 12 minutes to finish if autopart had to calculate all routes, and just five minutes if autopart was allowed to read previously-calculated routes. The memory it occupied did not top 700 MB.

Interestingly, our experiments show that, as the original ns script scales up, the memory required by Autopart increases linearly, and the running time increases quadratically. So in future even if we need to partition a million node and million stream simulation, it probably won't take more than 2G memory and an hour of running time on the same machine.

# REFERENCES


[1] Bagrodia, R., "Private communication," 1999.

[2] Bagrodia, R., Meyer, R., Takai, M., Chen, Y., Zeng, X., Martin, J., Park, B., and Song, H., "Parsec: A parallel simulation environment for complex systems," *IEEE Computer*, vol. 31, pp. 77–85, October 1998.

[3] Ballardie, T., Francis, P., and Crowcroft, J., "Core based trees (CBT) an architecture for scalable multicast routing," in *Computer Communications Review, Proceedings of ACM SIGCOMM 93*, pp. 85–95, September 1993.

[4] Bertolotti, S. and Dunand, L., "Opnet 2.4: an environment for communication network modeling and simulation," in *Proceedings of the European Simulation Symposium*, October 1993.

[5] Briscoe, R. and Tatham, M., "End to end aggregation of multicast addresses." Internet Draft, http://www.labs.bt.com/people/briscorj/projects/lsma/e2ama.html, 1997.

[6] Chandy, K. and Misra, J., "Distributed simulation: A case study in design and verification of distributed programs.," in *IEEE Transactions on Software Engineering*, September 1979.

[7] Cowie, J., Liu, H., Liu, J., Nicol, D., and Ogielski, A., "Towards realistic million-node internet simulations," in *International Conference on Parallel and Distributed Processing Techniques and Applications*, June 1999.

[8] Cowie, J. H., Nicol, D. M., and Ogielski, A. T., "Modeling the global internet," *Computing in Science and Engineering*, January 1999.

[9] Deering, S., Estrin, D., Farinacci, D., Jacobson, V., C.G.Liu, and Wei, L., "The PIM architecture for wide-area multicast routing," *IEEE/ACM Transactions on Networking*, vol. 4, pp. 153–162, April 1996.

[10] Deering, S. E. and Cheriton, D. R., "Multicast routing in datagram internetworks and extended LANs," *ACM Transactions on Computer Systems*, vol. 8, pp. 85–110, August 1994.

[11] El-Rewini, H. and Lewis, T. G., *Distributed and Parallel Computing*. Manning Publications Co., 1998.

[12] Fall, K. and Floyd, S., "Simulation-based comparisons of tahoe, reno, and sack tcp," in *Computer Communications Review*, July 1996.

[13] Foster, I., *Designing and Building Parallel Programs*. Addison-Wesley Publishing Company, 1995.

[14] Fujimota, R. M., Permualla, K., and Tacic, I., "Design of high performance RTI software," in *Distributed Simulation and Real-Time Applications 2000*, August 2000.

[15] Fujimoto, R., *Parallel and Distributed Simulation Systems*. Wiley Interscience, 1999.

[16] Fujimoto, R., "RTI-KIT v0.2 specification," March 1998.

[17] Fujimoto, R., Perumalla, K., Park, A., Wu, H., Ammar, M., and Riley, G., "Large-scale network simulation – how big? how fast?," in *Proceedings of International Symposium on Modeling, Analysis and Simulation of Computer Telecommunication Systems*, 2003.

[18] Fujimoto, R. M., "Lookahead in parallel discrete event simulation," pp. 34–41, 1988.

[19] Fujimoto, R. M., "Performance of time warp under synthetic workloads," pp. 23–28, 1990.

[20] Huang, P., Estrin, D., and Heideman, J., "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.

[21] Ion Stoica, T. E. N. and Zhang, H., "Reunite: A recursive unicast approach to multicast," in *Proceedings of IEEE Infocom 2000*, 2000.

[22] Jha, V. and Bagrodia, R. L., "Transparent implementation of conservative algorithms in parallel simulation languages," in *Proceedings of the 1993 Winter Simulation Conference*, pp. 677–686, December 1993.

[23] Karypis, G., Aggarwal, R., Schloegel, K., Kumar, V., and Shekhar, S., "Metis: Family of multilevel partitioning algorithms." Software online: http://www-users.cs.umn.edu/ karypis/metis/, 2003.

[24] Liljenstam, M., Liu, J., and Nicol, D. M., "Development of an internet backbone topology for large–scale network simulations," in *Proceedings of the 2003 Winter Simulation Conference (WSC'03)*, 2003.

[25] Liu, X. and Chien, A. A., "Traffic-based load balance for scalable network emulation," in *Proceedings of the ACM Conference on High Performance Computing and Networking, SC2003*, 2003.

[26] McCanne, S. and Floyd, S., "The LBNL network simulator." Software on-line: http://www.isi.edu/nsnam, 1997. Lawrence Berkeley Laboratory.

[27] Nicol, D., Johnson, M., Yoshimura, A., and Goldsby., M., "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.

