Enabling Large-Scale Multicast Simulation by Reducing Memory Requirements *

Donghua Xu[1]
George F. Riley[2]
Mostafa H. Ammar[1]
Richard Fujimoto[1]

[1]*College of Computing*
Georgia Institute of Technology
Atlanta, GA 30332-0280
{*xu,ammar,fujimoto*}*@cc.gatech.edu*

[2]*College of Engineering*
School of ECE
Georgia Institute of Technology
Atlanta, GA 30332-0250
riley@ece.gatech.edu

Abstract

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 paper we describe three approaches to substantially reduce the memory required by multicast simulations: 1) We introduce a novel technique called "negative forwarding table" to compress mutlicast 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*.*

1. Introduction

Unicast communication allows a sending host on the network to send data to one receiving host. However when a host needs to send data to many receivers, repeatedly unicasting the data will result in wasted bandwidth in the network and heavy computational burden on the sender. Multicast [5] was proposed to economize network bandwidth and the sender's workload by transmitting at most one copy of the data on any link, and only duplicating the data at branch points in the network where a router has to deliver the data onto two or more output interfaces.

Multicast has a number of potential applications that involve simultaneous delivery of identical data to multiple hosts, such as video conferencing, multi-user game, and content-distribution, just to name a few. The multicast research is often concerned with *large multicast networks*, where either the multicast group is large, the number of groups is large, or both. As a result, the scalability of the multicast network simulators becomes a serious concern. For example, Nonnenmacher and Biersack [9] used a mathematical model to analyze their scalable feedback mechanism for large groups, consisting of up to 1 million members. However they were not able to perform packet-level simulations to validate this analysis. As another example, Shi and Waldvogel [17] proposed a sender-based congestion control approach for multicast traffic and used *ns2* to evaluate their approach. However, they were unable to simulate large multicast groups while at the same time maintaining a large number of SRM flows in the simulation.

The limited scalability of existing multicast simulation methods is primarily due to the large amount of state maintained by the simulators, which is often on a high order of the input size (such as quadratic with the number of nodes, or the product of the number of nodes and number of groups). This state requires a proportional amount of memory in the simulator, which can often exceed the capacity of any contemporary computer system.

There are two general approaches that can be used to address this memory allocation problem in simulations, without loss of accuracy in the simulations:

1. We can *distribute* the simulation task across a number of computers, thereby utilizing the memory of a set of machines. A number of researchers are exploring methods and techniques for scalable network simulations using parallel and distributed simulation, such as [16, 15, 12, 11, 8, 4, 20, 1].

*This work is supported in part by NSF under contracts number ANI-9977544 and ANI-0136936

2. We can *reduce the memory* required to represent the state of the model. We observe that in typical simulations, a) the superfluous states (that are *not* needed by the simulations) often occupy excessive memory, which can be safely eliminated (e.g. [14, 6]); b) the necessary states (that are needed by the simulations) often contain significant amount of redundant information, which can be easily compressed/aggregated (e.g. [14]).

In this paper, we focus primarily on techniques for reducing the state memory requirements to enable large-scale multicast simulations. We employ both methods described above to reduce memory requirements: removing superfluous states and aggregating necessary states.

The remainder of this paper is organized as follows. In section 2 we give a brief overview of multicast routing, analyze the memory requirements of routing states and related objects in a typical simulator, and discuss existing work that has attempted to aggregate multicast routing state. In section 3 we discuss three memory saving techniques for multicast simulation, namely, the "Negative Forarding Table", the replicator aggregation, and the *NIx–Vectors* techniques. Section 4 presents experimental results demonstrating the memory savings of these three techniques. And in section 5 we give conclusions of this paper and future directions of our research.

2. Multicast Routing Memory Requirements

2.1. Background on Multicast Routing

Multicast data delivery services usually work 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.

If there are multiple sources in a group, the Source-Based Tree (SBT) approach[5, 13] constructs a different tree for every different source, so that packets from each source are delivered along this source's tree. Compared to Core-Based Tree (CBT) approach[2] 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. We are addressing exactly this scalability problem, hence we will assume the SBT approach in multicast routing for the remainder of this paper.

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 4 output interfaces on each router, the memory overhead per group per source on each node is 40 bytes, and the memory requirement of each output interface is 4 bytes, then the total memory requirement 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.

2.2. Existing Work on Aggregating Multicast Routing State in Real Networks

There is a significant body of work dealing with the problem of reducing the memory required to store multicast 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[18, 7] that attempted to store the mutlicast 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.

On the other hand, some other researchers also worked on aggregating multicast routing state without the sparse tree assumption. Briscoe and Tatham [3] proposed a completely new multicast address naming scheme that explicitly provides methods for routers to aggregate multicast routing state. Radoslavovet et al. [10] 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. Thaler and Handley [19] suggested that multicast routing state can be aggregated with just an "interface-centric" representation of multicast 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, (e.g., if the original set of addresses with 5 entries are $\langle 2, 5, 6, 7, 10\rangle$, it can be aggregated into the form $\langle (2, 1), (5, 3), (10, 1)\rangle$ that has only 3 entries. However the effectiveness of this method completely relies on the 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.

