An Evaluation of Three Application-Layer Multicast Protocols

Carl Livadas
Laboratory for Computer Science, MIT
clivadas@lcs.mit.edu

September 25, 2002

Abstract

In this paper, we present and evaluate three application-layer multicast (ALM) protocols, namely Narada [3–5], NICE [1,2], and an ALM protocol implemented using the Internet Indirection Infrastructure (*i3*) [11]. We evaluate these three ALM protocols according to the quality of their data delivery paths, their robustness to changing membership, changing network characteristics, and failures, and their overhead. Our evaluation focuses on the ability of each such protocol to scale to large multicast group sizes and to handle dynamic environments involving frequent membership changes and failures. We identify strengths and weaknesses of each such protocol and propose modifications that may remedy or mitigate some of these weaknesses.

1 Introduction

In the recent past, several implementations of the multicast communication service at the application-layer have been proposed. The earlier attempt to implement the multicast communication service at the IP layer, namely *IP multicast*, has not been as successful as initially expected. Although IP multicast affords latency characteristics comparable to IP unicast and minimizes packet duplication, it requires routers to maintain per-group state and to provide additional functionality. The resulting scalability, network management, and deployment issues have stifled its wide adoption. Conversely, application-layer multicast (ALM) protocols make use of IP unicast primitives and push all multicast related functionality onto the members of the multicast group. Although less efficient in terms of latency and network usage, this approach simplifies the issues of deployment and maintenance and, thus, constitutes a more viable multicast service implementation.

In this paper, we present and evaluate three ALM protocols, namely Narada [3–5], NICE [1, 2], and an ALM protocol implemented using the Internet Indirection Infrastructure (*i3*) [11], which we henceforth refer to as *i3*-MCAST [8]. Narada constructs a richly connected overlay network (referred to as a *mesh*) and disseminates multicast traffic along per-source spanning trees of this mesh. NICE organizes the multicast group members into a hierarchy of clusters. In particular, NICE partitions the members at each layer of this hierarchy into clusters, where proximate members at the given layer belong to the same cluster, and elects a leader to represent each such cluster at the higher layer of the hierarchy. *i3*-MCAST constructs a multicast forwarding tree using the *rendez-vous*-based indirection primitive provided by *i3*.

We evaluate these three ALM protocols according to the quality of their data delivery paths, their robustness to changing membership, changing network characteristics, and failures, and their overhead. Our evaluation focuses on the ability of each such protocol to scale to large multicast group sizes and to handle dynamic environments involving frequent membership changes and failures. We identify strengths and weaknesses of each such protocol and propose modifications that may remedy or mitigate some of these weaknesses.

This paper is organized as follows. Section 2 presents the metrics used in the literature to evaluate the performance of ALM protocols. Section 3 discusses the scalability issues pertaining to each such performance metric. Sections 4, 5, and 6 describe and evaluate Narada, NICE, and *i3*-MCAST, respectively. Section 7 briefly describes the factors affecting one's choice of an ALM protocol. Finally, Section 8 briefly summarizes our evaluation of each protocol.

2 ALM Protocol Performance

The performance metrics used to evaluate overlay-based application-layer multicast protocols are: i) the quality of the data delivery paths between sources and receivers, ii) the robustness of the overlay structure to membership changes, network characteristic changes, and failures, and iii) the protocol overhead.

Quality of Data Delivery The quality of data delivery paths between sources and receivers may be evaluated from the perspective of either the application or the network. From the application's perspective, the quality of

the data delivery path is measured in terms of data transmission metrics, such as latency and bandwidth. The performance of an ALM protocol in terms of latency and bandwidth is often compared to and normalized by that of IP multicast.[1] This comparison quantifies the performance cost of implementing the multicast service at the application rather than the IP level. The term *stretch* is used to denote the per-receiver ratio of the latency from the source to the particular receiver along the overlay network to the respective IP multicast (or, alternatively, IP unicast).

From the network's perspective, the quality of the data delivery path is measured in terms of *stress* and *resource usage*. Stress measures the concentration of overhead on particular links and hosts. Link stress is the count of identical packets that the protocol sends along each link of the underlying network. Thus, IP multicast, in which data is disseminated along trees of the underlying network, incurs unit link stress. Host stress corresponds to the number of copies of the same packet a particular host must forward; this corresponds to the out-degree (fanout) of the data path at the given host.

Presuming link delay is an indication of the cost associated with using a particular link, the resource usage of an ALM protocol is defined to be the quantity $\sum_{i \in L} d_i s_i$, where L is the set of underlying links used for data transmission, d_i is the delay of link i, and s_i is the link stress of link i. Once again, the resource usage of an ALM protocol compared to (normalized by) that of IP multicast is an indication of the overhead of implementing the multicast service at the application rather than the IP layers.

Protocol Robustness Protocol robustness refers to the ability of the application-layer multicast protocol to mitigate the effects of membership changes (member joins and leaves), network characteristic changes (*e.g.*, congestion), and overlay link and host failures. Measuring overlay robustness entails quantifying the extent to which the delivery of data is disrupted and the time it takes for the protocol to restore it.

Protocol robustness imposes several guidelines on the design of application-layer multicast protocols. First, overlay construction should minimize the introduction of single points of failure, such as the concentration of data forwarding responsibilities onto a small number of overlay nodes. Such failure points may partition the overlay and result in extended disruptions in the delivery of data and expensive overlay reconfigurations. Thus, ALM protocols should either assign equal responsibilities to each node of the overlay, or introduce redundancy. Second, routine overlay operations, such as member joins, should not put stress on particular overlay nodes. Such practice load the particular nodes and lead to a degradation of performance and, possibly, node or link failure. Finally, as argued by Chu *et al.* [3], it may be beneficial to design an ALM protocol to be *self-sufficient*, in the sense that, once a particular set of hosts have joined the overlay, routine overlay operations should not rely on external services. For example, overlay partition should be repairable without invoking some external bootstrapping mechanism.

Protocol Overhead Protocol overhead refers to the per-host memory, per-host processing, and control traffic requirements associated with the construction and maintenance of the overlay. Hosts may be required to maintain membership, topology, and routing information, compute routing tables, and exchange control packets to disseminate such information and coordinate the reconfiguration of the overlay.

One aspect of an ALM protocol's overhead, which is often overlooked, is inter-member distance estimation. Such estimates may be needed for the purposes of routing and overlay reconfigurations. Of course, the distance metric depends on the performance requirements of the application using the ALM protocol. Inter-member distance estimates can be obtained either passively, by monitoring data and control traffic, or actively, by explicit measurement probes. Although passive measurements introduce minimal overhead, active measurements may heavily contribute to an ALM protocol's overhead [3, 4]. This is especially the case when the distance metric used involves bandwidth. Thus, care must be taken on deciding when and how such measurements are performed.

3 ALM Protocol Scalability

The scalability of an ALM protocol refers to its ability to sustain good performance as the multicast group and, consequently, the overlay grows in size. In this section, we discuss the scalability issues that pertain to each of the performance metrics discussed in Section 2. Such issues guide our subsequent evaluation of the scalability of the three ALM protocols considered in this paper.

Quality of Data Delivery The scalability of an ALM protocol in terms of application-layer performance depends on the performance requirements of the application. We classify application-layer performance metrics into either *hop-independent* and *hop-cumulative* metrics. On one hand, the end-to-end cost associated with hop-independent metrics, such as bandwidth, is not explicitly affected by the number of overlay hops. On the other hand, the end-to-end cost associated with hop-cumulative

[1] In absence of IP multicast measurements, per-receiver IP unicast measurements are used for this comparison and normalization. This approximation is accurate only when IP multicast is implemented using source-specific spanning trees, such as DVMRP, and the IP unicast paths are symmetric.

metrics, such as latency, depends explicitly on the hop count. For instance, the end-to-end latency corresponds to the sum of the latency incurred along each overlay hop.

Hop-cumulative metrics may constrain the scalability of an ALM protocol. For the purpose of limiting link and host stress, ALM protocols often constrain the degree of the overlay network they construct. Thus, as the size of the multicast group grows, inevitably the diameter in terms of overlay hops increases. So as to sustain its performance in terms of hop-cumulative metrics, an ALM protocol must either minimize the overlay hops between sources and receivers, minimize overlay hop latencies, or, preferably, both. This suggests that the overlay of any scalable ALM protocol must conform to the locality of the underlying network topology, where locality is defined in terms of the hop-cumulative metric required by the application.

From the perspective of the network, the scalability of an ALM protocol is measured in terms of whether and to what degree the link and host stress increases as the size of the multicast group grows.

Protocol Robustness As the size of the overlay (multicast group) increases, the probability of some hosts failing increases. This is due to both the sheer number of hosts and the inevitable heterogeneous reliability and performance capabilities of the hosts and links comprising the overlay. Thus, as the size of the overlay increases, robustness becomes an increasingly important performance issue.

We evaluate the scalability of an ALM protocol in terms of robustness by analyzing the degree to which the ALM protocol: i) avoids the construction of overlays having single points of failure, ii) avoids over-stressing particular nodes of the overlay, and iii) constructs data delivery paths that limit the extent to which congestion and failures disrupt the delivery of data, such as distinct source-specific data delivery trees.

Protocol Overhead The scalability of an ALM protocol is highly dependent on the protocol's overhead in terms of per-host memory, per-host processing, and control traffic. Scalability with respect to the per-host memory is evaluated by estimating the amount of state that each host must store. This may include both membership information and routing information. Scalability with respect to the per-host processing is evaluated by identifying the processing requirements of each host, such as the cost and the frequency of routing table recalculation. Finally, scalability with respect to control traffic involves estimating the cost of maintaining the overlay and performing routine operations, such as handling a request to join the overlay and reconfiguring the overlay to reestablish connectivity after a failure.

In addition to how costly each overlay maintenance and repair operation is, it is important to identify: i) how often such an operation is invoked, ii) whether the overhead of each such operation is concentrated on particular hosts, and iii) whether such operations can occur in bursts. A burst of operations that stress particular hosts or underlying links may prevent the scalability of an ALM protocol.

4 Narada

Narada [3–5] is a mesh-based ALM protocol. As such, it performs two tasks: i) the construction and maintenance of a richly connected overlay graph of the members of the multicast group, henceforth referred to as the overlay *mesh*, and ii) the construction of per-source spanning trees within this overlay mesh for the purpose of multicast traffic dissemination.

Chu *et al.* [3–5] argue that a mesh-based approach to ALM is advantageous to tree-based approaches used by other ALM protocols. On one hand, shared spanning trees result in the concentration of multicast traffic on particular paths, are susceptible to single points of failure, and involve sub-optimal source to receiver paths. On the other hand, source-specific spanning trees incur the overhead of constructing and maintaining multiple overlays in the case of multi-source multicast transmissions.

Conversely, mesh-based approaches construct a and maintain a single overlay graph. The use of a single mesh averts the need to construct and maintain multiple overlays. Furthermore, mesh-based approaches take advantage of the connectivity of such overlay meshes to construct source-specific dissemination trees. Source-specific trees comprise better source to receiver paths and prevent single points of congestion and failure from disturbing the traffic from all sources. Of course, the performance of mesh-based approaches heavily depends on the mesh quality; that is, whether the quality of the path between any pair of members within the mesh is comparable to the quality of the unicast path between the same pair of members. Indeed, the mesh must continuously be reconfigured so as to improve the quality of the dissemination paths, avoid hot-spots in terms of node and link stress, and recover from failures and group membership changes.

