Available at
www.ComputerScienceWeb.com
POWERED BY SCIENCE DIRECT

Computer Communications 26 (2003) 1105–1128

computer communications

www.elsevier.com/locate/comcom

Distributed multicast tree generation with dynamic group membership

Frank Adelstein[a], Golden G. Richard III[b], Loren Schwiebert[c,*]

[a]Odyssey Research Associates, Ithaca, NY 14850, USA
[b]Department of Computer Science, University of New Orleans, New Orleans, LA 70148, USA
[c]Department of Computer Science, Wayne State University, 5143 Class Avenue, Detroit, MI 48202-3929, USA

Received 26 February 2002; revised 28 August 2002; accepted 10 October 2002

Abstract

As group applications have become more prevalent, efficient network utilization becomes a growing concern. Multicast transmission may use network bandwidth more efficiently than multiple point-to-point messages, however, creating optimal multicast trees is prohibitively expensive. For this reason, heuristic methods are generally employed. These heuristics are often based on a Steiner tree approach, which is known to produce multicast trees that achieve an efficient use of network resources. Many such algorithms, both centralized and distributed, have been proposed to generate 'good' multicast trees. Even these heuristics typically have significant execution times, however, so changes to the initial group of multicast participants during generation of the tree is likely. Furthermore, periodic rebuilding of multicast trees or sub-trees has been proposed to improve the efficiency of these trees as the group membership evolves. Changes in group membership are also possible during this rebuilding process. Existing algorithms, however, either do not support changes to the multicast group during building of the tree or they impose unrealistic restrictions, such as no overlapping modifications or regeneration of the tree after every change. These restrictions prevent the use of such algorithms in many situations, e.g.; networks with mobile hosts. To remedy this, we propose an efficient distributed algorithm that supports dynamic changes to the multicast group during tree building and allows concurrent join/leave operations. In this paper, we present the algorithm, a proof of correctness, and detailed simulation results.
© 2002 Elsevier Science B.V. All rights reserved.

Keywords: Multicast protocols; Dynamic multicast groups; Distributed Steiner algorithms; Mobile networks

1. Introduction

Networked multimedia applications that use multicast communication, such as distance learning, cooperative design tools, web-based broadcasts, and videoconferencing, are growing in popularity. These applications, which are often long-lived, place high demands on the underlying communication network, and often have a dynamic set of participants (called a *multicast group*). Furthermore, as the size of the multicast group for an application increases, efficient network utilization becomes more important. An important aspect of supporting a multicast session is building an efficient multicast tree, which defines the communication routes for the participants based on the underlying network topology.

Generation of optimal static multicast trees, which can be modeled as the Steiner Tree problem, has been shown to be NP-complete [12], so heuristic solutions are employed. Algorithms for generating the multicast tree typically balance 'goodness' of the generated tree, execution time, and storage requirements. Another distinguishing characteristic for tree generation algorithms is centralized versus distributed control. Centralized algorithms require less coordination and tend to be simpler, but the coordinating site can become a bottleneck and is a single point of failure. Generally, centralized algorithms also take longer to execute than distributed algorithms. Distributed algorithms can run more quickly and be more fault-tolerant, but they tend to have higher communication overhead and are more complex.

In real networks, a static group membership cannot be assumed. To be widely applicable, a multicast tree building protocol must allow new group members to join and allow existing group members to leave. Building efficient multicast trees for applications with a dynamic multicast group is very difficult, since changes to the multicast group can occur

* Corresponding author. Tel.: +1-313-577-5474; fax: +1-313-577-6868.
E-mail addresses: loren@cs.wayne.edu (L. Schwiebert), fadelstein@oracorp.com (F. Adelstein), golden@cs.uno.edu (G.G. Richard).

even during generation of the tree. To be effective, algorithms for building multicast trees must generate correct trees quickly even when multiple changes in the multicast group occur concurrently.

When nodes join or leave a multicast session, the efficiency of the tree, relative to an optimal tree, tends to degrade. In general, more membership changes increase the degradation of the multicast tree. This occurs because the structure of the tree is based on the group members and late additions to the group may have been better supported with a different tree structure. Further, some paths in the tree might have been chosen to accommodate deleted nodes. For long-running applications or applications with strict resource bounds, changes to the multicast group can degrade performance to the point where the current tree is too inefficient to support the application or consumes excess network bandwidth. When a multicast group changes significantly, it may be desirable to rebuild the tree or at least a degraded subtree. If the tree is rebuilt, the algorithm for generating the multicast tree must execute quickly while dealing with membership changes during regeneration. Future networks will require this support at least as much as present-day networks. For example, a network that includes mobile hosts is likely to see more group changes, as the mobile hosts move among base stations. Hence, the quality of the multicast tree can degrade more quickly and the benefits of periodically rebuilding the multicast trees will be more noticeable. A tree-building algorithm may also be used by self-healing protocols for recovering from network faults.

A network is a dynamic distributed environment, so serializing changes to the multicast group can have a prohibitively high overhead when it is even possible. In practice, assuming the multicast group membership does not change during the multicast tree setup or rebuild could result in incorrect trees. In general, the protocol cannot prevent concurrent changes to the multicast group, so protocols for use in real networks must be able to support these changes. For this reason, supporting concurrent updates to the group membership is very important. Previous work in this area has focused on minimizing the execution time of the tree construction algorithm or producing high-quality trees. To our knowledge, no research has been done on supporting concurrent changes to the multicast group *during* execution of the algorithm. Our distributed algorithm efficiently supports overlapping join and leave requests during generation of the multicast tree. This makes our algorithm suitable for a wide variety of settings, including networks with mobile hosts.

The rest of the paper is organized as follows. Section 2 presents background information, including the system model, problem statement, and a discussion of related work. Section 3 presents the proposed algorithm. A correctness argument is presented in Section 4. Section 5 describes our detailed simulation study and presents the results of these simulations. Finally, Section 6 concludes the paper with a discussion of future research directions.

2. Background

2.1. Related work

Multicast protocols define how multicast groups are maintained and the route each message takes to reach all members of the group [10,16]. Multicast protocols currently in use, such as the Distance Vector Multicast Routing Protocol (DVMRP) [17,24], which is used for the backbone of the Internet Multicast Backbone (MBONE), rely on flood and prune mechanisms to maintain multicast group membership. It is generally acknowledged that this approach generates too much network traffic for deployment in large networks when the set of participants is relatively sparse. As the demand for collaborative applications grows, an increase in multicast traffic is expected. More efficient use of the network bandwidth will be required to support these group applications. For this reason, we review only scalable protocols in this paper. In other words, we consider only protocols that do not rely on flood and prune mechanisms to build and maintain the multicast tree.

There has been considerable work on multicast route selection. Most of the work, both centralized and distributed, falls into one of three categories: Steiner trees, source-based routing, or center-based routing. Steiner tree-based algorithms produce efficient trees. Because the Steiner tree problem is NP-Complete, heuristics are used to generate good rather than optimal trees [26]. These algorithms generally use fewer network resources than the other two approaches, especially when there is a single source. Source-based schemes also build a tree rooted at each source, but do not use Steiner-tree heuristics. The generated trees tend to be less efficient, but the algorithms have reduced complexity or are easier to implement in a distributed environment. Center-based approaches are most appropriate when there are multiple sources in the multicast group. In this case, all the receivers of the group are part of the multicast tree and sources are optionally members. This approach has been proposed for use in both the Core-Based Tree (CBT) [4,5,6]. and Protocol Independent Multicast-Sparse Mode (PIM-SM) protocols, where the center is called the *core* or the *rendezvous point*, respectively. A node in the network is chosen as the center and the sources forward messages to the center, where all multicast communications originate. Since all sources must transmit through the center, traffic concentrations can be high. In addition, the resulting multicast tree is likely to be less efficient for each session than separate multicast trees. Steiner tree approaches also have some drawbacks, such as inefficient use of the network resources if multiple multicast trees exist simultaneously. In fact, the best choice of a tree-building approach remains an area of active research. The algorithm in this paper focuses on an important problem for Steiner tree-based algorithms; however, we expect the ideas in this paper to be extensible to other approaches to building multicast trees. This paper assumes that the multicast tree (or subtree) is being built (or rebuilt) from a given root node. This generalizes to all three classes of protocols described above, although some changes may be required to efficiently support source-based tree and center-based tree approaches.

Doar and Leslie [11] support using a 'naïve' approach to creating the multicast tree, which takes the union of all minimum cost paths from the source to the destinations. Simulations show that their trees generally have efficiencies within twice the optimal cost. Doar and Leslie argue that the simplicity of their approach compensates for generation of sub-optimal trees. They also point out that frequent multicast group changes can quickly degrade a near-optimal tree, while their algorithm is more resilient to changes. Although their algorithm exhibits good performance, it may not be well suited for all environments. For example, Doar and Leslie point out that when many nodes leave the multicast group, the performance of a multicast tree, relative to an optimal multicast tree, tends to degrade [11]. Our algorithm is suitable for such situations, since it could be used by a protocol that partially rebuilds a multicast tree.

Shaikh et al. [22] present a multicast route selection algorithm that requires no global cost information, while utilizing a localized, greedy approach. Its performance is sometimes worse than that of the global algorithms, but generally produces good trees. Although only localized information is required, the algorithm adds nodes to the multicast tree sequentially, so the algorithm is not completely distributed.

An approach, proposed by Im et al. [13], uses a delay-constrained algorithm that requires N rounds to construct a multicast tree with N receivers. The algorithm proceeds by adding the closest receiver to the existing multicast tree in each round. The authors mention the need for a tree-building algorithm that executes quickly. The need for supporting a dynamically changing network topology is also discussed. However, the authors assume a static multicast group during the multicast tree setup. Chakraborty et al. [9] investigate the construction of an optimal Steiner tree with dynamic joins and leaves. However, their algorithm requires that the joining and leaving times of all participants is known in advance. Kim et al. [14] presents a protocol for maintaining dynamic multicast trees for loosely coupled multimedia conference applications. Their protocol handles joins and leaves in a manner similar to PIM-SM. The paper does not discuss issues related to concurrent changes in the multicast group. Tsukada and Takai [23] point out that multiple multicast groups may be active at the same time, so there are advantages to arranging these multicast trees to distribute the overhead across the network. They then discuss an algorithm to generate multiple Steiner trees to achieve this goal and also extend this algorithm to support dynamic group membership.

