SIMULATING ROUTER ASSISTED OVERLAY MULTICAST

Thesis Project of

Nikolaos Bartsotas

Work performed in cooperation with Konstantinos Katsaros

Supervisor: George Xylomenos
Contents

1 Introduction ............................................................................................................4
  1.1 Application layer multicast ............................................................................4
  1.2 Router assisted overlay multicast ..............................................................6
2 Using the simulator ................................................................................................7
  2.1 ZipfGroupSizes distribution .........................................................................7
  2.2 Adding end hosts .........................................................................................7
  2.3 Measuring node stress .................................................................................8
  2.4 Measuring path stretch ...............................................................................12
  2.5 Measuring transmission stretch ....................................................................13
  2.6 Measuring Pastry signalling overhead .......................................................15
3 Code analysis .......................................................................................................16
  3.1 Proxy router Scribe test files ........................................................................16
  3.2 Modified files ...............................................................................................17
4 Simulation scenarios ............................................................................................18
  4.1 Topologies ....................................................................................................18
  4.2 Simulation Scenarios ...................................................................................21
  4.3 Scalability .....................................................................................................22
5 Running the simulations ......................................................................................25
6 Metrics produced ................................................................................................31
7 Future work ..........................................................................................................34
  7.1 Known Problems ..........................................................................................34
  7.2 Dynamic Application Behaviour .................................................................34
8 Acknowledgements ..............................................................................................38
9 Appendix: OverSim .............................................................................................39
  9.1 Direct Message Sending ..............................................................................39
  9.2 Dynamic Module Creation ...........................................................................39
  9.3 Scalars ..........................................................................................................40
  9.4 Other Capabilities of the Platform ...............................................................41
10 References ...........................................................................................................42
1 Introduction

While multicasting is considered valuable for content distribution, it is not widely supported on the Internet. Content providers have instead turned to peer assisted content distribution in order to efficiently serve large numbers of clients via unicast, thus removing the bandwidth bottleneck from their side. The redundant unicast transmissions of the same packet are not avoided however, they are just distributed between the peers. Since peer assisted content distribution has grown to represent a major fraction of total Internet traffic, schemes for reducing it would be of great interest to users and network operators alike. For this reason, we reconsider overlay multicast as a potential solution for mass content distribution. We present an overlay multicast scheme inspired by Scribe that exploits co-operative access routers so as to improve the multicast content distribution trees produced. We investigate the properties of our scheme compared to both regular Scribe and IP multicast by focusing on a full fledged simulation platform that can be used as a basis for the realistic evaluation of content distribution applications. Special attention has been paid to the scalability of the simulation environment, that is, its memory and processing time requirements, when using Internet like network topologies.

1.1 Application layer multicast

It has been long realized that the Internet is evolving from a network connecting pairs of end hosts to a substrate for information dissemination. Indeed, a major part of today’s Internet traffic is due to content distribution applications, in most cases peer assisted applications [8]. Peer assisted, or peer to peer, content distribution has been regarded as a very efficient method both for content providers (in terms of the required bandwidth) and the end users (in terms of download times). However, in terms of network resources this approach can be very inefficient: many nearby nodes may download the same data from a faraway node instead of from one another, since they make their choices independently [8]. The root cause of this problem is that the Internet lacks a multicast facility that could be used for efficient content distribution.

While IP multicast has been available for more than a decade, it has not been widely adopted. One reason for this is that IP routing does not scale well: unlike unicast addresses that can be aggregated in a single routing entry per network area, nearly identical multicast addresses may be referring to completely different member sets, therefore routers must allocate memory and perform separate signaling per group [14]. Another reason is that there is no mechanism for global multicast address allocation, therefore, applications may face address collisions [6]. Finally, there are no clear incentives for routers to provide multicast support. The lack of adoption of IP multicast by network providers has led to the development of a variety of alternative end host based solutions, which require no assistance from the network. One option is to employ an Application Layer Multicast (ALM) scheme [5], where multicast is simulated by multiple unicast transmissions between group members: the sender transmits packets to some members, those members relay them to others, and so on, until all members are served. In most ALM schemes however each group member needs to be aware of all other members to achieve good routing performance, meaning that these solutions are not scalable. A more scalable option is to create multicast trees over a Distributed Hash Table (DHT) substrate, such as Pastry [10],
that can route packets based on identifiers. One such system is Scribe [3] which uses the identifier based routing of a DHT substrate to provide a rendezvous (RV) point between senders and receivers to a group in a distributed manner. While Scribe seems promising, its performance is quite worse than that of IP multicast [4]; Our work showed that it is even worse when taking into account the entire path between the source and the receivers. In addition, it is hard to simulate realistic applications on top of Scribe so as to determine application level performance, since existing simulators either sacrifice realism for scalability [3, 4] or scalability for realism.

The reliance of Scribe on end hosts may however lead to inefficiencies. An end host that is an interior node in some trees will limit the bandwidth available to all those trees to that of its access link. This can be avoided by exploiting the properties of the underlying DHT to create a set of trees such that each node will be an interior node for only one of them [2]. However, this solution is tied to a specific overlay routing scheme (in this case, Pastry). In addition, an end host that is an interior node even for a single tree, may still be a bottleneck: as shown in Figure 1(a), data in transit has to enter and exit the RV point (peer \( b \)) and the other two internal end nodes \( c \) and \( d \), only one of which (peer \( c \)) is also a receiver, via their access links; if the access links are asymmetric, the tree bandwidth will be limited by the, typically lower, uplink bandwidth. Finally, neighboring end hosts may download the same content via separate tree branches, thus incurring unnecessary network load. For example, in Figure 1(a) the two peers \( e \) and \( f \) receive separately the content from their parent in the tree (peer \( d \)). Note that in this example peers \( b \) and \( d \) act as intermediate tree nodes without being receivers; this case arises when nodes participating in various multicast groups share the same DHT substrate, so as to amortize DHT maintenance costs among different groups and improve DHT routing performance by increasing the available overlay paths.

Figure 1: Overlay multicast (a) without router assistance and (b) with router assistance.
1.2 Router assisted overlay multicast

To avoid these problems, we propose using the access router of a peer as its proxy in the DHT substrate and overlay multicast scheme. This means that the access router participates in the DHT on behalf of the attached peer. If multiple peers are attached to the same access router, a single place will be held by it in the DHT, that is, the access router will be assigned a single, uniform part of the identifier space, regardless of the number of directly attached peers. On the other hand, an access router will not enter the DHT and act as a proxy unless at least one of its attached end hosts is a peer. In this manner, access routers are not burdened with the signaling overhead of maintaining the DHT unless there is a reason to do so. In addition, the access router also acts as a proxy for the peer in the Scribe trees. This means that the access router is responsible for joining the multicast groups indicated by the attached peers and forwarding the incoming traffic to them. The access router may also participate in a multicast tree as an interior node, subject to its position in the identifier space and the operation of regular Scribe, that is, if in regular Scribe its attached hosts were interior nodes of that tree. In this case, it forwards the incoming traffic to its tree descendant and to any interested attached peers.

The proposed role of access routers presents some significant advantages regarding the characteristics of the created distribution trees. First, as shown in Figure 1(b), data do not need to cross the access links of interior tree nodes at all, only crossing the, typically faster, downlink direction of those access links leading to nodes that are members of the group. For example, data will not cross the access link between peer d and router 5 at all, and it will only cross the access link between peer c and router 4 in the downlink direction so as to deliver the content to peer c. Second, multiple tree branches towards end hosts attached to the same access router can be aggregated in a single branch leading to that access router (in our example router 7). For the entire distribution tree of Figure 1, router assistance means that a packet transmitted to the group will only cross 12 instead of 20 links with regular Scribe (or 8 with an optimal IP multicast tree), avoiding the uplink direction of access links. Therefore, in router assisted overlay multicast the paths through the distribution trees become shorter and faster, while redundant transmissions over the access links towards intermediate nodes are prevented, something especially important in an environment where asymmetric access links are prevalent. Considering that overlay multicast is a response to the lack of support for IP multicast, proposing that access routers should undertake the task of maintaining an overlay multicast routing scheme on top of a DHT substrate seems counter intuitive. However, while in IP multicast all routers must participate in multicast routing, in our scheme access routers can optionally take part in DHT maintenance and overlay multicast routing: the DHT substrate and the overlay multicast scheme can operate without router assistance, albeit with reduced efficiency. Also, unlike in IP multicast where many routers have no incentive to participate in multicast routing, in our scheme the access routers have very clear incentives to act as proxies for their attached end hosts. First, they reduce their traffic load, since they eliminate transmissions over both their access links (for routers 2 and 5 in Figure 1) and their router-to-router links (for routers 5 and 7 in Figure 1). Second, they provide an enhanced service to their customers, as the end hosts will experience lower latencies and higher bandwidths in multicast applications such as content distribution. Note that these are essentially the same arguments that motivated many network providers to offer Web and other application proxy services to their clients.
2 Using the simulator

