A LIGHTWEIGHT ADAPTIVE MULTICAST ALGORITHM*

Lusheng Ji and M. Scott Corson
(lji, corson@isr.umd.edu)
Institute for Systems Research
A.V. Williams Building (115)
University of Maryland
College Park, Maryland 20742
(phone) 301-405-6630, (fax) 301-314-8586

Abstract

In this paper, we present a multicast protocol which is built upon the Temporally-Ordered Routing Algorithm (TORA). The protocol—termed the Lightweight Adaptive Multicast (LAM) routing algorithm—is designed for use in a Mobile Ad hoc NETwork (MANET) and, conceptually, can be thought of as an *integration of the CORE Based Tree (CBT) multicast routing protocol and TORA*. The direct coupling of LAM and TORA increases reaction efficiency (lowering protocol control overhead) as the new protocol can benefit from TORA's mechanisms while reacting to topological changes. Also during periods of stable topology and constant group membership, the LAM protocol does not introduce any additional overhead because it does not require timer-based messaging during its execution.

Technical Subject Areas: Mobile Communications Systems, Internet Technology, Military Communications

1 Introduction

We consider the problem of providing datagram multicasting service in a large Mobile Ad hoc NETwork (MANET) environment as described in [7]. In MANETs all units are equipped with wireless communication interfaces and can move at will. New links are established when two nodes move close enough together to permit wireless communications, and existing links are broken when the communications between two nodes is disrupted. The combination of node mobility and a wireless environment result in MANET topologies subject to rapid and unpredictable configuration changes.

While in the inter-domain multicast routing [2] research, both source-oriented and group-shared tree construction mechanisms are adopted [1, 4, 3, 5], group shared trees are better-suited for the wireless MANET environment because of two issues associated with the source-oriented tree approach. One issue concerns network control overhead where, with the source-oriented approach, one tree is needed for each (group, source) pair, whereas with the shared-tree approach, only one tree is needed per group. Due to the possibility of high topological change rates in MANETs, the burden of updating the greater number of source-oriented trees may create excessive control traffic. The second issue concerns aggregate bandwidth consumption by source-oriented trees. While source-oriented trees result in lower average data delivery latency, they consume more bandwidth on average than center-based trees—the type of trees formed by group-shared tree protocols [12]. Thus, from both network control and data traffic perspectives, source-oriented trees are less bandwidth-efficient than shared trees, thus favoring the use of group-shared tree approaches in bandwidth-constrained environments.

Existing inter-domain group-shared tree multicast routing protocols are not particularly well-suited for a MANET's dynamic network environment. Many of those protocols rely on distance vector information, which is provided by underlying unicast protocols, and a reverse-path forwarding mechanism to setup and update the multicast routing tree. However, in large MANETs, the shortest-path and reverse-path may change faster than an underlying shortest-path unicast routing protocol can compute, and to which the multicast protocol can react. This problem is more serious if the multicast protocol is designed atop a link-state unicast routing protocol (e.g. MOSPF [8]). A large, bandwidth-constrained MANET's topology may change faster than a link-state algorithm can track [11].

In this paper we propose a multicast routing algorithm which sits on top of a unicast routing protocol specially designed for large-scale, rapidly-changing MANETs — the Temporal Ordered Routing Algorithm (TORA) [9]. We argue that instead of making a multicasting protocol which theoretically functions above any unicast routing protocol, a tightly-coupled unicast-multicast routing suite will benefit the whole MANET routing architecture even more. Although a decoupled multicasting protocol is portable across unicast algorithms, it is not efficient in many cases. In this form of vertically-decoupled design, too often the upper layer protocols have to duplicate control overhead functionality that is already provided by some lower layer protocols resulting in bandwidth and energy inefficiency.

TORA provides a well-defined unicast routing infrastructure upon which to build group-shared tree

*Prepared through collaborative participation in the Advanced Telecommunications/Information Distribution Research Program (ATIRP) Consortium sponsored by the U.S. Army Research Laboratory under the Federated Laboratory Program, Cooperative Agreement DAAL01-96-2-0002.