In the next few sections, we give an overview of Narada. We describe how Narada manages the group membership, how routing is performed, and how the mesh is maintained. We conclude our presentation of Narada by summarizing the observed performance of Narada presented in [3–5] and by commenting on its virtues and shortcomings.

In our presentation and evaluation of Narada, we let N denote the number of multicast group members.

4.1 Group Management

In Narada, each member of the multicast group maintains the complete multicast group membership. Heartbeat messages are periodically exchanged by neighbor members within the mesh. These messages announce that the sender is still a member of the multicast group and propagate the membership information across the mesh. Moreover, heartbeat messages are annotates with monotonically increasing sequence numbers. The sequence number of a heartbeat message indicates how up-to-date the heartbeat message is.

The membership state maintained by each member i includes a tuple $\langle j, s_j, t_j \rangle$ for each member j of the multicast group known to i. The element s_j corresponds to the sequence number of the latest heartbeat message known by i to have been issued by j. The element t_j is the time at which i learned that j issued a heartbeat message with sequence number s_j.

The heartbeat messages of a host i include a tuple $\langle j, s_j \rangle$ for each member j of the multicast group known to i. Thus, i's heartbeat messages propagate the membership state information known to i to each of its neighbors. Upon receiving a heartbeat message from member i, each of i's neighbors updates its membership state to reflect any new membership information revealed by i's heartbeat message.

We let $T_{\text{HEARTBEAT}}$ denote the period with which multicast group members send heartbeat messages. In view of reducing control overhead, Chu *et al.* [3] also propose that membership information be piggybacked onto the routing messages exchanged by neighbor members. However, such a scheme presumes that heartbeat and routing messages are exchanged with the same period.

Member Join A host x joins the multicast group as follows. Through a bootstrap mechanism, the host x attains a set X of multicast group members. Then, the host x randomly selects a subset of X, contacts each host in this subset and requests to become its neighbor in the mesh. This process is repeated until x becomes the neighbor of one or more members in X. Subsequently, the exchange of heartbeat messages between x and its newly established neighbors informs x of the complete multicast group members and the remaining members of x's existence.

Member Leaves and Crashes A member x leaves the multicast group by simply notifying its neighbors. The fact that x has left the multicast group is propagated throughout the mesh through the exchange of heartbeat messages. In order to allow the routing to adapt to new topology and to minimize the effect of departures on data delivery, hosts are required to keep forwarding multicast packets for a short period of time $\Delta_{\text{FORWARDING}}$ following their departure from the multicast group.

The fact that a member x has crashed is detected when its neighbors in the mesh do not receive a heartbeat message from x for Δ_{FAILURE} time units. When a neighbor y of x suspects that x has crashed, it probes x. If this probe (or any such probe sent by some other neighbor of x) is not acknowledged, then y presumes that x has indeed crashed and propagates this information throughout the mesh through its heartbeat messages. The fact that x has crashed is maintained within the membership state information so that stale information pertaining to x does not get misinterpreted as information pertaining to a newly discovered member.

Mesh Partitions Mesh partitions are repaired as follows. Each member maintains a queue of all the members whose tuple in the membership state hasn't been updated for $\Delta_{\text{PARTITION}}$ time units. The elements of this queue are the members suspected of belonging to another part of a partition in the mesh. Periodically and with probability $P_{\text{PARTITION-REPAIR}}$, the member at the head of the queue is removed and probed. If this probe is not acknowledged, then the given member is presumed to have crashed and this information is propagated throughout the mesh. Otherwise, a link connecting the two members is added to the mesh. The probability $P_{\text{PARTITION-REPAIR}}$ is chosen based on the size of both the queue and the group so that even if several members detect the partition and attempt to repair it, only a small number of new links are added to the mesh.

4.2 Routing

Narada uses a distance vector routing protocol to compute shortest point-to-point routes among the members comprising the overlay mesh (multicast group). So as to avoid the *counting-to-infinity* problem, the routing table maintained by each member contains both the routing cost to every other member and the path that affords the given cost. Routing updates exchanged by neighbor members include the respective member's routing table; that is, the respective member's cost and path to each other member. We let $T_{\text{ROUTING-UPDATES}}$ denote the period with which multicast group members send routing updates.

Depending on the needs of the application using the Narada system, the distance vector routing protocol can be customized to optimize for a variety of application-layer performance metrics, such as latency, bandwidth. Of course, the routing table calculation relies on members estimating their distance to their neighbors in the overlay mesh. Ref. 3 describes how to customize the distance vector routing protocol to optimize for both latency and bandwidth. The authors observe that for conferencing applications, which impose both low latency and high bandwidth performance requirements, a dual metric in-

volving both latency and bandwidth affords better performance than using latency or bandwidth alone. We refer the reader to [3, 4] for the full description of how the dual metric involving both latency and bandwidth is incorporated within Narada.

Narada constructs per-source multicast dissemination trees for the overlay mesh using *reverse-path broadcasting* [9, 10]. Packets are thus forwarded as follows. Suppose that a member x receives a packet p from the source s through its neighbor x'. x proceeds to forward p if and only if x' is the next hop of x to s according to its routing table. If indeed x' is the next hop of x to s, then x forwards p to each of its neighbors x'' whose next hop to s is x. Thus, each member also maintains a bit indicating whether it is the next hop on the shortest path from each of its neighbors to each of the multicast transmission sources.

4.3 Mesh Maintenance

Narada incrementally improves the quality of the overlay mesh, with respect to a particular performance metric, by dynamically adding and removing overlay links between the members of the multicast group.

Links are added to the overlay mesh as follows. Each member x periodically (with a period T_{ADD}) chooses a random member in the multicast group that is not one of its neighbors and evaluates the *utility* of a link between itself and this random member. The utility of a link corresponds to the improvement in performance that the addition of the link would afford to x. Of course, a link's utility depends on the performance metric for which the overlay mesh is optimized, *e.g.*, latency or bandwidth. For example, in the case of latency, Chu *et al.* define a link's utility to be $\sum_{h \in H}(l_c(h) - l_n(h))/l_c(h)$, where H is the set of members of the multicast group, $l_c(h)$ is the current latency to h, and $l_n(h)$ is the new latency to h were the link in question added to the overlay mesh. If the utility of adding a link exceeds some threshold U_{ADD}, then the link is added to the routing table of x and propagated along the mesh to the other members of the multicast group.

Links are removed from the overlay mesh as follows. Each member x periodically (with a period T_{DROP}) computes the utility of the overlay links connecting it to its neighbors. In the case of removing a link, its utility corresponds to the importance of the link to each of its endpoints. For example, in [3] the utility of an existing link with respect to one of its endpoints is defined to be the number of members for which the given link comprises the next hop. This count is computed from the perspective of both endpoints and the link's utility is chosen to be the maximum of the two counts. If the utility of any link is below some threshold U_{DROP}, then the link is removed from the routing table of x and propagated along the mesh to the other members of the multicast group.

Both thresholds U_{ADD} and U_{DROP} are chosen based on the multicast group size and the number of neighbors of x.

When adding and removing links, caution must be taken so as to cause neither *instability*, nor mesh partition. Instability refers to situations in which links are added and, subsequently, immediately dropped or vice versa. Instability and mesh partition are avoided by: i) setting the threshold U_{DROP} lower than the threshold U_{ADD}, ii) overestimating the utility of a link when deciding if it should be removed, and iii) when deciding if a link should be removed, evaluating its utility from the perspectives of either endpoint and using the highest link utility value of the two.

The overlay degree of each member in the multicast group gets dynamically adjusted based on the capabilities of the member and the network it its vicinity. With the onset of congestion close to a particular member, its children in the data delivery tree witness a degradation in performance. Thus, the utility of the links to the congested member drops. These links are eventually removed from the mesh in favor of higher utility links. Chu *et al.* [3] argue that the onset of congestion will thus limit the degree of each member in the mesh. Alternatively, the authors suggest that the degree of each member in the mesh be explicitly constrained.

4.4 Reported Performance

The performance of Narada has been extensively analyzed both through internet experiments and simulations [3–5]. In the case of internet experiments, its performance has been analyzed along the following dimensions: i) the variability in bandwidth and latency limitations of the paths to participating hosts, *i.e.*, *host heterogeneity*, ii) the distance metric used to construct and maintain the mesh and to route multicast traffic, and iii) the source sending rate.

In terms of host heterogeneity, identical internet experiments were conducted on two sets of hosts. The first set, referred to as the *primary set*, involved 13 well connected hosts whose unicast paths from source to receivers could support the source's sending rate. The second set, referred to as the *extended set*, involved 20 hosts of varying degree of connectivity. The extended set, which included all the primary set, also included bandwidth limited hosts that could not support the source's sending rate. In terms of its distance metric, Narada was implemented using latency (denoted LATENCY), bandwidth (denoted BANDWIDTH), and a dual metric involving bandwidth and latency (denoted BANDWIDTH/LATENCY). In terms of source sending rates, Narada was analyzed at sending rates of either 1.2Mbps or 2.4Mbps. In all internet experiments the performance of Narada was compared to that of: i) sequential unicast, where traffic is sequentially

unicast to all receivers, and ii) Random-Narada, where a connected mesh is randomly generated, remains fixed over time, and routing is carried out as in Narada.[2]

We first consider the results of the experiments involving the primary set of receivers. At a source sending rate of 1.2Mbps, BANDWIDTH/LATENCY performs slightly worse than the sequential unicast scheme in terms of latency but comparably to it in terms of bandwidth. In some cases, BANDWIDTH and BANDWIDTH/LATENCY in fact take advantage of internet routing pathologies and achieve higher bandwidth to some receivers than the sequential unicast transmissions. Finally, LATENCY and BANDWIDTH/LATENCY outperform BANDWIDTH and Random-Narada in terms of latency.

At a source rate of 2.4Mbps, the BANDWIDTH/LATENCY still performed slightly worse than the sequential unicast scheme in terms of latency but performed comparably to it in terms of bandwidth. LATENCY, however, performs poorly in terms of bandwidth. For the extended set of receivers and a source rate of 2.4Mbps, the performance of BANDWIDTH/LATENCY is close to that of sequential unicast and outperforms both LATENCY and BANDWIDTH; LATENCY performs poorly in terms of bandwidth and BANDWIDTH performs poorly in terms of latency.

These experiments showed that: i) Narada performs comparably to sequential unicast (stretch on the order of 1.3–1.5 and comparable bandwidth), in particular when the dual metric of bandwidth and latency is used, ii) using the dual metric involving both bandwidth and latency is important for meeting both bandwidth and latency performance requirements, and iii) in the case of BANDWIDTH/LATENCY, control traffic comprised 10–15% of all traffic, 90% of which was due to active bandwidth probes.

Chu *et al.* [3–5] also evaluated Narada through extensive simulations. These simulations involved medium-sized multicast groups, on the order of 256 receivers, over underlying networks of 1000 routers and 3000 links. In these simulations, Narada was implemented using a dual metric of bandwidth and latency and was compared to IP multicast (DVMRP) and the Random-Narada scheme. While mean latency for IP multicast was found to be relatively independent of group size, mean latency for Narada increased with group size. This is possibly due to the increase in the number of application-level hops traversed by each packet. In addition, Narada achieved lower worst-case stress than the Random-Narada scheme. However, worst-case stress on members and links was found to increase with group size. Finally, Narada's overhead, not including bandwidth probes, was found to be independent of source sending rate and to increase linearly with group size.

[2] Narada is also compared to other schemes but due to space constraints we are omitting them in this paper. The reader is referred to Ref. 3–5 for the complete performance analysis results.