2.1 ZipfGroupSizes distribution

In order to start the simulation, we must first create topologies via GT-ITM. The topologies simulated are described in Table 1. Each topology consists of a set of transit domains containing transit routers. Each transit router is connected to a set of stub domains, and each stub domain contains stub routers. After building the topology, our BRITE patch can be used to transform the file to the proper format, in order for the routing policy weights to be taken into account by OMNeT++. Note that we must ensure that the .ned file of the topology is included in the nedfilelist

<table>
<thead>
<tr>
<th></th>
<th>Topo-0</th>
<th>Topo-1</th>
<th>Topo-2</th>
<th>Topo-3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transit domains</td>
<td>5</td>
<td>7</td>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td>Avg. routers per transit domain</td>
<td>5</td>
<td>4</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>Stub domains per transit router</td>
<td>5</td>
<td>7</td>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td>Avg. routers per stub domain</td>
<td>5</td>
<td>7</td>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td>Stub routers</td>
<td>625</td>
<td>1372</td>
<td>2916</td>
<td>5000</td>
</tr>
<tr>
<td>Transit routers</td>
<td>25</td>
<td>38</td>
<td>36</td>
<td>50</td>
</tr>
<tr>
<td>Total routers</td>
<td>650</td>
<td>1400</td>
<td>2952</td>
<td>5050</td>
</tr>
</tbody>
</table>

Table 1: Details of simulated topologies.

The ZipfGroupSizes.cc class determines which node will participate in which group. Specifically, a Zipf-Like distribution is used where \( r \)-th group has a size equal to \( \left\lfloor Nr^{-2.5} + 0.5 \right\rfloor \), where \( N \) is the total number of overlay nodes, as in [3]. The maximum number of members for a group is \( N \) and the minimum is 11, a size typical of instant messaging applications [3]. Based on this lower bound and the number of participating hosts, the number of groups in the Sparse scenario (500 hosts) is 21, in the Medium scenario (1000 hosts) it is 36 and in the Dense scenario (4000 hosts) it is 110. The members of each group were randomly selected from the entire population of end hosts, meaning that each end host may have participated in multiple groups. For each group, a random identifier was chosen and a non-member end host was randomly selected as the sender to the tree.

2.2 Adding end hosts

After the initialization of the topology, we can start joining the end hosts in the overlay. The initialization is finished when the file which contains the topology has been loaded and the topology has been built, that is, the modules of all routers have been loaded and all network connections have been made between the routers and the backbone routers. How the end hosts will join groups is defined in the class ZipfGroupSizes which is explained in the previous section. This class uses the distribution described in [1] and it provides to MultiScribeTest the information required in order to add the node to a specific group.

---

1 The nedfilelist file is a file which contains all the .ned files that Oversim is going to need. The path to a new .ned file must be inserted in that file either by hand or executing a makemake command for Oversim. Note that if you put it by hand, the path to the file must be correct. It is needed because Oversim loads all the .ned files before running any simulation scenario.
When an end host tries to join the overlay there are two possibilities: either the access router for that host has not joined the overlay, or it has already joined the overlay. In the first case, when the host tries to join the overlay the IPV4UnderlayConfigurator module starts loading the overlay modules of the corresponding access router, using dynamic module loading. After the initialization of the router overlay modules, the IPV4UnderlayConfigurator informs the ProxyRouterScribeTestModule of the router that a node is trying to connect to it, by sending the TransportAddress of the attaching node with a message via a direct_in_port. In order to start joining nodes with the MultiscribeTest.cc class, we have ensured that the overlay is ready. This is implemented by ensuring that a TierReadyMessage has arrived before any joins are made by the router to any group on behalf of any node. We must ensure that the overlay is ready before making the joins, because the following scenario may occur: if an end host has not joined the overlay then it is possible for one end host to be the RV point at the beginning of the initialization phase and another one to become the RV point later on as new hosts join. In other words, we must be sure that the RV point is the final RV point of the overlay tree and that it will not change. In the second case, if the router has already joined the overlay the IPV4UnderlayConfigurator just sends to the router the TransportAddress of the attaching node.

In the proxy based scenario, we do not build the modules of the end hosts, because after sending the message to the router, there is no use for them. This reduces the active modules that run during the simulation, and saves as much memory as possible. Because the simulation stops when all the end hosts have joined all the designated groups and the access routers or the end hosts cannot migrate or disconnect, there are no side effects to the measurements as long as we take this into account. If someone wants to modify the code, this must be considered. In the no proxy scenario, this is not the case as the end hosts run the overlay modules themselves. In this scenario, the routers do not run the overlay modules and no messages are sent from the IPV4UnderlayConfigurator to the MultiScribeTest class. It is the normal scenario for Pastry and Scribe.

2.3 Measuring node stress

An important metric of a content distribution system’s performance is the forwarding load it imposes on participating nodes. In any multicast scheme forwarding load can be expressed by the branching factor of each node, that is, the number of descendants it forwards traffic to; in a multi-tree scenario, attention must also be paid to the number of groups served by a node. In Scribe each node maintains forwarding information in a separate children table per group, with each table’s entries referring to the children of the node for that specific group. Hence, we calculate two metrics of node stress, as in [3, 4]. The first metric is the number of children tables maintained by each node, which corresponds to the number of multicast groups the node is forwarding traffic for. In the case of our router assisted scheme, this number includes the number of groups for which a router is not forwarding data to other overlay nodes, but only acts as a receiver on behalf of one or more attached nodes. The second metric is the number of children entries maintained by each node, which corresponds to the

---

2 The overlay is ready when all the overlay modules are running. In our case, Pastry should be ready. By saying ready we mean that there are no cached State messages, that all the entries of Pastry tables have been determined and the proximity metrics has been received.
number of nodes the node is forwarding traffic to, across all multicast groups. Again, in our scheme this number includes the number of group recipients directly attached to an access router, as the access router is forwarding traffic towards them.

In the Non Proxy Based scenario the Node Stress is defined by the number of children tables maintained by each node and the number of children entries maintained by each node. In the Proxy Based scenario we add to the first metric the number of groups for which the router is not a forwarder but a receiver and to the second metric the sum of the number of attached end hosts nodes belonging to all of these groups.

After all end hosts have been attached to routers, routers have joined the groups and have sent a message to the RV point so as to determine which node it is, we can start measuring. In order to measure Node Stress we have altered Scribe’s source code. The MultiScribeTest module of every router sends a message to Scribe to start recording the Node Stress that the particular Scribe Node has in its children tables. Since Node Stress has to do only with the overlay, there is no difference in measurements whether we have the routers or the end hosts join the overlay. There is only a change to the definition which is being taken into account in the measurements. This change is due to the fact that we have assumed that the overlay routers operate in the overlay on behalf of the end hosts.

![Figure 2: Node stress: children tables.](image-url)
Figures 2 and 3 depict the two node stress metrics achieved for the Sparse, Medium and Dense scenarios in Topo-1, aggregated over all nodes in the network; results from other topologies indicate that both metrics are mostly dependent on end host density. The proposed router assisted scheme incurs a significant forwarding state overhead increase, especially considering that this state is concentrated on the access routers serving end hosts participating in the overlay rather than being distributed among the participating end hosts as in regular Scribe. This increase however is largely an artifact of our definition of the node stress metrics. For example, an end host that was a leaf for a multicast tree in regular Scribe, would only require an entry in the end host serving as its parent, while in our scheme it requires an entry both in the access router serving as a proxy for its parent, as well as in the access router that is acting as a proxy for itself. Note however that these leaf entries do not require the same maintenance overhead as regular Scribe entries.
In order to assess how the amount of forwarding state affects the nodes participating in the overlay multicast routing process, Figures 4 and 5 show the number of overlay nodes per tree, as a function of the number of end hosts participating in the DHT substrate, either directly or via their access routers; for clarity, the figures include only the values for the ten most popular groups in each scenario. As expected, with more end hosts (Dense) and fewer access routers (Topo-1), Figure 4 shows that the forwarding load in the router assisted approach is concentrated on fewer nodes than in regular Scribe, while with fewer end hosts (Sparse) and more access routers (Topo-3),
Figure 5 shows that the forwarding load of the router assisted approach is distributed in roughly the same manner as in regular Scribe.