0-7803-4984-9/98/$10.00 ©1998 IEEE.

multicasting. TORA is distributed and highly-adaptive. Most network control is localized in TORA. These are desirable characteristics to have underlying one's multicasting protocol. In addition, TORA provides multiple routes. If failures occur along one route, the next packet will likely take an alternative route at the node just prior to the failure point. The portion of the route before this point stays unchanged. Mapping this feature into the multicasting domain when one multicasting tree link is failing, which may happen at a high rate in MANETs, the change to the multicasting tree can be localized near the failure point. The change is not typically propagated to places far away from the failure point. Finally, TORA maintains a global order over all nodes in the network involved in unicast routing. By borrowing these totally-ordered labels, loop freedom in multicasting can be achieved with minimal effort.

LAM offers the basic functionalities of a multicasting protocol, that is to construct and maintain routing path tree(s) inter-connecting members of the multicasting group so that all multicast data can be forwarded using only the tree paths. Atop LAM, algorithms such as those using routing labels [6] can be implemented.

2 Related Work

2.1 Core Based Tree (CBT)

The design of LAM is greatly inspired by Ballardie's Core Based Tree (CBT) inter-domain multicast routing protocol, which builds a group-shared distribution tree centered at a single special node called a "CORE". Many behaviors of LAM look very similar to those of CBT. Such behaviors are also very common in this family of "group-shared tree" multicasting protocols. The protocol of CBT is well known and described in detail in [1], and is therefore omitted here.

2.2 Temporally-Ordered Routing Algorithm (TORA)

LAM is built upon the TORA routing protocol. TORA has been shown [11] to be well-suited for large, bandwidth-constrained networks with high topological change rates. In TORA, all nodes that are involved in routing to a certain destination are totally ordered by labels called "heights". By accessing the height information maintained in TORA, LAM is able to achieve loop-freedom when constructing and maintaining a multicasting tree. The following gives a brief description of TORA—its full definition and specification can be found in [9, 10].

TORA is a "source-initiated" routing protocol; i.e. routes are built reactively, or "on demand", as needed to reflect current traffic patterns[1]. On each node involved in routing from source to destination, one version of TORA is running for the destination. Using available network links, TORA builds a Directed Acyclic Graph (DAG) over the physical network topology. Each link in this DAG is directed from a node with a higher height "downwards" to a neighboring node with a lower height[2]. The destination always has the lowest height in the DAG, denoted as the "zero" height. The routed packets "flow" downwards along the directed links to reach their destination. Thus, sequences of links with "down" directions form routes towards the destination. Typically TORA builds multiple routes towards the destination. The heights of the nodes along any route strictly decrease. TORA assigns and updates the heights.

Three types of control packets are employed in TORA: query (QRY), update (UPD), and clear (CLR). Initially every node with no routes towards the destination has a NULL height. When a source has a message to send to the destination, but is in lack of a route, broadcasts a QRY packet to all of its neighbors to discover a path to the destination. If the receiving node does not have a route either, it rebroadcasts the QRY message. If the receiving node has a route already, it broadcasts a UPD message to announce its height. Once a node that is waiting for a route hears the UPD message, it assigns itself a height according to the height in the UPD message, then announce its new height using an UPD packet. With its own height, and the knowledge of neighbors' heights, a node assigns directions to its links. Any "downstream" links (directed to a lower neighbor) can be used in future routing towards the destination.

To minimize the amount of control messages, TORA does not react to downstream link losses as long as there are still downstream links available. In the event of a node losing its last downstream link, it may need to discover a new route towards the destination. At this time, it sets its height to be the "highest" in the network (using the current time value, which is typically the newest and therefore the highest time value of any height in the network) and announces its new height to all of its neighbors in a UPD packet. This UPD packet will typically reverse some links' directions on some neighbors (those previously which were "upstream" of the UPD sender with respect to the destination). If such a link reversal causes another node to lose its last downstream link, it adjusts its height to be "lower" than the height of the sender of the UPD packet and broadcasts an UPD packet. If the node still has downstream links after it updates its neighbor height information using the new height in the UPD packet, it does not change its height. This process of link reversal continues until either all nodes have at least one downstream link or a network partition is detected. In the case of network partition (when there are no possible routes to the destination), routes are erased by a CLR packet sent by a node who detects the network partition.