[28] Nicol, D., "Scalability, locality, partitioning, and synchronization in pdes," in *Proceedings of the Parallel and Distributed Simulation Conference*, 1998.

[29] Nonnenmacher, J. and Biersack, E. W., "Scalable feedback for large groups," *IEEE/ACM Transactions on Networking*, July 1999.

[30] Pavlin I. Radoslavov, R. G. and Estrin, D., "Exploiting the bandwidth-memory tradeoff in multicast state aggregation." Technical Report 99-697, USC Computer Science Department, 1999.

[31] Perumalla, K., Fujimoto, R., and Ogielski, A., "Ted - a language for modeling telecommunications networks," *Performance Evaluation Review*, vol. 25, March 1998.

[32] Perumalla, K. S. and Fujimoto, R. M., "Efficient large-scale process-oriented parallel simulations," in *Proceedings of the Winter Simulation Conference*, December 1998.

[33] Perumalla, K. S. and Fujimoto, R. M., "Virtual time synchronization over unreliable network transport," in *15th Workshop on Parallel and Distributed Simulation*, May 2001.

[34] Perumalla, K., "libsynk/rti." Software online: http://www.cc.gatech.edu/computing/pads/kalyan/ libsynk.htm, 2004.

[35] Preiss, B. R. and Loucks, W. M., "The impact of lookahead on the performance of conservative distributed simulation," in *Proceedings of European Multiconference–Simulation Methodologies, Languages and Architectures*, pp. 204–209, June 1990.

[36] Pusateri, T., "Distance vector multicast routing protocol." Internet Engineering Task Force, April 1997. Internet Draft.

[37] Riley, G. F., Ammar, M. H., and Fujimoto, R. M., "Stateless routing in network simulations," in *Proceedings ofthe Eighth International Symposium on Modeling, Analysis and Simulation of of Computer and Telecommunication Systems*, August 2000.

[38] Riley, G. F., Ammar, M. H., Fujimoto, R. M., Xu, D., and Perumalla, K., "Distributed network simulations using the dynamic simulation backplane," in *Proceedings of the 21st Annual Conference on Distributed Computing Systems*, April 2001.

[39] Riley, G. F., Fujimoto, R. M., and Ammar, M. H., "A generic framework for parallelization of network simulations," in *Proceedings of Seventh International Symposium on Modeling, Analysis and Simulation of of Computer and Telecommunication Systems*, October 1999.

[40] Riley, G. F., Fujimoto, R. M., and Ammar, M. H., "Parallel/Distributed ns." Software on-line: www.cc.gatech.edu/computing/compass/pdns/ index.html, 2000. Georgia Institute of Technology.

[41] Riley, G. F., Fujimoto, R. M., and Ammar, M. H., "Reduced-state models for distributed network simulations," in *Submitted for publication.*, 2000.

[42] Schloegel, K., Karypis, G., and Kumar, V., "Graph partitioning for high performance scientific simulations," 2000.

[43] Shi, S. and Waldvogel, M., "A rate-based end-to-end multicast congestion control protocol," in *Proceedings of ISCC 2000*, (Antibes, France), pp. 678–686, July 2000.

[44] Tandri, S. and Abdelrahman, T. S., "Computation and data partitioning on scalable shared memory multiprocessors," in *International Conference on Parallel and Distributed Processing Techniques and Applications*, 1995.

[45] Tang, J. and Neufeld, G., "Forwarding state reduction for sparse mode multicast communication," in *Proceedings of IEEE Infocom 1998*, 1998.

[46] Thaler, D. and Handley, M., "On the aggregatability of multicast forwarding state," in *Proceedings of IEEE Infocom 2000*, 2000.

[47] Wagner, D. B. and Lazowska, E. D., "Parallel simulation of queueing networks: Limitations and potentials," *Performance Evaluation Review*, vol. 17, pp. 146–155, May 1990.

[48] Xu, D., "Automatic partitioning tool for pdns." Software online: http://www.cc.gatech.edu/ xu/partitioning/, 2004.

[49] Xu, D. and Ammar, M. H., "Benchmap: Benchmark-based, hardware and model-aware partitioning for parallel and distributed network simulation," in *Proceedings of the Ninth International Symposium on Modelling, Analysis, and Simulation of Computer and Telecommunication Systems*, 2004.

[50] Xu, D., Riley, G. F., Ammar, M. H., and Fujimoto, R. M., "Split protocol stack network simulations using the dynamic simulation backplane," in *Proceedings of the Ninth International Symposium on Modelling, Analysis, and Simulation of Computer and Telecommunication Systems*, 2001.

[51] Yocum, K., Eade, E., Becker, J. D. D., Chase, J., and Vahdat, A., "Toward scaling network emulation using topology partitioning," in *International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, 92.

[52] Zeng, X., Bagrodia, R., and Gerla, M., "GloMoSim: a library for parallel simulation of large-scale wireless networks," in *Proceedings of the 12th Workshop on Parallel and Distributed Simlations*, May 1998.