Ryu et al. [19] propose techniques for supporting dynamic changes in ATM multicast groups. Their algorithm assumes an initial multicast tree and handles dynamic changes that occur during the lifetime of the multicast connection. The multicast tree is built by repeatedly selecting the minimum cost path from the current tree (initially, just the source) to destinations that are not yet connected to the tree. It prunes leaf nodes that depart, but does not reroute existing connections based on late adds or deletes. Rerouting is avoided because of the complexity of maintaining in-order delivery of cells over an ATM connection while modifying the path. The algorithm also uses the probability of nodes joining or leaving the multicast group to design trees that facilitate the sharing of paths and that produce a relatively large number of leaf nodes. The algorithm is centralized and assumes knowledge of all multicast branch points in the network. The authors also assume a static multicast group during the multicast tree setup.

Bauer and Varma [8] propose a distributed algorithm to establish a multicast tree in a point-to-point network using shortest path heuristics (SPH) and Kruskal-based shortest path heuristics (K-SPH). Only members of the multicast and nodes in the neighborhood of the multicast tree participate in this algorithm. Their algorithm builds the tree from 'fragments', initially consisting of just the individual multicast nodes. These fragments combine with each other to form new fragments, with a single node assigned as the leader of each fragment. Each leader runs a distributed algorithm that is either in a discovery phase or in a connection phase. The discovery step uses a 'flood to N' approach to find out which nodes are close to it. It also propagates information about the fragment leader to nodes within the fragment and sends updated shortest path information from each node to the fragment leader. In the connection step, each fragment picks a 'preferred fragment' and attempts to negotiate a merger with it. Nodes in the discovery step respond to a request to merge with a busy reply. If the merger succeeds, the fragments are combined and the leader with the lowest index becomes the leader of the new fragment. This process continues until there is a single fragment remaining, containing all of the nodes participating in the multicast. One problem with this approach is that the 'discovery step' can impose a high overhead when the network topology is relatively stable, since the same information is recomputed many times. In addition, a request for merging may result in a series of busy replies followed by additional requests to the same node. Another problem is that their tree-building algorithm does not support changes to the multicast group during generation of the tree.

Raghavan et al. [18] propose a similar algorithm for restructuring an existing multicast tree or a subtree of the multicast tree. Rather than minimizing the resource requirements of the multicast tree, their protocol attempts to minimize the *delay* of the resulting multicast tree. This is beneficial for many multicast trees for voice or video applications. When changes to a subtree degrade the performance of the subtree to the point where it can no longer maintain the required quality of service, the subtree is rebuilt. As with Bauer and Varma's protocol, only one change in group membership is allowed at any given time. That change must be processed before another membership change can occur—a restriction that is clearly unenforceable in a distributed network environment. The authors also do not permit any changes while the subtree is being rebuilt.

2.2. Problem statement

Given an arbitrarily connected communication network $G = (V, E)$, a set of multicast participants $Vm \subseteq V$, and a source $Vs \in Vm$ for the multicast, form an efficient Steiner tree to connect the multicast nodes. All leaves of the multicast tree are multicast nodes, although non-participating nodes (called *Steiner nodes*) may be required to form the tree. The algorithm should be fully distributed and must handle concurrent changes to the multicast group during execution.

2.3. System model

We assume an arbitrarily connected point-to-point network of $N = \|V\|$ nodes, any of which may participate in a given multicast session. Routing information to each potential destination, including the next node in the route and the associated cost, is available at each node in a routing table or other suitable structure. The cost function for an edge between two nodes is symmetric. The network delivers messages in order in finite time and does not drop or corrupt messages. We assume that no node or link failures occur during the execution of the algorithm. This is consistent with previous work, which has not discussed fault tolerance. A fault tolerant solution is the subject of on-going work.

3. The algorithm

3.1. Overview

In Section 3.2, we describe our basic distributed algorithm for generating an efficient multicast tree. The algorithm generates a correct tree provided the following conditions hold:

- The initial multicast group, *Vm*, is known to all participants.
- The multicast group does not change once execution of the algorithm has begun.

Certain aspects of the basic algorithm resemble Bauer and Varma's [8], such as connection of fragments through the shortest path and the concept of a preferred node or fragment, but there are substantial differences. To make the description of the algorithm clear, we assume that a given set of nodes is involved in the generation of at most one multicast tree. Concurrent generation of multicast trees for different groups is possible by simply tagging the various messages used in the algorithm with a unique multicast session ID and keeping separate data structures for each invocation of the algorithm (Fig. 1).

Each node stores the following local variables:

ID	(the node's unique identifier)
FragID	(identifier for the fragment and the fragment leader)
F	(list of multicast group members in this fragment)
Vm	(nodes wishing to participate in the multicast)
Vp	(the identifier of this node's preferred fragment)
Vs	(multicast source node identifier).

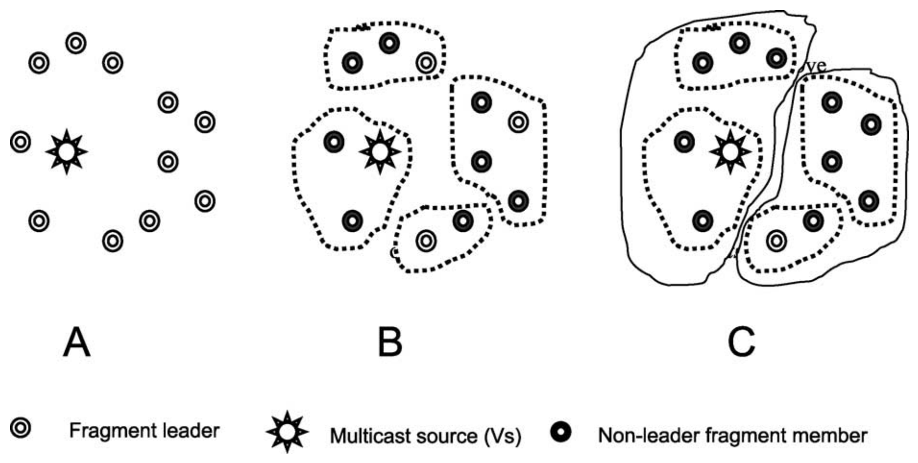

Fig. 1. The proposed algorithm begins with fragments each containing only a single multicast member (A). These fragments are gradually merged (B, C), with a single member serving as the leader of each larger fragment. In the next step (not shown), the two large fragments in C will merge to form a single fragment containing all the multicast members.

Initialization. The basic algorithm performs the following actions. In the initialization step, a separate fragment is created for each node interested in participating in the multicast. Initially, each node is the fragment leader of its singleton fragment. Each fragment is aware of all other nodes in the initial multicast group. This information is used to build a list of other fragments (potential merge candidates). Each node has access to internal routing tables and can determine the cost/distance of transmitting a message to other nodes. The fragment leader is responsible for coordinating mergers with other fragments and for keeping other fragment members updated regarding the fragment membership and leader. Each node forwards multicast messages to members of its fragment to which it is directly connected (sometimes called its 'children').

Merge negotiation. Each node searches its routing table to find the closest multicast participant, which becomes its *preferred node*. Because fragments may not have complete information on the other fragments in the tree, each fragment must choose its preferred node based on local information. This information is accumulated from the nodes within the fragment, but nodes outside the fragment are not queried. In the case of multiple same-cost choices, we assume that all nodes execute the same a priori mechanism for breaking ties, such as selecting the node with the lowest ID.

Once a fragment leader selects a preferred node, it sends a MERGE REQUEST message to that node and waits for a reply. The MERGE REQUEST message contains information that informs the recipient of the sending fragment's *ID* and the nodes at the head and tail of the shortest path connecting these two fragments. (This allows the receiving fragment to determine the proposed path between the two fragments.) When a fragment leader receives a MERGE REQUEST, if the sender is the preferred node, then it sends an ACCEPT message and both leaders enter the connection phase. If the sender is not the preferred node, then the request is noted by sending a BUSY reply to the sender. If a non-leader receives a MERGE REQUEST, it forwards the MERGE REQUEST message to its own fragment leader for processing and transmits a BUSY reply with its *FragID* attached. This informs the sender of the MERGE REQUEST of the identity of the fragment leader, so that this sender will be able to correctly interpret the response from the actual fragment leader.

A node receiving a BUSY reply to a MERGE REQUEST will receive a MERGE REQUEST initiated by the sender of the BUSY at a later time. In the meantime, the node receiving the BUSY reply waits for this MERGE REQUEST but may ACCEPT a MERGE REQUEST from a different fragment while waiting, if the cost is less than or equal to its current preferred fragment. This prevents deadlocks in the protocol.

Connection phase. The purpose of the connection phase is to join two fragments. The fragments are joined

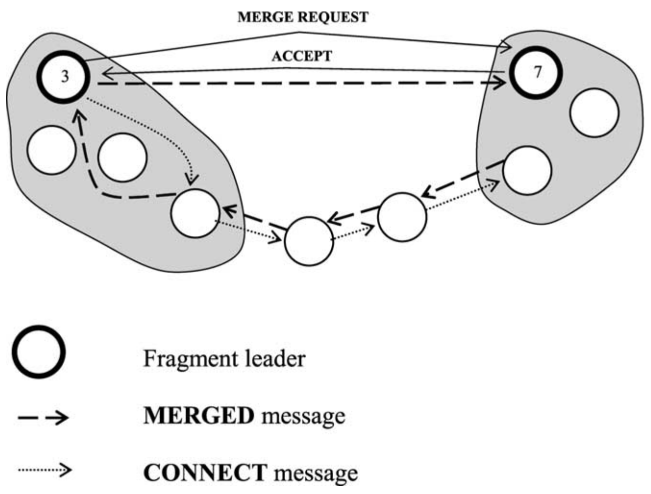

Fig. 2. Two fragments merge successfully. The MERGE REQUEST from the leftmost leader results in an ACCEPT. A chain of CONNECT messages reserve nodes along the shortest path between the fragments, then a chain of MERGED messages finalizes the reservations. The rightmost leader, upon receiving the MERGED message, sends a list of its fragment members to be incorporated into the merged fragment's member list. The leftmost leader node becomes the leader for the merged fragment.

using the shortest path between them and nodes along this path are incorporated into the merged fragment. If any of the nodes along the shortest path connecting the fragments already belong to a different fragment, the merge attempt fails.