2.4 Measuring path stretch

The most obvious metric for evaluating overlay routing schemes is the overhead incurred by not strictly following the shortest path IP routes. This is usually expressed by the term *stretch* which is defined as the ratio between the overlay path length (or delay) and the length (or delay) achieved by IP routing. In essence, *stretch* expresses the degree to which the performance achieved is worse than the performance of the underlying IP substrate, a penalty paid in return for the scalability and robustness of the overlay solution. Multicast complicates this metric, not only because we need to take into account the entire distribution tree, but also because the Internet does not provide multicast routing in the first place. The usual convention, also followed in this thesis, is that we use as a substitute for IP multicast the tree that would be formed by merging the optimal unicast paths between the sender and each receiver. In reality however, a sender desiring to reach all receivers would have to send duplicate copies of the same message over all those paths.

![Figure 6: Path Stretch](image)

In this thesis we define *path stretch* as the ratio between the number of IP hops comprising the path from the sender to a single receiver in the multicast overlay tree to the number of hops that comprise the shortest unicast path between these two nodes, averaged over all trees and paths. The overlay paths are comprised of two segments: from sender to RV point, and from RV point to receiver. For the first segment we use the shortest path between these two nodes, since the sender can cache the IP address of the RV point, which is discovered in the message sent to initially locate the RV point for the group, thus avoiding the overlay path for all subsequent data transmissions. For the second segment we use the path constructed by the overlay scheme. Figure 6 shows the path stretch achieved for the Sparse, Medium and Dense.
scenarios in Topo-1; by definition the stretch of IP multicast is 1. It is clear that our approach yields significantly lower stretch values in all scenarios (40-45% decrease), regardless of end host density.

It should be pointed out that our results diverge from those shown in the papers evaluating Scribe [3, 4] in two ways. First, we measure the entire path between the sender and each receiver, as opposed to the path between the RV point and each receiver, since the source to RV point segment may be quite long, even up to the diameter of the network, and it is the entire path length that applications have to contend with. Second, we use network hops rather than delays to calculate stretch, since in a static environment we can only calculate the propagation delay and, if we assume a specific packet size, the transmission delay of each path. Since these delays may only represent a small part of the actual delays faced by applications in a dynamic environment where queueing delays due to congestion may dominate overall delays, we felt that defining stretch using static delays would be misleading.

In order to measure Path Stretch we are using a recursive DFS algorithm. First, we ensure that the initialization of the overlay has finished and that all the trees are fully formed and stabilized. Then we use the topology extraction tool in order to derive a sub-topology of the main topology which includes all routers and end hosts of the group. Then we start from the RV point of every group and traverse towards each leaf of the tree. During the recursion and until we reach a leaf, we count the underlay hops in order to calculate the hops that a message crosses to reach the leaf node. After reaching the leaf we add to a scalar statistic the underlay hops until the node plus one; this takes into account the link between the final router and the end host. After finishing the recursion we print the scalar statistic for the group which provides the average, the maximum, the minimum values and other statistics. This is performed for every group that has been constructed. The underlay hops from the sender to the RV point are added when making the plots; they are recorded in a separate metric file.

Second, we measure the paths from the sender to the receivers for the IP multicast case. In that case for every leaf node we use the weighted shortest path algorithm, available from the OMNeT++ forum, from the leaf node to the sender of the group and put the paths in a vector. The path vector consists of the routers that a message will cross to reach from the sender to the receiver. Then every path vector is added to a group vector which consists of all the path vectors of the group. It must be noted that in the vector path are included both the sender and the receivers. To record the path we use again the scalar tool that provided by OMNeT++. We record for each tree the length of the Sender-Receiver path. It must be noted that the path in the proxy based scenario is the path vector size plus one and in the non-proxy scenario is the path vector size minus one. This is because each path vector includes all the routers participating in the current path. But when we translate this into links for the proxy based scenario, the number of the links is the number of the routers plus the link for the end host; the opposite is the case for the non-proxy scenario.

2.5 Measuring transmission stretch

Another important performance metric is the efficiency of the multicast trees produced in terms of the total load imposed on the network for data distribution. We define as the total transmission load the number of hop-by-hop transmissions required
in order for a single packet originating from the sender of each group to reach all
group members of the corresponding group. In order to provide a normalized metric,
we define *transmission stretch* as the ratio between the total transmission loads
imposed by each overlay multicast scheme to the total transmission load imposed by
IP multicast, where the IP multicast trees are formed as described above.

![Figure 7: Transmission Stretch](image)

Figure 7 shows the transmission stretch achieved for the Sparse, Medium and Dense
scenarios in Topo-1; by definition the stretch of IP multicast is again 1. Exploiting
router assistance results in a decrease of the transmitted packets in all cases (7-19% decrease).
This is due to the fact that the proposed router assisted multicast overlay
scheme leads to the formation of multicast trees with fewer IP hops compared to the
regular Scribe trees, leading to a more efficient utilization of the network. Recall also
that, as mentioned above, our router assisted scheme eliminates packet transmissions
in the uplink direction of access links leading to intermediate nodes, thus removing a
bottleneck imposed by asymmetric access links.

The transmission stretch for the overlay is measured via a global variable. After
running the ScribeTreeDFS function, we record the value to a scalar. For the IP
multicast case, we put the elements of every path vector of the group in a set. After
inserting all the routers to the set for the proxy based scenario the transmission stretch
is the set size plus the number of end hosts that participate in the current group. In the
non-proxy based scenario it is the set size minus one. This metric is calculated in this
way because in the proxy scenario it is not possible to measure the links between the
leaf router and the end host, as the end hosts simply do not exist; we know however
that we must take them into account by adding their links to the metric. The link
between the end host and the sender does not need to be added, as the sender is
counted in addition to the router, but the router to receiver links must be added. These
are equal to the number of end hosts that participated in the overlay. In the non-proxy
scenario the size of the set is one higher so we subtract one.
2.6 Measuring Pastry signaling overhead

As mentioned on the section of NodeStress, the proposed router assisted scheme incurs a significant forwarding state overhead increase. However there is a notable reduction to number of the overlay participants. This reduction leads to a corresponding reduction in the number of overlay signaling messages exchanged to establish the DHT substrate (Pastry). As demonstrated in Figure 8 for the Sparse, Medium and Dense scenarios in Topo-1, the router assisted approach greatly reduces (by 45%) the signaling overhead for DHT maintenance in the Dense scenario, as in this case the access routers participate in the overlay on behalf of many end hosts. On the other hand, in the Sparse scenario each router hardly ever serves more than one end host, thus we see only a slight reduction (4%); in the Medium scenario the reduction lies between these extremes (25%). Note also that in our router assisted scheme these messages travel shorter distances since they do not need to cross the access links to and from the end hosts. Therefore, there exists a tradeoff between placing the burden of forwarding and tree maintenance on the access routers on the one hand and reducing path stretch, link stress and DHT related signaling on the other hand.

![Pastry Signaling Overhead](image)

Figure 8: Pastry Signaling Overhead

The Pastry signaling load is provided by the Pastry implementation and is recorded to the scalar file. As we found by studying the code, every Pastry message includes a Boolean variable which defines whether this message is a signaling message (JOIN, STATE, …) or a message from the upper layers. Thus the system knows when it has a message for Pastry and records it.
3 Code analysis

In order to make measurements, we had to write code to manage which node will join which group and to make routers join the overlay on behalf of the end hosts. In order to implement this functionality we had to make small modifications in some classes. Our aim was to make the least modifications possible, in order to have fewer side effects to the rest of the OverSim’s functionality. All our classes are in the ProxyRouterScribeTest directory. Their functionality will be analyzed next.