[1] It should be noted that is easy to modify TORA to run in a "destination-initiated" mode, a.k.a. a "proactive" or "traditional" routing mode, as well. In this mode, all nodes, at the request of the destination, actively seek to maintain routes to the destination regardless of current traffic demands. Such a mode will likely be defined in future versions of [10].

[2] The exact composition of a TORA height and the rules for its modification are not important here and are omitted.

3 The LAM Protocol

As with CBT, LAM builds a group-shared multicast routing tree for each multicast group. This tree is centered at a pre-selected node called a CORE. It shares some common behaviors with other protocols in this family, especially CBT. However, LAM and CBT differ significantly in the following ways.

1. *Scalability and Adaptivity:* Although in general group-shared tree algorithms are more scalable than source-driven tree algorithms, CBT in particular is not as scalable as it could be. In CBT when an on-tree node discovers the link between itself and its parent is broken, it must flush the sub-tree below itself. After all nodes on the sub-tree remove themselves from the multicasting tree, they start rejoining the tree individually. When a link that is close to the CORE breaks, there can be a large number of nodes in the sub-tree under the broken link. They will all be flushed out. Multicasting service is interrupted to these nodes. It also takes a lot of effort for them to rejoin the tree resulting in higher than necessary time and communication complexity. In a MANET environment, such flushing behavior is not desirable because links may break at a much higher rate than in fixed network. In LAM, there is no flush operation. The portion of the tree that is affected following any failure is typically highly localized.

2. *Lightweightness:* LAM's tree maintenance phase does not utilize timers. Its tree maintenance control traffic is proportional to the rate of network changes. Therefore when tree links in the network are stable, no control messages are generated by LAM. Because LAM is closely coupled with TORA, whenever there is a neighbor link status change, TORA will notify LAM through their interface. Therefore LAM control operations are event-triggered. LAM does not need to check link status through periodically polling.

3. *Simplicity:* For on-tree non-CORE nodes, LAM only needs three types of control messages which perform very straightforward functions. The protocol's simplicity is achieved by taking advantage of TORA state information. Since TORA already builds a DAG destined at the CORE, LAM does not need to worry about loop formation as long as all child-to-parent tree links follow the downstream links in the TORA DAG. The resulting multicast protocol is simpler than CBT.

LAM is implemented as a sub-layer in the network layer. In the protocol stack, LAM runs on top of TORA. LAM makes the same assumptions about lower layers as does TORA [9]. They are the following: (1) a lower-level protocol ensures that each node knows their neighbors; (2) all transmitted packets are received correctly and in order of transmission; and (3) each node is able to "broadcast" to all of its neighbors.

We now present a conceptual description of LAM—its specification is given in Appendices B and C.

Data Object Values	State
(**Parent**==NULL) && (**PParent**==NULL)	off-tree
(**Parent**==NULL) && (**PParent**!=NULL)	join-waiting
(**Parent**!=NULL) && (**PParent**==NULL)	on-tree
(**Parent**!=NULL) && (**PParent**!=NULL)	not valid

Figure 1: Node States

3.1 Data Structures

In each node's LAM layer, two variables, **Potential-Parent** and **Parent**, and two lists, **Potential-Child-List** and **Child-List**, are kept for recording the tree notation. The **Parent** variable is used to remember the parent node in the multicasting tree. The **Child-List** stores identities of one-hop children in the multicasting tree. **Potential** data objects are used when the node is in a "join" (or "rejoin") waiting state. Lastly, LAM needs to remember the multicast group for which it is running which is stored in the **Mcast-Group** variable, and a flag **member** to remember if a node wants to be a member of the group. Initially, in an off-tree node, all data objects are set to NULL. There is no variable explicitly describing the state of a node as either "on-tree", "join-waiting", or "off-tree", but the state information is implied by the combination of the two parent data objects as shown in figure 1.