Chu *et al.* [4] also analyzed the time it takes Narada to adapt to the onset of congestion on a particular overlay link. With a routing table exchange period of 10sec, Narada detects the need to adapt the overlay mesh within 20–35sec and recovers from the congested link within 20–45sec. Of course, the adaptation time scale depends heavily on the frequency with which routing tables are exchanged among neighboring members. Clearly, whether the adaptation timescale of tens of seconds is sufficient depends on the performance requirements of the application. Chu *et al.* mention that, while a higher frequency of routing updates would reduce the detection and recovery times of Narada, it would also increase the chances of the overlay becoming unstable by trying to adapt to the highly dynamic network congestion characteristics.

4.5 Evaluation

The mesh-based approach used by Narada affords several advantages. First, it decouples the membership management from the data path construction. Thus, while per-source spanning trees are constructed, a single copy of the multicast group membership is maintained.

Second, the use of per-source spanning trees mitigates the disruption of the data delivery due to congestion and failures. In the case of overlay link congestion and failure, the data delivery on only some of the per-source spanning trees may be disrupted. In the case of member failures, since each member may be at different levels of each per-source spanning tree, its failure disrupts the data delivery on the per-source spanning trees to different degrees. Thus, Narada prevents single points of failure, where the failure of a particular host causes the disruption of all multicast traffic.

Finally, per-source spanning trees distribute the traffic by different sources onto different overlay paths, thus reducing the stress sustained by overlay links. By extension, the stress sustained by the underlying links is also reduced. By constraining (either implicitly, or explicitly) the degree of each member in the mesh, Narada achieves to further reduce link stress.

Apart from the bootstrapping mechanism, Narada is also relatively robust to frequent joins. A host joins the multicast group by contacting a random set of members. Thus, provided the bootstrapping mechanism provides either a large or a random set of members to the joining host, the load of handling joins is distributed among all members of the group.

By having each member of the group maintain the complete multicast group membership, Narada is robust to the failure of a substantial percentage of links or hosts. Even if a large number of links or members fail, members can eventually discover other members that are still operational and reestablish connectivity. In this respect, Narada is self-sufficient, in the sense that connectivity

may be reestablished without resorting to an external bootstrapping mechanism. Chu *et al.* [3, 4] argue that self-sufficiency distinguishes Narada from other ALM protocols.

4.5.1 Overhead

Throughout this section, we presume that Narada constrains (either implicitly or explicitly) the mesh degree to d.

Per-Host Memory Narada's use of distance vector routing introduces considerable scalability concerns. Each host in Narada records its distance to each other member in the multicast group and the path along the mesh that affords this distance. Presuming that each of the per-source spanning trees are of considerable degree and are full, each member's routing table size is $O(N \log N)$.

In addition, each member must record whether it is the next hop from any of its neighbors to any of the multicast transmission sources. This information is used to forward multicast packets along per-source spanning trees according to the reverse-path broadcasting scheme. Thus, each member's memory requirement pertaining to the per-source spanning trees is $O(dN)$.

Per-Host Processing Each member of the multicast group must update its routing table each time it receives a routing update from one of its neighbors. The cost of such an operation in the worst-case is $O(N \log N)$, because it must check whether reaching each member through the sender of the given routing update is preferable to the current path and if so verify that the resulting path has no loops. Thus, the processing requirements of each member are, in the worst case, $O(dN \log N)$ every $T_{\text{ROUTING-UPDATES}}$ time units. Of course, routing updates in most cases can be carried out much faster, since the routes to only some members are updated as a result of each routing update.

Routing Update Overhead The scalability concerns regarding Narada due its per-host memory requirements are reinforced by the overhead associated with routing updates. As described above, each member of the multicast group exchanges its routing information with its neighbors every $T_{\text{ROUTING-UPDATES}}$ time units. Thus, each member sends $O(d)$ routing updates every $T_{\text{ROUTING-UPDATES}}$ time units. Each routing update is size $O(N \log N)$. Thus, $O(dN \log N)$ bytes every $T_{\text{ROUTING-UPDATES}}$ time units.

Although reducing the frequency of routing updates would reduce this overhead, such a reduction would slow down the convergence rate of the routing tables and the overlay's adaptation to joins, leaves, failures, and changes in network characteristics.

Analysis of Overlay Operations The process of joining the multicast group entails contacting a certain number of members and becoming their neighbor. Presuming that Narada constrains (either implicitly or explicitly) the degree of each member and that the set of members attained through the bootstrapping mechanism are reachable, the joining process involves a constant number of probes.

The cost of adding a link involves the exchange of routing information and the calculation of the distance between two hosts. This cost is incurred every T_{ADD} time units by every member. The cost of removing a link involves the exchange of routing information between two hosts. This cost is incurred every T_{DROP} time units by every member.

Partitions are repaired by periodically probing a member from the partition queue. Provided that the partition queue is non-empty, this cost is incurred every $T_{\text{PARTITION-REPAIR}}$ time units with probability $P_{\text{PARTITION-REPAIR}}$.

In summary, the process of adding and removing links and repairing partitions involves a constant number of message exchanges. The exchange of routing information does however involve the transfer of $O(N \log N)$ bytes of data.

4.5.2 Concerns and Suggestions

The effect of Narada's parameters on its performance is not well addressed in the literature [3–5]. In terms of evaluating its scalability, it is important to observe how each parameter affects the protocol's performance as the size of the multicast group increases. For instance, consider the case of partition detection and repair. In Narada, hosts suspect partitions through timeouts; that is, if the membership status of a particular host has not been updated for $\Delta_{\text{PARTITION}}$ time units, then the host is suspected of belonging to another part of a partition.

As the multicast group grows and presuming the degree of each member is constrained (either implicitly or explicitly), the number of hops between members increases. The increase in hop-count among members of the multicast group may increase the associated latency. Thus, as the group size grows and the inter-member latencies increase, members will begin falsely suspecting overlay partitions. Such suspicions will induce extensive and unwarranted partition repair probing. Clearly, this is not the intended behavior of Narada. Rather, each member x should use per-member timeouts each being proportional to x's latency to the respective member along the overlay network. A similar scheme should also be used for failure detection; that is, choosing per-member values for the timeouts (Δ_{FAILURE}) used to detect failures that are proportional to the latency to the respective member along the overlay network.

Although Narada can potentially handle a high frequency

of joins, it is unclear whether Narada's overlay can adapt fast enough afford good performance in either highly dynamic environments or large multicast group sizes. This is the case for a couple of reasons. First, the frequency with which heartbeat and routing messages are exchanges may not be increased so as to adapt quicker to the highly dynamic environment. Both Chu *et al.* [3, 4] and Banerjee *et al.* [1] have observed that increasing the frequency of heartbeat and routing messages leads to routing instability. Moreover, increasing the frequency of heartbeat and routing messages introduces additional control traffic.

Second, as the multicast group size increases, the time required by the random link addition scheme to discover efficient routes also increases. The number of candidate overlay links at any point in time is N^2 and the number of links evaluated every T_{ADD} time units is N. Thus, as the size of the multicast group grows a smaller fraction of the overlay links are probed every T_{ADD} time units. As N increases, more attempts are required to discover high utility links to be added to the mesh. For instance, consider the scenario where a host joins a video conference close to its source and that all remaining receivers are far away. Since it joins by contacting random members, it will contact members that are far and connect to the source through them. Since all members are far away, chances are that the given member will keep on probing far away members and never discover the overlay link directly to the source. Heuristics that direct Narada's search for high utility links may prove highly beneficial in terms of accelerating Narada's convergence to high quality overlays.

Another concern is the high cost of active bandwidth probes. Chu *et al.* [3, 4] observe that for conferencing applications the use of a dual metric of bandwidth and latency is highly beneficial for building a quality mesh. In their experimental results however, Chu *et al.* observed that bandwidth probes accounted for 90% of the overhead.

5 NICE

The NICE Internet Cooperative Environment (NICE) [1, 2] is an ALM protocol that employs a tree-based (hierarchical) data distribution structure. NICE arranges the members of the multicast group into a hierarchy and uses this hierarchy to disseminate multicast traffic among multicast group members. This hierarchy is constructed and maintained so as to minimize a particular performance metric, such as end-to-end latency.

We proceed by briefly describing the member hierarchy, how it is maintained, and its reported performance. We conclude by evaluating NICE's overall design and scalability and suggesting some possible improvements.

In our presentation and evaluation of NICE, we let N denote the number of multicast group members.

5.1 Hierarchy Overview

NICE arranges members of the multicast group into a hierarchy. The members at each layer (level) of the hierarchy are partitioned into clusters ranging in size from k to $3k - 1$ members, where $k \in \mathbb{N}^+$ is NICE's cluster size parameter. This partition observes the locality of the members at the particular layer; that is, members that are close together, with respect to the distance metric for which the hierarchy is being optimized, belong to the same cluster. The member that constitutes the graph-theoretic center of each cluster is considered to be the leader for the respective cluster and represents it at the higher layer of the hierarchy.[3] Thus, while the lowest layer in the hierarchy is comprised of all members of the multicast group, higher layers are comprised of progressively fewer members.

Letting L_i, for $i \in \mathbb{N}$, denote the i-th layer of the NICE hierarchy, with L_0 corresponding to the lowest layer of the hierarchy, the NICE hierarchy satisfies the following properties:

- a member belongs to only one cluster in any layer,
- if a member is present in a layer L_i, then it is also present in any lower layer; in fact, it is its cluster's leader in each such layer,
- if a host is not present in layer L_i, then it is not present in any higher layer,
- provided the multicast group is comprised of at least k members, each cluster at each layer of the hierarchy is comprised of at least k and at most $3k-1$ members,
- at each layer, cluster leaders are the graph-theoretic centers of their clusters,
- letting N be the number of multicast group members, the hierarchy is comprised of at most $\log_k N$ layers.

The structure of the hierarchy is maintained by the members in soft state. Each member stores the members in each of the clusters it belongs to (its cluster peers at each layer at which it is present), the distance estimates to all these cluster peers, and the members of its *super-cluster*. Suppose the distinct members x and y belong to the cluster X_i at some layer L_i and y is the leader of X_i. Then, the cluster X_{i+1} to which y belongs at layer L_{i+1} is the super-cluster of x. Similarly, X_{i+1} is said to be the super-cluster of X_i. Since the number of members per cluster is limited to $3k - 1$, the per-member memory required is $O(k)$ for each cluster it belongs to. Let x be a member that is present in layer L_i and not present in any higher

[3] The graph-theoretic center of a cluster corresponds to the cluster member whose maximum distance to any other member in the cluster is the minimum among all other cluster members.

layer. The memory requirement for x is $O(ki)$; the memory requirement of the member at the top of the hierarchy is $O(k \log N)$.

Control Path Cluster peers periodically exchange heartbeat messages. Such messages include the membership view of each cluster member pertaining to the given cluster; that is, it contains a list of the cluster members that are known to the sender. The heartbeat messages sent out by cluster leaders also inform the cluster members of the members comprising their super-cluster. Let $T_{\text{HEARTBEAT}}$ denote the period with which cluster members send out heartbeat messages and x be a member that is present at L_i but at no higher layer. Then, x must send out $O(ki)$ heartbeat messages every $T_{\text{HEARTBEAT}}$ time units; the member at the top of the hierarchy must send $O(k \log N)$ messages every $T_{\text{HEARTBEAT}}$ time units.

Data Path Multicast traffic is disseminated throughout the multicast group as follows. Suppose that x and y are distinct members of a particular cluster at layer L_i. If x receives a multicast packet from y, then it forwards the packet to each of the members of any other cluster it belongs to. This routing strategy forwards multicast traffic along per-source spanning trees; however, these trees may share a substantial number of overlay links.