Fragment leaders entering the connection phase perform the following actions, as illustrated in Fig. 2. The fragment leader with the lower *ID*[1] sends a CONNECT message along the shortest path between the fragments, from the head of the shortest path to the tail. In other words, the path taken is from the closest node in this fragment to the closest node in the other fragment. Upon receiving the CONNECT message, if a node is not a member of another fragment and is not reserved, it tentatively becomes a member of the combined fragment and marks itself reserved. It then forwards the CONNECT message along the shortest path. If a node receiving a CONNECT is reserved or already a member of another fragment, the path connecting the merging fragments is *blocked* and the merge fails. The node sends a NACK backward along the shortest path (toward the sender of the CONNECT message) and each node receiving the NACK cancels its reservation and reverts to its previous status. When the NACK arrives at the leader, it sends a NACK to the other leader and the merge fails. The procedure then repeats from selection of a preferred node. If the CONNECT message reaches the tail of the shortest path between the fragments, then MERGED messages are sent back along the shortest path between the fragments.

[1] If the root of the multicast tree is the leader of one of the two fragments, it operates as if it were the one with the lowest node ID.

The MERGED messages make the reservations permanent (the nodes become members of the combined fragment) and also propagate a list of Steiner node *ID*s back to the leader. The leader of the combined fragment adds the new members to the fragment membership list *F*.

The fragment leader with the lower node *ID* becomes the leader for the new, combined fragment. The node with the higher node *ID* sends its fragment membership list, via an UPDATE TABLES message, to the new leader, which includes these members in the new fragment membership list. The leader of the combined fragment calculates a new preferred node and then multicasts an UPDATE TABLES message to the other fragment members. This message contains the leader's identity, the fragment membership list *F*, *Vm*, and the suggested preferred node and its distance. When a fragment member receives the UPDATE TABLES message, it updates the fragment *ID*, updates *F* and *Vm*, and then calculates its preferred node. If the calculated preferred node is closer than the one suggested by its leader, it sends an UPDATE back to the leader with its preferred node and the cost, otherwise it sends back an ACK message.

The fragment leader gathers the UPDATE/ACK messages and determines the closest multicast participant that is not a member of the fragment, which becomes the new preferred node. The leader then sends out a MERGE REQUEST and the whole process repeats. The algorithm terminates when the leader determines that there are no other fragments (that is, when $Vm \subseteq F$). A detailed description of the algorithm in the form of pseudocode is available in Appendix A.

3.2. Dynamic algorithm

A practical distributed algorithm must be able to handle changes to the multicast group during tree setup. Two types of changes are possible: additional nodes may wish to join the multicast group and current members of the multicast group may wish to leave. The modifications proposed in this section extend the basic algorithm to support concurrent changes to the multicast group during generation of the tree. If no dynamic changes occur, the algorithm operates as previously described. An additional data structure, called *BUSY_Q*, is required to support dynamic changes. *BUSY_Q* tracks the *ID* of nodes to which BUSY messages have been sent, along with the unique identifier for the BUSY message. In addition, the fragment members list, *F*, and the multicast participants list, *Vm*, must be augmented to allow entries to be marked as *added* and/or *deleted*.

Section 3.2.1 discusses late additions to the multicast group. Section 3.2.2 describes how dynamic deletions are performed. Section 3.2.3 discusses merging fragment information in the presence of late adds and deletes.

3.2.1. Addition requests

Requests for entering the multicast group after the tree setup has started are handled as follows: the new node becomes a new singleton fragment, contacts a member of the multicast group, and then sends a merge request to its preferred fragment.[2] In this section, we describe how these steps are performed and how the multicast group information is updated and consistency is maintained.

The new node uses its own *ID* for its fragment identifier and considers itself the leader of this singleton fragment. Two possibilities exist: the multicast tree has already been established or the tree generation is still underway. The new node is unaware of the status of the tree, but knows the identity of the source node, *Vs*. Therefore, in either case, it sends a JOIN REQUEST toward the multicast source. If the tree already exists (i.e., the protocol described in this paper to generate the tree has already terminated), this request can be processed in the network by an independent protocol that dynamically adds group members to an existing tree [7,11].

Otherwise, the JOIN REQUEST must be intercepted by our tree-building protocol and processed as a late add. As the JOIN REQUEST propagates toward the multicast source, it either encounters another fragment or reaches the source. Obviously, the source always remains a member of any valid multicast group, so the JOIN REQUEST is guaranteed to eventually contact another multicast group member. The multicast group member that receives the JOIN REQUEST forwards the message to its fragment leader, which adds this node to its multicast group membership list, *Vm*. Updating the multicast group membership list prevents the tree building algorithm from terminating between the time when the JOIN REQUEST is received and the new fragment merges with another fragment in the tree. In order to ensure that all nodes in a fragment have a consistent view of the multicast group, the multicast membership lists are merged when fragments merge. This procedure is discussed in Section 3.2.3.

Regardless of which group member the new node contacts, a LATE ADD REPLY is returned to the new node. This message contains the current list of multicast group members known to that fragment. Upon receipt of this reply, the new node determines its preferred node by referencing its routing table and sends a MERGE REQUEST to that fragment. When the preferred fragment's leader receives the MERGE REQUEST, it adds this node to its copy of *Vm*. From this point on, the MERGE REQUEST is processed in the same manner as all other MERGE REQUESTs.

[2] If a node that wants to join the multicast is already a member of a fragment but is currently marked as deleted or is currently a Steiner node, it simply sends a message to the fragment leader requesting to be a 'full member.' The fragment leader forwards this information to other fragment members the next time an UPDATE TABLES message is sent.

If the preferred fragment has deleted itself from the multicast group, then the MERGE REQUEST is processed as described in Section 3.2.2 and the new fragment selects another candidate as its preferred node. An exception is whenever all multicast group members known to the responding fragment decide to delete themselves from the multicast group after the fragment leader responds to the JOIN REQUEST. It is also possible that all the nodes in the original multicast group have deleted themselves from the multicast group. In either of these two pathological cases, the new node sends a MERGE REQUEST to the source and notifies the source that no member of the original multicast group wishes to participate. Other late adds may still be in the tree building algorithm, but the source may begin to send at this point. Hence, it is possible that not all nodes are in the multicast tree when the multicast begins, however, the multicast session cannot begin if any of the *original* members of the multicast group are still in the tree-building algorithm. If any late add nodes are still in the multicast tree-building algorithm, these nodes will merge with the source later.

There are a few other issues that arise when allowing late adds. First, it is possible that a fragment can ACCEPT a MERGE REQUEST from a node that is not actually the closest candidate, since it might not be aware of closer late adds. However, this affects only efficiency, not correctness. Second, a node can decide to join the multicast group at any time, even while it is being used to form a connection between two fragments. If this occurs, the late add waits to see if the fragments merge before sending the JOIN REQUEST. Finally, a connection attempt by two fragment leaders may be blocked by a node of which neither is aware. When the blocking node sends a NACK, its *ID* is attached so the node can be added to *Vm* at the receiver. This allows the blocking node to become a candidate for preferred node.

3.2.2. Deletion requests

Delete requests can be more complicated than add requests, since the node to be deleted may have already been incorporated into a fragment. If the node is still a singleton fragment, it simply sends a NOT INTERESTED response to any MERGE REQUESTs. The node receiving the NOT INTERESTED response to a MERGE REQUEST marks the node as deleted in its fragment list, *F*, and multicast group membership list, *Vm*. This is done to allow accurate determination of the group membership list when merging fragments, as described in Section 3.2.3.

If the node is in a fragment with more than one member, it removes itself from the multicast group by sending a DELETE message to the fragment leader. The deleted node continues to handle future MERGE REQUESTs the same way as other non-leader nodes in the fragment. If the node that wishes to be deleted from the multicast group is the fragment leader and there are at least two other nodes in the fragment, it selects the member of the fragment that is closest to the fragment leader and requests that the node take over leadership duties via a CHANGE LEADER message. (If there is only one other node in this fragment, the fragment is dissolved and the other node becomes a singleton fragment.) The closest node is selected to minimize the disruption to the current tree structure. After the fragment leader sends the CHANGE LEADER message to this fragment member, it waits for either a NACK or an ACK from this node. If a NACK is received, the candidate node has deleted itself and another candidate must be chosen. If there is only one other multicast group member still in the fragment, this fragment is dissolved and the remaining node in this fragment functions as a singleton node. This process is repeated until an ACK is received or the fragment is dissolved. If an ACK is received, the new fragment leader has been 'elected' and pending MERGE REQUESTs (i.e. entries in the *BUSY_Q*) as well as up-to-date copies of *Vm* and *F* are forwarded to the new fragment leader. The new leader subsequently updates other member nodes in the fragment with an UPDATE TABLES message.

Once the ACK has been received and the fragment leader is changed, the fragment must be restructured so that the root of this subtree is the new fragment leader. This could potentially require changing the orientation of multicast tree edges, replacing existing edges with new edges, and modifying the set of Steiner nodes in the fragment. Although this may create a lower cost subtree, the overhead of restructuring could be quite expensive. Furthermore, if additional nodes in that subtree add or delete themselves during this restructuring, the processing becomes very complicated. Rather than introduce this complexity and overhead into the protocol, a simple restructuring mechanism is used. When the candidate for becoming the new fragment leader sends the ACK back to the current fragment leader, the edges on that path are reversed. The result is that the fragment is now rooted at the new fragment leader. A robust fault-tolerant version of this procedure for restructuring subtrees was introduced by Schwiebert and Chintalapti [21], where it was used for restructuring CBTs.

Before a fragment is dissolved, one additional issue must be handled. Nodes that received a BUSY response from the dissolving fragment must be informed so they do not wait forever. In order to track outstanding BUSY messages, an entry is added to the *BUSY_Q* whenever a BUSY message is sent in response to a MERGE REQUEST. This entry contains the unique identifier for the BUSY message and the *ID* of the recipient of the BUSY. Before the fragment is dissolved, the leader sends FRAGMENT DISSOLVED notifications to each node with an entry in the *BUSY_Q*. Each of the FRAGMENT DISSOLVED messages carries the unique identifier of the BUSY message so the receiving node can determine which fragment is being dissolved (since the original sender of the BUSY message may have merged into a new fragment and lost the leadership role).

3.2.3. Combining fragment information

When two fragments are merged, they may have an inconsistent view of the multicast group. Any inconsistent information must be reconciled by the new fragment leader before being multicast to all nodes in the newly merged fragment. This can be done by including this information in the UPDATE TABLES message described in Section 3.2. Inconsistent views may be caused by several events. Late adds are seen by a fragment when the node is merged into the fragment, when a JOIN REQUEST is received, when a MERGE REQUEST is received, or when a node blocks a merge with a NACK. Deletes are seen by a fragment only when a NOT INTERESTED reply is returned from a MERGE REQUEST or a node in the fragment sends a DELETE REQUEST. Finally, the *BUSY_Q* sets for the two fragments must be combined.