3.2 Join

When a node is interested in joining a particular multicast group, it needs to join the multicast forwarding tree. The join process starts with LAM setting the **member** flag indicating that this node is interested at the group, then generating a JOIN message containing the group id, the target CORE id. By looking at the link status table in the version of TORA running for the target CORE, LAM picks the neighbor with the lowest height as the receiver of this JOIN message and sends away the message. The **Potential-Parent** variable is set to this lowest-height neighbor. The joining node is now in a join-waiting state.

When an off-tree node receives a JOIN message, it first checks if the link over which the message has arrived is listed as a downstream link in the link status table of the version of TORA running for the target CORE. Since the JOIN message is only supposed to travel along a "downwards" path in the TORA DAG with respect to the target CORE, if a JOIN message has received over an upstream link, it is considered invalid. When an invalid JOIN message is received, the node needs to send a LEAVE message back to the joiner as a rejection. Otherwise, the node performs the same algorithmic steps as a join initiator, with the additional step of putting the JOIN sender into its **Potential-Child-List**. The purpose of the **Potential-Child-List** is to record all neighbors from which a JOIN message has been received while the current node is in the join-waiting state.

The LEAVE message back to sender is necessary for LAM to run correctly. Although the JOIN sender

will only send the message via a TORA "down" link, while TORA is adjusting its DAG such state information may be wrong and two neighboring nodes may disagree about the state of the link between them. Among all possible disagreeing situations, only the case where *both* ends think the link between them is a downstream link may cause problem since JOIN operations only occur over downstream links. Such disagreement is only temporary. Eventually one of nodes will be corrected after TORA settles. If the JOIN receiver is the mistaken party, to the sender everything looks fine so it will not know that its JOIN message has been discarded due to the receiver's temporary link status information. Some mechanism must be used to reset the state on the sender side. That is why LEAVE message must be sent back to the JOIN sender if the JOIN message is from an invalid link.

After a JOIN sender receives a LEAVE message from its joining target, it needs to erase the current join waiting state, then choose the current best join candidate and send a new JOIN message towards it. It is possible that this new candidate is the same as the one picked before. However, after one round trip delay (first the JOIN message, then the LEAVE message), the TORA on the candidate node may have changed its link status information so that new JOIN will be accepted.

If the node who has just received the JOIN message is already in a waiting state, it does not need to send another JOIN message. However, it does need to add the sender of that JOIN message to its **Potential-Child-List**. Since each node checks if the JOIN message is not from a TORA "down" link, only JOIN messages that are on valid routes towards the target CORE are processed.

It might occur that the underlying TORA does not have a route towards the target CORE when the LAM process is generating the JOIN message. In this case, LAM will request TORA to build a route towards the CORE. After TORA succeeds in route construction, TORA will send a link status change (some link status change from "unknown" to "down") notification to LAM. Triggered by this notification, LAM starts the join process. The details is in the 3.5 section.

3.3 Join Acknowledgment

If a JOIN message reaches the CORE, the CORE will acknowledge the JOIN. Also, when a JOIN message is received by an on-tree node, and the message is received via a valid non-downstream TORA link, it is accepted. The receiving node replies by sending an ACK message to the joining node. The on-tree node adds the joining node to its **Child-List**.

Reception of a valid ACK message changes a node's state from join-waiting to on-tree. Only an ACK from the waiting node's **Potential-Parent** is considered valid. After the ACK is recognized, the receiver copies the value of the **Potential-Parent** to **Parent**. If the **Potential-Child-List** is not NULL, the node broadcasts an ACK to all nodes in its **Potential-Child-List**. Then, it copies all entries in the **Potential-Child-List** to **Child-List**, and sets both the **Potential-Parent** and **Potential-Child-List** to NULL.