3.1 Proxy router Scribe test files

MultiScribeTest: This class makes the joins and defines when initialization has finished so that we may start the measurements. In order to avoid mass joining, we have introduced some delays to the joining of the groups. This class ensures that before we start joining the trees the overlay is ready, by waiting for the Tier Ready Message. The initialization is done when all joins are made and all RV points have been discovered. Senders discover RV points by having their MultiScribeTest module send a data message to them. The RV point must be discovered in order for the measurements to be made as it is provided by the senders of the group when measuring the paths between the RV point and the receivers.

compareWithMultiScribeTest: This class has the same functionality as MultiScribeTest but it is used for the non proxy scenario whereas the MultiScribeTest is for the proxy based scenario.

ProxyRouterScribeTest: This is the class for the messages used for our measurements. There are three types of message:

- **MultiScribeTestNotification**: This is sent by the IPV4UnderlayConfigurator to MultiScribeTest, when an end host is trying to connect to a specific router. When this message is received by MultiScribeTest, it is the signal to start the joins of the nodes to the groups according to the Zipf-like distribution.

- **MultiScribeTestJoinGroupNotification**: This message is triggered in order to have random joining and not a straight join of the groups.

- **MultiScribeTestSendDataNotification**: This message triggers a send data message to Scribe.

ZipfGroupSizes: This class implements the Zipf-like distribution of end hosts to groups. It calculates how many end hosts each group will have, following a Zipf-like distribution. It also implements the Path Stretch and Transmission Stretch measurements. This class has two methods that traverse the trees created and collect the values required to calculate the above metrics. It works with either the Proxy based scenario or with the non Proxy scenario, as long as the omnetpp.ini file has the proper values. The two measurement methods are called measureStretch and measureStrechEndHostsOnly. There are also two more functions, ScribeTreeDFS traverses the overlay tree and collects metrics and ShortestPathIPMulticastHops traverses the IP Multicast tree and collects the same metrics as ScribeTreeDFS in order to contrast the results. The only difference is that in ScribeTreeDFS we separately measure the RV to Sender path and the other distances. In the
ShortestPathIPMulticastHops we measure the Sender to Receiver paths directly, as there is no RV point in the IP Multicast case.

### 3.2 Modified files

**IPV4UnderlayConfigurator:** In the IPV4UnderlayConfigurator class some changes were made in order to support the proxy based functionality. More specifically, we have the addRouterToOverlay method which is called when an end host is trying to connect to the router. This takes place when we call the create method in the IPV4UnderlayConfigurator class. This method is called when a new end host tries to connect and from this place it calls our method. We also modified it so that when we are running the proxy based scenarios, the Overlay Host module will not even be created, as mentioned before. The addRouterToOverlay method first checks whether the router is already in the overlay or not. If it is not, it starts the overlay modules of the custom router. If it is already in the overlay, it does not do anything. The second thing that this method does is to send a MultiScribeTestNotification message to the MultiScribeTest class, since a new end host is connecting to the router.

**GlobalParameters:** In globalParameters we added some global counters and access methods to the IPV4UnderlayConfigurator in order to have access to them from our measurement application. These counters help MultiscribeTest to find out when the initialization face has finished and we are ready to start measuring.

**GlobalStatistics:** In this class we added some scalar variables in order to print out our metrics using the scalar statistic capabilities that OMNeT++ offers. If someone wants to get a pointer to the GlobalParameters or GlobalStatistics classes, he can use their access classes: their get methods return a pointer which can be used to access all the variables and methods of each class.

**Scribe:** In the Scribe.cc class we have added the method measureNodeStress. This method exploits the GroupList map which contains all the information necessary to calculate the node stress, either for the proxy based or the non proxy scenario. The GroupList is a map which has as a key the OverlayKey of the Group and as a second argument a ScribeGroup object. This object maintains all the children entries each ScribeNode has. Thus the size of the GroupList gives the number of Children Tables and the method numChildren of the object ScribeGroup gives the children entries of the table. This is correct for the non proxy scenario. In the proxy scenario, there is a small modification to that definition: we add to the children entries the sum of the number of groups that each directly attached node has joined and to the children tables the number of groups for which a router is not a forwarder but a receiver. Both are provided by MultiScribeTest with the ALMInitNodeStressMeasurementMessage which has two variables. The first variable depicts the total number of end hosts connected to the router and have joined different groups and the second one is an array with all the group joins that the router has made on behalf of the end hosts.
4 Simulation scenarios

The set of scenarios selected was based on the JSAC paper [1] in order to be able to contrast these results with ours. Another factor that we took into account was the memory limitations that have been opposed. The System that ran the simulations had an Intel® Core™2 Duo CPU (P9500 @ 2.53 GHz), 4 GB of RAM, and ran Ubuntu Linux as a VMWare image with 2GB of RAM (the host ran Windows Vista 32 bit).

4.1 Topologies

Our initial concern in building a simulation environment relates to the underlying network topology. The SimpleUnderlay model provides a scalable routing substrate since no network protocols are actually in operation. In this model, packets are directly sent to end hosts by simply using a global routing table, with packet delivery delay being determined by the two communicating ends’ distance in the Euclidean space. Furthermore, each end host can be assigned to a logical access network for which the access delay, bandwidth and packet loss characteristics may be set. Though it has been shown that this model provides a scalable solution for the simulation of large numbers of overlay nodes [1], it suffers from serious limitations: the lack of protocol functionality and step-by-step routing are major drawbacks of the model, since important aspects of a real system are neglected, such as the queuing of packets in intermediate nodes (routers) along a path and therefore the packet delays, and even losses, that arise due to network congestion. As a result, this model cannot be used to evaluate the dynamic performance properties of content distribution applications.

On the other hand, the IPv4Underlay model provides a good approximation of real networking conditions by incorporating the operation of almost all widely deployed networking protocols. We have therefore focused on using this model for our work, exploring its memory and processing time requirements for the simulation of very large network topologies. Apart from scalability, the other major drawback of this model is that it lacks support for routing policy weights, such as those produced by the Georgia Tech Internet Topology Model (GT-ITM) [13] that has been used for previous studies of Scribe performance [3, 4]. Instead, the model employs an unweighted shortest path algorithm (Dijkstra’s) which calculates the shortest paths between any pair of network nodes, regardless of their placement on the network. Hence, in contrast with reality, it is possible for a path between two routers in a single stub network to pass through several transit routers, something very likely to influence the results produced, especially when it comes to routing issues. By not taking routing policy weights into account, routing paths may become shorter than in reality, and since Pastry employs proximity metrics in the selection of overlay neighbors, this could result in the selection of the wrong node as an overlay neighbor.

In order to avoid routing inaccuracies, we constructed a conversion tool that allows the use of GT-ITM topologies within the OverSim platform. Our tool was implemented as an extension to the BRITE topology generator export tool [12] which already allows the parsing of GT-ITM topologies and the conversion of BRITE topologies into OMNeT++ format. However, the existing tool did not provide any support for weighted topologies, nor did it make a distinction between transit and stub routers, producing flat topologies. We solved these problems by piggybacking the routing weights inside the channel definition of the produced OMNeT++ topology,
using fields whose values are not provided by the GTITM model, but are instead later read from configuration files. Furthermore, we incorporated the distinction between transit and stub routers in the tool and translated it into IPv4Underlay’s distinction between backbone and access routers. The support for routing policy weights was completed by employing a weighted shortest path algorithm, leading to a platform that captures all the intricacies of GT-ITM and is therefore comparable to earlier simulation studies that are based on that topology generator.

We next provide the README file which explains how the topologies are generated and imported to OverSim.

importing gt-itm t/s topologies in omnet++ (compatible with oversim ipv4underlay)

// copyright 2008 Konstantinos V. Katsaros
// ntinos@aueb.gr
// http://mm.aueb.gr/~katsaros

(1) install GT-ITM topology generator.
(http://www.cc.gatech.edu/projects/gtitm/gt-itm/gt-itm-linux.tar.gz)

(a) Generate a T-S topology as described in gt-itm’s docs and convert it to the alternate format (ALT) (using "bin/sgb2alt").

(b) Change the produced .alt file extension to ".gtts".

(2) Install BRITE (http://www.cs.bu.edu/brite/download.html)

(3) Fix the CLASSPATH (see http://www.omnetpp.org/forum/viewtopic.php?forum=12&showtopic=189) to include the BRITE/Java directory. This would be more convenient to be made in .bashrc/.profile so that we do not have to care about it every time we want to run BRITE.


