Building Dynamic Multicast Trees in Mobile Networks

Frank Adelstein
Odyssey Research Assoc.
Ithaca, NY 14850-1250
fadelstein@oracorp.com

Golden G. Richard III
Dept. of Computer Science
University of New Orleans
New Orleans, LA 70148
golden@cs.uno.edu

Loren Schwiebert
Dept. of Electrical and Computer Engineering
Wayne State University
Detroit, MI 48202
loren@ece.eng.wayne.edu

Abstract

As group applications become more prevalent, efficient network utilization becomes a major concern. Multicast transmission may use network bandwidth more efficiently than multiple point-to-point connections. Many algorithms have been proposed for generating "good" multicast trees, however, these algorithms have significant limitations for networks containing mobile hosts. Existing algorithms either do not support changes to the multicast group while building the tree or impose unrealistic restrictions, such as prohibiting overlapping modifications or forcing regeneration of the tree after each change. Clearly, to extend the range of applications that depend on multicast communication to mobile hosts, multicast tree algorithms must provide a flexible means for participants to join and leave the multicast group. We propose an efficient distributed algorithm that supports dynamic changes to the multicast group during tree building and allows overlapping join/leave operations. In this paper, we present the algorithm and initial simulation results.

Keywords: mobile networks, dynamic multicast groups, multicast protocols, distributed Steiner algorithms.

1. Introduction

Networked multimedia applications that use multicast communication, such as distance learning, cooperative design tools, and videoconferencing, are growing in popularity. There is substantial interest in making these applications available to mobile hosts. These applications, which are often long-lived, place high demands on the underlying network, and can have a dynamic set of participants. Further, 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.

Heuristic techniques are generally employed to generate static cost-optimal multicast trees, since generation of optimal trees (which can be modeled as the Steiner Tree problem) is NP-Complete [14]. Algorithms for generating multicast trees must typically balance "goodness" of the generated tree, execution time, and storage requirements. Another factor for tree generation algorithms is centralized versus distributed control. Centralized algorithms tend to be simpler, but the coordinating site can become a bottleneck and a single point of failure. Distributed algorithms can run more quickly and be more fault-tolerant, but tend to have higher communication costs and more complexity.

Building efficient multicast trees in environments with dynamic multicast groups is very difficult, since changes to the multicast group can occur even during generation of the tree. To be effective, algorithms to create multicast trees must generate correct trees quickly, even if multiple changes to the multicast group occur concurrently. When nodes join or leave a multicast session, the efficiency of the tree tends to degrade. In general, more membership changes increase the degradation of the multicast tree. When a multicast group changes significantly, it may be desirable to rebuild the tree, and again, the algorithm for generating the multicast tree must execute quickly while dealing with membership changes during generation. 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 may be more noticeable.

In a dynamic environment, requiring that the multicast group not change during the multicast tree setup could result in incorrect trees, so supporting concurrent updates is very important. Previous work in the area has focused on minimizing the execution time of the tree construction algorithm. 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.

2. Related Work and Problem Statement

There has been considerable work done on multicast route selection [5]. Most of the work, both centralized and distributed, takes one of three approaches: Steiner trees, source-based routing, or core-based trees. Steiner tree-based algorithms produce efficient trees. Heuristics are used to generate "good" rather than optimal trees, since generation of optimal trees is NP-Complete [14]. These trees generally use fewer network resources than the other two approaches, especially when there is a single source. Source-based schemes build a tree rooted at each source, but do not use Steiner-tree heuristics and tend to require more network resources. Other advantages, such as minimizing delay, are often the goal of these algorithms. Early schemes typically relied on periodic broadcasts to determine and maintain group membership, and so do not scale well to large multicast groups. Other approaches, such as Protocol Independent Multicast – Sparse Mode (PIM-SM), are being developed to address the problem of defining multicast groups in large networks. Core-based approaches [7] are most appropriate when there are multiple sources in the multicast group. One node is chosen as the center and all multicast traffic is routed to and through this central node. All sources must transmit through the core, so traffic concentrations can be high. In addition, the resulting multicast tree is likely to be less efficient for each source than separate multicast trees. There are drawbacks with Steiner trees as well, such as inefficient use of 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 addresses an important problem for Steiner tree based algorithms; however, the ideas should be extensible to other approaches to building multicast trees. Due to space constraints, we discuss related work focused on Steiner tree-based approaches.

Doar and Leslie [6] support using a "naïve" approach to create the multicast tree, which takes the union of all minimum cost paths from the source to the destinations. Simulations show that the generated trees generally have efficiencies within a factor of two of optimal. 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, when many nodes leave the multicast group, the performance 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. [11] present a multicast route selection algorithm that requires no global cost information and generally produces good trees. Although only localized information is required, the algorithm joins nodes to the multicast tree sequentially, so the algorithm is not completely distributed. Another approach, proposed by Im et al. [8], uses a delay-constrained algorithm, requiring *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. Supporting a dynamically changing network topology is discussed, however, the authors assume a static multicast group during the multicast tree setup.