When an ACK is received from a node which is not the current node's **Potential-Parent**, a LEAVE message is sent back to the acknowledger to remove the corresponding child entry. This is done to avoid loop formation as a loop may be formed starting from the child, leading to the common ancestor of the child and the false parent, then to the false parent, and finally back to the child.

If, during the course of a join, some of TORA's link status's change, LAM is notified of these changes. Upon notification, LAM nullifies all data object entries corresponding to these links. Depending on the changes and its current state, LAM will act accordingly as specified in the subsequent section on tree maintenance.

3.4 Leave

When an on-tree node who is not interested in the multicasting group has an empty **Child-List**, it can leave the tree by sending a LEAVE message to its parent, and then removing all information regarding this multicast group. Upon reception of this LEAVE, the parent removes the sender from its **Child-List**. When the parent's **Child-List** becomes empty and it is not interested in the group membership, it can also leave the tree by sending a LEAVE message to its parent.

If a join-waiting node that is not interested in the group sees that both of its **Child-List** and **Potential-Child-List** become NULL, it can leave the tree as well by sending a LEAVE message to its **Potential-Parent**. Its **Potential-Parent**, in turn, removes the sender of the LEAVE message from its correct child lists.

A LEAVE message serves multiple purposes in LAM protocol. When a LEAVE message is received by a node from one of its children, it means a request for tree branch pruning or a rejection to an ACK (which have the same effect on the receiving node). When a LEAVE message is received by a join waiting node from its **Potential-Parent**, it means a rejection to a JOIN. All other cases are invalid. Although a LEAVE message may carry different semantics, the processing of LEAVE messages with different meanings is almost identical. Based on this observation, and the fact that no different information is needed for different uses of a LEAVE message, it is not necessary to add new message types. Upon receiving a LEAVE message, a node removes the corresponding occurrence of the message sender from its data record. If the LEAVE message sender does not appear in any of these data objects, the message is simply discarded. To help the LEAVE message receiver interpret the true meaning of the message, the message has a flag (**As-Child**) indicating the relationship between the sender and the receiver. When the sender sends the LEAVE message as a pruning request or a rejection to an ACK, it sets this bit. Otherwise this bit is unset. When the receiver sees that this bit is set in the LEAVE message, it only checks its child entries (**Potential-Child-list** and **Child-List**) for occurrence of the message sender. Otherwise it only checks its parent entries (**Potential-Parent** and **Parent**).

In general, the reception of a valid LEAVE message will cause the removal of some entry from LAM's data objects. Such removal may have consequences if any of the data objects are nullified. This processing is described in the next subsection and in the Appendix.

3.5 Maintenance

When there is any topological change, TORA reacts accordingly to maintain its DAG. This usually results in link status changes involving nodes near the topology change location. At these nodes, after TORA finishes its reaction, it notifies LAM about link status change through an interface operation. A list of links whose status have just been changed is sent to LAM. LAM checks all of its data objects for the usage of these links, and nullifies entries corresponding to these links. When any of LAM's data objects becomes empty because of this removal, further actions may be required.

After a node's LAM finishes updating its data objects due to the reception of a link state change notification from TORA, or a valid LEAVE message, if its **Potential-Parent** and **Parent** are both NULL, a node reacts depending on the situation of its child lists. After the removal, if a node's **Child-List** and **Potential-Child-List** are both empty, but it still has a parent, it needs to check whether to generate a LEAVE message. If a node loses all links to its children (both **Child-List** and **Potential-Child-List**), then unless the node itself is a member of the group, this node can forget about the multicasting group and send a LEAVE message if its **Potential-Parent** or **Parent** is not NULL. In all the other situations the node needs to generate a new JOIN message towards the CORE, and start the rejoin process. Figure 2 shows the decision tree for the operation.