(5) What BRITE actually does depends on the provided .conf file (see conf/psirp.gtts_import.conf example .conf file). The converter has been also extended to produce topologies in which a certain percentage of the routers also participates in the overlay. (This is accomplished with the new "OverlayParticipation" parameter). (The provided GUI seems not to function properly so we shall use the command line.) However, as is, the Omnet++ BRITE extension seems to flatten the hierarchical Transit-Stub Model (no distinction between transit-stub router, no link weights). So we must replace the following files we the ones provided and build again.

- "BRITE/Java/Import/GTTSImport.java" (parses the ".gtts" file)

- "BRITE/Java/Export/OmnetppExport.java" (concerts to the desired format and writes it to a .ned file)
- "BRITE/Java/Graph/RouterEdgeConf.java" (now keeps the link weight too)

- "BRITE/Java/Model/FileModel.java" (for retrieving the OverlayParticipation parameter from the .conf file)

- "BRITE/Java/Main/ParseConfFile.java" (for retrieving the OverlayParticipation parameter from the .conf file)

(5) We run BRITE with following command:


This should produce the file GTITM_TS_IPv4Underlay.ned under the:

OverSim-20080416/Underlay/IPv4Underlay/GTITM_TS_IPv4Underlay
directory. Note that the location of the ".gtts" file is indicated in the .conf file!

(6) We must add this .NED file to our nedfiles list (i.e. OverSim-20080416/nedfiles.lst) so that Oversim is aware of it.

(7) In the comments at the beginning of the produced NED file, there are several parameters that should be provided to Omnet++/Oversim so that the produced topology can be loaded. These are actually the parameter values that should be placed in the omnetpp.ini file. All you have to do is simply copy and paste them there.

(8) Furthermore, we have to replace the following files in Omnet++ and Oversim (and re-build):

- "/OverSim-20080416/Underlay/IPv4Underlay/IPv4UnderlayConfigurator.cc"

- "/omnetpp-3.3/src/sim/ctopo.cc"

- "/omnetpp-3.3/include/ctopo.h"

The latter ctopo.* files include Pasquale Gurzi's implementation of Weighted Shortest path algorithm, not implemented in the current release of Omnet++ (omnetpp-3.3)(see http://www.omnetpp.org/listarchive/msg10970.php).

They also contain code for retrieving the weight values from the produced NED file.

How things work:

Currently, there is provision for placing the weight values on the nodes, which seems not so appropriate since these weights refer to a link. So, at first, we have to pass this information to our channel (omnet-wise...).

The problem is that our topology generation is performed off-line (i.e. not while initiating the simulation), and in effect the weights
must be inside the NED file. However, the standard channel definition allows only for the error, delay, and rate attributes to be set in the NED file i.e. no weight attribute. Hence, until omnet++ allows for specifying additional channel attributes in the NED file, we pass the weight information in place of "error" attribute, that is the error attribute has the weight value. Note: it is possible to add an attribute to a channel once the simulator is initializing, however its value has to be retrieved from the NED file!

We have added code to "ctopo.cc" that, when the simulation begins, it dynamically creates a weight attribute for each channel and passes the "error" value specified in the NED file. Next, in order to fix the error value, we have added appropriate parameters in our omnetpp.ini file that are used to pass the correct values. We have provisioned for different values depending on whether we have a transit-transit, transit-stub or stub-stub link.

The attributes expected by our code are:

```
# GT-ITM TS link attributes
**.ttDelay = 0.0018
**.ttError = 1e-6
**.ttDatarate = 128000
**.tsDelay = 0.0017
**.tsError = 1e-7
**.tsDatarate = 17000
**.ssDelay = 0.0016
**.ssError = 1e-8
**.ssDatarate = 126000
```

(9) Good luck!

NOTE:

In this image, everything has been install and running properly. In order to produce a topology just run the

"/opt/make_topology.sh" script.

### 4.2 Simulation scenarios

We have already described the parameters of the topologies that we have used in Table 1. In this section we explain why these parameters were chosen. Apart from the runs we made, so as to produce the results that are being discussed in the main thesis, there was a number of other runs we made, in order to investigate the scaling properties of the platform and to produce a number of results that are available for
anyone who wants to use them, without running the simulations. The largest topology has the same size as the JSAC paper, and the other topologies are smaller versions of it. Specifically, the topologies simulated were:

- **650 Routers Small JSAC**
  - 1000 End Hosts
- **1400 Routers Quarter JSAC**
  - 1000 End Hosts
- **2952 Routers Half JSAC**
  - 500 End Hosts
  - 1000 End Hosts
  - 4000 End Hosts
- **5050 Routers JSAC**
  - 500 End Hosts
  - 1000 End Hosts

The above runs have been simulated for five different topologies for each router scenario, which were all produced from the GT-ITM topology creator.

### 4.3 Scalability

As shown in [1], OverSim enables the simulation of scenarios with even 100,000 overlay nodes. However, in the simulations presented in that paper, the underlying network was either too simplistic (*SimpleUnderlay*) or too small (*IPv4Underlay* with 20 backbone and 20 access routers). In [1] it is claimed that the *IPv4Underlay* network requires approximately 70KB per node, which is twice the per node memory requirement of the *SimpleUnderlay* network, due to the complexity of the model. However, this does not clarify the actual memory footprint of each distinct node type. If the same number of overlay end hosts is used with both models, then the memory requirements of the *IPv4Underlay* model will also be larger due to the presence of additional network elements (routers and links) and their operation. It is also expected that the memory of a router will differ from that of an overlay end host.

In this thesis we investigate the simulator memory requirements for the considerably larger network sizes described in Table 1. Figure 9 shows the memory footprint of these topologies: the x-axis presents the size of the network in terms of the total number of participating routers, while the curves indicate the memory footprint of scenarios either without any end hosts or with 1000 end hosts in the proposed router assisted (*Proxies*) or in the regular Scribe scheme (*No Proxies*). It is clear that the memory footprint of the networking topology increases drastically with the number of participating routers, in a non-linear fashion. This increase is due to the increasing number of links between the participating routers.
The memory requirements of the network topologies have a severe impact in the feasibility of large scale scenarios, since, for example, a topology with 5000 access routers and 50 backbone routers, already requires approximately 1800 MB of memory; for a realistic simulation, we also need to add end hosts, along with their access links and their overlay maintenance code.
It is important to point out why the simulations for the proposed router assisted scheme require less memory than the regular Scribe scheme: as all overlay functionality is provided by the access routers, we did not create the actual end hosts at all, so as to reduce the memory footprint of the simulation. Furthermore, our scheme requires fewer messages for the establishment of the overlay, since each router joins the overlay only once, regardless of the number of end hosts that it is a proxy for. Note also that these messages travel smaller distances since they do not need to cross the access links; only a single message is required for an end host to initially ask its access router to be its proxy. These memory savings are investigated in Figure 10 which shows the memory requirements of the 1000 end hosts alone, with or without router assistance. The reason that the curve for the router assisted case is less flat is that as the number of routers grows, the end hosts are distributed among more access routers, meaning that more access routers need to maintain state for their attached host.
5 Running the simulations

After replacing the OverSim code with our own modified code, execute `makemake` and then `make`. In order to use the topologies produced by the GT-ITM tool and extracted to the proper OMNeT++ format via our patch, make sure that the `nedfilelist` contains the path for the ned topology files. We have already produced some topologies from the GT-ITM and they can be found in the topologies directory. As they are very large, we have not included all the topologies in the `nedfilelist` in order to avoid memory overloads, as OMNeT++ loads all ned files before starting the simulation. Our advice is to add to the `nedfilelist` only the topology you want to simulate in the current scenario.

