Internet Draft
draft-ietf-manet-tora-spec-00.txt

V. Park
Naval Research Laboratory
S. Corson
University of Maryland

Submitted: Nov 26, 1997
Expires: May 26, 1997

Temporally-Ordered Routing Algorithm (TORA) Version 1 Functional Specification

Status of this Memo

This document is an Internet-Draft. Internet-Drafts are working documents of the Internet Engineering Task Force (IETF), its areas, and its working groups. Note that other groups may also distribute working documents as Internet-Drafts.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

To learn the current status of any Internet-Draft, please check the "1id-abstracts.txt" listing contained in the Internet-Drafts Shadow Directories on ftp.is.co.za (Africa), nic.nordu.net (Europe), munnari.oz.au (Pacific Rim), ds.internic.net (US East Coast), or ftp.isi.edu (US West Coast).

Abstract

This document provides a detailed specification of Version 1 of the Temporally-Ordered Routing Algorithm (TORA)--a distributed routing protocol for mobile, multihop, wireless networks. Its intended use is for routing of Internet Protocol (IP) datagrams within an autonmous system. The basic, underlying algorithm is neither a distance-vector nor a link-state; it is one of a family of algorithms referred to as ''link reversal'' algorithms. The protocol's reaction is structured as a temporally-ordered sequence of diffusing computations, each computation consisting of a sequence of directed link reversals. The protocol is highly adaptive, efficient and scalable; and is well-suited for use in large, dense, mobile networks. In these networks, the protocol's reaction to link failures typically involves only a localized ''single pass'' of the distributed algorithm. This desirable behavior is achieved through the use of a physical or logical clock to establish the ''temporal order'' of topological change events. The established temporal ordering is subsequently used to structure (or order) the algorithm's reaction to topological changes.

1 Introduction

The Temporally-Ordered Routing Algorithm (TORA) [1] is a highly adaptive routing protocol, which has been tailored for operation in a Mobile Ad hoc Network (MANET). Such a network can be envisioned as a collection of routers (equipped with wireless receiver/transmitters), which are free to move about arbitrarily. The status of the communication links between the routers, at any given time, is a function of their positions, transmission power levels, antenna patterns, cochannel interference levels, etc. The mobility of the routers and the variability of other connectivity factors result in a network with a potentially rapid and unpredictably changing topology. Congested links are also an expected characteristic of such a network as wireless links inherently have significantly lower capacity than hardwired links and are therefore more prone to congestion. TORA's design is predicated on the notion that a routing algorithm that is well-suited for operation in this environment should possess the following properties:

* Executes distributedly
* Provides loop-free routes
* Provides multiple routes (i.e., to reduce the frequency of reactions to topological changes and potentially to alleviate congestion)
* Establishes routes quickly (i.e., so they may be used before the topology changes)
* Minimizes communication overhead by localizing algorithmic reaction to topological changes when possible (i.e., to conserve available bandwidth and increase scalability)

Routing optimality (i.e., determination of the shortest-path) is of less importance. It is also not necessary (nor desirable) to maintain routes between every source/destination pair at all times. The overhead expended to establish a route between a given source/destination pair will be wasted if the source does not require the route prior to its invalidation due to topological changes.

TORA is based, in part, on the work presented in [2] and [3]. TORA is designed to minimize reaction to topological changes. A key concept in its design is that it decouples the generation of potentially far-reaching control message propagation from the rate of topological changes. Control messaging is typically localized to a very small set of nodes near the change without having to resort to a dynamic, hierarchical routing solution with its attendant complexity. TORA includes a secondary mechanism, which allows far-reaching control message propagation as a means of infrequent route optimization and soft-state route verification. This propogation occurs periodically at a very low rate and is independent of the network topology dynamics.

TORA is distributed in that nodes need only maintain information about adjacent nodes (i.e., one hop knowledge). It guarantees all routes are loop-free, and typically provides multiple routes for any source/destination pair that requires a route. TORA is "source initiated" and quickly creates a set of routes to a given destination only when desired. Since multiple routes are typically established and having a single route is sufficient, many topological changes require no reaction at all. Following topological changes that do require reaction, the protocol quickly re-establishes valid routes. This ability to initiate and react infrequently serves to minimize communication overhead. Finally, in the event of a network partition, the protocol detects the partition and erases all invalid routes.