Similarly to above, a member x that is present at L_i but at no higher layer must forward each data packet to $O(ki)$ other members; the member at the top of the hierarchy must forward each data packet to $O(k \log N)$ members. Clearly the higher in the hierarchy that a member is present, the higher its forwarding overhead. However, by amortizing the forwarding overhead, the average per-member forwarding overhead tends to $O(k)$ with increasing N.

In order to reduce the concentration of the overhead at the members present at the higher layers of the hierarchy, Banerjee *et al.* [2] sketch a scheme where the leader of each cluster delegates the responsibility of forwarding data packets. In particular, each cluster leader instructs each member in the given cluster to forward packets to members in the given cluster's super-cluster. Since clusters are comprised of at least k and at most $3k-1$ members, each cluster member is delegated the responsibility of forwarding packets to at most 3 more members. Using this delegation scheme and a more intricate data delivery path, Banerjee *et al.* [2] reduce the per-member forwarding overhead to $O(k)$.

5.2 Hierarchy Maintenance

Member Join A host x initiates the process of joining the multicast group using a bootstrap mechanism. NICE presumes that x knows of a particular host, referred to as the *rendez-vous point* (RP), through which it learns the member y at the top of the NICE hierarchy. Subsequently, x contacts y and learns the cluster peers of y at the next layer down the hierarchy. Then, x probes each such member, determines which of these members is closest, and asks this closest member for its cluster peers an the next layer down the hierarchy. x proceeds to explore successively lower layers of the hierarchy in view of finding and joining its closest L_0 layer cluster.

During the joining process, a host must query $O(k)$ members at each layer of the hierarchy. Thus, the joining process incurs an overhead of $O(k \log N)$ messages and spans a time interval of $O(\log N)$ RTTs. In view of shortening the delay is receiving multicast transmissions, the joining host successively peers with the cluster leader of each cluster whose members it queries as it successively explores lower layers of the hierarchy.

Member Leaves/Crashes Graceful leaves are carried out as follows. Prior to leaving the multicast group, the member intending to leave sends a `remove` message to its peers in each of the clusters it belongs. These messages initiate a leader selection process in each of the affected clusters. In each such cluster, each member estimates which of the peers should be the cluster's leader and a leader is elected through heartbeat message exchanges among the remaining cluster peers. In the cases when multiple leaders are selected, further heartbeat messages are used to select a single leader among them.

Once a new cluster leader is selected among the remaining cluster peers, the new cluster leader joins the higher layer by joining its super-cluster. If the new cluster leader is unsuccessful in joining its super-cluster, *e.g.*, due to stale super-cluster state, then it contacts the RP and initiates the process of joining the next highest layer of the NICE hierarchy. This process is identical to the process of a new host joining the multicast group, with the exception that the process terminates when the cluster leader discovers the appropriate cluster to join at the appropriate layer. For instance, suppose a cluster leader x at layer L_i wants to join layer L_{i+1}. It contacts the RP and begins exploring the NICE hierarchy top-down in search of the appropriate cluster to join. This joining process terminates when it discovers its closest member y at layer L_{i+1} and joins y's cluster at that layer.

When a member crashes, its peers in each of the clusters it belongs to stop receiving heartbeat messages. In each such cluster, the remaining peers initiate the process of selecting a new cluster leader as described above.

Member Migration In order to allow the hierarchy to adapt to changing network characteristics and to correct possible cluster selection errors when hosts join the hierarchy, NICE allows members to migrate between clusters as follows. Suppose x be a member that is present at layer L_i and no higher layer, X_i be the cluster to which x

belongs at layer L_i, and X_{i+1} be the super-cluster of x at layer L_{i+1}. Periodically, x estimates its distance to each of the members in X_{i+1}. If it discovers that it is closest to some member y in X_{i+1} than to the cluster-leader of X_i, then it leaves X_i and joins the layer L_i cluster of y.

Cluster Splitting and Merging Cluster leaders periodically check the size of their clusters and appropriately decide whether to split the clusters into two equally sized clusters or to merge their clusters with other clusters in their vicinity.

For instance, if a cluster leader x determines that the size of its cluster X_i at layer L_i exceeds $3k-1$, then it initiates the process of splitting X_i. Based on the pairwise distances between the cluster's peers, the cluster leader x splits X_i into two clusters such that the maximum of the radii of the two clusters is minimized. Moreover, it selects the leaders of each of the two new clusters and informs the members of the original clusters of the split and of their new leaders. Presumably, although not clearly specified in [1,2], x removes itself from any clusters it belongs to at layers higher than L_i and the two new leaders join their super-cluster at layer L_{i+1} as the leaders of the two new clusters at layer L_i.

If x determines that the size of X_i has fallen below k, then it initiates the process of merging X_i with another cluster in its vicinity. Let X_{i+1} be the layer L_{i+1} cluster to which x belongs, y be the member of X_{i+1} that is closest to x, and Y_i be the layer L_i cluster to which y belongs. x initiates the merging of X_i and Y_i by sending a cluster merge request to y and informing its peers in X_i of the merge. Upon receiving such a cluster merge request, y informs its peers in Y_i of the merge. Following the merge, x removes itself from any clusters it belongs to at layers higher than L_i.

5.3 Reported Performance

Banerjee *et al.* [1, 2] have extensively analyzed the performance of NICE using both simulations and wide-area network experiments. In their simulations, the authors compared the performance of NICE to that of Narada, under single source transmission scenarios. In their wide-area network experiments they validated their simulation results.

In summary, their simulation and experimental findings were that:

- NICE and Narada converge to trees of similar path lengths,
- the stretch achieved by NICE is comparable to that of Narada,
- the stress imposed by NICE is lower than that of Narada, especially as the multicast group size increases — for larger multicast groups, NICE converges to topologies with 25% less average stress than Narada,
- the failure recovery in both schemes is comparable,
- the overhead of NICE is much lower than that of Narada, especially when the refresh rate of Narada is increased so as to achieve comparable failure recovery to that of NICE, and
- the worst-case control overhead at members running the NICE protocol increases logarithmically with group size.

The simulation experiments of Banerjee *et al.* [1, 2] are, however, biased in favor of NICE. First, Banerjee *et al.* seem to have incorrectly implemented the Narada protocol. In their brief overview of Narada, they claim that members must exchanges heartbeat/routing updates with every other member in the multicast group. Thus, they estimate the aggregate control traffic to be $O(N^2)$, where N is the size of the multicast group. Clearly, this is not the intended behavior of Narada. As explained in Section 4, each member in Narada periodically exchanges heartbeat/routing updates with its neighbors in the overlay mesh. Chu *et al.* [3] state that a low degree overlay is indeed preferred so as to reduce overhead and stress. Moreover, the authors claim that the increase in load and congestion at high degree members will induce the overlay to reconfigure. Thus, Narada implicitly constrains the degree of the overlay mesh. Chu *et al.* concede that when the degree is unsuccessfully constrained implicitly, an explicit scheme for limiting its degree should be employed. In fact, in their simulation results, which model neither congestion nor interference from other transmissions, Chu *et al.* explicitly constrained the degree of Narada's overlay mesh within particular bounds.

It is unclear whether Banerjee *et al.* conducted a fair comparison of the overhead of NICE and Narada. This depends on whether Banerjee *et al.* implemented Narada such that members exchange heartbeat/routing messages with all members in the group. The particulars of their implementation affects also the results pertaining to the recovery time of Narada. If heartbeat/messages are exchanged using a complete graph, then members are alerted to failures potentially sooner than in the case of a bounded-degree mesh and the recovery is quicker.

5.4 Evaluation

5.4.1 Quality of Data Delivery

NICE's cluster-based hierarchy results in a data delivery path that has two highly desirable properties. First, the hierarchy guarantees that multicast packets traverse at most $O(\log N)$ application-level hops. Second, since clusters capture the underlying locality (in terms of whichever performance metric used), the application-level hops tra-

verse incrementally larger regions of the underlying topology and, thus, afford good aggregate end-to-end performance. These two properties allow NICE to scale to large multicast groups while still maintaining good application-level performance.

Unless the data forwarding responsibilities of cluster leaders are delegated, the stress subjected on underlying links by NICE's data delivery path may prohibit its use for large groups and high bandwidth applications. Consider a member x that is present at layer i and no higher layer. This member forwards data packets to the $O(ki)$ members that comprise the clusters at layers i and below to which x belongs. In the worst case, the stress sustained by the underlying links emanating from x is $O(ki)$. In the worst-case scenario, the links emanating from the member at the top of the hierarchy sustain a stress of $O(k \log N)$. Clearly, the delegation of the data forwarding responsibilities of cluster leaders, as suggested by Banerjee *et al.* [1, 2], is necessary. Using delegation, the stress sustained in the worst case by underlying links is $O(k)$.

5.4.2 Robustness

Disruption of Data Delivery Due to hierarchical structure of NICE's data delivery paths, failures at higher layers of the hierarchy disrupt the data delivery to larger sets of receivers. Moreover, since the per-source data delivery trees within the NICE overlay share all their higher layer members, failures affect to a similar degree the data stream of each source. Consider the failure of a member that it present at layer i and no higher layer. In such a scenario, packets that are forwarded to it from its peers at layer L_i do not get forwarded to any of the members in the sub-hierarchy for which it is the leader. Moreover, any packets forwarded from its cluster peers at layer L_0 don't get forwarded but to their L_0 cluster.

The delegation of data forwarding responsibilities by cluster leaders improves the robustness of the data delivery to congestion and failures. Through delegation, congestion and failures affect more per-source data delivery paths but to a lesser degree. Thus, a more graceful degradation of overall data delivery is achieved.

Failure Repair Once NICE repairs a failure, for instance, by electing new leaders to represent the particular clusters affected by the failure, the data delivery resumes as before, that is, with comparable performance characteristics. In contrast, Narada repairs failures by randomly adding links and, subsequently, gradually improving the overlay mesh by adding and removing overlay links.

Member Joins NICE's joining process constitutes a robustness and scalability concern. This process involves contacting the RP and then progressively exploring the hierarchy top-down in search of the most appropriate cluster for the new host to join. For either large multicast groups or highly dynamic environments very frequent joins may stress the RP and the higher layers of the hierarchy. Were a distance metric whose active measurement is costly, the problem would be aggravated. Further research as to how to relieve this concentration of joining overhead at the top of the hierarchy is needed.

Highly dynamic environments involving frequent joins and leaves would probably result in more frequent cluster merges and splits. In addition, more situations would arise in which newly elected cluster leaders are unable to reach any of the super-cluster members and resort to querying the RP and rejoining the hierarchy from scratch.

Frequent merges, splits, re-joins may be mitigated by adjusting the minimum and maximum cluster size bounds. Increasing the value of k would lower the chances of all members of a group failing or becoming unreachable. Increasing the maximum cluster size to $6k-1$, for instance, would reduce the frequency of both merges and splits; the resulting clusters would have roughly $3k$ members, so at least $2k$ members would have to leave or crash and $3k$ members would have to join for the cluster to merge or split once again.

5.4.3 Overhead

Per-Host Memory The memory overhead for a member that is present at layer i and no higher layer is $O(ki)$; this, includes the information pertaining to each of the i clusters it belongs to. In the worst case, the member at the top of the hierarchy has a memory requirement of $O(k \log N)$. This is a major advantage to Narada which has a memory requirement of $O(N \log N)$.