After the above, proper values must be given to the `omnetpp.ini` file. An example of two different scenarios follows. The first is a proxy based scenario and the second a non proxy based one. A description of all the parameters related to our code is also provided inline with the text. Note that you must use our `default.ini` file, included in our modified OverSim code. First, an example for a proxy based scenario:

```ini
[Run 36]
description = "GTITM TS Proxy Router Scribe Test "
network = GTITM_TS_IPv4Network

**.removalProbability=0.0
**.overlay.measureNetwInitPhase=true #variable: gather statistics when bootstrapping
**.numTiers=2
**.overlayType = "PastryModules"
**.tier1Type = "MulticastScribe"
**.tier2Type = "ProxyRouterScribeTestModule"
**.childTimeout = 6000000 #seconds until a node assumes that a particular child has failed
**.parentTimeout = 6000000 #seconds until a node assumes that the parent node has failed

These values are so large in order to avoid timeouts, whether childTimeout or parentTimeout, as our purpose is to construct the trees, not to measure their dynamic properties (this is a goal for future work).

**.numDataMessages = 1

Number of data messages to send.
```
**.overlay.iterativeLookup=false
**.overlay.useCommonAPIforward=true

**.overlay.pastry.optimizeLookup=false #whether to search the closest node in findCloserNode() calls
**.overlay.pastry.optimisticForward=false #forward message immediately in recursive mode, otherwise ping first
**.overlay.pastry.avoidDuplicates=false # when node seems unreachable but msg already sent, do not retry

These parameters are related to Pastry, we have just used some appropriate values for them.

**.hopCountMax=10000
**.outRouterNum=0
**.startIP = "10.1.0.1"
**.overlayTerminalType = "Host"
*.underlayConfigurator.churnGeneratorTypes = "NoChurn"
**.initPhaseCreationInterval = 1
**.waitOverlayEstablishment = true
**.bootstrapOracle.maxNumberOfKeys = 80000

This value defines the number of Overlay keys we are going to use. This value is greater than the one needed, but it is preferable to use this value to avoid a lack of Overlay keys.

**.targetOverlayTerminalNum = 1000  # 12000 800 25000 27722

The value of this parameter defines the number of the end hosts that will be used in the simulation. When we reach this number of end hosts, the simulation stops and the measurements start. It The number of end hosts is not random, it is related to the totalGroupNumber in order to follow the Zipf-like distribution. For this purpose, we used an .xls file to calculate the proper totalGroupNumber for a particular number of end hosts or vice versa. One must be careful to ensure that the last group must have at least 11 end hosts. The .xls file is provided in the distribution.

**.accessRouterNum = 625  # 5000 2916 1372 625

This parameter defines the number of access Routers in the topology. The value of this parameter is derived from the NED topology file produced by our topology conversion tool.
**.overlayAccessRouterNum = 0

This is set to zero because at the beginning of the simulation no routers are running the overlay protocols.

**.backboneRouterNum = 25

This is a synonym for access router.

**.overlayBackboneRouterNum = 0

Set to zero for the same reason as OverlayAccessRouterNum.

#**.ttError = 0

#**.tsError = 0

#**.ssError = 0

**.channelTypes="fiberline"

**.ppp[*.queue.frameCapacity = 1000000

**.globalParameters.rpcUdpTimeout = 100000.0

**.globalParameters.rpcKeyTimeout = 500000.0

A large number is used to avoid RPC timeouts, as they are beyond the scope of our work.

**.proxyBased = true

Determines whether we are running a proxy based scenario or not.

**.fixedGroupSize = false

When this variable is true it means that end hosts will randomly join groups according to the following parameters. When false, it means that we will use the Zipf-Like distribution of end hosts to groups. This variable must be set to false as the code for the other case is not ready.

**.groupsPerRouter = 1

How many groups a router will be allowed to join.

**.groupsPerHost = 1

How many groups an end host will be allowed to join.

**.totalGroupNumber = 36

How many groups will be created during the simulation.
**.exponent = -1.25  #Ignored when fixedGroupSize = true

**.constant = 0.5  #Ignored when fixedGroupSize = true

*.globalObserver.useGlobalFunctions = 1

*.globalObserver.globalFunctionsType = "ZipfGroupSizes"

output-scalar-file = "GTITM_TS_Proxy_Router_Scribe_Test_Run_36.sca"

This name of the scalar file to be created.

**.scheduleMsgDelay = 10

**.joinDelay = 100

**.dataDelay = 1000

These are set to large values to avoid message congestion when trying to find the RV point or when we send Join messages. Use them as they are.

The second example is for a non-proxy based scenario. The only difference is to the **.proxyBased = false parameter. The name of the scalar file also differs to indicate whether a proxy or a non proxy scenario was run.

[Run 37]

description = "GTITM TS Proxy Router Scribe Test "

network = GTITM_TS_IPv4Network

**.removalProbability=0.0

**.overlay.measureNetwInitPhase=true #variable: gather statistics when bootstrapping

**.numTiers=2

**.overlayType = "PastryModules"

**.tier1Type = "MulticastScribe"

**.tier2Type = "compareWithProxyRouterScribeTestModule"

**.childTimeout = 60000 #seconds until a node assumes that a particular child has failed

**.parentTimeout = 60000 #seconds until a node assumes that the parent node has failed

**.numDataMessages = 1

**.overlay.iterativeLookup=false
**.overlay.useCommonAPIforward=true

**.overlay.pastry.optimizeLookup=false #whether to search the closest node in findCloserNode() calls

**.overlay.pastry.optimisticForward=false #forward message immediately in recursive mode, otherwise ping first

**.overlay.pastry.avoidDuplicates=false # when node seems unreachable but msg already sent, do not retry

**.hopCountMax=10000

**.outRouterNum=0

**.startIP = "10.1.0.1"

**.overlayTerminalType = "OverlayHost"

*.underlayConfigurator.churnGeneratorTypes = "NoChurn"

**.initPhaseCreationInterval = 1

**.waitOverlayEstablishment = true

**.bootstrapOracle.maxNumberOfKeys = 80000

**.targetOverlayTerminalNum = 4000 #8000 #12000 #27722

**.accessRouterNum = 1372 # 5000 2916 1372 625

**.overlayAccessRouterNum = 0

**.backboneRouterNum = 28 # 50 36 28 25

**.overlayBackboneRouterNum = 0

#**.ttError = 0

#**.tsError = 0

#**.ssError = 0

**.channelTypes="fiberline"

**.ppp[*].queue.frameCapacity = 10000000

**.globalParameters.rpcUdpTimeout = 100000.0

**.globalParameters.rpcKeyTimeout = 500000.0

**.proxyBased = false

**.fixedGroupSize = false
**.groupsPerRouter = 1  #Ignored when fixedGroupSize = false
**.groupsPerHost = 1    #Ignored when fixedGroupSize = false
**.totalGroupNumber =110 #200  #Ignored when fixedGroupSize = true
**.exponent = -1.25     #Ignored when fixedGroupSize = true
**.constant = 0.5       #Ignored when fixedGroupSize = true
*.globalObserver.useGlobalFunctions = 1
*.globalObserver.globalFunctionsType = "ZipfGroupSizes"
output-scalar-file = "GTITM_TS_End_Hosts_Scribe_Test_Run_37.sca"
**.scheduleMsgDelay = 10
**.joinDelay = 10
**.dataDelay = 1000
6 Metrics produced

As the simulation runs, some output files are created, the format and content of which is explained below. The files produced are the following:

- GTITM_TS_End_Hosts_Scribe_Test_Run_[Run].sca
- NODE_STRESS_Measurements_[Run]_.txt
- STRETCH_EndHostsOnly|ProxyRoutersOnly_[Run]_.txt
- TOTAL_OVERLAY_NODES_PER_GROUP_EndHostsOnly|ProxyRoutersOnly_[Run]_.txt

Each file includes the number of the run that produced it in [Run], according to the run numbers in omnetpp.ini. Depending on whether a proxy or a non-proxy scenario was executed, the name of the file differs for the last two files (the two options are separated by the | symbol).

The first file contains the scalar metrics recorded at the end of the Simulation. Its contents are as follows:

1. RVtoEndHostPathLengths_ProxyRouterOnly_Group_[]

   This scalar is printed in a proxy scenario and measures for each group the path length in terms of IP hops from the RV point to the end hosts. When traversing the tree, when we reach a leaf we record the IP hops required to reach that particular host. It must be noted that the end host in the proxy based scenario is a router. At the end of the tree traversal, this scalar will have recorded all the paths from RV point to end hosts. In the same manner, the scalar metric RVtoEndHostPathLengths_EndHostsOnly_Group_[] is recorded for the non-proxy scenario. The only difference is that the end hosts now are not routers but actual end hosts.