2 MANET Routing Functional Description

A protocol for routing packets in a MANET can be divided into two functional distinct components--a link status sensing mechanism and a routing mechanism. Although these two components are somewhat orthogonal, they can be designed to work together in a synergistic fashion. Originally, these two mechanisms were being designed together as a part of the TORA protocol specification. However, since the basic functionality provided by a link status sensing mechanism is required by a variety of different routing mechanisms, it seemed appropriate that a more generic link status sensing mechanism should be designed and specified as a separate, underlying protocol. This underlying protocol--the Internet MANET Encapsulation Protocol (IMEP) [4]--is being designed to provide several other basic functions that are commonly required by a routing mechanism. Thus, TORA provides only the routing mechanism and depends on IMEP for other underlying functions. Should it later be decided that the two protocols should be combined, IMEP's functionality will be incorporated back into the TORA specification.

2.1 Internet MANET Encapsulation Protocol

IMEP and TORA have been designed to work together synergistically. TORA relies on IMEP for the following underlying functions and services:

* Link status sensing (i.e., monitoring and maintaining the status of connectivity with the set of neighboring routers)
* Control packet delivery (i.e., reliable, in-order control packet delivery to the set of neighbors)
* Network-layer address resolution and mapping
* Security authentication

IMEP provides a rich interface for use by a variety of different mobile wireless routing protocols, which may have varied needs for underlying services.

2.2 Temporally-Ordered Routing Algorithm

This section provides a functional description of TORA. A detailed specification of TORA is provided in a subsequent section.

2.2.1 Notation

A network can be modeled as a graph with a finite set of nodes connected by a set of initially undirected links--wherea node represents a router and a link represents communication connectivity between two routers. Each node i in the network is assumed to have a unique node identifier (ID), and each link (i, k) is assumed to allow two-way communication (i.e., nodes connected by a link can communicate with each other in either direction). Due to the mobility of the nodes, the set of links in the network is changing with time (i.e., new links can be established and existing links can be severed). Each initially undirected link (i, k) may subsequently be assigned one of three states; (1) undirected, (2) directed from node i to node k, or (3) directed from node k to node i. If a link (i, k) is directed from node i to node k, node i is said to be "upstream" from node k, while node k is said to be "downstream" from node i. For each node i, we define the "neighbors" of i, to be the set of nodes k such that there exists a link between nodes i and k. The following assumptions account for the functionality provided by the IMEP. It is assumed that each node i is always aware of its set of neighbors. Additionally, it is assumed that when a node i transmits a packet, it is broadcast to all of its neighbors and that all transmitted packets are received correctly and in order of transmission.

2.2.2 Foundation and Basic Structure

A logically separate version of TORA is run for each destination to which routing is required. The following discussion focuses on a single version running for a given destination, j.

TORA can be separated into three basic functions: creating routes, maintaining routes, and erasing routes. Creating a route from a given node to the destination requires establishment of a sequence of directed links leading from the node to the destination. This function is only initiated when a node with no directed links requires a route to the destination. Thus, creating routes essentially corresponds to assigning directions to links in an undirected network or portion of the network. The method used to accomplish this is an adaptation of the query/reply process described in [2], which builds a directed acyclic graph (DAG) rooted at the destination (i.e., the destination is the only node with no downstream links). Such a DAG will be referred to as a "destination-oriented" DAG. Maintaining routes refers to reacting to topological

changes in the network in a manner such that routes to the destination are re-established within a finite time--meaning that its directed portions return to a destination-oriented DAG within a finite time. Gafni and Bertsekas (GB) described two algorithms [3], which are members of a general class of algorithms designed to accomplish this task. However, the GB algorithms are designed for operation in connected networks. Due to instability exhibited by these algorithms in portions of the network that become partitioned from the destination, they are deemed unacceptable for the current task. TORA incorporates a new algorithm, in the same general class, that is more efficient in reacting to topological changes and capable of detecting a network partition. This leads to the third function--erasing routes. Upon detection of a network partition, all links (in the portion of the network that has become partitioned from the destination) must be marked as undirected to erase invalid routes.

TORA accomplishes these three functions through the use of three distinct control packets: query (QRY), update (UPD), and clear (CLR). QRY packets are used for creating routes; UPD packets are used for both creating and maintaining routes; and CLR packets are used for erasing routes.

2.2.3 General Class of Algorithms

It is beneficial at this point to briefly review the earlier GB algorithms. Consider a connected DAG with at least one node (in addition to the destination) that has no downstream links. Such a DAG will be referred to as "destination-disoriented." The following excerpts (punctuation added for clarity) from [3] loosely describe the two algorithms designed to transform a destination-disoriented DAG into a destination-oriented DAG.

Full Reversal Method: At each iteration, each node (other than the destination) that has no outgoing links reverses the direction of all its incoming links.