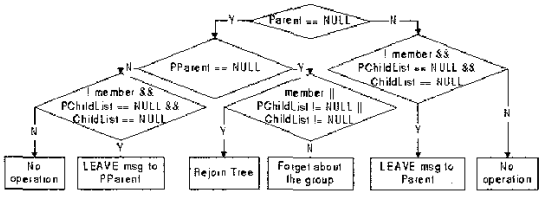

Figure 2: LAM decision tree

The rejoin process is the same as the join process. The only difference is that the child list might not be empty during the rejoin process. Since this node was on-tree before the link breakage, its **Child-List** may contain entries. During the rejoin, LAM does not modify the list unless TORA signals a link status change on those links to entries in the list. The sub-tree underneath the change is kept as much as possible. This way if after the change there are still downstream links in the TORA DAG available for this node, the effect of the change is kept local (i.e. it is not spread to the sub-tree underneath) since TORA does not react while there are still downstream links available. If there is no downstream link towards the CORE, TORA will change the link status's at both this node and its children. So the children will know that their parent is gone (indirectly through TORA) and they will start rejoining by themselves. The effect of the worst case is similar to a sub-tree tear-down in CBT. However, to guarantee loop-freedom, in CBT this relatively expensive sub-tree tear-down is performed every time a node wants to rejoin. This operation is performed recursively until it destroys the whole sub-tree below the breaking point. In LAM, the tear-down is actually done *indirectly* by TORA changing neighbor link status instead of relying on any LAM messages. The tear-down is stopped at nodes which still have downstream TORA links after the state of the link to its parent changes. The loop freedom in LAM is ensured by the parent-child constraint that a link from child to parent (an upstream tree link) must be a downstream link in the TORA DAG.

3.6 Data Packet Forwarding

Data packets are multicasted using a mechanism that is similar to CBT's which is: if the source is on-tree, it sends data packets to all of its tree neighbors; if the source is not part of the multicasting group, it sends the data packets towards the CORE. The off-tree data source has a choice of sending the data packet in a unicast encapsulated multicast data packet format or a multicast data packet. If the former is used, the data packet will be delivered to the CORE, who extracts the multicast data packet and forwards the data packet to all tree neighbors. If the latter is used, the multicast data packet is sent along a path towards the CORE. Along this path, each node's LAM checks whether it is on the tree for which the data packet is intended. If the answer is yes, LAM accepts the packet and then forwards it via all tree neighbors. Otherwise, LAM only keeps forwarding the data packet along the path towards the CORE. The first mechanism is more flexible since it does not require all nodes along the path to CORE to be LAM-aware. The second mechanism is more efficient because the data packet gets on the tree earlier. Also it reduces traffic congestion around the CORE node.

When an on-tree node receives a data packet from one of its tree neighbors, it accepts the packet then forwards the packet to all tree neighbors except for the neighbor from which the packet is received. During forwarding, if one node sees that it needs to forward the same packet to more than one tree neighbors, it will broadcast to "all neighbors" once instead of sending multiple times to each tree neighbor[3]. When data is forwarded, the node from which the data is received is excluded.

4 Conclusions and Future Work

In this paper, we have presented a lightweight multicasting protocol suitable for MANETs. The LAM protocol is closely coupled with TORA unicast routing to achieve efficiency and simplicity.

LAM is very simple. However, with only one CORE for a group, LAM is not very robust. The CORE is the crucial point of the whole group, and the group

[3]Here it is assumed that all neighbors are on the same wireless communication channel. So one broadcast will cover all neighbors. However in reality neighbors are often on different channels (because of the use of different radio frequencies, different technologies, etc.). Therefore, a more sophisticated algorithm may need to be carried out to decide the minimum set of channels to use to cover all tree neighbors.

is vulnerable to attacks on the CORE. Also, single CORE configuration may create data traffic concentration at the CORE.

IC-LAM: Towards a Two-level Hierarchical Protocol: To solve this problem, another protocol is being developed as an integrated part of the LAM-TORA suite. This protocol, termed Inter-Core LAM (IC-LAM) is a tunnel-based multicast protocol interconnecting multiple COREs for the same multicasting group. By allowing multiple COREs, IC-LAM avoids total group failure due to a single CORE failure. As long as there is one functioning CORE, the whole multicasting group remains functioning. Even when the network is partitioned, members in the same partition as any CORE will be connected together.