Per-Host Processing The only substantial processing performed by the members of the multicast group is the cluster split operation. Banerjee *et al.* [2] state that the processing overhead of splitting a cluster C is $O(|C|^3)$. Thus, presuming that k is relatively small, that each cluster leader initiates the process of splitting its cluster soon after it exceeds the upper size bound, and that splits do not occur that frequently, this processing overhead seems manageable.

Control Traffic The heartbeat messages that each member exchanges with all its cluster peers comprises the control traffic overhead of each member. A member sends $O(k)$ heartbeat messages every $T_{\text{HEARTBEAT}}$ time units for each cluster it belongs to. Thus, a member present at layer i and no higher layer sends $O(ki)$ heartbeat messages every $T_{\text{HEARTBEAT}}$ time units. Of course, the worst such overhead is incurred by the member at the top of the hierarchy; it sends $O(k \log N)$ heartbeat messages.

Since each such heartbeat message contains cluster membership information, it is of size $O(k)$ bytes. Thus, a

member at layer i and no higher layer sends out $O(k^2 i)$ bytes every $T_{\text{HEARTBEAT}}$ time units. Of course, the worst such overhead is incurred by the member at the top of the hierarchy; it sends $O(k^2 \log N)$ bytes every $T_{\text{HEARTBEAT}}$ time units.

5.4.4 Concerns

An important concern regarding the NICE hierarchy is that the migration of members from cluster to cluster is insufficient to correct for inaccurate placement and changes in the network characteristics. For example, consider a host x that due to packet losses during its joining process, is erroneously misled down the wrong branch in the hierarchy and joins a cluster that is locally optimal but globally sub-optimal. For instance, suppose that x joins the cluster X_0 at layer L_0, y is the leader of X_0, and X_1 is the cluster of y at layer L_1. If indeed x joins X_0 is error, it is possible that x is closer to y than to any other member in X_1, but that it closer to the cluster leader of another L_0 layer cluster somewhere else in the hierarchy. In this scenario, x is incapable of migrating and joining its globally optimal cluster.

Chu *et al.* [3,4] have observed that a dual metric involving both bandwidth and latency is crucial for achieving good performance for conferencing applications. Using a dual metric of bandwidth and latency as the distance metric in the NICE hierarchy is questionable. Recall that in order for a host to join the hierarchy, it successively explores the layers of the hierarchy is search of the lowest layer cluster that is closest to it. During this exploration, it performs $O(k \log N)$ distance probes. Thus, if a dual metric were used, then each join operation would involve $O(k \log N)$ high overhead bandwidth probes [3,4]. A solution to this overhead problem might be to have hosts join the hierarchy based on a latency metric and then migrate using the dual metric. However, as explained above, this might result in hosts joining sub-optimal clusters and getting stuck with poor performance.

Furthermore, NICE is dependent on the RP for recovering from situations in which newly elected cluster leaders fail to join their super-cluster. Thus, NICE doesn't satisfy the self-sufficiency requirement put forth by Chu *et al.* [3]; self-sufficiency is the property that once a set of hosts have joined the multicast group, the ALM protocol should be able to recover from failures and reestablish data delivery without relying on out-of-band mechanisms.

5.4.5 Suggestions

A plausible solution to the problem of insufficient migration is for each member to maintain a list of all leaders under which it resides in the hierarchy. This data can maintained by having newly elected cluster leaders inform all the members in their respective region of their election. Then, periodically each member would check whether it is misplaced with respect to the cluster of its leader at a randomly chosen layer of the hierarchy. Of course, such probes would have to be less frequent for higher layers of the hierarchy. If at any point in time a member were to determine, through a probe at a cluster X_i at layer L_i, that it is misplaced, then it would migrate at the region by jump-starting a joining process from the cluster X_i. Although this is a plausible scheme, it introduces additional overhead. First, each member would have to maintain a list of all its higher layer leaders. This amounts to a memory requirement of $O(\log N)$. In addition, this information would probably have to be exchanges within L_0 clusters so as to ensure consistency. Thus, a member at layer i and no higher layer would have to send out $O(ki \log N)$ bytes every $T_{\text{HEARTBEAT}}$ time units. Of course, the worst such overhead is incurred by the member at the top of the hierarchy; it sends $O(k \log^2 N)$ every $T_{\text{HEARTBEAT}}$ time units.

All of the aforementioned concerns can potentially be addressed by maintaining partial group membership information as is done in [6, 7]. In SCAMP [7], each member maintains a list of $(c+1) \log N$ members, where c is a parameter, and even when a fraction $c/(c+1)$ of the underlying links failing, this membership information guarantees connectivity through gossiping. In *lpbcast* [6], each member maintains a fixed-size view of the membership. Using random gossiping, the membership views of the members are exchanged and continuously modified. We propose the design of a similar scheme, where each member maintains a partial view of the membership of size $O(\log N)$ which is continuously exchanged and modified as in *lpbcast*.

A member could use this membership information to periodically probe remote areas of the hierarchy in view of discovering a more appropriate cluster to join. Since the partial membership view would constantly be changing, different regions of the hierarchy would be randomly explored. With migration in tact, dual metrics could potentially also be handled. As described above, hosts would join based on latency alone and slowly migrate based on the dual metric. Finally, in the event of a hierarchy partition, this partial membership information would also be useful in discovering other regions of a hierarchy; thus, rendering NICE self-sufficient. Clearly, more work is needed to see if such a scheme is feasible, provides robustness and connectivity guarantees similar to those claimed in [6, 7], and indeed produces viable solutions to the concerns regarding the NICE protocol.

6 Large-Scale Multicast Using *i3* (*i3*-mcast)

Lakshminarayanan *et al.* [8] are currently designing an implementation of a large-scale multicast service using

the Internet Indirection Infrastructure (*i3*) [11]. In this section, we present and evaluate the current version of their ALM implementation, which we refer to as *i3*-MCAST. We begin by describing the functionality of the *i3* system and conclude by describing and evaluating *i3*-MCAST.

6.1 The *i3* System

i3 is an overlay-based system that enables the implementation of a collection of communication services. The *i3* system involves an overlay network comprised of a dynamic set of *i3* servers. Loosely speaking, clients interact with the *i3* system by: i) inserting *triggers* into the *i3* system, ii) removing *triggers* from the *i3* system, and iii) sending packets addressed to *i3* identifiers. Triggers correspond to forwarding instructions; that is, a trigger instructs the *i3* system as to how to forward packets addressed to a particular *i3* identifier (or set of *i3* identifiers). Each *i3* server is responsible for: i) storing in soft-state all triggers pertaining to a particular subset of the identifier space, and ii) forwarding all packets addressed to this particular subset of the *i3* identifier space according to the triggers currently stored. A client inserts a trigger by submitting it to any of the *i3* servers. The *i3* system forwards this trigger along the *i3* overlay to the *i3* server that is responsible for storing it. This server inserts the trigger by storing it in soft-state. Triggers are removed from the *i3* system analogously. All triggers stored within the *i3* system must periodically be refreshed; otherwise, they expire and are discarded. A client sends a packet by submitting it to any of the *i3* servers. Subsequently, the *i3* system forwards this trigger along the *i3* overlay to the *i3* server that is responsible for forwarding it. In turn, this server forwards the packet according to the triggers pertaining to the *i3* identifier to which the packet is addressed.

In our presentation of *i3*, we let N_S denote the number of *i3* servers (Chord nodes) comprising the *i3* system.

6.1.1 Packets and Triggers

In their simplest form, packets are of the form $\langle$*i3*-id, *data*$\rangle$; that is, pairs of *i3* identifiers and data (payload) of the packet. In their simplest form, triggers are of the form $\langle$*i3*-id, IP-addr$\rangle$; that is, pairs of *i3* identifiers and IP addresses.[4] A packet submitted to the *i3* system is forwarded based on its *i3* identifier to the *i3* server responsible for storing all triggers pertaining to the given *i3* identifier. Then, the given server forwards the packet according to any trigger whose identifier *matches* the identifier of the packet. If no such trigger exists, then the packet is discarded.

[4] The IP address of a trigger may also include a particular port designation.

The *i3* system uses an inexact identifier matching strategy. Letting m denote the bit-length of the *i3* identifiers, the *i3* system introduces an *exact match threshold* of k bits, where $k < m$. A trigger identifier id_t is said to match a packet identifier id_p if and only if: i) id_t and id_p have a prefix match of at least k bits, and ii) no other trigger in the *i3* system has a longer identifier prefix match with id_p than id_t. The exact match threshold k is presumed to be chosen large enough such that the probability that two randomly chosen identifiers match is negligible.

Suppose that a packet $\langle id_p, data\rangle$ is submitted to the *i3* system and forwarded along the *i3* overlay to the server s that is responsible for handling it. If s maintains a trigger $\langle id_t, a\rangle$ whose identifier id_t matches id_p, then it replaces the *i3* identifier of the packet with the IP address a and forwards the packet using IP. In the cases when multiple triggers match a packet's identifier, the packet is copied and forwarded to multiple IP addresses.

This indirection scheme can be used by a client h_1 to send packets to a client h_2 as follows: client h_2 inserts a trigger $\langle id_2, a_2\rangle$, where id_2 is some *i3* identifier known to h_1 and a_2 is its IP address, and client h_1 simply sends packets addressed to the *i3* identifier i_2. In effect, the server handling the packets addressed to the *i3* identifier i_2 (or, abstractly, the identifier itself) serves as the communication *rendez-vous* point.

Forwarding Preferences: The $m-k$ least significant *i3* identifier bits may be used to encode packet forwarding preferences (or trigger matching preferences). For example, these bits can be used to distribute client requests to a collection of web servers or to direct client requests to web servers that are geographically close to the respective clients. The former is achieved as follows. Clients address requests to *i3* identifiers whose $m-k$ least significant bits are chosen at random and web servers insert triggers with *i3* identifiers whose $m-k$ least significant bits are chosen at random. Thus, client requests will be forwarded to the web server whose trigger *i3* identifier shares a longer prefix with the *i3* identifier of the request.

The latter is achieved by having both clients and web servers encode their location into the $m-k$ least significant bits (presuming this encoding is hierarchical in the sense that a longer prefix match implies that clients and web servers are closer, either geographically or in terms of latency).

Trigger Chains: The *i3* system supports additional levels of indirection by allowing clients to insert triggers of the form $\langle$*i3*-id, *i3*-id$\rangle$. For example, suppose that a client sends a packet $\langle id_1, data\rangle$ to any *i3* server. This packet is forwarded along the overlay to the *i3* server s_1 responsible for handling packets addressed to id_1. Moreover, suppose that a trigger of the form $\langle id_1, id_2\rangle$ is stored at s_1. Since this trigger matches the packet $\langle id_1, data\rangle$,

the *i3* identifier of the packet is replaced with the destination *i3* identifier id_2 of the trigger and the packet is forwarded once again. In this case however, the packet is forwarded along the overlay to the *i3* server s_2 that is responsible for handling packets addressed to id_2.

Such triggers allow clients to set up complex packet forwarding chains, such as the large-scale multicast dissemination trees suggested by Lakshminarayanan *et al.* [8].

Identifier Stacks: The *i3* system also supports packet and trigger identifier stacks. In their most general form, packets are of the form $\langle \mathrm{id}_{stack}, data \rangle$ — pairs involving an identifier stack and a payload — and triggers are of the form $\langle i3\text{-id}, \mathrm{id}_{stack} \rangle$ — pairs involving an *i3* identifier and an identifier stack. An identifier stack corresponds to a list of either *i3* identifiers or IP addresses.