Partial Reversal Method: Every node i (other than the destination) keeps a list of its neighboring nodes k that have reversed the direction of the corresponding links (i, k). At each iteration, each node i that has no outgoing links reverses the directions of the links (i, k), for all k which do not appear on its list, and empties the list. If no such k exists (i.e., the list is full), node i reverses the directions of all incoming links and empties the list.

These two algorithms are subsequently re-stated in the context of a generalized numbering scheme that will be summarize here; however, much detail will be left out. For a thorough understanding, one should review the original paper. Essentially, a value is associated

with each node at all times, and the values are such that they can be totally ordered. For example, in the full reversal method, a pair (alpha[i], i) is associated with each node where i is the unique ID of the node and alpha[i] is an integer. The pairs can then be totally ordered lexicographically (e.g. (alpha[i], i) > (alpha[k], k) if alpha[i] > alpha[k] ,or if alpha[i] = alpha[k] and i > k). The value associated with each node i will be referred to as its "height" and denoted h[i]. Now, assume that we assign an initial height to each node in the destination-disoriented DAG such that node i is upstream from node k if and only if h[i] > h[k]. Then it is clear that node i has no downstream links when, measured by its height, it is a local minimum with respect to its neighbors, h[i] < h[k] for all neighbors k. To achieve the desired behavior in the full reversal method, node i must select a new height such that it becomes a local maximum with respect to its neighbors (i.e., h[i] > h[k] for all neighbors k). Node i simply selects a new value alpha[i] = max {alpha[k] such that k is a neighbor of i}+1 and broadcasts the value to all of its neighbors. The partial reversal method can neither be viewed conceptually nor explained as easily. Again, a node selects a new height only when it is a local minimum, but it does not always become a local maximum. To reverse only some of its links (i.e., partial reversal), a node selects a new height that is higher than its own previous height and the height of some of its neighbors, but not higher than the height of all of its neighbors.

The algorithms that belong to this general class are shown to be loop-free, and terminate in a finite number of iterations to a destination-oriented DAG [3]. Furthermore, only nodes that have lost all downstream paths to the destination react to a given failure. The new algorithm incorporated into TORA for the maintaining routes process is a member of this class, and thus inherits many of these properties. The new algorithm is similar to the partial reversal method in that it often reverses only some of its links. However, in order to provide a partition detection capability, the rules for the selection of a new height are significantly more complex. These rules are discussed in detail in section 2.2.4.2.

The basic idea is as follows. When a node loses its last downstream link (i.e., becomes a local minimum) as a result of a link failure, the node selects a new height such that it becomes a global maximum by defining a new "reference level". By design, when a new reference level is defined, it is higher than any previously defined reference levels. This action results in link reversals that may cause other nodes to lose their last downstream link. Any such node executes a partial reversal with respect to its neighbors that have heights already associated with the newest (highest) reference level. In this manner, the new reference level is propagated outward from the point of the original failure (re-directing links in order to re-establish

routes to the destination). This propagation will only extend through nodes that (as a result of the initial link failure) have lost all routes to the destination. Some nodes may experience link reversals from all neighbors (as a result of the same initial link failure). Any such node must select a new height such that it becomes a local maximum. This is accomplished by defining a higher sub-level associated with the new reference level, which will be referred to as the "reflected reference level". This node essentially "reflects" this higher sub-level back toward the node that originally defined the new reference level. Should this reflected reference level be propagated back to the originating node from all of its neighbors, then it is determined that no route to the destination exists. The originating node has then detected a partition and can begin the process of erasing the invalid routes.

2.2.4 Detailed Description

At any given time, an ordered quintuple, HEIGHT = (tau[i], oid[i], r[i], delta[i], i), is associated with each node i, where i is the unique ID of the node. Conceptually, the quintuple associated with each node represents the height of the node as defined by two parameters: a reference level and a offset with respect to the reference level. The reference level is represented by the first three values in the quintuple, while the offset is represented by the last two values. A new reference level is defined each time a node loses its last downstream link due to a link failure. The first value representing the reference level, tau[i], is a time tag set to the "time" of the link failure. For now, it is assumed that all nodes have synchronized clocks. This could be accomplished via interface with an external time source such as the Global Positioning System (GPS) [5] or through use of an algorithm such as the Network Time Protocol [6]. This time tag need not actually indicate or be "time," nor will relaxation of the synchronization requirement invalidate the protocol. The second value, oid[i], is the originator-ID (i.e., the unique ID of the node that defined the new reference level). This ensures that the reference levels can be totally ordered lexicographically, even if multiple nodes define reference levels due to failures that occur simultaneously (i.e., with equal time tags). The third value, r[i], is a single bit used to divide each of the unique reference levels into two unique sub-levels. This bit is used to distinguish between the original reference level and its corresponding, higher, reflected reference level. When a distinction is not required, both the original and reflected reference levels will simply be referred to as "reference levels." The first value representing the offset, delta[i], is an integer used to order nodes with respect to a common reference level. This value is instrumental in the propagation of a reference level. How delta is selected will be clarified in a subsequent section. Finally, the second value