The combined fragment list, *F*, is the union of the two original fragment lists. Thus, nodes marked as deleted in either list are marked deleted in the combined fragment list. Similarly, nodes added to either fragment become members of the combined fragment.

The combined multicast group membership list, *Vm*, consists of the union of the multicast group membership list in each fragment. Nodes marked as deleted in both of the lists, or marked deleted in one list but appearing unmarked in the other, or marked as deleted in one list but not appearing in the other, are marked deleted in the combined list. Nodes marked as deleted in one list but added in the other are marked as added in the combined list. This ensures that a node that has sent a NOT INTERESTED reply to one fragment and a JOIN REQUEST to the other remains a candidate for merging. If this node is in fact deleted, then a subsequent MERGE REQUEST from the combined fragment will result in a NOT INTERESTED reply and the node will be properly deleted. Finally, the *BUSY_Q* set for the combined fragment is the union of the *BUSY_Q* sets for the individual fragments.

3.3. Termination condition and tree refinement

The algorithm terminates (for a particular group) when there is only one fragment remaining, whose membership consists of the nodes in *Vm*. At some point, additional changes to the multicast group must be postponed so that a multicast tree can be established. Otherwise, the multicast session may be postponed indefinitely as late adds and deletes are being processed forever. (If changes occur this rapidly, there is little advantage in attempting to construct a good multicast tree, since many changes will occur after the tree is completed and it is likely that the tree will be severely degraded within a short time period.) This situation can be handled by bounding the number of additions that a fragment accepts, which forces eventual termination. Subsequent JOIN REQUESTs are then buffered until the tree is built or processed as if the tree has already been built. Obviously, if the number of adds is bounded, then the number of deletes is also implicitly bounded by $\|Vm\|$.

Once the algorithm has completed, it may be beneficial to run an optional protocol that prunes leaf nodes that either are marked deleted or are Steiner nodes. These can be present because of dynamic changes during execution of the tree-building algorithm. When a node is pruned, the algorithm will recurse up the tree and determine if the parent of this leaf node should also be pruned. This process repeats until a parent is found with more than one child, or the parent is a group member, or the source node is reached. The state information maintained by both multicast group members and Steiner nodes may be reduced or eliminated, depending on the needs of the protocol that supports the multicast session after the tree has been built. In our simulations, pruning resulted in average savings of 6% for multicast groups with dynamic changes. More details on the simulation study appear in Section 5.

4. Correctness argument

In the following, we show that the algorithm produces a correct Steiner Tree and always terminates.

Definition 1. The *preferred fragment* of fragment I is the fragment J, denoted preferred(I) = J, if and only if cost(path(I,J)) = min(cost(path(I,x))) for all x where $x \in Vm$ at I.

Note that *Vm* is local to I, so I's notion of preferred fragment can change if it discovers the existence of a late add that is closer than other nodes in *Vm*, or if another fragment in *Vm* gets closer because of a Steiner node.

Definition 2. A pair of fragments (I, J) is a *preferred pair* if and only if preferred(I) = J and preferred(J) = I.

Lemma 1. *The algorithm produces a Steiner Tree of the nodes in Vm.*

Proof. The algorithm proceeds by merging fragments. Initially, each fragment contains a single node, and there are $n = \|Vm\|$ such fragments. Obviously, there are no cycles in the singleton fragments and each is a valid Steiner Tree. Now consider the case of k fragments, where $k < n$. Of these k fragments, two distinct fragments, F1 and F2, merge by adding a path from one fragment to the other. No cycles exist within F1 or F2 and there is only a single path connecting F1 to F2. Hence, there are no cycles in the new fragment formed by merging F1 and F2, and this combined fragment remains a valid Steiner Tree. There are now $k - 1$ fragments, all of which form valid Steiner Trees. By induction, all fragments remain acyclic and are valid Steiner Trees. Since the algorithm terminates when there is only a single fragment remaining, which contains all nodes in *Vm*, the final fragment is a valid Steiner Tree for the nodes in *Vm*. □

Lemma 2. *During the connection phase, at least one connection succeeds.*

Proof.

(1) Without loss of generality, assume fragments A and C are closer than any other pair of fragments. If A is aware of C, A sends a MERGE REQUEST to C. If C sends an ACCEPT message, the pair have recognized each other. If, due to incomplete information, C has selected another fragment, B, then either C merges with B in the current round (because B has already sent an ACCEPT), or C merges with A, because B sent a BUSY reply. Again, if A and C merge, they have recognized each other. If C merges with B, then it notes the request from A, sends a BUSY reply, and merges with A during the next round. Recall that the distance from A to C is less than the distance between any other pair of nodes, so even if C and B merge, when C and B combine their pending merge list, A is closer to the B–C fragment than any other fragment. Therefore, the two closest fragments find each other and initiate a merge. If A is unaware of C, it will send a MERGE REQUEST to another node, D. This will succeed unless A receives a BUSY reply from D. In this case, if C sends a MERGE REQUEST then A can ACCEPT. Otherwise, D merges with some other node, since that other node is closer to D than A is.

(2) Clearly, there is always some pair of fragments that are closest (ties are broken by an a priori ordering). Thus, this closest pair always attempts to merge, based on the above argument in (1), after some sequence of rounds in which at least one merger takes place.

(3) The only problem ignored above is if the connection phase does not succeed because the path between the two joining fragments routes through some intervening fragment. First, we consider the case without any late adds.

Case 1. *No late adds*. Refer to Fig. 3. Assume fragment A wants to merge with fragment C and node B, which lies between A and C, belongs to another fragment and is a Steiner Node. Since B was not originally in *Vm*, neither A nor C know about B. If they were aware of B, then B would be in the preferred fragment for both A and C.

Not only does node B lie between A and C, but since it is a Steiner Node, it must lie between two nodes within its own fragment, say nodes A′ and C′ (which are in *Vm*), where nodes A′ and C′ are a preferred pair. Therefore, by Definitions 1 and 2:

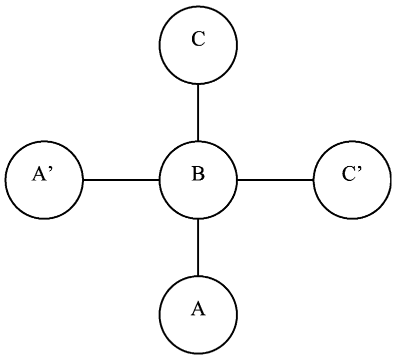

Fig. 3. Two merging fragments collide.

cost(path(A′,C′)) = min(cost(path(A′,x))) for all x, or alternatively,
cost(path(A′,C′)) = min(cost(path(y,C′))) for all y
AND
cost(path(A,C)) = min(cost(path(A,x))) for all x, or
cost(path(A,C)) = min(cost(path(y,C))) for all y.

But, B is an element of path(A,C) and path(A′,C′). It can be thought of as a 4-way intersection with B as the common node, as illustrated in Fig. 3.

Assume cost(path(x,B))|x = (A,A′,C,C′) is not equivalent, for all given values of x. Without loss of generality, also assume that cost(B,C) is the minimal cost of the group. Additionally, without loss of generality, assume the ID(C) < ID(C′). In the case where cost(path(x,B))|x = (A,A′,C,C′) is equal for all given values of x, since C has the lowest ID number, it is preferred by all other fragments and this case reduces to the case where the distances are distinct. Therefore, without loss of generality, we assume that not all the costs are equal. This implies that C is the preferred fragment for A. It also means that C is the preferred fragment for A′, not C′, since cost(path(B,C)) ≤ cost(path(B,C′)), which means cost(path(A′,C)) ≤ cost(path(A′,C′)). This means that A′ tries to merge with C instead of with C′. Thus A′ and C′ cannot already be merged, so B is not a Steiner Node for an A′–C′ fragment, which implies B is still available and the connection does not block.

Since C only ACCEPTs a MERGE REQUEST from a single fragment per round, any other fragment(s) that send MERGE REQUESTs receive a BUSY response and do not attempt to connect to C, therefore the path to C is not blocked. Hence, without late adds the connection phase always succeeds.

(4) Now we consider the case in which late adds occur.

Case 2. *Late adds occur*. The late add case is the only case in which a connection attempt can fail. Again, referring to

Fig. 3, this time we assume node B is a late add, unknown to A and C. (Note that late adds that are not in the path between two merging fragments cannot block a connection for those fragments, and are thus not considered.) In this case, A sends a CONNECT message to C and B is on the path of this message. Since B has not agreed to merge with A or C, B rejects the CONNECT message by sending a NACK back to A. When A receives the NACK, A forwards the NACK directly to C, informing C of the failed connection. Thus, the connection is blocked temporarily.

At this point, there are two possibilities. One is that B already knew about the existence of A and C when it received the list of multicast members, in response to its JOIN REQUEST. The other is that B does *not* know about A or C because they are late adds themselves and contacted a different initial node with their JOIN REQUESTs. If B knows about A and C, then B chooses the closer of A and C as its preferred node (in this round or a later round), and sends a MERGE REQUEST to the closer one. Without loss of generality, assume that node is C. Since B is closer to C than A is, C accepts B's proposal and merges with B.

If A and C were not included in Vm for node B, B then adds them to its copy of Vm and recalculates its preferred node and then proceeds as above, i.e.; attempts to merge with the closer node. Also, note that upon receiving a NACK from B, node A then becomes aware of B's membership in Vm and B becomes the preferred node for A. At this point, A no longer ACCEPTs any MERGE REQUESTs from C, since C is farther away than B, and sends C a BUSY reply. Although C is unaware of B initially, B is added into Vm at node C when C receives the NACK forwarded from A (and notices B is not in Vm), at which point B is added to Vm and becomes the preferred fragment for C. Having received a BUSY response from A, C is free to ACCEPT a MERGE REQUEST from B.

In either case, B merges with C, as well as with A, progress is made after the failed connection, and the process then continues the same as in the case without late adds. Therefore, even when late adds are allowed, at least one connection succeeds. Thus, in each round at least one connection succeeds.

Theorem 1. *The dynamic algorithm always terminates with a correct Steiner tree.*

Proof. We assume that a finite number of changes occur. The number of changes can be limited as described in Section 3.3. From Lemma 1, the resulting tree is free of cycles—it is a valid Steiner tree. From Lemma 2, at least one pair of fragments merges each round, reducing the number of fragments by one. Since the number of fragments is finite, the algorithm eventually terminates. □

5. Simulations

A simulator, **mcSIM**, was written in C using version 18 of CSIM [20]. **mcSIM** is a detailed, realistic simulation of the protocol, which simulates the protocol running on every node in the network. Each node operates in a truly distributed fashion. The only sharing of information among nodes is through the exchange of messages. Nodes not interested in the multicast run a low-cost Steiner code that just forwards messages and consumes few resources. Any 'intelligence' required in the protocol occurs in only group member nodes. The tree construction protocol terminates when the multicast tree is built.