2. SendertoEndHostPathLengths_ProxyRoutersOnly_Group_[]

   This is the same as the previous metric but for the IP multicast tree from the Sender to the end hosts, since there is no RV point in IP Multicast. For a non-proxy scenario the SendertoEndHostPathLengths_EndHostsOnly_Group_[] scalar is recorded instead.

3. Children Tables

   This scalar records the children tables that each node has, so as to use these statistics for the Children Tables metrics. After the initialization has finished, each Scribe node knows how many Children Tables it has. Thus, each Scribe Node writes to a global scalar and by collecting this value and printing it we will be able to calculate Node Stress.

4. Number of Elements In Children Tables
This is used for exactly the same reason as Children Tables scalar.

5. Overlay Multicast Hops

This measures all the overlay hops that a group has.

6. Underlay Hops of Overlay Multicast

This measures the size of the tree of each group, in terms of IP hops. Each overlay link consists of some of underlay links, and these are recorded here.

7. Underlay Hops of Overlay Multicast, No Sender

This is the same as the above but without taking into account the links between the RV and the Sender.

8. Underlay Hops For IP Multicast

This metrics measures the IP hops for the IP multicast tree

The second file contains the metrics required to calculate node stress. It is formatted as a sequence of lines, as follows:

```
15.118.0.1   1   1
13.142.0.1   2   85
14.45.0.1    2   2
15.182.0.1   1   1
21.88.0.1    2   2
```

In these 5 lines we can see that there are three tab separated columns. The first column is the IP address of the node, the second is the number of children tables in this node and the third is the number of entries in all the children tables of the node.

The third file contains the metrics required to calculate stretch. An example of the format of the file follows:

```
GroupIndex  OverlayHops  UnderlayHopsOfOverlay  UnderlayHopsOfOverlayNoSender  UnderlayHopsOfIP
1           817   5800   5791   2322
2           383   2983   2974   1200
3           238   1824   1817   786
4           170   1611   1604   598
```

The first column is the group index which is an identifier for the group (largest group first), the second column is the number of overlay hops in the group, the third column is the number of underlay (IP hops), the fourth column is the same as the third but excluding the hops between the RV point and the sender, and the fifth column is the number of IP hops in the IP multicast case.

The fourth file contains the metrics for the routers participating in each group. The format of the file is as follows:
Total Overlay Routers in all Simulation: 826
GroupIndex  OverlayRouterPerGroup
1           817
2           384
3           239
4           171
5           129
6           103
7           86

The first line shows the number of routers that participated in the Overlay. In the following lines, the first column is the group index (identifier) of the group and the second column the number of routers in the group.

Finally, one more file can be produced to measure memory and CPU time consumption via the top command in UNIX like systems. To get these data, when the simulation starts you need to open a second terminal and type ps –e to see the process ID of OverSim. Then type top –p pid > “filename.txt”. Next type fprsl enter d 60 enter. The letters stand for Group Name (F), Swapped Size (P), Code Size (R), CPU Time (L), Data+Stack Size (S) and time interval (D 60 for 60 seconds). We need to distinguish three different meanings of “time” here:

- **Real Time**: This is the time that the simulation takes to complete. It was not measured because it is not usable as it varies from run to run, even when simulating runs with the same properties, depending on the other processes running in the system.
- **CPU Time**: This is the time used by the simulation process (OverSim) from the CPU. This is the time used in our metrics and it is produced by the above command. It is the most useful time metric for the scalability of the platform.
- **Simulation Time**: This is the virtual time in the simulation required in order for all joins to be made and the measures to be calculated. It is normally larger than the real time because the simulation events are being accelerated in order to finish quickly. This time metric is recorded in the scalar file that is being produced and it can be found in the GlobalStatistics metrics.
7 Future work

7.1 Known problems

The first thing to do is obviously to fix known bugs. At this time one known bug is that our application cannot run in graphical environment, but only from the command line, using the following command:

Run –UCmdenv –r [The run number in the omnetpp.ini file]

7.2 Dynamic application behavior

The performance evaluation has concentrated on the static properties of the multicast distribution trees built by either regular Scribe or our router assisted Scribe variant. Routing however is only a means to an end, which in our case is improving the performance of content distribution applications via multicast. Unfortunately, the static properties of the multicast trees are insufficient to characterize the dynamic performance of actual applications, and this is the exact reason why we have evaluated the feasibility of executing full fledged simulations of overlay multicast: only by including a model of the application in the simulation will we be able to predict application level performance.

As a first example of the impact of dynamic application behavior on performance, consider the trade-off discussed above between burdening the access routers with forwarding and tree maintenance and improving path stretch, link stress and DHT related signaling. If an application uses overlay multicast to distribute very large amounts of data to many recipients, our router assisted approach will be preferable to regular Scribe due to the reduced path stretch and link stress that it provides. If however an application uses overlay multicast to send small amounts of data to a large number of small groups, regular Scribe may be preferable due to its reliance on end hosts only. In both cases, a full fledged simulation will be needed to assess metrics such as path delays, as these depend not only on the links traversed by each packet, but also on the load placed on each link by each overlay scheme.

As a second example, consider the case of applications with dynamic multicast group membership, an example of which is peer assisted content distribution. In our work as, to the best of our knowledge, in all previous studies of Scribe, it is assumed that the overlay network is fully formed before Scribe signaling begins and never changes thereafter. A full fledged simulation incorporating dynamic group membership would reveal that node arrivals and departures, in addition to increasing the maintenance requirements of the underlying DHT, also affect the structure of the multicast trees, since not only the paths between the nodes, but even the location of the RV points may change. In our router assisted scheme, since access routers participate in the overlay on behalf of multiple hosts, group dynamics may have a smaller effect than in regular Scribe, but this would depend on the actual application under study.
As a third example, consider the idea of global access router participation to the DHT substrate, regardless of the presence of interested end hosts. This would lead to a very stable DHT substrate since routers are less volatile than end hosts, as well as to improved routing due to the additional possibilities offered, but it would also cause the DHT maintenance overhead to increase due to the larger number of nodes participating in the overlay. Again, in order to assess whether this change would be beneficial to application level performance or not, one would need to undertake a full fledged simulation study incorporating actual application behavior.

In this direction we have developed some code in order to support the dynamic joining and leaving of routers from the overlay and the groups. Although it is not ready yet we will describe its usability and what it needs to be done.

The goal is to add overlay functionality to the routers of a GTITM Topology, which will allow them to act as agents for the end-hosts to the overlay. The multicast trees created after the initialization of the topology and the joining of terminals must also be measured. The final step is to compare the overlay trees with the IP multicast trees created by using the terminals in the overlay. The steps required to complete the project are listed below:

1. Create Overlay Nodes which have no Overlay Functionality (Dummy Nodes).
2. Make the IPv4Network add an access Router to the Overlay when a dummy Node connects to it.
3. Develop and rebuild ScribeTest to simulate the functionality described above in order to run over Scribe.

The creation of dummy nodes is simple: the only thing we had to do was to remove the overlay modules from the Ned file. Modifying the IPv4Network to add access routers to the overlay is also partially done. When a Host logs onto a router, if the router has no overlay functionality IPv4 load the overlay code for that router by calling addRouterToOverlay. This simply runs the modules of the overlay with dynamic module building, it sets up the connections and it calls the initialize method of each new module. After loading the overlay functionality it sends a MultiScribeTestNotification message to the MultiScribeTest of the router through the direct_in gate in order to log the node into the list of the MultiScribeTest. If the router is already in the overlay it simple sends the MultiScribeTestNotification.

When the churnGenerator tells to a node to leave the IPv4Network, it sends a SCRIBELeaveMessage to scribeTest in order to erase the Node from the list and from the groups that it has joined. The leaving code has been removed from the IPVUnderlayConfigurator and it is commented out in MultiScribeTest. All that needs to be done in order for this to work is to send in the direct_in gate a leave message with the TrasportAddress of the node it the MultiSribeTest (methods createNode, migrateNode) and to remove the comments from the MultiScribeTest. We have also supplied the code in a different file.