Ryu et al. [9] propose centralized techniques for supporting dynamic changes in ATM multicast groups. The algorithm assumes a static multicast group during the multicast tree setup. The multicast tree is built by repeatedly selecting the minimum cost path from the current tree 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 joins or leaves. The algorithm uses the probability of nodes joining or leaving the multicast group to design trees that facilitate the sharing of paths and produce a relatively large number of leaf nodes.

Bauer and Varma [4] 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). 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. In the discovery step, a "flood to N" approach is used to find the closest nodes, to propagate information about the fragment leader to fragment members, and to send 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. 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 merge request 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. This could be especially problematic in a multicast group with mobile hosts, since the location of these hosts could change while the multicast tree is being built.

2.1. 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. Leaves of the multicast tree are multicast nodes, although non-participating

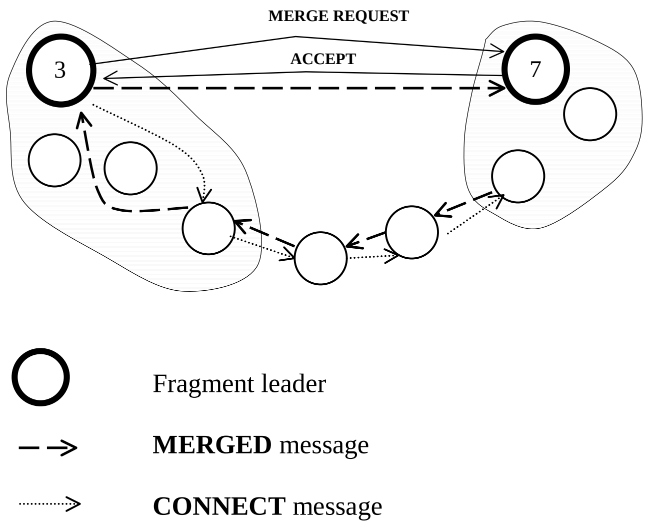

Figure 1. Two fragments merge successfully.

nodes (*Steiner nodes*) may be required to form the tree. The algorithm is fully distributed and handles changes to the multicast group during execution.

2.2. 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. The network includes mobile hosts, which are connected to the fixed network through base stations. Participating mobile hosts must be leaf nodes or the source. 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. Distinct edges may have different costs, but the cost for an edge between two nodes is the same in both directions. Although the network may lose packets due to congestion or noise, the transport layer delivers messages in order in finite time and does not drop or corrupt messages. This implies that no permanent 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

In Section 3.1, we describe our basic distributed algorithm for building an efficient multicast tree. Section 3.2 extends the algorithm to handle changes to the multicast group during generation of the tree. Section 3.3 addresses termination of the algorithm and pruning the generated tree of unneeded connections. The *basic* algorithm generates a correct tree provided the following conditions hold:

✓ The 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 [4], such as connecting fragments through the shortest path and selecting a preferred 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 sources is possible by maintaining a unique multicast session ID and separate data structures for each tree.

3.1. Basic Algorithm

Each node stores the following local variables:

✓	*ID*	(the node's unique identifier)
✓	*FragID*	(identifier for the fragment)
✓	*F*	(list of nodes in this fragment)
✓	*Vm*	(nodes wishing to be in the multicast)
✓	*Vs*	(multicast source node identifier)

Initialization

In the initialization step, a fragment is created for each node in *Vm*. Initially, each node is the *fragment leader* of its fragment. Each fragment has a list of all nodes in the initial multicast group. This information is used to build a list of merge candidates. Each node has access to a routing table and can determine the cost of transmitting a message to other nodes. The fragment leader is responsible for coordinating mergers with other fragments and for updating group members in its fragment. Each node forwards multicast messages to its children — the other members of its fragment to which it is directly connected.

Merge Negotiation

Each node looks through 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.

Once a fragment leader selects a preferred node, it sends a MERGE REQUEST containing *FragID* to that node and waits for a reply. 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 leader for processing and transmits a BUSY reply, with its *FragID* attached, to inform the sender of the identity of the fragment leader.

When a node receives a BUSY reply to a MERGE REQUEST, the fragment sending the BUSY initiates a MERGE REQUEST later, so the node receiving the BUSY waits for a MERGE REQUEST. A node receiving a BUSY reply to a MERGE REQUEST may ACCEPT a

MERGE REQUEST from another fragment, if the cost is less than or equal to its current preferred fragment.

Connection phase

The purpose of the connection phase is to join two fragments. The fragments are joined 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 belong to a different fragment, the merge attempt fails.