3. Techniques to Reduce Memory Requirement 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.

3.1. 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 take the reverse 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, or 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 4 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 en-

try $\langle grp1, src13\rangle$, 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.

The ideal cases where the NFT approach works best are: for a group *grp*, the interface needs to deliver packets that come from either none or all of the sources of *grp*. If the interface needs to deliver none, then there is no entry for *grp* in either PFT or NFT. 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.

The effectiveness of the NFT technique (i.e., how much saving the NFT approach could achieve over the PFT-only approach) depends on how likely different trees overlap on each interface, which largely depends on the following three factors.

1) Number of sources of a group whose trees pass through a router. The more sources a group has on a router, the more likely different trees will overlap on an individual interface.

2) The receiver *density* of the group. This density can be defined as the ratio of the number of receivers of a group divided by the total number of nodes in the network. It determines the average number of output interfaces that a $\langle$grp, src$\rangle$ pair can have on a router. The denser a group, the more output interfaces a tree would require on a router, the more likely different trees would overlap on an interface.

3) The connectivity of the network, i.e., the availability of alternative paths between any two nodes in the network. The fewer alternative paths that exist, the more likely that different trees will overlap on the same router, hence the more saving the NFT approach will achieve. The worst 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 requirement.

Multiple-source multicast is useful for conferencing applications and multi-user games. In the traditional multicast routing schemes that operate on the IP layer, the Internet is so large, and the topology is so widely spread that most routers on the trees only carry a small number of all the possible sources of the group, and most trees are sparse, (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 put 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.

3.2. Aggregating the Replicator Objects in *ns2*

The second technique we propose deals with multicast-related objects other than the multicast routing states. 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 that the tree passes through, *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 takes about 1.5K bytes memory. Using $R = 1.5K$ in the expression we have seen in Section 2.1, the total multicast memory requirement 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 requirement 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, much more tractable than 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 replica-

tors with each ⟨grp, src⟩ 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 ⟨grp, src⟩ pair on a network node. However as the simulations scale up, it may be more desirable to sacrifice this advantage in order to reduce the excessive memory requirement 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 takes 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 percent, which would not be a worthwhile effort in our estimation.

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 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 requirement 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* [14] 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 re–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.

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 set of Pentium-III 866MHZ systems 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.

We constructed the network topology in the simulation as follows: first we constructed a tree topology with a fanout

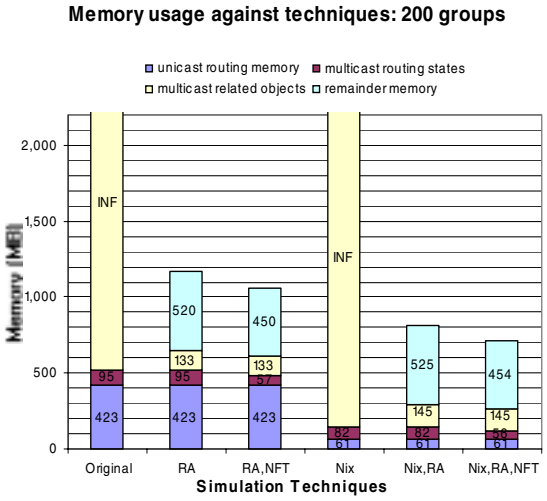

Figure 1: Simulation topology: fanout=6, depth=5, CrossLinkRate=100, with 200 multicast groups, each group having 100 senders and 200 receivers.

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 a connected graph. 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. 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. 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 1 shows the memory usage of the 6 different combinations of the 3 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 bottom up, 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* 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. 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 take 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 take 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 take 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 requirement 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 taken 17Gb memory if they are all

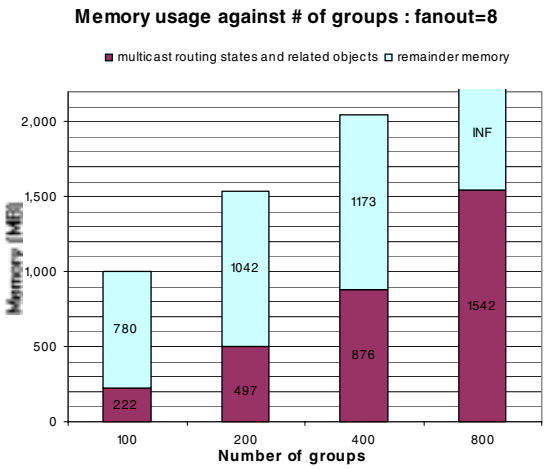

Figure 2: Simulation topology: depth=5, fanout=8, CrossLinkRate=100. All three techniques Nix, RA and NFT are applied. Each multicast group has 100 senders and 200 receivers.