A packet is forwarded by the *i3* system according to the identifier on the top of the packet's identifier stack. If this identifier is an IP address, then the packet is forwarded to particular IP address through IP. A client processes a packet addressed to an identifier stack either by ignoring the identifier stack and simply delivering the packet to the application or by popping the stack, processing the packet, and subsequently sending another packet (possibly containing the results of processing the packet received) addressed to the remainder of the identifier stack. The choice as to how such packets are processed may be determined by either the clients or additional flags within the packet headers that dictate how each packet should be processed by the client.

If the identifier on the top of a packet's identifier stack is an *i3* identifier, then the *i3* system forwards the packet to the server responsible for handling packets addressed to the particular *i3* identifier. For any matching trigger, this server pops the packet's identifier stack, pushes the trigger's destination identifier (or, identifier stack) onto the stack, and once again forwards the packet with the updated identifier stack once again. For example, suppose that the triggers $\langle id_1, id_3 | a_1 \rangle$ and $\langle id_1, a_2 \rangle$ are stored at the *i3* server s and that s receives a packet $\langle id_1 | id_2, data \rangle$. Then, the server s forwards the following packets $\langle id_3 | a_1 | id_2, data \rangle$ and $\langle a_2 | id_2, data \rangle$ to the server responsible for id_3 and to the client at a_2, respectively.

The use of identifier stacks enables clients to implement several communication primitives, such as *service composition* and *heterogeneous multicast*. Service composition is implemented by having the sender address packets to an identifier stack. Each identifier in the packet's identifier stack results in the processing of the packet and the subsequent forwarding of the result of such processing. For example, in order for a client to send an HTML web page to a wireless client, it may first forward the packet to a wireless application protocol gateway that translates HTML to WML (simplified mark-up language for wireless devices) prior to delivering the packet to the wireless client. This can easily be done using the *i3* system by addressing such packets to an identifier stack $id_{HTMl\text{-}to\text{-}WML} | id_{wireless-client}$, where $id_{HTMl\text{-}to\text{-}WML}$ is the rendez-vous *i3* identifier for the wireless application protocol gateway that translates HTML to WML and $id_{wireless-client}$ is the rendez-vous *i3* identifier for the wireless client.

Heterogeneous multicast refers to the service where clients with different streaming capacities simultaneously subscribe to a particular multicast transmission. For example, consider a wireless client that wants to subscribe to a high bandwidth MPEG stream. This client may redirect the MPEG stream through an MPEG to H.263 transcoder by inserting a trigger of the form $\langle id_{\mathrm{MPEG}}, id_{MPEG-to-H.263} | a \rangle$, where id_{MPEG} is the *i3* identifier for the high bandwidth MPEG stream, $id_{MPEG-to-H.263}$ is the *i3* identifier for an MPEG to H.263 transcoder, and a is the IP address of the wireless client.

6.1.2 Routing Using Chord

The Chord protocol is used to route packets and triggers to the *i3* servers that are responsible for forwarding and storing them, respectively. Chord is efficient, robust, and scalable. Using the Chord protocol, each *i3* server maintains routing information regarding $O(\log N_S)$ other servers and routes packets and triggers to the appropriate *i3* server within $O(\log N_S)$ steps. Moreover, the overhead in maintaining the routing information when *i3* servers join or leave the Chord system involves $O(\log^2 N_S)$ messages. The reader is referred to [12] for an extensive analysis of Chord's performance and robustness analysis.

Although Stoica *et al.* [11] use Chord in their presentation of the *i3* system, any distributed lookup protocol similar to Chord may be used. Of course, it is important that such a protocol be efficient, robust, and scalable.

6.1.3 Routing Efficiency

Although routing packets and inserting/removing triggers through the overlay network using the Chord protocol is efficient, it is typically less efficient than routing directly to the appropriate server using IP. The *i3* system addresses this inefficiency by exposing to client originally sending the packet or inserting/removing the trigger the IP address of the *i3* server that is responsible for handling and storing the packet or trigger, respectively. Once the *i3* server's IP address is cached, the client uses IP to send subsequent packets or triggers pertaining to the particular *i3* identifier to the appropriate server. If, subsequently, another server takes responsibility of the particular *i3* identifier, packets and triggers will be routed to the new server using Chord and the IP address of the new server will be cached.

Routing efficiency may also be improved by having clients probe the overlay network in search of a server that is close-by in terms of RTT latency. For instance, a client c with IP address a may probe the *i3* system as follows. It selects a random *i3* identifier id, then it inserts a trigger of the form $\langle id, a \rangle$, sends a dummy packet addressed to id, and measures the packets RTT latency. Presuming that the mapping of *i3* identifiers onto servers is relatively stable, this operation need only be done off-line and infrequently.

6.1.4 Robustness

In terms of routing packets and inserting/removing triggers, the *i3* system inherits its robustness to node failures from the underlying Chord location protocol. Since triggers are maintained by the *i3* system as soft-state, it is also robust. This is because all triggers must periodically be refreshed/reinserted by clients. Thus, even if a server storing a particular trigger fails, the trigger will be reinserted into the *i3* system the next time its client refreshes it.

Trigger re-insertion is not however immediate. One approach to improve the robustness of the *i3* system to the interruption of service due to the delay in re-inserting lost triggers is replication. Replication can be done by either the clients or the overlay network. In the former solution, receivers may generate multiple triggers (hopefully stored on distinct servers) and senders can address packets to an identifier stack involving the *i3* identifier of each such trigger. If the trigger corresponding to the identifier on the top of the stack is lost, then the packet is forwarded to the subsequent *i3* identifiers in the stack in search of the trigger replicas.

6.1.5 Relieving Hot-Spots

Replication can also be employed to avoid or relieve hot-spots in the underlying overlay network. When a server becomes overwhelmed with traffic, it may duplicate triggers pertaining to particular sets of *i3* identifiers to other servers and, thus, distribute the load. Of course, this replication must ensure that: i) the triggers are copied to a server that will process a substantial amount of the packets routed to the swamped server, and ii) all triggers that have a prefix match of k bits must be replicated so that the new server can match triggers correctly.

6.1.6 Scalability

The scalability of the *i3* system involves both the scalability of the underlying Chord service and that of the storage requirements introduced by the *i3* system. Chord comprises a highly scalable overlay-based lookup protocol. In terms of the storage requirements when implementing communication services using the *i3* system, each point-to-point flow involves two triggers. However, only a subset of the triggers are stored by each *i3* server. Presuming that the *i3* identifiers are uniformly distributed, the number of point-to-point flows is n, each server is on average required to store n/N_S triggers. Of course, the implementation of more complex communication services using the *i3* system may require the storage of a larger number of triggers.

6.2 The *i3*-mcast Protocol

i3-MCAST uses *i3* identifier chains to construct a source-specific multicast spanning tree of the members of the multicast group within the *i3* infrastructure. The *i3*-MCAST protocol can be implemented in either a strict peer-to-peer sense, or a client-server sense. In the strict peer-to-peer sense, the multicast group members implement the *i3* system and the underlying lookup protocol. In the client-server sense, the multicast group members are clients to the *i3* service, which is provided as a service by some other entity. In our presentation and evaluation, we let N denote the number of multicast group members and N_S denote the number of *i3* servers (Chord nodes). The strictly peer-to-peer setting is obtained by letting $N = N_S$ and, of course, realizing that the members inherit the overhead responsibilities of the *i3* servers.

Multicast Spanning Tree *i3* identifiers and triggers play the role of the nodes and the edges, respectively, of the multicast spanning tree. A set of *i3* identifiers, one corresponding to each member of the multicast group, comprise the nodes of the multicast spanning tree. Each member h_k subscribes to its distinct *i3* identifier id_k by inserting (and continuously refreshing) a trigger $\langle id_k, h_k \rangle$ into the *i3* system. A set of triggers comprise the edges between the nodes of the multicast spanning tree. For instance, an edge between node id_{k_1} and id_{k_2} is established by inserting (and continuously refreshing) the trigger $\langle id_{k_1}, id_{k_2} \rangle$.

The identifier id_r of the root node comprises the multicast group address; that is, the identifier to which any multicast traffic should be addressed. The host h_r subscribing to the root node of the multicast tree is the source since the multicast tree is constructed such that the latency between h_r and any other member of the multicast group is minimized.

Lakshminarayanan *et al.* limit the degree of the multicast tree by imposing a limit $D + 1$ on the number of triggers stored for any given *i3* identifier — one trigger connecting the node to its member and up to D triggers connecting the node to up to D other nodes (children). We say that a node is *joinable* if it has less than D children nodes and *full* if it already has D children nodes.

The members of the multicast group maintain two addi-

tional multicast groups per node. Let *jHash* and *fHash* be hashes from the *i3* identifier space to itself. For each node id, the multicast groups with *i3* identifiers $jHash(id)$ and $fHash(id)$ consist of the hosts that are directly connected to the joinable and full, respectively, children nodes of id. These multicast groups are flat in the sense that the server handling their *i3* identifier stores triggers pointing to each of the members of the group.

Member Joins A new host h_j joins the multicast group by choosing a random *i3* identifier id_j (preferably one that is handled by an *i3* server that is close-by), subscribing to it, and attaching it to a node of the multicast spanning tree by inserting the appropriate trigger. The node onto which to attach id_j is chosen so as to afford good performance to h_j. In particular, the joining host h_j traverses the multicast tree in search of a joinable node that affords good latency between the root host h_r and itself along the multicast tree.

This traversal starts at the root node id_r and proceeds down the multicast tree according to a branch-and-bound scheme. At any point in time during this traversal, the joining host h_j records the joinable node id^* that up to that point in the traversal affords the minimum latency between h_r and h_j; thus, up to that point in the traversal, id^* is the best candidate node to which the joining node should attach its node id_j.

The joining host h_j traverses the multicast tree top-down visiting one node per-level starting at the root node id_r. For each node id it visits, it performs several tasks. First, it estimates the distance from h_r to itself along the multicast tree through each of the joinable children nodes of id. This is done by sending a JOIN-PROBE control packet to the multicast group $jHash(id)$. Any host that receives such a packet responds to h_j. Such responses include relative timing information that enable h_j to determine which of the joinable nodes of id affords the minimum latency from h_r to itself. Let id_{jmin} denote the node among the joinable nodes of id that affords the minimum such latency. If the latency afforded by id_{jmin} is less than that afforded by id^*, then id_{jmin} is recorded as the best candidate node so far; that is, id^* is reset to id_{jmin}.

Subsequently, using the multicast group $fHash(id)$, h_j determines which of the full children nodes of id affords the minimum latency from h_r to itself. Let id_{fmin} denote the node among the full nodes of id that affords the minimum such latency. If the latency afforded by id_{fmin} is less than that afforded by id^*, then h_j continues the traversal of the tree by exploring the subtree rooted at id_{fmin} in the same fashion. Otherwise, h_j ceases its traversal.

Once h_j ceases its traversal, subscribes to id_j, and attaches id_j to the multicast tree at the node id^* by inserting the trigger $\langle id_j, h_j \rangle$ and $\langle id^*, id_j \rangle$, respectively. The trigger $\langle id_j, h_j \rangle$ instructs id_j to forward the multicast packets to h_j and the trigger $\langle id^*, id_j \rangle$ instructs id^* to forward the multicast packets to id_j.