By combining the two protocols, LAM and IC-LAM, the result is a two-level hierarchical multicast routing protocol. On the bottom, the simple LAM protocol described here connects regular nodes to COREs These CORE-based-trees form a forest, in which each tree is rooted at a CORE in the CORE set for the multicasting group. On the top level, IC-LAM interconnects all COREs to form a complete forwarding tree covering the whole multicasting group. While the majority of the multicasting members (non-CORE nodes) need only the simplicity of the LAM protocol, a small set of nodes (CORE nodes) can also perform more sophisticated operations to provide a highly-reliable multicasting service. Together, this hierarchical protocol can provide highly-adaptive and survivable multicasting service to large-scale, rapid-changing, MANETs. Due to space limitations, IC-LAM is not described in detail here. Its description and specification will be described in a subsequent paper.*

References

[1] A. Ballardie. "Core Based Trees (CBT) Multicast Routing Architecture". *RFC2201*, September 1997.

[2] S. E. Deering and D. R. Cheriton. "Multicast Routing in Datagram Internetworks and Extended LANs". *ACM Transactions on Computer Systems*, 1990.

[3] D. Estrin et. al. "Protocol Independent Multicast-Sparse Mode (PIM-SM): Protocol Specification". *RFC2117*, June 1997.

[4] S. Deering et. al. "Protocol Independent Multicast Version 2, Dense Mode Specification". *Internet Draft, draft-ietf-idmr-pim-dm-05.txt*, 1997.

[5] T. Pusateri et. al. "Distance Vector Multicast Routing Protocol". *Internet Draft, draft-ietf-idmr-dvmrp-v3-05.txt*, 1997.

[6] B. Levine and J. J. Garcia-Luna-Aceves. "Improving Internet Multicast with Routing Labels". *Proc. IEEE ICNP'97, Atlanta, Georgia*, October 1997.

[7] J. Macker and M. S. Corson. "Mobile Ad hoc Networking and the IETF". *ACM Mobile Computing and Communications Review*, 2(1), January 1998. (first article of an ongoing series).

[8] J. Moy. "Multicast Extensions to OSPF". *RFC1584*, March 1994.

[9] V. Park and M. S. Corson. "A Highly Adaptive Distributed Routing Algorithm for Mobile Wireless Networks". *Proceeding, IEEE INFOCOM '97, Kobe, Japan*, 1997.

[10] V. Park and M. S. Corson. "Temporally-Ordered Routing Algorithm (TORA) Version 1 Functional Specification". *Internet Draft, draft-ietf-manet-tora-spec-00.txt*, 1997.

[11] V. Park and M. S. Corson. "A Performance Comparison of the Temporally-Ordered Routing Algorithm and Ideal Link-State Routing". *Proceeding, Third IEEE Symposium on Computers and Communications*, July 1998.

[12] L. Wei and D. Estrin. "The Trade-offs of Multicast Trees and Algorithms". In *International Conf. on Comp. Comm. and Networks*, 1994.

*The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Army Research Laboratory or the U.S. Government.

Appendix A. LAM-TORA Interface

```
interface ToraToLamItfc {
   enum LinkStatusType {DN, UN, UP, NOP};
   typedef long NodeIdType;
   struct ToraHeightType {
      long tau;   NodeIdType oid;   boolean r;
      int delta;  NodeIdType i;};
   struct LinkType {
      NodeIdType       neighbor;
      ToraHeightType   neighborHeight;
      LinkStatusType   status;};
   typedef sequence <LinkType> LinkTableType;
   readonly attribute NodeIdType toraDestination;
   readonly attribute LinkTableType toraLinkTable;
   readonly attribute ToraHeightType toraHeight; };
interface LamToToraItfc {
   enum LinkStatusType {DN, UN, UP, NOP};
   typedef long NodeIdType;
   struct LinkType {
      NodeIdType       neighbor;
      LinkStatusType    status;};
   typedef sequence <LinkType> LinkListType;
   void LinkStatusChange(in LinkListType changedLinks,
                         in NodeIdType toraDestination); };
```