The simulation suite includes a network generation program called **Bessie**[3] [3] that generates and displays network topologies as described by Waxman [25] and Doar and Leslie [11]. **Bessie** also provides an improved edge model to provide greater control over the generated topologies. When the simulation finishes, **mcSIM** writes an output file that can be displayed by **Bessie**. **Bessie** verifies the integrity of the generated multicast tree and provides various statistics, such as overall tree cost.

The simulator has been stress-tested with thousands of simulations run in which the number of changes to the network was unbounded. Although testing does not guarantee correctness, examination of the interaction among the nodes during these tests shows that even extremely rare interactions were encountered in at least some of the simulations and were properly handled. Messages sent to nodes must pass through all of the intermediate nodes defined in the network topology. By running extensive simulations, we were able to verify the protocol's correct behavior in the presence of race conditions and unexpected messages. This is especially important when simulating the effects of dynamic changes to the multicast group, because it is nearly impossible to anticipate all the possible interactions that can occur as a result of these spontaneous changes. In addition, as a debugging tool, a watchdog process runs in the background to warn of deadlock and starvation. This was useful during testing to identify implementation bugs.

The protocol runs on every node in the network. Each node has a fixed probability of changing its status with regard to participating in the multicast. Specifically, nodes that are not participating in the multicast can 'decide' to join at any time. Similarly, nodes that are participating in the multicast, or in the process of joining the group can decide to remove themselves from the multicast. There is no constraint on the number of times an individual node may choose to join or leave the multicast. The ratio of joins to leaves was initially set at

[3] The name 'Bessie' is a tribute to the blues singer Bessie Smith.

Table 1
Relative performance with average degree = 3, 10% multicast, no late joins/leaves

No. of nodes	Worst case (%)	Average case (%)	95% Conf. interval (%)	Best case (%)
50	−4.73	24.85	8.87	82.37
100	2.17	27.78	5.96	54.96
200	8.33	26.92	5.86	76.61
500	11.26	24.50	3.11	41.31

Table 3
Relative performance with 200 nodes, 10% multicast, no late joins/leaves

Average node degree	Worst case (%)	Average case (%)	95% Conf. interval (%)	Best case (%)
3	8.33	26.92	5.86	76.61
4	11.61	33.09	5.11	70.95
5	17.83	38.51	5.81	73.43
6	8.50	44.92	5.26	69.79

4:1, but was varied from 9:1 to 1:1 in subsequent test runs. To prevent network instability, there is a system-wide limit on the total number of multicast group changes. By default, it is 100 changes, although that is a run-time configurable parameter. Although these limits were imposed on the runs shown in the simulations, no such limits were used for the stress tests.

In addition, we have an implementation of the naïve approach as described by Doar and Leslie [11] to serve as a comparison. We have also compared our dynamic algorithm with a version that defers multicast group changes until the multicast tree is built and then sequentially performs and the late additions and deletions. This is done to quantify the advantages of incorporating these changes *while* building the multicast tree. The other comparable algorithms [7,8] do not support dynamic changes, so a comparison with those algorithms is not possible. In addition, implementing either of these two protocols is likely to be a significant undertaking. For example, the simulations conducted by Bauer and Varma used only a partial implementation of their protocol [7]. The Doar and Leslie algorithm is run on the final set of multicast group members in the networks. Since the path from each node to the root in Doar and Leslie is independent of every other node participating in the multicast, the exact sequence in which late additions (joins) and deletions (leaves) occur is irrelevant to the Doar and Leslie algorithm. Hence, the output of mcSIM consisting of the network topology and the final list of nodes in the multicast is used as the input for the Doar and Leslie algorithm.

Note that we are only comparing the final costs of the multicast trees, not the computational overhead or running time of the two different methods. Our protocol is implemented in a simulator with low-level features such as message passing and message parsing, and handles dynamic changes in the multicast group during the simulation. Our implementation of Doar and Leslie is a centralized algorithm that simply computes the cost of the final tree. Thus, the execution time of the two protocols is too dissimilar for any meaningful analysis to be made. A careful comparison of our protocol and Bauer and Varma's protocol, however, shows that our protocol requires fewer messages, partially due to Bauer and Varma's use of localized flooding. Our dynamic protocol uses larger messages than Bauer and Varma's protocol in order to support the dynamic group membership, however, the basic version of our protocol can support the same assumptions as Bauer and Varma's protocol with shorter messages.

5.1. Simulation results

The results of the simulation study are shown in Tables 1–7, in which **mcSIM** is compared with Doar and Leslie's protocol. We use their protocol as a baseline for comparison because their algorithm also handles dynamic changes and has been shown to produce good results for many cases. Each row in each table is derived from

Table 2
Relative performance with 200 nodes, average degree = 3, no late joins/leaves

% Multicast	Worst case (%)	Average case (%)	95% Conf. interval (%)	Best case (%)
5	2.12	29.36	8.21	87.27
10	8.33	26.92	5.86	76.61
20	11.02	21.52	2.78	37.91
25	8.68	23.27	3.30	36.82
30	7.95	20.67	3.08	36.87

Table 4
Relative performance with average degree = 3, 10% multicast, join/leave ratio = 4:1

No. of nodes	Worst case (%)	Average case (%)	95% Conf. interval (%)	Best case (%)
50	−16.33	19.18	8.50	82.37
100	1.26	13.37	2.90	31.56
200	−0.23	16.70	3.50	34.23
500	8.88	17.48	2.95	40.23

Table 5
Relative performance with 200 nodes, average degree = 3, join/leave ratio = 4:1

% Multicast	Worst case (%)	Average case (%)	95% Conf. interval (%)	Best case (%)
5	1.05	16.72	4.39	55.89
10	−0.23	16.70	3.50	34.23
20	5.27	12.94	1.87	21.86
25	7.17	14.82	1.65	24.09
30	7.34	14.79	2.07	24.72

Table 7
Relative performance with 200 nodes, 10% multicast, average degree = 3, varying add/delete ratio

Join/leave ratio (%)	Worst case (%)	Average case (%)	95% Conf. interval (%)	Best case (%)
90	3.21	16.69	2.54	28.44
80	−0.23	16.70	3.50	34.23
70	2.15	13.8	3.06	29.27
60	3.75	13.76	2.47	22.99
50	−7.52	11.66	3.38	24.00

the results of runs on 25 random networks. For each experiment we varied one parameter (e.g.; number of nodes in the network, node degree, etc.) while fixing the remaining parameters. The study encompasses over 600 individual simulation runs. Although we attempted to be as thorough as possible, a complete study of all possible combinations of parameter values in a four dimensional space would have been prohibitively expensive. However, our simulation runs showed consistent behavior over a wide range of parameter values, which we believe indicates that the following results are indicative of the general performance of our algorithm.

The results in Tables 1–3 show the percentage improvement of our protocol over the baseline averaged over 25 random networks per row. These three tables present the performance of our basic algorithm, a situation with no late joins or leaves. In Table 1, networks with 10% multicast nodes and an average of three links per node were simulated for varying network sizes. The results show that, in most cases, our protocol generates significantly better trees; in some cases, as much as 80% better, with an average improvement of about 25%. In Table 2, the network size is fixed at 200 nodes and the number of multicast members is varied. Although the average improvement of our protocol decreases as a larger percentage of nodes in the network become group members, our protocol consistently yields more efficient multicast trees. In Table 3, the average degree of nodes in the network is varied. Across node degrees of 3–6, our protocol generates trees that are, on average, 35% more efficient than trees produced by the baseline algorithm. As the average node degree increases, our protocol generates correspondingly better multicast trees relative to the baseline. This is because we are better able to exploit the increased number of shared paths in the network to build better trees.

Table 6
Relative performance with 200 nodes, 10% multicast, join/leave ratio = 4:1

Average node degree	Worst case (%)	Average case (%)	95% Conf. interval (%)	Best case (%)
3	−0.23	16.70	3.50	34.23
4	3.73	19.19	4.19	46.13
5	14.64	29.80	2.94	44.92
6	15.86	32.06	4.13	54.36

In Tables 4–6, late joins and leaves are introduced, with a join/leave ratio of 4:1. In other words, 80% of the changes to the multicast group are late adds and 20% are deletes. The rate at which adds and deletes occur is uniformly distributed throughout each simulation run. Some changes occur at the beginning of the simulation, some at the end, and most in the middle of the simulation. This pattern of changes mimics the behavior expected in a real network environment—changes occur in the multicast group membership in a completely distributed manner. Recall that at most 100 changes to the multicast group are simulated, using a random choice of type of change and node to change. This limit is set intentionally high to allow for more changes than would normally be expected. Other than the requirement that the source of the multicast tree is not permitted to delete itself, there is no restriction on which nodes can add themselves or delete themselves. As stated earlier, a node might change its membership status multiple times during the building of the multicast tree.

The results for these simulations are consistent with the results when there are no dynamic changes to the group membership. In Table 4, the size of the network is varied from 50 to 500 nodes. Table 4 shows that although the relative improvement of our algorithm is not as great as shown in Table 1, our protocol still generates trees with an average improvement of 15–20% over the baseline protocol. In Tables 5 and 6 we fix the number of nodes in the network at 200 and vary the number of nodes initially in the multicast group and the average node degree, respectively. Table 5 displays results similar to the relative performance shown in Table 2. Although the relative improvement of our algorithm does not decrease consistently with increases in multicast group size, the same general trend appears. Similar to Table 3, Table 6 shows that better trees are generated as node degree increases. Although there are a few individual cases where Doar and Leslie's protocol generates a better tree, these cases are rare and the average performance of our protocol is very good. Since each multicast membership change as the tree is being built potentially degrades the quality of the tree, this is good evidence that our protocol is resilient to these dynamic changes.

In Table 7, the join/leave ratio is varied by progressively reducing the percentage of changes that are late adds. The relative performance of our algorithm decreases as the number of deletions increases. Because shared paths are not used by Doar and Leslie's algorithm, as the number of late adds increases, its relative performance decreases. Note that it does not make sense to have more deletes than late adds, since this implies that if our protocol runs long enough, there will be no remaining group members. However, such a ratio may make sense if only a very limited number of changes to the multicast group is permitted.