Finally, MulyiScribeTest needs to be rebuilt. MultiScribeTest is running over Scribe and in our scenario it is running only in the routers. When a new node is connected to the router, the overlay functionality is loaded in the router. After the initialization, the node is only connecting or disconnecting from the router. All the work is performed
by the router. MultiScribeTest has a map and a list. The map contains all the groups that the router belongs to (on behalf of the nodes) as a key and how many nodes, connected to it, have joined the group. The list has the nodes that are connected to the router as jets and it has as elements objects of the class Node.h which depicts the node and the groups that the node is in. The picture bellow shows the structure:

<table>
<thead>
<tr>
<th>&lt;int,int&gt;map</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>groupNumer</td>
<td>#of joined Nodes int the Group</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>#5</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>#3</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>#1</td>
<td></td>
</tr>
</tbody>
</table>

When a node is connected to the router, it sends a message and router adds the Node into its list by calling addNode method. This method adds the node into the list of ScribeTest. When a node is disconnected from the router it sends a message to the router and it removes the node from the list by calling removeNode. It also checks in which groups the node was included and informs the groups in the map by removing the node from these groups. Last it iterates the map to find groups that are empty and leaves them. After a node has been logged into the list of the router, there are three scenarios. The scenarios are executed randomly according to a uniform distribution.

1. Node wants to join a group
2. Node wants to leave a group
3. Node wants to send data to a group

In the first case a node is chosen randomly from the nodes that are logged in and it is also randomly chosen whether he is going to join a group that the router has not already joined, or a group that the router has already joined. In the first case the node joins the group by calling addGroupToNode and the router makes a new entry in the map for the group while it sends a join(group) message to Scribe in order to join. In the second case the node adds the group to its list by calling again addGroupToNode and the router just increases the entry, of the group, in the map by one.

In the second case, when a node leaves a group, it chooses a group to leave, it removes the group from the Node list and it subtracts one from the entry of the group in the map. Finally, it checks whether the group is empty, in which case it removes the entry and sends a leave(group) message to the scribe in order to leave the group.
In the third case, the node just checks if there is a group in the map, it chooses a random group from the map and sends data to the group.

The Node class is a class which keeps information about the Nodes attached to the Router. It maintains the TransportAddress of the attached Node and a list with the groups that the attached node wanted to join, and includes all the necessary methods to control the structures of the class. For example, the method addGroup(int group) adds a group to the list of the class.

In order to make the above fully functional, some tasks remain. First, we have to substitute the random joining and leaving mechanism with an appropriate distribution. Second, when a node leaves the Overlay and disconnects from the router, if this end host was the last Node connected to the specific router, the router does not need any more the overlay functionality, therefore a function to kill the overlay functionality from the routers with no connected end hosts is needed. The removeNode method which implements the leaving procedure triggers the procedure of the router leaving the overlay, but it is commented out. There is also an if condition from which we can call the procedure for the router leaving the overlay.
8 Acknowledgements

I want to thank my supervisor Assistant Professor George Xylomenos who gave me the opportunity to see how academic research is performed and for trusting me during all the time needed to fulfill all the tasks that he asked me to do.

I want to also thank the Ph.D. student Konstantinos Katsaros for all the help he offered me and for his patience in explaining things in sufficient detail. I also thank him for his co-operation the led to my work being completed.
9 Appendix: OverSim

OverSim [1] is a very useful simulation platform which provides implementations of many Overlay Routing schemes, such as Chord, Pastry, etc. It offers a very good specification of its classes and methods via doxygen. Although doxygen is good, there is no functional description of the implemented code. Many variables that should be inserted in the omnetpp.ini file are not explained and it is very difficult to understand what they do. Furthermore, there is no explanation of the functional operation of OverSim’s classes. Generally, one should read a lot of code from OverSim in order to understand what is going on in each class. On the other hand, OMNeT++ has a very good documentation file which is very helpful for someone who tries to implement code for it. We will now explain some definitions that one needs to be aware of in order to better understand this thesis.

9.1 Direct message sending

Sometimes it is necessary or convenient to ignore the gates/connections of the simulator and send a message directly to a remote destination module. The sendDirect() function does that:

```
sendDirect(cMessage *msg, double delay, cModule *mod, int gateId)
sendDirect(cMessage *msg, double delay, cModule *mod, const char *gateName, int index=-1)
sendDirect(cMessage *msg, double delay, cGate *gate)
```

In addition to the message and a delay, it also takes as parameters the destination module and gate. The gate should be an input gate and should not be connected. In other words, the module needs dedicated gates for receiving via sendDirect(). To leave a gate unconnected in a compound module, you need to specify connections nocheck: instead of plain connections: in the NED file. An example:

```
cModule *destinationModule = parentModule()->submodule("node2");
double delay = truncnormal(0.005, 0.0001);
sendDirect(new cMessage("packet"), delay, destinationModule, "inputGate");
```

At the destination module, there is no differentiation between messages received directly and those received over connections.

9.2 Dynamic module creation

In some situations you need to dynamically create and maybe destroy modules. It is often convenient to use direct message sending with dynamically created modules, as we did. Once created and started, dynamic modules are not any different from static modules; for example, one could also delete static modules during simulation (although this is rarely useful.)

To understand how dynamic module creation works, you have to know a bit about how OMNeT++ normally instantiates modules. Each module type (class) has a corresponding factory object of the class cModuleType. This object is created under
the hood by the Define_Module() macro, and it has a factory function which can instantiate the module class (this function basically only consists of a return new module-class(...) statement). The cModuleType object can be looked up by its name string (which is the same as the module class name). Once you have its pointer, it is possible to call its factory method and create an instance of the corresponding module class -- without having to include the C++ header file containing module's class declaration into your source file. The cModuleType object also knows what gates and parameters the given module type has to have. (This info comes from compiled NED code.) Simple modules can be created in one step. For a compound module, the situation is more complicated, because its internal structure (submodules, connections) may depend on parameter values and gate vector sizes. Thus, for compound modules it is generally required to first create the module itself, second, set parameter values and gate vector sizes, and then call the method that creates its submodules and internal connections. Simple modules with activity() need a starter message. For statically created modules, this message is created automatically by OMNeT++, but for dynamically created modules, you have to do this explicitly by calling the appropriate functions. Calling initialize() has to take place after insertion of the starter messages, because the initializing code may insert new messages into the FES, and these messages should be processed after the starter message.

9.3 Scalars

You can use output scalars

- to record summary data at the end of the simulation run
- to do several runs with different parameter settings/random seed and determine the dependence of some measures on the parameter settings. For example, multiple runs and output scalars are the way to produce Throughput vs. Offered Load plots.

Output scalars are recorded with the recordScalar() method of cSimpleModule, and you'll usually want to insert this code into the finish() function. An example:

```c++
void Transmitter::finish()
{
    double avgThroughput = totalBits / simTime();
    recordScalar("Average throughput", avgThroughput);
}
```

You can record whole statistics objects by calling their recordScalar() methods, declared as part of cStatistic.

Generally the calls write into the output scalar file which is named omnetpp.sca by default. The output scalar file is preserved across simulation runs (unlike the output vector file which gets deleted at the beginning of every simulation run). Data are always appended at the end of the file, and output from different simulation runs are separated by special lines.
9.4 Other capabilities of the platform

The OverSim platform offers four different types of churn Generators. A churn generator is a tool which defines when an end host will join, leave or migrate from the topology. The first generator that is being offered is Lifetime Churn, which is supporting a specific time of an end host remaining alive. The second generator is ParetoChurn and the third is RandomChurn. The latter implements random joining of the end hosts and accordingly the leaving. The final one is TraceChurn.

Another very important tool offered by the OMNet++ platform is the extraction topology tool. This tool uses the cTopology class which was designed primarily to support routing in telecommunication or multiprocessor networks.

A cTopology object stores an abstract representation of the network in graph form:

- each cTopology node corresponds to a module (simple or compound), and
- each cTopology edge corresponds to a link or series of connecting links.

You can specify which modules (either simple or compound) you want to include in the graph. The graph will include all connections among the selected modules. In the graph, all nodes are at the same level, there's no submodule nesting. Connections which span across compound module boundaries are also represented as one graph edge. Graph edges are directed, just as module gates are. If you're writing a router or switch model, the cTopology graph can help you determine what nodes are available through which gate and also to find optimal routes. The cTopology object can calculate shortest paths between nodes for you.

You can extract the network topology into a cTopology object by a single function call. You have several ways to select which modules you want to include in the topology:

- by module type
- by a parameter's presence and its value
- with a user-supplied boolean function

For more details check the OMNeT++ documentation and user manual.
10 References