Appendix B. LAM, Functions

```
NodeIdType QueryToraLink(destination) {
  returns the lowest downstream neighbor (with respect
  to ''destination'') among all neighbors; or NULL if none.}
void OnTreeRejoin() {
  Parent = NULL;
  if ((Child-List != NULL) or (Potential-Child-List != NULL) or
      (self interested in the group)) {
    Potential-Parent = QueryToraLink(My-Core);
    If (Potential-Parent != NULL)
      Send(Potential-Parent, MakeMessage(JOIN, My-Core));}
  else {
    My-Core = Parent = Potential-Parent = Child-List
              Potential-Child-List = NULL;}}
void DealConsequences() {
  if (Parent == NULL) {
    if (Potential-Parent == NULL) {
      if ((member) || (Child-List != NULL) ||
          (Potential-Child-List != NULL))
        OnTreeRejoin();
      else
        My-Core = Child-List = Potential-Child-List = NULL;}
    else {
      if ((!member) && (Child-List == NULL) &&
          (Potential-Child-List == NULL)) {
        Send(Potential-Parent, MakeMessage(LEAVE, TRUE));
        My-Core = Potential-Parent = NULL;}}}
  else {
    if ((!member) && (Child-List == NULL) &&
        (Potential-Child-List == NULL)) {
      Send(Parent, MakeMessage(LEAVE, TRUE));
      My-Core = Parent = NULL;}}}
```

Appendix C. LAM, Protocol

```
/* For simplicity reason, the following specification is for one
   group only.  Also it does not deal with CORE change in the event
   of CORE failure. */
Join Multicast Group M {
  Mcast-Group = M
  My-Core = GetClosestCore(Mcast-Group);
  member = TRUE;
  Potential-Parent = QueryToraLink(My-Core);
  if (Potential-Parent == NULL)
     request TORA to build route towards My-Core;
  else
     Send(Potential-Parent, MakeMessage(JOIN, My-Core));}
JOIN message Msg received from neighbor L {
  if (L is "down" link wrt Msg.Target-Core)
    Send(Msg.Sender, MakeMessage(LEAVE, TRUE);
  else {
    if ((self is the CORE) || (Parent != NULL)) {
      AddToList(Child-List, Msg.Sender);
      Send(Msg.Sender, MakeMessage(ACK));}
    else if (Potential-Parent != NULL)
      AddToList(Potential-Child-List, Msg.Sender);
    else {
      My-Core = Msg.Target-Core;
      AddToList(Potential-Child-List, Msg.Sender);
      Potential-Parent = QueryToraLink(My-Core);
      if (Potential-Parent == NULL)
        request TORA to build route towards My-Core;
      else
        Send(Potential-Parent, MakeMessage(JOIN, My-Core));}}
ACK message Msg received from neighbor L {
  if (L==Potential-Parent) {
     Parent = Potential-Parent;
     for each child in Potential-Child-List {
       AddToList(Child-List, child);
       Send(child, MakeMessage(ACK, NULL));}
     Potential-Child-List = Potential-Parent = NULL;}
  else
    Send(L, MakeMessage(LEAVE, TRUE));}
LEAVE message Msg received from neighbor L {
  if (Msg.flag) {
    RemoveOccurrence(Potential-Child-List, L);
    RemoveOccurrence(Child-List, L);}
  else {
    if (L == Parent) Parent = NULL;
    if (L == Potential-Parent) Potential-Parent = NULL;}
  DealConsequences();}
TORA link state changed notification received {
  for each L in Changed-Link-List {
    RemoveOccurrence(Potential-Child-List, L);
    RemoveOccurrence(Child-List, L);
    if (Potential-Parent == L) Potential-Parent = NULL;
    if (Parent == L) Parent = NULL;}
  DealConsequences();}
```