Overall, the performance of our algorithm is significantly better than Doar and Leslie in terms of the cost of trees generated. In virtually all cases, our algorithm generates substantially better trees. Substantial testing with a large number of different networks confirms these results.

6. Conclusion and future work

We have presented a distributed algorithm for the construction of a multicast tree in environments in which the multicast group membership is dynamic. Unlike existing algorithms, nodes may join or leave the multicast while the algorithm is executing, and concurrent membership changes are permitted. The algorithm builds correct trees and integrates dynamic changes to produce high-quality trees even in the presence of dynamic group membership. This is accomplished without a significant increase in the message complexity of the algorithm. To be practical, a distributed algorithm must support dynamic group membership, since it is impossible to prevent these changes from occurring.

This algorithm is suitable for use in many situations that require the generation of multicast trees. Examples of such applications include multicast groups with mobile hosts, dynamic regeneration of subtrees that have experienced substantial degradation due to local changes in the multicast group, and self-healing protocols for resolving faults in a fault-tolerant system. On the other hand, the requirement that all participants are aware of all the initial group members seems to preclude the use of this protocol for IP multicasting. However, modifications of this algorithm to work in the Internet for IP multicasting are possible. For local rebuilding of subtrees, especially within a network domain, knowledge of all participants is not needed. In addition, this requirement of membership knowledge could be relaxed further if information on local participants is available. This would allow efficient construction or reconstruction of the multicast tree, since the most important information—the existence of neighboring participants, would be known. Performance of this algorithm with limited, but local, information on participants is important future work.

Because we avoid the discovery step, the message complexity of the inter-fragment communication is lower than the algorithm proposed by Bauer and Varma. In practice, our algorithm should be faster because it does not require a discovery step and cannot have loops where MERGE REQUESTs to the same fragment are repeatedly rejected until finally accepted. The intra-fragment communication, which is used to determine preferred fragments and maintain consistent information within a fragment, introduces additional message overhead. Future work includes the design of better protocols to reduce intra-fragment communication and different network topology models.

In this paper, we have described the protocols and design decisions for creating a multicast tree building algorithm that allows dynamic changes to the multicast group. Generating trees with a distributed algorithm can reduce execution time, especially for a large multicast group. A detailed simulation study of the algorithm described in this paper was conducted, which demonstrated the advantages of our proposed protocol. This allowed us to verify the quality of the trees generated by our algorithm relative to those generated by the Doar and Leslie algorithm.

Other topics for future work include extending the protocol to incorporate fault tolerance—a topic that, to our knowledge, has not been addressed. We will also incorporate this algorithm into a multicast protocol that supports dynamic trees with periodic rebuilds of locally inefficient subtrees. This would be useful for long-running multicast sessions with periodic changes in the multicast group during the actual multicast session.

Acknowledgements

The authors are very grateful to Christine Richard for editorial assistance.

Appendix A

Algorithm pseudocode

Basic concepts
Vm = the initial set of multicast participants. Initially, there are ‖*Vm*‖ fragments, each containing a single distinct element of *Vm*. The algorithm terminates for members of a particular group when *F* = *Vm*. Any message not explicitly handled is an error. State functions are shown in italics.

```
Possible node states = {ConnectionPhase, Initialize, Steiner, TryMerge, UpdatePhase}.

Message types = {ACCEPT, ACK, BUSY, CHANGE_LEADER, CL_ACK, CONNECT,
DELETE, FRAGMENT_DISSOLVED, JOIN_REQUEST, LATE_JOIN_REPLY,
MERGE_REQUEST, MERGED, NEW_LEADER, NACK, NOT_INTERESTED, TERMINATE,
UPDATE, UPDATE_TABLES}.

Each fragment member (node) maintains the following local variables:

        F = set of fragment members.
        State = current state.
        status = (added, deleted, terminating)
        ID = node's ID
        FragID = identifier for fragment, also ID of fragment leader.
        n = ||Vm|| (the number of multicast participants).
        Vs = the root of the multicast tree.
        Vm = the set of multicast participants known to this fragment.
        Vp = preferred fragment.
        msgID = unique identifier each node assigns to each message.
        Terminated = True if the algorithm has terminated, False otherwise.
        BUSY_Q = nodes to which BUSY but no_MERGE REQUEST has been sent.
        SteinerQ = list of Steiner nodes in a fragment.

Initialize
/* initial participants begin with a copy of Vm; late joins must request
   a copy by sending a JOIN_REQUEST message
*/

State = Initialize.
Terminate = False.
Deleted = False.
F = {ID}.
FragID = ID.
Vs = source.

If (Vm == { }) {               /* I'm a late join */
        Send JOIN_REQUEST toward multicast source.
        Receive LATE_JOIN_REPLY, containing Vm.
        Vm = Vm + {ID}.
}

Goto TryMerge.
/* end function */


SteinerInit (ID, source)
/* initialize Steiner nodes */

terminating = False.
deleting = False.
FragID = ID.
Vs = source.
msgID = 0.

Goto SteinerState.
/* end function */
```

```
UpdatePhase
/* update preferred node decision.  The fragment leader provides
   updated FragID, F, preferred node information to fragment members
*/

State = UpdatePhase.
/* find preferred node */
Vp = the node n in Vm such that distance(n, F) is minimal.  The routing
tables are used for this determination.

If (FragID == ID and F == Vm) {       /* I'm the leader and all known fragments are merged  */
        Send TERMINATE message to each fragment children.
        Terminate = true.
        goto Done.
} Else If (FragID == ID) {            /* I'm the leader and some fragments aren't merged */
        If (I want to delete myself) {
                LeaderChanged = false.
                Mark node as deleted.
                Move node from F to SteinerQ.
                While (F has more than one element and not LeaderChanged) {
                        Choose untried node n in F closest to ID.
                        Send CHANGE_LEADER(F, Vm, BUSY_Q) to n.
                        ChangeDecision = false.
                        While (!ChangeDecision) {
                              Receive msg.
                              If (msg is CL_ACK) {
                                 LeaderChanged = true.
                                 ChangeDecision = true.
                                 Reverse multicast tree path toward n.
                              } Else If (msg is NACK) {
                                 /* n is deleted-need to try another node */
                                 Mark n as deleted.
                                 Move node from f to Steiner Q.
                                 ChangeDecision = true.
                              } Else If (msg is MERGE_REQUEST) {
                                 Send BUSY to msg.sender.
                              } Else If (msg is ACCEPT) {
                                 Send NACK to msg.sender.
                              } Else If (msg is CONNECT) {
                                 Send NACK to msg.sender.
                              }
                        }
                }
                If (F has only 1 element ) {
                     /* no new leader, fragment is dissolved */
                     Remove multicast tree path information.
                     Send FRAGMENT_DISSOLVED to remaining node in F.
                } Else If (F = {}) {
                     /* No nodes in F. */
                     Remove multicast tree path information.
                     Send FRAGMENT_DISSOLVED to nodes in BUSY_Q.
                }
                goto Steiner.
        } Else {
                Find preferred node, Vp.
                Send UPDATE_TABLES(FragID, F, Vm, Vp) message to fragment children.
                Wait for ACKs/UPDATEs from children and handle
                unexpected messages (late adds, deletes, out of sequence/
                misdirected messages, etc.).  I.e., Save ms if MERGE_REQUEST;
                add node to Vm if LATE_ADD; add node to Vm and send
                LATE_ADD_REPLY if JOIN_REQUEST; remove node from f and
                add to SteinerQ if NOT_INTERESTED.
                If (children respond with closer fragments via UPDATE messages) {
                        Update routing tables.
                        Choose the closest as Vp.
                }
                Goto TryMerge.
        }
} Else {                      /* non-leader nodes loop here */
```

```
        Elected = False.
        Deleted = False.
        Terminate = False.
        While (not Terminate and not Deleted and not Elected) {
                If (I want to delete myself) {
                        Deleted = true.
                } Else {
                        Receive msg.
                        If (msg is TERMINATE) {
                                Terminate = true.
                                goto Done.
                        } Else If (msg is UPDATE_TABLES) {
                                Receive UPDATE_TABLES(FragId_new, F_new, Vm_new, Vp_new) from leader.
                                FragID = FragID_new.
                                F = F_new.
                                Vp = Vp_new.
                                Vm = Vm_new.
                                Forward UPDATE_TABLES message to fragment children.
                                If (there is a node n in Vm s.t. distance(n, ID) < distance(Vp, F)) {
                                        /* send better choice for preferred node to leader */
                                        Send UPDATE(n,distance(n,ID)) to node FragID.
                                } Else {
                                        Send ACK to node FragId.
                                }
                        } Else If (msg is MERGE_REQUEST) { /* mis-directed MERGE REQUEST */
                                Send BUSY to sender.
                                Forward MERGE_REQUEST to node FragID.
                        } Else If (msg is NACK or msg is JOIN_REQUEST) {
                                /* to receive this message, I must be the path between the
                                   two fragment leaders during the failed merge attempt */
                                Forward msg to along the path to FragID.
                        } Else If (msg is FRAGMENT_DISSOLVED) {
                                Elected = True.
                                Delete nodes in F and SteinerQ.
                                FragID = ID.
                        } Else If (msg is MERGED) {
                                If (fragID != tempfragID) {
                                        Set multicast tree path for this node.
                                        FragID = tempFragID.
                                }
                                Forward message up the tree.
                        } Else If (msg is CL_ACK) {
                                Forward message up the fragment to the fragment leader.
                                Reverse these edges on the fragment.
                        } Else If (msg is CONNECT) {
                                /* ID is connection point in this fragment */
                                Forward msg to FragID.
                        } Else If (msg is CHANGE_LEADER) {
                                /* msg contains F_new, Vm_new, BUSY_Q_new */
                                F = F_new.
                                Vm = Vm_new.
                                BUSY_Q = BUSY_Q_new.
                                Elected = true.
                                Send ACK to FragID.
                                FragID = ID.
                        }
                }
        }

        If (Deleted) {
                goto Steiner.
        } Else If (Elected) {
                goto UpdatePhase.
        }
}
/* end function */
```