Fragment leaders entering the connection phase perform the following actions, as illustrated in Figure 1. The fragment leader with the lower *ID* sends a CONNECT message along the shortest path between the fragments. 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 a member of another fragment or reserved, the merge fails. The node sends a NACK backward along the shortest path. 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 restarts with the selection of another preferred node. If the CONNECT message reaches the tail of the shortest path between the fragments, a MERGED message is sent back along the shortest path between the fragments. The MERGED message makes the reservations permanent and propagates a list of Steiner node *ID*s back to the leader, who adds the new members *F*.

The leader node with the lowest node *ID* becomes the leader of the combined fragment. The node with the higher *ID* sends its fragment membership list, via an UPDATE TABLES message, to the new leader, who adds these members into the new fragment membership list. The leader of the combined fragment calculates a new preferred node and multicasts an UPDATE TABLES message to the other fragment members. This message contains the leader's identity, the list *F*, the current preferred node, and its cost. When a fragment member receives the UPDATE TABLES message, it updates *FragID* and *F*, and then computes its own preferred node. If this preferred node is closer than the one suggested by its leader, it returns an UPDATE message 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 process repeats. The algorithm terminates when the leader determines that there are no other fragments ($Vm \subseteq F$). Detailed pseudocode appears in [1].

3.2. Dynamic Algorithm

A practical distributed algorithm must 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. The *BUSY_Q* tracks the *ID* of nodes to which BUSY messages have been sent. In addition, *F* and *Vm* must be augmented to allow entries to be marked as *added* and *deleted.*

Join 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. (Join requests from nodes already in a fragment are handled locally within that fragment.) In this section, we describe how the 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, it sends a JOIN REQUEST toward *Vs*. If the tree has already been built, this request is processed in the network by an independent protocol that dynamically adds group members to an existing tree [3], [6].

Otherwise, the JOIN REQUEST must be intercepted by our tree-building protocol and processed as a late join. As the JOIN REQUEST propagates toward the multicast source, it either encounters another fragment or reaches the source. The fragment member that receives the JOIN REQUEST forwards it to the fragment leader, which adds this node to its copy of *Vm*. 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 the fragments merge.

A LATE JOIN REPLY is returned to the new node. This message contains the current list of multicast group members known to the responding fragment. Upon receipt of this reply, the new node determines its preferred node using 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 *Vm*.

If the preferred fragment has deleted itself from the multicast group, then the MERGE REQUEST is rejected 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. For this pathological case, 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.

There are two more issues with late joins. First, a fragment could ACCEPT a MERGE REQUEST from a node that is not the closest, as it is unaware of closer late joins. This does not affect correctness, only efficiency. Second, 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. The blocking node then becomes a candidate for preferred node.

Leave Requests

Leave requests are more complicated than join 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 *F* and *Vm*.

If the node is in a fragment with more than one member, it sends 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 fragment leader wishes to leave the multicast tree, it selects the remaining member of the fragment with the smallest *ID* and requests that that node take over leadership duties via a CHANGE LEADER message. Once the new fragment leader has been “elected”, 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 nodes in the fragment with an UPDATE TABLES message. When no other group members remain in a fragment, the fragment is dissolved.

Before the fragment is dissolved, nodes that received a BUSY response from this fragment must be informed so they do not wait forever. To track BUSY messages, an entry is added to the *BUSY_Q* whenever a BUSY is sent in response to a MERGE REQUEST. When the fragment is dissolved, the leader sends FRAGMENT DISSOLVED messages to each node with an entry in the *BUSY_Q*.

Combining Fragment Information

When two fragments are merged, they may have an inconsistent view of the multicast group. Inconsistent views may be caused by late joins and leaves. Any inconsistent information is reconciled by the new fragment leader and then multicast to all nodes in the newly merged fragment. This can be done by including this information in the UPDATE TABLES message.

The combined *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 *Vm* is the union of the two original multicast group membership lists. 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 sent a NOT INTERESTED reply to one fragment and a JOIN REQUEST to the other remains a candidate for merging. Finally, the *BUSY_Q* set for the combined fragment is the union of the two *BUSY_Q* sets.

3.3. Termination and Tree Refinement

The algorithm terminates 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 built. This can be done by bounding the number of joins that a fragment accepts. Subsequent JOIN REQUESTs are then processed as if the tree has already been built.

Once the algorithm has completed, it may be beneficial to run an optional protocol that prunes leaf nodes that are marked deleted or are Steiner nodes. The state information maintained by multicast group members and Steiner nodes may be reduced or eliminated once the tree is built.

4. Simulations

The simulation suite includes a network generation program called **Bessie** [2] that generates and displays network topologies as described by Waxman [13] and Doar [6]. Bessie also provides an improved edge model to provide greater control over the generated topologies.