representing the offset, i, is the unique ID of the node itself. This ensures that nodes with a common reference level and equal values of delta (and in fact all nodes) can be totally ordered lexicographically at all times.

Each node i (other than the destination) maintains its height, HEIGHT. Initially the height of each node in the network (other than the destination) is set to NULL, HEIGHT = (-, -, -, -, i), where i is the unique ID of the node. Subsequently, the height of each node i can be modified in accordance with the rules of the protocol. The height of the destination j is always ZERO, HEIGHT = (0, 0, 0, 0, j), where j is the unique ID of the destination for which the algorithm is running). In addition to its own height, each node i maintains a height table with an entry HT_NEIGH[k] for each neighbor k. Initially the height of each neighbor is set to NULL, HT_NEIGH[k] = (-, -, -, -, k). If the destination j is a neighbor of node i, node i sets the corresponding height entry to ZERO, HT_NEIGH[j] = (0, 0, 0, 0, j).

Each node i (other than the destination) also maintains a link-status table with an entry LNK_STAT[k] for each link (i, k), where node k is a neighbor of node i. The status of the links is determined by the height of the node, HEIGHT, and its height entry for the neighbor, HT_NEIGH[k]. The link is directed from the higher node to the lower node. If a neighbor k is higher than node i, the link is marked upstream (UP). If a neighbor k is lower than node i, the link is marked downstream (DN). If the neighbor's height entry, HT_NEIGH[k], is NULL, the link is marked undirected (UN). Finally, if the height of node i is NULL, then any neighbor's height that is not NULL is considered lower, and the corresponding link is marked downstream (DN). When a new link (i, k) is established (i.e., node i has a new neighbor k), node i adds entries for the new neighbor to the height and link-status tables. If the new neighbor is the destination j, the corresponding height entry is set to ZERO, HT_NEIGH[j] = (0, 0, 0, 0, j); otherwise it is set to NULL, HT_NEIGH[k] = (-, -, -, -, k). The corresponding link-status entry, LNK_STAT[k], is set as outlined above. Nodes need not communicate any routing information upon link activation.

2.2.4.1 **Creating Routes**

Creating routes requires use of the QRY and UPD packets. A QRY packet consists of the destination-ID, j, which identifies the destination for which the algorithm is running. An UPD packet consists of the destination-ID, j, and the height of the node i that is broadcasting the packet, HEIGHT.

Each node i (other than the destination) maintains a route-required flag, which is initially un-set. Each node i (other than the

destination) also maintains the time at which the last UPD packet was broadcast and the time at which each link (i, k), where node k is neighbor of node i, became active.

When a node with no directed links and an un-set route-required flag requires a route to the destination, it broadcasts a QRY packet and sets its route-required flag. When a node i receives a QRY it reacts as follows:

a) If the receiving node i has no downstream links and its route-required flag is un-set, it re-broadcasts the QRY packet and sets its route-required flag.

b) If the receiving node i has no downstream links and the route-required flag is set, it discards the QRY packet.

c) If the receiving node i has at least one downstream link and its height is NULL, it sets its height to HEIGHT = (tau[k], oid[k], r[k], delta[k] + 1, i), where HT_NEIGH[k] = (tau[k], oid[k], r[k], delta[k], k) is the minimum height of its non-NULL neighbors, and broadcasts an UPD packet.

d) If the receiving node i has at least one downstream link and its height is non-NULL, it first compares the time the last UPD packet was broadcast to the time the link over which the QRY packet was received became active. If an UPD packet has been broadcast since the link became active, it discards the QRY packet; otherwise, it broadcasts an UPD packet.

If a node has the route-required flag set when a new link is established, it must broadcast a QRY packet.