Finally, h_j also updates the joinable and full multicast groups of id^*. h_j determines if id_j is joinable or full by inserting (and, subsequently, immediately removing) a dummy trigger $\langle id_j, id_{dummy} \rangle$. If this trigger insertion is successful, which indicates that id_j is joinable, then h_j joins the joinable group of id^* by inserting the trigger $\langle jHash(id^*), h_j \rangle$. Otherwise, h_j joins the full group of id^* by inserting the trigger $\langle fHash(id^*), h_j \rangle$. Clearly, unless by chance another member of the multicast group is also subscribed to id_j, h_j will join the joinable multicast group of id^*.

Multicast Tree Maintenance Since triggers are stored as soft-state at the *i3* servers, they must periodically be refreshed. This includes both the triggers pertaining to the multicast tree and those pertaining to the joinable and full multicast groups.

In order for each member of the multicast group to maintain the data path from the root node to itself, it must periodically refresh all the triggers comprising the trigger chain from the root node to itself. Having clients refresh all the triggers comprising their respective trigger chains is not scalable, because triggers high up in the dissemination tree, which are shared by numerous receivers, would get refreshed by a multitude of clients.

Lakshminarayanan *et al.* propose two techniques for reducing the number of times each trigger gets refreshed within each refresh period. First, a client uses randomization to vary the points in time at which it refreshes each of the triggers in its trigger chain. Thus, not all members that share a particular trigger refresh the given trigger all at once. Second, when a client refreshes a trigger $\langle id_{l_1}, id_{l_2} \rangle$, it also sends out a REFRESH-ACK control packet addressed to id_{l_2}. This control packet is disseminated throughout the subtree rooted at id_{l_2} and suppresses any other refresh messages for the given trigger. In conjunction, these techniques reduce the number of times a trigger is refreshed within each refresh period. More sophisticated randomization schemes can further reduce duplicate refreshing of triggers [8].

In addition to refreshing the trigger chain from the root node to itself, a host h_k must also maintain its membership to either the joinable or full multicast groups of its parent node. Thus, it periodically determines whether id_k is joinable or full, as described above, and re-subscribes to the appropriate multicast group.

If at any point in time all the members of the joinable and full multicast groups of a node leave the multicast group or crash, then hosts that are joining the group may be prevented from either exploring promising branches of the multicast tree or joining altogether. Such scenarios are avoided by having members periodically probe their parent nodes to determine whether any member is directly attached to it. If not, then the member migrates

to its parent node and joins the joinable or full multicast group of the node that was its grandparent prior to migrating.

Multiple Sources The multicast tree construction process described above attempts to minimize the latency from the root host h_r to each of the other members of the multicast group. Although this results in a source-specific multicast tree, any member h of the multicast group may send packets to the multicast group by addressing them to id_r. Of course, the latency of multicast traffic from members other than the root host will incur the additional latency of reaching the root node.

6.3 Evaluation and Suggestions

In our analysis of *i3*-MCAST, we presume that the multicast tree constructed by *i3*-MCAST is at all times full and, thus, of depth $O(\log N)$.

6.3.1 Quality of Data Delivery

Multicast Path Length Multicast packets are forwarded by *i3* to the root of the *i3*-MCAST multicast tree, along the tree, and finally to the multicast group members. The forwarding of a packet to the root node and between nodes of the multicast tree correspond to Chord lookups. Thus, they involve in most cases at most $O(\log N_S)$ application-level hops [12]. Each node-to-member edge in the tree corresponds to a single application-level hop. Thus, the transmission of each multicast packet involves $O(\log N \log N_S)$ application-level hops. Thus, each packet multicast using *i3*-MCAST incurs more application level hops than in NICE.

Stoica *et al.* [11] suggest that hosts should cache the IP address of the *i3* server that handles the leading packet of a stream of packets addressed to a particular *i3* identifier and, subsequently, submit the later packets to the appropriate *i3* server directly. Thus, the $O(\log N_S)$ application-level hops incurred by a Chord lookup may be avoided for the later packets within the stream.

Using this caching scheme, the sources of a multicast transmission can avoid the $O(\log N_S)$ application-level hops involved in forwarding packets to the root node of the *i3*-MCAST multicast tree. This caching idea can be potentially employed within the *i3* system itself. For instance, consider an *i3* server x storing a trigger of the form $\langle id_1, id_2 \rangle$. By caching the IP address of the *i3* server y responsible for handling the *i3* identifier id_2, subsequent packets matching id_1 can be sent directly to y, rather than incurring a Chord lookup involving $O(\log N_S)$ application-level hops. Thus, caching all edges in the multicast tree successfully would reduce the number of application-level hops required to deliver multicast packets from $O(\log N \log N_S)$ to $O(\log N)$.

Alternatively, other techniques for reducing the latency of Chord lookups, such as those presented in Section 6.1.3, may be required so as to match the performance of either Narada or NICE.

Stress We estimate the stress that *i3*-MCAST imposes on the underlying network by calculating the number of times each Chord node must forward the same packet; this should be an upper bound on the stress sustained from the network links emanating from given Chord node. Each multicast transmission is routed to each node of the multicast tree using a Chord lookup and to each member of the multicast group along an application-level hop. Thus, the number of application-level messages involved in the multicast transmission of each packet is $O(N \log N_S)$ (or, $O(N)$ using caching, if effective). Presuming a well balanced Chord system, the number of application-level messages sent by each Chord node is $O((N \log N_S)/N_S)$ (or, $O(N/N_S)$ using caching, if effective). Clearly, unless our proposed caching scheme is effective, the stress sustained by underlying links may prohibit the use of *i3*-MCAST for high bandwidth applications.

Multicast Tree Concerns Several issues arise concerning the process with which the *i3*-MCAST multicast tree is constructed. First, a new host determines where to join the multicast tree based on the latency to the source host. In a scenario involving multiple sources, this approach gives a latency advantage to the members close to the root node of the multicast tree.

Second, the branch-and-bound search scheme used during the joining process is not exhaustive and may result in the construction of a sub-optimal multicast tree. This could be either an oversight (since this work is still in progress) or a conscious attempt to reduce the cost and duration of the joining process. The branch-and-bound traversal, as described in [8], explores only the most promising full node at each level. Thus, a promising full node may attract joining nodes down a branch that is less favorable than others. It is worth evaluating the trade-off between joining cost and multicast tree quality when using a simplistic versus a full-fledged branch-and-bound search during the joining process.

Finally, the quality of the multicast tree depends on the order in which hosts join the multicast group. Consider for instance the scenario where a number of hosts that are far away from the source join the multicast tree and fill up the first level nodes. Then, suppose that a host next to the source joins the multicast group. Since the first level is full, it is forced to join further down within the multicast tree. Thus, although the last host is very close to the source, multicast packets are forwarded to the far away hosts and back. Clearly, it would be beneficial to devise a scheme with which hosts can join higher up in

the tree by pushing other nodes further down.

In summary, it is debatable whether the multicast tree construction of Lakshminarayanan *et al.* [8] introduces enough locality to afford comparable performance to Narada or NICE. In the worst case, the dissemination of multicast packets would involve $O(\log N \log N_S)$ application-level hops, each of whose latencies may be arbitrary.

6.3.2 Robustness

The main advantage of *i3*-MCAST is that, by taking advantage of layering, it inherits its robustness from the *i3* system (which in turn inherits it from Chord, or whichever distributed lookup protocol is used). Since the triggers comprising the multicast tree are stored in soft-state and are periodically refreshed, *i3*-MCAST is highly robust to failures. Once the underlying lookup protocol has recovered from a failure, the multicast tree is restored when the triggers comprising it are refreshed.

Of course, failures may interrupt the multicast transmission to particular subtrees of the *i3*-MCAST multicast tree until the appropriate triggers are refreshed, possibly as long as T_{REFRESH} time units. This interruption may also be compounded by staggered failures. The replication-based scheme proposed by Stoica *et al.* [11] and presented in Section 6.1.4 may mitigate such interruptions.

Since each member of the multicast group is responsible for refreshing all the triggers comprising its trigger chain, the departure of members does not disrupt the data delivery. On the other hand, the departure of *i3* servers (Chord nodes) may temporarily disrupt the data delivery until Chord manages to redistribute the responsibility of handling lookups. Of course, in the strictly peer-to-peer setting member leaves correspond to server leaves and may thus temporarily disrupt data delivery.

6.3.3 Overhead

The overhead of *i3*-MCAST involves the overhead associated with the underlying lookup protocol and the construction and maintenance of the multicast tree.

Overhead of Lookup Protocol The overhead of the underlying lookup protocol depends on which such protocol is used. In the case of Chord, the memory requirements are $O(\log N_S)$ memory and the cost of nodes joining and leaving is, with high probability, no more than $O(\log^2 N_S)$ messages.

Each node of the multicast tree is implemented by at most $2D + 1$; $D + 1$ triggers implement its edges to its multicast group member and its children nodes and D triggers implement its joinable and full multicast groups. Thus, $O(DN)$ triggers comprise the multicast tree. Presuming a well balanced Chord system, the memory requirements at each node are $O(DN/N_S)$. Of course, in a peer-to-peer setting, the average number of triggers stored per-node is $O(D)$.

Trigger Refreshing Since all triggers are stored at the *i3* servers as soft-state, they are periodically refreshed. Each multicast group member must refresh: i) the trigger attaching it its node in the multicast tree, ii) the trigger establishing its membership to either the joinable or the full multicast group of its parent node, and iii) all the triggers comprising its trigger chain from the root to its own node.

The calculation of the expected number of triggers refreshed by each member depends on the probabilistic scheme used to limit the number of members refreshing each edge of the multicast tree. Nevertheless, we estimate the aggregate cost of refreshing triggers by presuming that the number of times each trigger gets refreshed during each refresh period is bounded by a constant. Thus, the number of refresh messages are $O(DN)$; recall, the *i3*-MCAST multicast tree is comprised of at most $O(DN)$ triggers. Thus, the number of application-level messages is $O(DN \log N_S)$ (or, $O(DN)$ using caching).

Suppression of Trigger Refreshing Each time a member refreshes one of its triggers, it also sends a REFRESH-ACK control packet to suppress other members from refreshing the same trigger during the particular refresh period. Such a packet is disseminated throughout the subtree emanating from the edge established by the particular trigger. At any level i of the multicast tree, there are D^i nodes each of which has D node-to-node edges (triggers). A REFRESH-ACK packet for each of these triggers is disseminated along $D^{\log_D N - i} - 1$ node-to-node edges and $D^{\log_D N - i}$ node-to-member edges. Node-to-node edges involve $O(\log N_S)$ application-level hops and node-to-member edges involve single application-level hops. Thus, the total number of application-level messages sent for suppression purposes is $\sum_{i=0}^{\log_D N} D^{i+1} \left[\left(D^{\log_D N - i} - 1 \right) \log N_S + D^{\log_D N - i} \right] = O(DN \log N \log N_S)$. In a well balanced Chord system, each node would have to send $O((DN \log N \log N_S)/N_S)$ application-level messages ($O(D \log^2 N)$ in a strictly peer-to-peer setting).

Presuming that the proposed caching scheme if effective, the total number of application-level messages sent for suppression purposes is reduced to $\sum_{i=0}^{\log_D N} D^{i+1} \left(2D^{\log_D N - i} - 1 \right) = O(DN \log N)$. Again, in a well balanced Chord system, each node would have to send $O((DN \log N)/N_S)$ application-level messages ($O(D \log N)$ in a strictly peer-to-peer setting).