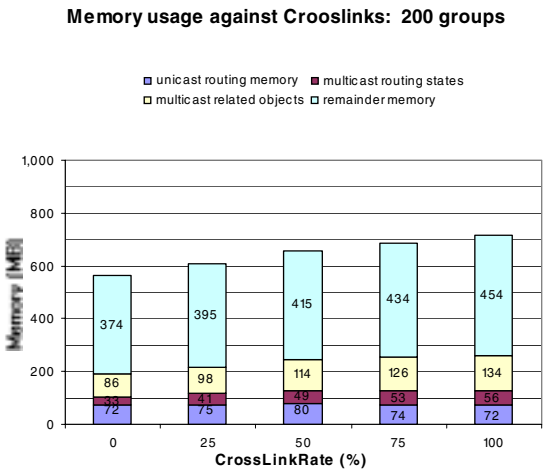

Figure 3: Simulation topology: fanout=6, depth=5, with 200 multicast groups, aach group having 100 senders and 200 receivers. All three techniques Nix, RA and NFT are applied.

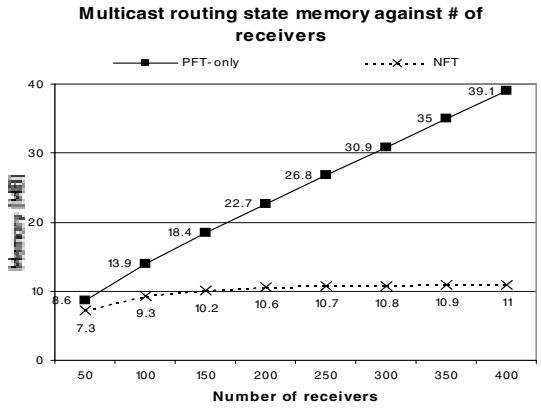

Figure 4: Simulation topology: fanout=8, depth=4, CrossLinkRate=100. Nix and RA are both applied. There are 100 multicast groups. Each multicast group has 100 senders.

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.

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 2 shows the memory usage as the number of groups increases, with all three techniques applied. This topology contains 4681 nodes, and simulation can only be carried out with the *NIx–Vector* technique. In this chart 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.

Figure 3 shows how the network connectivity impacts the effectiveness of the NFT technique. As the $CrossLinkRate$ increases, the multicast routing memory also increases. With the provided number of senders(100) and density of receivers (200/1555), when CrossLinkRate is 0%, the multicast routing states take about 33MB memory; when the CrossLinkRate is 100%, the multicast routing states take up about 56MB memory. This confirms our expectation that the more alternative paths the network has, the less effective NFT would be.

Figure 4 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 idealistic for End-System Multicast where the density is exactly 1.

5 Conclusions and Future Work

In this paper we analyzed the memory requirement 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 requirement by the replicator objects.
3. We remove the *unicast* routing table, so that simulations of large network topology do not run out of memory simply because of the $O(n^2)$ memory requirement 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.

As future work, we will attempt to distribute the multicast simulations onto multiple systems using *pdns*[15], which each system running a portion of the entire simulation, in order to achieve even larger multicast simulations.

Also, the NFT technique introduced in this paper can be used not only in simulations, but also potentially within actual multicast routers to reduce the routing state memory requirement of multi-source multicast groups. We plan to explore this possibility on actual multicast routers or Application Layer Multicast end systems.

References

[1] R. Bagrodia, R. Meyer, M. Takai, Y. Chen, X. Zeng, J. Martin, B. Park, and H. Song. Parsec: A parallel simulation environment for complex systems. *IEEE Computer*, 31(10):77–85, October 1998.

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

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

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

[5] S. E. Deering and D. R. Cheriton. Multicast routing in datagram internetworks and extended LANs. *ACM Transactions on Computer Systems*, 8(2):85–110, August 1994.

[6] P. Huang, D. Estrin, and J. Heideman. Enabling large-scale simulations: selective abstraction approach to the study of multicast protocols. In *Proceedings of the International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, July 1998.

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

[8] D. Nicol, M. Johnson, A. Yoshimura, and M. Goldsby. Ides: A java-based distributed simulation engine. In *Proceedings of the International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, July 1998.

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

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

[11] K. Perumalla, R. Fujimoto, and A. Ogielski. Ted - a language for modeling telecommunications networks. *Performance Evaluation Review*, 25(4), March 1998.

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

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

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

[15] G. F. Riley, R. M. Fujimoto, and M. H. Ammar. 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.

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

[17] S. Shi and M. Waldvogel. A rate-based end-to-end multicast congestion control protocol. In *Proceedings of ISCC 2000*, pages 678–686, Antibes, France, July 2000.

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

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

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