When a node i receives an UPD packet from a neighbor k, node i first updates the entry HT_NEIGH[k] in its height table with the height contained in the received UPD packet. Node i then updates the entry LNK_STAT[k] in its link-status table and reacts as follows:

a) If the route-required flag is set (which implies that the height of node i is NULL), node i sets its height to HEIGHT = (tau[k], oid[k], r[k], delta[k] + 1, i)--where HT_NEIGH[k] = (tau[k], oid[k], r[k], delta[k], k) is the minimum height of its non-NULL neighbors, updates all the entries in its link-status table, un-sets the route-required flag and then broadcasts an UPD packet that contains its new height.

b) If the route-required flag is not set, node i need only react if it has lost its last downstream link. The section on maintaining routes discusses the reaction that occurs if reception

of the UPD packet resulted in loss of the last downstream link.

2.2.4.2 Maintaining Routes

Maintaining routes is only performed for nodes that have a height other than NULL. Furthermore, any neighbor's height that is NULL is not used for the computations. A node i is said to have no downstream links if HEIGHT < HT_NEIGH[k] for all non-NULL neighbors k. This will result in one of five possible reactions depending on the state of the node and the preceding event. Each node (other than the destination) that has no downstream links modifies its height, HEIGHT = (tau[i], oid[i], r[i], delta[i], i), as follows:

Case 1 (Generate):

Node i has no downstream links (due to a link failure).

(tau[i], oid[i], r[i])=(t, i, 0), where t is the time of the failure.

(delta[i],i)=(0, i)

In essence, node i defines a new reference level. The above assumes node i has at least one upstream neighbor. If node i has no upstream neighbors it simply sets its height to NULL.

Case 2 (Propagate):

Node i has no downstream links (due to a link reversal following reception of an UPD packet) and the ordered sets (tau[k], oid[k], r[k]) are not equal for all neighbors k.

(tau[i], oid[i], r[i])=max{(t[k], oid[k], r[k]) of all neighbors k}

(delta[i],i)=(delta[m]-1, i), where m is the lowest neighbor with the maximum reference level defined above.

In essence, node i propagates the reference level of its highest neighbor and selects a height that is lower than all neighbors with that reference level.

Case 3 (Reflect):

Node i has no downstream links (due to a link reversal following reception of an UPD packet) and the ordered sets (tau[k], oid[k], r[k]) are equal with r[k] = 0 for all neighbors k.

(tau[i], oid[i], r[i])=(tau[k], oid[k], 1)

(delta[i],i)=(0, i)

In essence, the same level (which has not been "reflected") has propagated to node i from all of its neighbors. Node i "reflects" back a higher sub-level by setting the bit r.

Case 4 (Detect):

Node i has no downstream links (due to a link reversal following reception of an UPD packet), the ordered sets (tau[k], oid[k], r[k]) are equal with r[k] = 1 for all neighbors k, and oid[k] = i (i.e., node i defined the level).

(tau[i], oid[i], r[i])=(-, -, -)

(delta[i],i)=(-, i)

In essence, the last reference level defined by node i has been reflected and propagated back as a higher sub-level from all of its neighbors. This corresponds to detection of a partition. Node i must initiate the process of erasing invalid routes as discussed in the next section.

Case 5 (Generate):

Node i has no downstream links (due to a link reversal following reception of an UPD packet), the ordered sets (tau[k], oid[k], r[k]) are equal with r[k] = 1 for all neighbors k, and oid[k] != i (i.e., node i did not define the level).

(tau[i], oid[i], r[i])=(t, i, 0), where t is the time of the failure

(delta[i],i)=(0, i)

In essence, node i experienced a link failure (which did not require reaction) between the time it propagated a reference level and the reflected higher sub-level returned from all neighbors. This is not necessarily an indication of a partition. Node i defines a new reference level.

Following determination of its new height in cases 1, 2, 3, and 5, node i updates all the entries in its link-status table; and broadcasts an UPD packet to all neighbors k. The UPD packet consists of the destination-ID, j, and the new height of the node i that is

broadcasting the packet, HEIGHT. When a node i receives an UPD packet from a neighbor k, node i reacts as described in the creating routes section and in accordance with the cases outlined above. In the event of the failure a link (i, k) that is not its last downstream link, node i simply removes the entries HT_NEIGH[k] and LNK_STAT[k] in its height and link-status tables.