This overhead is substantial, especially since it is incurred during each refresh period T_{REFRESH}. We proceed by sketching an alternative scheme for refreshing the triggers comprising the *i3*-MCAST multicast tree. Instead of

requiring each member to refresh each trigger in its trigger chain from the root to its own node, we require it to simply refresh the edge from its parent to its own node. Suppose that a member h_k is attached to the multicast tree at the node id_k and that the parent node of id_k is id'_k. We propose that h_k be responsible for refreshing only the edge from id'_k to id_k, *i.e.*, the trigger $\langle id'_k, id_k \rangle$.

Using this scheme, each member relies on its ancestor members to refresh the prefix of its trigger chain leading to its parent node. Member failures could thus break the chain and partition the tree. To prevent such partitions, we also require that each member send heartbeat messages to its children members; the members that are subscribing to either its joinable or full multicast groups. If at any point in time a member ceases to receive heartbeat messages from its parent member, then it determines whether its parent has either left the multicast group or crashed by probing its parent node (in an attempt to migrate upward in the tree). If the parent node indicates that no member is directly attached to it (*i.e.*, that the parent member has either left the multicast group or crashed), the child member migrates upward in the tree and informs its prior children members of its migration.

Our proposed refreshing scheme involves: i) $O(N \log N_S)$ application-level messages (or, $O(N)$ using caching) to refresh the triggers comprising the multicast group; each member must refresh the trigger attaching it to its node in the multicast tree, the trigger connecting its node to its parent node, and the trigger establishing its membership to either the joinable or the full multicast group of its parent node, and ii) $O(DN \log N_S)$ application-level messages (or, $O(DN)$ using caching, if effective) for each member to send heartbeat messages to its D children members. The second count of messages replaces the $O(DN \log N \log N_S)$ (or, $O(DN \log N)$ using caching, if effective) application-level messages used to suppress duplicate trigger refreshes. Thus, our proposed scheme reduces the number of application-level messages for suppression by a factor of $\log N$.

Of course, a more rigorous correctness analysis of the this alternative scheme for refreshing the *i3*-MCAST multicast tree triggers would need to be conducted. If such a scheme were to fail in refreshing the appropriate trigger chains in highly dynamic environments, then potentially a combination of the two schemes would work. When the system is stable the proposed scheme is used and when failures are detected members revert to refreshing all their trigger chains.

Per-Host Memory Each member of the multicast group is in charge of refreshing the complete trigger chain from the root node to itself. Since the depth of the *i3*-MCAST multicast tree is $O(\log N)$, its memory requirements are $O(\log N)$.

6.3.4 Extensibility

Since *i3*-MCAST is implemented using the *i3* system, it can easily be extended to provide other services, such as service composition and heterogeneous multicast (presented in Section 6.1.1). Lakshminarayanan *et al.* [8] describe how the *i3*-MCAST system can be extended using additional *i3* system functionality so as to provide the reliable multicast service. Thus, *i3*-MCAST affords the added advantage of easy extensibility.

7 Choosing an ALM Protocol

Given a particular application, the choice of ALM protocol depends on the following factors:

Transmission Properties Clearly, each application has distinct transmission property requirements. For example, a conferencing application requires low latency and high bandwidth. As we have seen, some protocols may be able to cater to multiple application-level performance metrics easier than others. For instance, while Narada has been implemented using a dual metric of bandwidth and latency, a dual metric is trickier to support in NICE and *i3*-MCAST due to their costlier join operations.

Robustness The degree to which the ALM protocol is robust to congestion and failures may also affect the decision as to which protocol to choose. This also depends on the environment within which an application is used. For instance, an application that either operates within a highly dynamic environment or interacts with heterogeneous clients, would require an ALM protocol that is highly robust. For example, consider a highly dynamic environment where host joins, leaves, and failures may occur in bursts. In the case of joins, while Narada distributes the load of a burst of join operations, the joining members may load the members high up in the NICE hierarchy and those at the top of the *i3*-MCAST multicast tree.

Scalability Scalability becomes an important issue when the application is intended for large multicast group, such as streaming video involving millions of receivers. The per-source memory, processing, and control overhead of some ALM protocols may prohibit their use for multicast groups of such size. Narada seems to suffer from such scalability constraints. Taking advantage of layering, *i3*-MCAST inherits its robustness from the underlying distributed lookup protocol. This may allow *i3*-MCAST to scale to larger multicast group sizes.

8 Summary

We conclude this paper by summarizing our ALM protocol evaluations. We begin by the Narada protocol. As described earlier, Narada disseminates multicast traffic along per-source spanning trees of a richly connected overlay mesh. This mesh is continuously reconfigured to afford better application-layer performance and adapt to changes in the group membership, changes in the network characteristics, and failures. By distributing the load of joins among its members, Narada is capable of handling frequent joins. Moreover, by having each member of the group maintain the complete multicast group membership, Narada is capable of reestablishing connectivity even in cases of a significant number of failures. Notably, this is achieved without relying on an external bootstrap mechanism; that is, even in the case of a substantial number of failures, Narada is self-sufficient. Narada's per-member memory and control overhead requirements may prohibit its scalability. In particular, the per-member memory requirement is $O(N \log N)$. Furthermore, this state is periodically exchanged among neighbors in the overlay mesh. Thus, presuming the degree of the mesh is d, each member must periodically exchange $O(d)$ messages, each of which is of size $O(N \log N)$. It is also questionable whether Narada's random link addition scheme can discover high utility links in a large and highly dynamic multicast setting fast enough to afford an efficient overlay.

NICE is a hierarchy-based ALM protocol. It partitions the members at each layer of this hierarchy into clusters, where proximate members at any given layer belong to the same cluster, and elects a leader to represent each such cluster at the higher layer of the hierarchy. NICE's hierarchy guarantees that multicast packets traverse at most $O(\log N)$ application-level hops. Moreover, since clusters capture the underlying locality, these application-level hops traverse incrementally larger regions of the underlying topology. Thus, NICE achieves good aggregate end-to-end performance. Through delegation of the forwarding responsibilities of members at the higher layers of the hierarchy, NICE imposes $O(k)$ stress on the links and members due to the data path. In the case of the control path, however, members at layer i must send $O(ki)$ heartbeat messages (in the worst case, $O(k \log N)$ for the member at the top of the hierarchy). NICE's shortcomings are that: i) the join process concentrates load on the higher layers of the hierarchy, ii) in some situations, which will probably arise in highly dynamic environments, NICE recovers from hierarchy partitions by resorting to an external bootstrapping mechanism, and iii) NICE's migration process is incapable of correcting clustering errors and adapting to large changes in the network characteristics. Our proposed modifications to NICE, presented in Section 5.4.5, compensate for the latter two of these drawbacks.

i3-MCAST constructs a multicast forwarding tree using the *rendez-vous*-based indirection primitive provided by *i3*. Each of the edges in *i3*-MCAST's multicast tree constitute Chord lookups. Thus, the number of application-level hops required to deliver multicast packets is, in most cases, $O(\log N \log N_S)$. However, the degree of locality captured by the multicast tree is questionable so the latency incurred by each such application-level hop may be arbitrary. The stress sustained by the underlying links may be as high as $O(\log N_S)$. Stress may thus prevent the use of *i3*-MCAST for high bandwidth applications. Furthermore, *i3*-MCAST incurs substantial overhead for refreshing the triggers comprising the multicast tree and for subcasting suppression messages so as to limit the number of refreshes per trigger.

We propose two schemes for improving the applicability and scalability of *i3*-MCAST. First, we propose that *i3* servers be augmented with a caching scheme in which, for each of their triggers, they cache the *i3* server that is responsible for the destination *i3* identifier of the trigger. Thus, packets can be forwarded directly to the appropriate *i3* server and need not be routed using a Chord lookup. If viable, this scheme has the potential of reducing the number of application-level hops required to deliver multicast packets to $O(\log N)$ and reducing the stress sustained by underlying links to $O(D)$. Second, instead of each member being responsible for refreshing all the triggers comprising its trigger chain, we propose that each member only refresh the trigger comprising the edge from its parent node to its own node. Thus, each member relies on its ancestor members to refresh the triggers earlier in its trigger chain. To avoid partitions, we propose using heartbeat messages from parent to children members. Thus, when children notice their parents have either left or failed, they take their place. This scheme reduces the aggregate number of application-level messages associated with refreshing the triggers comprising the multicast tree.

References

[1] BANERJEE, S., BHATTACHARJEE, B., AND KOMMAREDDY, C. Scalable Application Layer Multicast. In *Proc. ACM Special Interest Group on Data Communication (ACM/SIGCOMM'02)* (Pittsburgh, PA, Aug. 2002).

[2] BANERJEE, S., BHATTACHARJEE, B., AND KOMMAREDDY, C. Scalable Application Layer Multicast. Tech. Rep. UMIACS TR-2002-53 and CS-TR 4373, Dept. of Computer Science, University of Maryland, College Park, MD, May 2002.

[3] CHU, Y.-H., RAO, S. G., SESHAN, S., AND ZHANG, H. A Case for End System Multicast. *IEEE Journal on Selected Areas in Communication (JSAC), Special Issue on Networking Support for Multicast*. To appear.

[4] CHU, Y.-H., RAO, S. G., SESHAN, S., AND ZHANG, H. Enabling Conferencing Applications on the Internet using an Overlay Multicast Architecture. In *Proc.*

ACM Special Interest Group on Data Communication (ACM/SIGCOMM'01) (San Diego, CA, Aug. 2001), pp. 55–67.

[5] Chu, Y.-H., Rao, S. G., and Zhang, H. A Case for End System Multicast. In *Proc. International Conference on Measurement and Modeling of Computer Systems, ACM Special Interest Group on Measurement and Evaluation (ACM/SIGMETRICS'00)* (Santa Clara, CA, June 2000), ACM Press, New York, pp. 1–12.

[6] Eugster, P., Handurukande, S., Guerraoui, R., Kermarrec, A.-M., and Kouznetsov, P. Lightweight Probabilistic Broadcast. In *Proc. International Conference on Dependable Systems and Networks (IEEE/DSN'01)* (Göteborg, Sweden, July 2001), IEEE Computer Society, pp. 443–452.

[7] Ganesh, A. J., Kermarrec, A.-M., and Massoulié, L. SCAMP: Peer-to-Peer Lightweight Membership Service for Large-Scale Group Communication. In *Proc. 3rd International Workshop on Networked Group Communication* (London, UK, Nov. 2001), J. Crowcroft and M. Hofmann, Eds., vol. 2233 of *Lecture Notes in Computer Science*, pp. 44–55.

[8] Lakshminarayanan, K., Rao, A., Stoica, I., and Shenker, S. Flexible and Robust Large Scale Multicast Using *i3*. Tech. Rep. CS-02-1187, University of California, Berkeley, 2002. Working draft as of 2002/09/07. Work still in progress.

[9] Peterson, L. L., and Davie, B. S. *Computer Networks: A Systems Approach*, second edition ed. Morgan Kaufmann Publishers, Inc., San Francisco, CA, 2000.

[10] Semeria, C., and Maufer, T. Introduction to IP Multicast Routing. Internet-Draft (Informational), Internet Engineering Task Force, July 1997. Also, Technical Memo, Networking Solutions Center, 3Com Corporation.

[11] Stoica, I., Adkins, D., Zhuang, S., Shenker, S., and Surana, S. Internet Indirection Infrastructure. In *Proc. ACM Special Interest Group on Data Communication (ACM/SIGCOMM'02)* (Pittsburgh, PA, Aug. 2002).

[12] Stoica, I., Morris, R., Karger, D., Kaashoek, M. F., and Balakrishnan, H. Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. In *Proc. ACM Special Interest Group on Data Communication (ACM/SIGCOMM'01)* (San Diego, CA, Aug. 2001), pp. 149–160.