TryMerge

```
/* only leaders execute this code--see if preferred fragment will agree to merge */
```

```
State = TryMerge.
gotACCEPT = False.
gotRetry = False.
MergeFailure = False.
closest_req = Nodes[ID].dist.
Nodes[ID].deferredMR = -1.
If (Vp is on the BUSY_Q) {
        source = sender of the BUSY msg. /* Respond for the expected node in this fragment. */
        Remove Vp from BUSY_Q.
} Else {
        source = ID.
}
If (Vp == OrigVp AND sender != ID {
        /* we preempted another MR so send it again */
        gotBUSY = True.
        BusyID = -1.
} Else If (Vp != origVp) {
        gotBUSY = False.
        BusyID = -1.
} Else If (gotRetry) {
        /* got a MR from original Vp, it's on our BusyQ */
        gotBUSY = False.
        BusyID = -1.
} Else {
        gotBUSY = True.
        BusyID = Vp.
        /* already got a BUSY, don't send another MR */
        add Vp to BusyIDQ.
}

sender = closest child to Vp in f.
Send MERGE_REQUEST(ID, sender) to Vp.

While (not gotACCEPT and not MergeFailure) {
        Receive msg.
        If (msg is MERGE REQUEST) {
                /* message format is MERGE_REQUEST(source, sender, MRdist) */
                If (source not in Vm) {
                        Add source to Vm.
                }
                If (msg.sender == Vp || sender == BusyID || source in BusyIDQ) {
                        If (gotBUSY && BusyID == sender || source in BusyIDQ) {
                                If (source in f) {
                                        /* this was a forwarded request from
                                           within the fragment */
                                        Vp = source.
                                        If (sender is in this fragment) tail = source.
                                        else tail = sender.
                                        dist = MRdist.
                                }
                        }
                        gotACCEPT = True.
                        Send ACCEPT to Vp.
                } Else {
                        /* MERGE_REQUEST came from the new leader of Vp */
                        If (sender == Vp) {
                                Vp = msg.sender.
                                gotACCEPT = True.
                                Send ACCEPT to Vp.
                        } Else If ((MRdist - dist) < EPSILON && gotBusy) {
                                /* we'll accept a request from a closer node */
                                origVp = Vp.
                                origdist = dist.
                                Put current request (Vp) on BusyQ.
                                Vp = msg.sender.
                                dist = MRdist.
                                If (sender is in this fragment) tail = source.
                                else tail = sender.
                                gotACCEPT = True.
                                Send ACCEPT to Vp.
                } Else {
                        If ((MRdist - closest_req) < EPSILON) {
                                /* got a MERGE_REQUEST from a closer
                                   unknown node, just save it */
                                closest_req = MRdist.
                                deferredMReq = msg.sender.
                                deferredMReqDist = MRdist.
                        }
                        Send BUSY to msg.sender.
                        Add msg.sender to BusyQ.
                }
        }
} Else If (msg is ACCEPT) {
        If (msg.sender == Vp || msg.sender == BusyID || source in BusyIDQ) {
                gotACCEPT = True.
        } Else {
                Error.
        }
} Else If (msg is NOT_INTERESTED) {
        /* preferred node is deleted */
        If (msg.sender == Vp || msg.sender == BusyID || source in BusyIDQ) {
                MergeFailure = True.
                Mark Vp deleted in Vm.
        } Else {
                Error.
        }
} Else If (msg is BUSY) {
        /* message format is BUSY(busyVp) */
        BusyID = Vp.
        Add Vp to BusyIDQ.
        gotBUSY = True.
        Vp = busyVp.
        If (Vp in BusyQ) {
                remove Vp from BusyQ.
                Send Merge_Request to Vp.
        } Else If (deferredMReq != -1) {
                /* if there's a MERGE_REQUEST that's closer, use it */
                origVp = Vp.
                origDist = dist.
                Vp = deferredMReq.
                tail = Vp.
                dist =  deferredMReqDist.
                BusyID = -1.
                gotBUSY = False.
                Send MERGE_REQUEST to Vp.
        } Else If (BusyID != Vp & Vp not in BusyIDQ) {
                /* if preferred fragment has changed, send new req. */
                BusyID = -1.
                gotBUSY = False.
                Send MERGE_REQUEST to Vp.
        }
} Else If (msg is DELETE_SIGNAL) {
        MergeFailure = True.
        deleting = True.
} Else If (msg is FRAGMENT_DISSOLVED) {
        If (msg.sender == Vp) {
                MergeFailure = True.
        }
        Mark msg.sender as deleted in Vm.
} Else If (msg is JOIN_REQUEST) {
        /* message format is JOIN_REQUEST(late_add) */
        Add late_add to Vm.
        Delete late_add from SteinerQ if present.
        Add late_add to f.
        Send LATE_ADD_REPLY(Vm) to late_add.
} Else If (msg is CONNECT) {
        /* message format is CONNECT(source) */
        /* find path to send the NACK to */
        sender = neighbor(ID, source).
        Send NACK to sender.
} Else {
```

```
                Ignore message.
        }
}
If (MergeFailure) {
        goto UpdatePhase.
} Else {
        goto ConnectionPhase.
}
/* end function */
```

ConnectionPhase

```
/* leaders have agreed to merge, so see if the shortest path connecting
   the merging fragments is clear.  If it is, reserve the nodes along
   the path and complete the merge.  If it's not, abort the merge.
*/
State = ConnectionPhase.
MergeDecision = False.
If ((ID < Vp && Vp != Vs) || ID == Vs) {
        /* I'm the "active" leader */
        head = closest child to Vp in f.
        If (head == ID) {
                /* we're at the edge of a fragment, we connect outside frag */
                next = neighbor(ID, tail).
        } Else {
                /* we're still inside fragment, route directly to edge of frag */
                next = head.
        }
        Send CONNECT(head, tail, ID, Vp) to next node on the shortest path to a node in fragment Vp.
        While (not MergeDecision) {
                Receive msg.
                If (sender is head and msg is NACK) {
                        MergeDecision = True.
                        MergeSucceeded = False.
                        /* NACK contains blocked node's ID */
                        Add blocked node's ID to Vm.
                        Send NACK with attached blocked node's ID to Vp
                                (only if Vp did not send NACK).
                } Else If (sender is head and msg is ACCEPT) {
                        /* got an ACCEPT from someone else -- refuse it */
                        Send NACK to msg.sender.
                } Else If (sender is head and msg is NOT_INTERESTED) {
                        If (msg.sender == Vp) {
                                MergeDecision = True.
                                MergeSucceeded = False.
                                VpDeleted = True.
                                Mark Vp as deleted in Vm.
                        }
                } Else If (sender is head and msg is MERGED) {
                        /* message format is MERGED(source, F_Vp, F_Vm, BUSY_Q_Vp, SteinerQ_Vp) */
                        If (source != Vp) {
                                Print warning.
                        } Else {
                                MergeDecision = True.
                                MergeSucceeded = True.
                                Add node IDs in F attached to MERGED msg to F
                                Add node IDs in SteinerQ attached to MERGED msg to SteinerQ
                                Extract F_Vp from msg.
                                Extract F_Vm from msg.
                                Extract BUSY_Q_Vp from msg.
                                Extract SteinerQ_Vp from msg.
                                Merge F_Vp into Vp.
                                Merge Vm_Vp into Vm.
                                Merge BUSY_Q_Vp into BUSY_Q.
                                Send NEW_LEADER to nodes in SteinerQ_Vp.
                                Merge SteinerQ_Vp into SteinerQ.
                        }
                } Else If (msg is MERGE_REQUEST) {
                        /* message format is MERGED(source, sender) */
                        If (source == Vp or sender == Vp) {
```

```
                        /* the ACCEPT was lost because of late adds, resend */
                        Vp = source.
                        Send ACCEPT to Vp.
                    } Else {
                        Send BUSY with attacked msgID to msg.sender.
                        Add (sender's ID, identifier for BUSY) to BUSY_Q.
                    }
                } Else If (msg is CONNECT) {
                    /* Two fragments that don't know about us are
                       trying to Merge through us. */
                    Send NACK with ID along path toward sender.
                } Else If (msg is DELETE_SIGNAL) {
                    deleting = True.
                } Else If (msg is JOIN_REQUEST) {
                    /* message format is JOIN_REQUEST(late_add) */
                    /* Received a join request from a late add.
                       Add node to Vm and pass it our Vm */
                    Add late_add to BusyQ.
                    Remove late_add from SteinerQ if present.
                    Send LATE_ADD_REPLY with attached Vm to late_add.
                }
        }
} Else {            /* I'm the "passive" leader */
        While (not MergeDecision) {
                Receive msg.
                If (msg is NACK) {
                    MergeDecision = True.
                    MergeSucceeded = False.
                    /* NACK contains blocked node's ID, assumed to be a late add. */
                    Add blocked node's ID to Vm.
                } Else If (msg is ACCEPT) {
                    If (msg.sender != Vp) {
                        /* got ACCEPT from someone else, refuse it */
                        Send NACK to msg.sender.
                    }
                } Else If (msg is NOT_INTERESTED) {
                    If (msg.sender == Vp) {
                        /* Vp has deleted itself.  Have to try again with another node */
                        MergeDecision = True.
                        MergeSucceeded = False.
                        VpDeleted = True.
                        Mark Vp as deleted in Vm.
                    }
                } Else If (sender is Vp and msg is CONNECT) {
                    /* message format is CONNECT(head, tail, MergedFragID, OtherID) */
                    If (MergedFragID != Vp) {
                        /* Got a CONNECT from someone else, refuse it */
                        Send NACK to msg.sender.
                    } Else {
                        MergeDecision = True.
                        MergeSucceeded = True.
                        FragID = Vp.
                        tempFragID = Vp.
                        Set multicast tree path for this node.
                        Send MERGED(F, Vm, BUSY_Q, SteinerQ) to Vp.
                    }
                } Else If (msg is MERGE_REQUEST) {
                    /* message format is MERGE_REQUEST(source, sender) */
                    If (source == Vp or sender == Vp) {
                        /* ACCEPT msg was lost, probably because of a late add. Resend it. */
                        Vp = source.
                        Send ACCEPT to Vp.
                    } Else {
                        Send BUSY with attacked msgID to msg.sender.
                        Add source to Vm with BusySender set to ID.
                        Add (sender's ID, identifier for BUSY) to BUSY_Q.
                    }
                } Else If (msg is DELETE_SIGNAL) {
                    deleting = True.
                } Else If (msg is JOIN_REQUEST) {
                    /* message format is JOIN_REQUEST(late_add) */
```

```
                        Add late_add to Vm.
                        Remove late_add from SteinerQ if present.
                        Send LATE_ADD_REPLY with attached Vm to late_add.
                }
        }
}

If (MergeSucceeded || deleting || VpDeleted) {
        goto UpdatePhase.
} Else If (blockersFragID == ID) {
        /* don't try to merge with ourselves */
        goto UpdatePhase.
} Else {
        /* a NACK was received.  Try to merge with that node. */
        origVp = Vp.
        origdist = dist.
        Vp = blockersFragID.
        child = closest child to Vp in f.
        dist = distance between child and blocker.
        tail = blocker.
        goto TryMerge.
}
/* end function */
```