2.2.4.3 Erasing Routes

Following detection of a partition (case 4), node i sets its height and the height entry for each neighbor k to NULL (unless the destination j is a neighbor, in which case the corresponding height entry is set to ZERO), updates all the entries in its link-status table, and broadcast a CLR packet. The CLR packet consists of the destination-ID, j, and the reflected reference level of node i, (tau[i], oid[i], 1). In actuality the value r[i] = 1 need not be included since it is always 1 for a reflected reference level. When a node i receives a CLR packet from a neighbor k it reacts as follows:

a) If the reference level in the CLR packet matches the reference level of node i; it sets its height and the height entry for each neighbor k to NULL (unless the destination j is a neighbor, in which case the corresponding height entry is set to ZERO), updates all the entries in its link-status table and broadcasts a CLR packet.

b) If the reference level in the CLR packet does not match the reference level of node i; it sets the height entry for each neighbor k (with the same reference level as the CLR packet) to NULL and updates the corresponding link-status table entries. Thus, the height of each node in the portion of the network that was partitioned is set to NULL and all invalid routes are erased. If (b) causes node i to lose its last downstream link, it reacts as in case 1 of maintaining routes.

3 Protocol Specification

In the previous description of TORA, some simplifications were made for clarity. In particular, j was used to represent the unique ID of the destination for which the algorithm was running. However, when forwarding IP datagrams in an internetwork, "destinations" to which routing is required are usually identified by an IP address and mask. Together, these two values may correspond to an individual interface on a specific machine, or an aggregation of addresses (e.g., a network address). Thus, in the subsequent discussion a destination "j" refers to a typical IP destination. Another significant simplification pertains to the link between to nodes. In the most general case, a MANET router may have multiple wireless and hardwired

interfaces with differing communication technologies. Therefore, it is necessary to make a distiction between a physical communication "connection" between two routers and a logical communication "link" between two routers. The previous description also ommited any discussion about how the next-hop forwarding decision is made. It is assumed that the IP packet forwarding performed by the kernel in accordance with a standard IP routing table maintained in kernel space. The TORA process must have access to the information in the table and be able to manipulate the table entries. The details regarding how the routing table manipulations made by TORA will be described in detail in the subsequent sections. Since the subsequent description is intended to be in sufficient detail to serve as a template for implementations, some additional terminology is defined first.

3.1 Terminology

The following definitions are identical to the definitions used in the IMEP specification. Many of these definitions may be replaced by or merged with those of the MANET working group's terminology draft [7] now under development.

MANET router or router:
A device--identified by a "unique Router ID" (RID)--that executes a MANET routing protocol and, under the direction of which, forwards IP packets. It may have multiple interfaces, each identified by an IP address. Associated with each interface is a physical layer communication device. These devices may employ wireless or hardwired communications, and a router may simultaneousl employ devices of differing technologies. For example, a MANET router may have four interfaces with differing communications technologies: two hardwired (Ethernet and FDDI) and two wireless (spread spectrum and impulse radio).

adjacency:
The name given to an "interface on a neighboring router".

medium:
A communication channel such as free space, cable or fiber through which connections are established.

communications technology:
The means employed by two devices to transfer information between them.

connection:
A physical-layer connection--which may be through a wired or wireless medium--between a device attached to an interface of one

MANET router and a device utilizating the same communications technology attached to an interface on another MANET router. From the perspective of a given router, a connection is a (interface, adjacency) pair.

link:
A "logical connection" consisting of the logical *union* of one or more connections between two MANET routers. Thus, a link may consist of a heterogeneous combination of connections through differing media using different communications technologies.

neighbor:
From the perspective of a given MANET router, a "neighbor" is any other router to which it is connected by a link.

topology:
A network can be viewed abstractly as a "graph" whose "topology" at any point in time is defined by set of "points" connected by "edges." This term comes from the branch of mathematics bearing the same name that is concerned with those properties of geometric configurations (such as point sets) which are unaltered by elastic deformations (such as stretching) that are homeomorphisms.

physical-layer topology:
A topology consisting of connections (the edges) through the *same* communications medium between devices (the points) communicating using the *same* communications technology.

network-layer topology:
A topology consisting of links (the edges) between MANET routers (the points) which is used as the basis for MANET routing. Since "links" are the logical union of physical-layer "connections," it follows that the "network-layer topology" is the logical union of the various "physical-layer topologies."

IP routing fabric:
The heterogeneous mixture of communications media and technologies through which IP packets are forwarded whose topology is defined by the network-layer topology.

3.2 State Variables

For each destination "j" to which routing is required, a router maintains the following state variables.

```
HEIGHT[j]    The height metric of this router.
RT_REQ[j]    Flag indicating route to the destination is required.
TIME_UPD[j]  Time an UPD packet was last sent by this router.
```

For each destination "j" to which routing is required, a router maintains a separate instance of the following state variables for each neighbor "k".

HT_NEIGH[j][k]	The height metric of neighbor "k."
LNK_STAT[j][k]	The assigned status of the link to neighbor "k."
TIME_ACT[j][k]	Time the link to neighbor "k" became active.