The simulator, **mcSIM**, was written in C using CSIM18 [10]. The Phase I implementation is intended to verify the correctness of the protocol without late joins or leaves. (The Phase II implementation will include late joins and leaves; Phase III will include fault-tolerance.) **mcSIM** simulates the protocol running on every node in the network. 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 only in non-Steiner nodes. The tree construction protocol terminates when the multicast tree build is complete. **mcSIM** then writes an output file that can be displayed by **Bessie**. **Bessie** checks the integrity of the generated multicast tree and provides various statistics, such as overall tree cost.

4.1. Simulation Results

The results of the Phase I implementation of the simulator are shown in the following three tables, where **mcSIM** is compared with Doar and Leslie’s protocol. The results show the percentage improvement of our protocol versus Doar and Leslie's for 25 networks. In the first table, networks with 10% multicast nodes and an average of three links per node were simulated for varying network sizes. The results show that, in some cases, our protocol is significantly better, with an average improvement of about 25%. Similarly, in table 2, the network size is fixed at 200 nodes and the number of multicast members is varied. In table 3, the average number of links is varied. All these results show that our protocol produces high-quality trees.

Table 1. Avg. node degree = 3, 10% multicast nodes.

# 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 2. 200 nodes, average node degree = 3.

% 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 3. 200 nodes, 10% multicast nodes.

Avg 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%

5. Conclusion and Future Work

We have presented a distributed algorithm for construction of a multicast tree in environments in which the multicast group membership is dynamic. 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 algorithm is suitable for use in many situations that require the generation of multicast trees. Examples include multicast groups with mobile hosts and dynamic regeneration of sub-trees that have experienced substantial degradation due to local changes in the multicast group.

Future work includes extending the protocol to incorporate fault tolerance and building this algorithm into a multicast protocol that supports dynamic trees with periodic rebuilds of locally inefficient sub-trees. This would be useful for long-running multicast sessions with periodic multicast group changes. The combination of our technique with *geocasting* algorithms for mobile networks [12] is also under consideration.

6. Acknowledgments

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

7. References

[1] F. Adelstein, G. G. Richard III, and L. Schwiebert, "Distributed Multicast Tree Generation with Dynamic Group Membership", University of New Orleans, Computer Science Technical Report UNOCS-TR97-01.
[2] F. Adelstein, G. G. Richard III, and L. Schwiebert, ""Bessie: Portable Generation of Network Descriptions for Simulation," Proc. 7th International Conference on Computer Communications and Networks, pp. 787-791, October 1998.
[3] F. Bauer and A. Varma, "ARIES: A Rearrangeable Inexpensive Edge-Based On-Line Steiner Algorithm," IEEE Journal on Selected Areas in Communications, 15(3), pp. 382-397, April 1997.
[4] F. Bauer and A. Varma, "Distributed Algorithms for Multicast Path Setup in Data Networks," Proceedings of GLOBECOM '95, pp. 1374-1378, Nov. 1995.
[5] C. Diot, W. Dabbous, and J. Crowcroft, "Multipoint Communication: A Survey of Protocols, Functions, and Mechanisms," IEEE Journal on Selected Areas in Communications, 15(3), pp. 277-290, April 1997.
[6] M. Doar and I. Leslie, "How Bad is Naïve Multicast Routing?," IEEE INFOCOM, pp. 82-89, San Francisco, California, March 1993.
[7] S. K. S. Gupta, and P. K. Srimani, "An Adaptive Protocol for Reliable Multicast in Mobile Multi-Hop Radio Networks," Proc. 2nd IEEE Workshop on Mobile Computing Systems and Applications, pp.111-122, February 1999.
[8] Y. Im, Y. Lee, S. Wi, and Y. Choi, "Delay Constrained Distributed Multicast Routing Algorithm," Computer Communications, 20(1), pp. 60-66, January 1997.
[9] B. H. Ryu, M. Murata, and H. Miyahara, "A Dynamic Application-Oriented Multicast Routing for Virtual-Path Based ATM Networks," IEICE Trans. on Communications, E80-B(11), pp. 1654-1663, November 1997.
[10] H. Schwetman, "CSIM: A C-Based, Process-Oriented Simulation Language," Proc. 1986 Winter Simulation Conference, pp. 387-396.
[11] A. Shaikh, S. Lu, and K. Shin, "Localized Multicast Routing," Proc. IEEE GLOBECOMM '95, pp. 1352-1356.
[12] Y.-B. Ko, N. H. Vaidya, "Geocasting in Mobile Ad Hoc Networks: Location-Based Multicast Algorithms," Proc. 2nd IEEE Workshop on Mobile Computing Systems and Applications, pp.101-110, February 1999.
[13] B. Waxman, "Routing of Multipoint Connections," IEEE Journal on Selected Areas in Communications, 6(9) pp. 1617-1622, December 1988.
[14] P. Winter, "Steiner Problem in Networks: A Survey," Networks, 17(2) , pp. 129-167, 1987.