Steiner

```
/* nodes that are either not members of a fragment (i.e., have never been multicast participants) or
   have deleted themselves (are now "inactive" members of a fragment) execute the following code:
*/


If (FragID == ID) {
        Reserved = False.    /* only applicable to true Steiner nodes, not deleted frag members */
} Else {
        Reserved = True.
}

WantToJoin = False.  /* nodes can decide to (re-)join the multicast session */
HaveJoined = False.  /* true once a LATE_ADD_REPLY is received */
AddToLeader = True.  /* Used to make sure that the multicast path information
                        isn't updated in a way that forms a loop. */

While (not Terminate and not HaveJoined) {
        Receive msg.
        If (msg is CONNECT) {
                /* message contains (head, tail, MergedFragID, OtherID) */
                If (Reserved && FragID != OtherID && FragID != MergedFragID) {
                        next = neighbor of ID and head.
                        Send NACK(head, my ID) to msg.sender.
                } Else If (WantToJoin && not Reserved) {
                        next = neighbor of ID and head.
                        Send NACK(head, my ID) to msg.sender.
                } Else If (deleting && ID == OtherID) {
                        next = neighbor of ID and head.
                        Send NACK(head, my ID) to msg.sender.
                } Else {
                        /* forward the CONNECT message on */
                        If (FragID == OtherID) {
                                If (Already on a multicast tree path) {
                                        next = neighbor in the path.
                                } Else {
                                        next = neighbor of ID and OtherID.
                                }
                        } Else If (FragID == MergedFragID && ID != head) {
                                next = neighbor of ID and head.
                                If (next == msg.sender) {
                                        /* This Steiner node is farther than
                                           any node in Vm,so we actually want
                                           to route it to the tail, since head
```

```
                                    will take it backwards into the
                                    source fragment */
                                 next = neighbor of ID and tail.
                                 /* set flag so we won't set a loop
                                    in the multicast tree */
                                 AddToLeader = False.
                         }
                 } Else {
                         /* we're routing it towards the other frag */
                         next = neighbor of ID and tail.
                 }
                 If (not Reserved) {
                         Reserved = True.
                 }
                 tempFragID = MergedFragID.
                 If (AddToLeader && msg.sender != FragID) {
                         Set temporary multicast tree path for this node.
                 }
                 Forward CONNECT(head, tail, MergedFragID, OtherID) to next node on the path.
         }
         AddToLeader = True.
 } Else If (msg is NACK) {             /* merge is failing */
         Forward NACK(head, blocked node's ID) along temporary path.
         /* undo changes to temp variables */
         Reset temporary path and FragID to current path and FragID.
         If (FragID == ID) {
                 Reserved = False.
         }
         If (WantToJoin && not Reserved) {
                 next = neighbor of ID and Vs.
                 Send JOIN_REQUEST to next.
         }
 } Else If (msg is MERGED) {           /* merge is succeeding */
         /* message contains (MergedFragID) */
         Alone = False.
         If (MergedFragID != tempFragID) {
                 /* do nothing */
         } Else If (FragID != tempFragID) {
                 /* commit changes that are in temp variables */
                 Set multicast tree path for this node.
                 FragID = tempFragID.
         }
         Set next to next node along the path.        /* shortest path toward head. */
         Attach my ID to MERGED msg.
         Forward MERGED(head) msg to next.

         If (WantToJoin) {
                 /* CONNECT has succeeded, tell FragID we are late add */
                 IsolatedNode = False.
                 Send JOIN_REQUEST to FragID.
         }
 } Else If (msg is CL_ACK) {
         Send CL_ACK up the tree.
         Reverse multicast tree path toward n.
 } Else If (msg is CHANGE_LEADER) {
         Send NACK to msg.sender.
 } Else If (msg is NEW_LEADER) {
         FragID = msg.sender.
         tempFragID = msg.sender.
 } Else If (msg is MERGE_REQUEST) {
         If (FragID != ID) {
                 /* node is already in fragment, forward to the leader. */
                 Send BUSY with attacked msgID to msg.sender.
         } Else If (not WantToJoin) {
                 Send NOT_INTERESTED to msg.sender.
         } Else {
                 /* We are in the process of joining a multicast
                    group and will be a singleton fragment, so
                    send a BUSY and push this on the BUSY Q. */
                 Add (msg.sender's ID, identifier for BUSY) to BUSY_Q.
                 Add msg.sender to Vm.
```

```
                        Send BUSY with attacked msgID to msg.sender.
                }
        } Else If (msg is JOIN_REQUEST) {
                /* This is a late add, forward it to the source */
                next = neighbor of ID and Vs.
                Forward JOIN_REQUEST message to next.
        } Else If (msg is JOIN_SIGNAL) {
                WantToJoin = True.
                deleted = False.
                added = True.
                If (FragID != ID) {    /* already part of a fragment */
                        IsolatedNode = False.
                        Send JOIN_REQUEST with ID to FragID.
                } Else If (not Reserved) {
                        IsolatedNode = True.
                        next = neighbor of ID and Vs.
                        Send JOIN_REQUEST with ID to next.
                }
        } Else If (msg is DELETE_SIGNAL) {
                WantToJoin = False.
        } Else If (msg is LATE_ADD_REPLY) {
                If (WantToJoin) {
                        /* Add ourselves to f.  Only node in this fragment */
                        HaveJoined = True.
                        Add ID to f.
                        Extract Vm from message and add to Vm (or create Vm if it does not exist)
                        If (IsolatedNode) {
                                Vp = minimum distance of ID to nodes in Vm, f, and SteinerQ.
                                tail = Vp.
                                goto TryMerge.
                        } Else {
                                /* this node has been added to the multicast
                                   group by the fragment leader already.  The
                                   node will receive f and Vm when it gets
                                   an UPDATE_TABLES message. */
                                Goto UpdatePhase.
                        }
                }
        } Else If (msg is UPDATE_TABLES) {
                If (deleting) {
                        FragID = msg.sender.
                        tempFragID = msg.sender.
                }
                Send NOT_INTERESTED to msg.sender.
        } Else If (msg is FRAGMENT_DISSOLVED) {
                If (msg.sender = FragID) {
                        Reserved = False.
                        FragID = tempFragID = ID.
                        Remove multicast tree path information.
                } Else {
                        Send NOT_INTERESTED to msg.sender.
                }
        }
}
/* end function */
```

References

[1] F. Adelstein, F. Hosch, G.G. Richard III, L. Schwiebert, Bessie: portable generation of network descriptions for simulation, Proceedings of the Seventh International Conference on Computer Communications and Networks (IC3N'98) (1998) 787–791.

[2] A. Ballardie, Core Based Trees (CBT version 2) Multicast Routing Protocol Specification, RFC 2189, September 1997.

[3] A. Ballardie, Core Based Trees (CBT) Multicast Routing Architecture, RFC 2201, September 1997.

[4] A. Ballardie, B. Cain, Z. Zhang, Core Based Trees (CBT version 3) Multicast Routing Protocol Specification, Internet Draft draft-ietf-idmr-cbt-spec-v3-01, August 1998.

[5] F. Bauer, A. Varma, ARIES: a rearrangeable inexpensive edge-based on-line Steiner algorithm, IEEE Journal on Selected Areas in Communications 15 (3) (1997) 382–397.

[6] F. Bauer, A. Varma, Distributed algorithms for multicast path setup in data networks, IEEE/ACM Transactions on Networking 4 (2) (1996) 181–191.

[7] D. Chakraborty, C. Pornavalai, G. Chakraborty, N. Shiratori, Distributed routing for dynamic multicasting with advance resource reservation information, Proceedings of the 15th International Conference on Information Networking (ICOIN'01) February (2001).

[8] C. Diot, W. Dabbous, J. Crowcroft, Multipoint communication: a survey of protocols, functions, and mechanisms, IEEE Journal on Selected Areas in Communications 15 (3) (1997) 277–290.

[9] M. Doar, I. Leslie, How bad is naïve multicast routing?, IEEE INFOCOM March (1993) 82–89. San Francisco, CA.

[10] M. Garey, D. Johnson, Computers and Intractability: A Guide to the Theory of NP-Completeness, Freeman, San Francisco, CA, 1979.

[11] Y. Im, Y. Lee, S. Wi, Y. Choi, Delay constrained distributed multicast routing algorithm, Computer Communications 20 (1) (1997) 60–66.

[12] S.G. Kim, Y.-M. Choi, S.T. Kim, Y.S. Kim, A dynamic multicast tree for loosely coupled conference, International Multimedia Conference, Poster Paper, November 1999, Available online at http://www.kom.e-technik.tu-darmstadt.de/acmmm99/ep/kimsanggil/.

[13] J. Lin, R.-S. Chang, A comparison of the internet multicast routing protocols, Computer Communications 22 (2) (1999) 144–155.

[14] T. Pusateri, Distance Vector Multicast Routing Protocol, Internet Draft draft-ietf-dvmrp-v3-09, September 1999.

[15] S. Raghavan, G. Manimaran, C. Siva Ram Murthy, A rearrangeable algorithm for the construction of delay-constrained dynamic multicast trees, IEEE/ACM Transactions on Networking 7 (4) (1999) 514–529.

[16] B.H. Ryu, M. Murata, H. Miyahara, A dynamic application-oriented multicast routing for virtual-path based ATM networks, IEICE Transactions on Communications E80-B (11) (1997) 1654–1663.

[17] H. Schwetman, CSIM: a C-based, process-oriented simulation language, Proceedings of the 1986 Winter Simulation Conference (1986) 387–396.

[18] L. Schwiebert, R. Chintalapati, Improved fault recovery in core based trees, Computer Communications 23 (9) (2000) 816–824.

[19] A. Shaikh, S. Lu, K. Shin, Localized multicast routing, Proceedings of the IEEE GLOBECOMM'95 (1995) 1352–1356.

[20] M. Tsukada, Y. Takai, Distributed algorithms for dynamic Steiner tree problem. Information Processing Society of Japan SIGNotes Distributed Processing System, No. 095-010, 1999.

[21] D. Waitzman, C. Partridge, S. Deering, Distance Vector Multicast Routing Protocol, RFC 1075, November 1988.

[22] B. Waxman, Routing of multipoint connections, IEEE Journal on Selected Areas in Communications 6 (9) (1988) 1617–1622.

[23] P. Winter, Steiner problem in networks: a survey, Networks 17 (2) (1987) 129–167.