3.3 Auxiliary Variables

For each destination "j" to which routing is required, a router may maintain the following auxiliary variables. Although each of the variables can be computed based on the entries in the LNK_STAT table, maintaining the values continuously may facilitate implementation of the protocol.

num_active[j]	Number of neighbors (i.e., active links).
num_down[j]	Number of links marked DN in the LNK_STAT table.
num_up[j]	Number of links marked UP in the LNK_STAT table.

3.4 Height Data Structure

Each HEIGHT[j] and HT_NEIGH[j][k] entry requires a data structure that comprises five components. The first three components of the Height data structure represent the reference level of the height entry, while the last two components represent an offset with respect to the reference level. The five components of the Height data structure are as follows.

Height.tau	Time the reference level was created.
Height.oid	Unique id of the router that created the reference level.
Height.r	Flag indicating if it is a reflected reference level.
Height.delta	Value used in propagation of a reference level.
Height.id	Unique id of the router.

To simplify notation in this specification, a height may be written as an ordered quintuple--e.g., HEIGHT[j]=(tau,oid,r,delta,id). The following two predefined values for a height are used throughout the specification of the protocol.

NULL=(-,-,-,-,id)	An unknown or undefined height. Conceptually, this can be thought of as an infinite height.
ZERO=(0,0,0,0,id)	The assumed height of a given destination. Note that here "id" is the unique id of the given destination.

3.5 Determination of Link Status

Each entry in the LNK_STAT table is maintained in accordance with the following rule.

```
if        HT_NEIGH[k]==NULL    then   LNK_STAT[k]=UN;
else if   HEIGHT==NULL         then   LNK_STAT[k]=DN;
else if   HT_NEIGH[k]<HEIGHT   then   LNK_STAT[k]=DN;
else if   HT_NEIGH[k]>HEIGHT   then   LNK_STAT[k]=UP;
```

3.6 TORA Packet Formats

TORA packets are encapsulated in IMEP messages, which are sent as "raw" IP datagrams with protocol number ?. The bit level format of the TORA packets has yet to be defined.

3.7 Event Processing

3.7.1 Initialization

TBD.

3.7.2 Connection Status Change

The TORA process receives notification of connection status changes from the the IMEP process. The interface between these two processes has yet to be defined. However, it is anticipated that the TORA process will have access to all the information maintained by the IMEP process about the connections. Thus, upon notification, TORA will have sufficient information to determine if any new links have been established or any existing links have been severed. If either is the case, then TORA must proceed as outlined in appropriate subsequent section (3.7.3 or 3.7.4). In addition, it is also possible for a connection that was used in the routing table to be severed without resulting in the corresponding link being severed. In this case TORA must modify the appropriate routing table entries as outlined in section 3.7.5.

3.7.3 Link with a New Neighbor "k" Established

TBD.

3.7.4 Link with Prior Neighbor "k" Severed

TBD.

3.7.5 Connection Used in Routing Table Severed

TBD.

3.7.6 QRY Packet Regarding Destination "j" Received from Neighbor "k"

If the RTE_REQ flag set then I) else II). nk with Prior Neighbor "k" Severed

I) Event Processing Complete.

II) If HEIGHT[j].r==0 then A) else B).

A) If TIME_ACT[j][k]>TIME_UPD[j] then 1) else 2).

1) Set TIME_UPD to the current time. Create an UPD packet and place it in the queue to be sent to all neighbors. Event Processing Complete.

2) Event Processing Complete.

B) If HT_NEIGH[j][n].r==0 for any n then 1) else 2).

1) Find m such that HT_NEIGH[j][m] is the minimum of all height entries with HT_NEIGH[j][n].r==0. Set HEIGHT[j]=HT_NEIGH[j][m]. Increment HEIGHT.delta. Set HEIGHT[j].id to the unique id of this node. Update LNK_STAT[j][n] for all n. Set TIME_UPD to the current time. Create an UPD packet and place it in the queue to be sent to all neighbors. Event Processing Complete.

2) Set the RT_REQ flag. If num_active>1 then a) else b).

a) Create a QRY packet and place it in the queue to be sent to all neighbors. Event Processing Complete.

b) Event Processing Complete.

3.7.7 UPD Packet Regarding Destination "j" Received from Neighbor "k"

Update the entries HT_NEIGH[j][k] and LNK_STAT[j][k]. If the RT_REQ flag is set and HT_NEIGH[j][k].r==0 then I) else II).

I) Set HEIGHT[j]=HT_NEIGH[j][k]. Increment HEIGHT.delta. Set HEIGHT[j].id to the unique id of this node. Update LNK_STAT[j][n] for all n. Unset the RT_REQ flag. Set TIME_UPD to the current time. Create an UPD packet and place it in the queue to be sent to all neighbors. Event Processing Complete.

II) If num_down[j]==0 then A) else B).

A) If num_up[j]==0 then 1) else 2).

1) If HEIGHT[j]==NULL then a) else b).

a) Event Processing Complete.

b) Set HEIGHT[j]=NULL. Set TIME_UPD to the current time. Create an UPD packet and place it in the queue to be sent to all neighbors. Event Processing Complete.

2) If all HT_NEIGH[j][n], for all n such that HT_NEIGH[j][n] is non-NULL, have the same reference level then a) else b).

a) If HT_NEIGH[j][n].r==0, for any n such that HT_NEIGH[j][n] is non-NULL, then i) else ii).

i) Set HEIGHT[j]=HT_NEIGH[j][n], where n is such that HT_NEIGH[j][n] is non-NULL. Set HEIGHT[j].r=1. Set HEIGHT[j].delta=0. Set HEIGHT[j].id to the unique id of this node. Update LNK_STAT[j][n] for all n. Set TIME_UPD to the current time. Create an UPD packet and place it in the queue to be sent to all neighbors. Event Processing Complete.

ii) If HT_NEIGH[j].oid==id, where id is the unique id of this node, then x) else y).

x) Save the current values of HEIGHT[j].tau and HEIGHT[j].oid in temporary variables. Set HEIGHT[j]=NULL. Set num_down=0. Set num_up=0. For every active link n, if the neighbor connected via link n is the destination j, set HT_NEIGH[j][n]=ZERO and LNK_STAT[j][n]=DN else set HT_NEIGH[j][n]=NULL and LNK_STAT[j][n]=UN. Create a CLR packet, with the previously saved values of tau and oid, and place it in the queue to be sent to all neighbors. Event Processing Complete.

y) Set HEIGHT[j].tau to the current time. Set HEIGHT[j].oid to the unique id of this node. Set HEIGHT[j].r=0. Set HEIGHT[j].delta=0. Set HEIGHT[j].id to the unique id of this node. Update LNK_STAT[j][n] for all n. Unset the RT_REQ flag. Set TIME_UPD to the current time. Create an UPD packet and place it in the queue to be sent to all neighbors. Event Processing Complete.

b) Find n such that HT_NEIGH[j][n] is the maximum of all

non-NULL height entries. Find m such that HT_NEIGH[j][m] is the minimum of the non-NULL height entries with the same reference level as HT_NEIGH[j][n]. Set HEIGHT[j]=HT_NEIGH[j][m]. Decrement HEIGHT.delta. Set HEIGHT[j].id to the unique id of this node. Update LNK_STAT[j][n] for all n. Set TIME_UPD to the current time. Create an UPD packet and place it in the queue to be sent to all neighbors. Event Processing Complete.

B) Event Processing Complete.

3.7.8 CLR Packet Regarding Destination "j" Received from Neighbor "k"

If HEIGHT[j].tau and HEIGHT[j].oid match the values of tau and oid from the CLR packet and HEIGHT[j].r==1 then I) else II).

I) Save the current values of HEIGHT[j].tau and HEIGHT[j].oid in temporary variables. Set Height[j]=NULL. Set num_down=0. Set num_up=0. For every active link n, if the neighbor connected via link n is the destination j, set HT_NEIGH[j][n]=ZERO and LNK_STAT[j][n]=DN else set HT_NEIGH[j][n]=NULL and LNK_STAT[j][n]=UN. If num_active>1 then A) else B).

A) Create a CLR packet, with the previously saved values of tau and oid, and place it in the queue to be sent to all neighbors. Event Processing Complete.

B) Event Processing Complete.

II) Set HT_NEIGH[j][k]=NULL and LNK_STAT[j][k]=UN. For all n such that HT_NEIGH[j][n].tau and HT_NEIGH[j][n].oid match the values of tau and oid from the CLR packet and HT_NEIGH[j][n].r==1, set HT_NEIGH[j][n]=NULL and LNK_STAT[j][n]=UN. If num_down==0 then A) else B).

A) If num_up==0 then 1) else 2).

1) If HEIGHT[j]==NULL then a) else b).

a) Event Processing Complete.

b) Set HEIGHT[j]=NULL. Set TIME_UPD to the current time. Create an UPD packet and place it in the queue to be sent to all neighbors. Event Processing Complete.

2) Set HEIGHT[j].tau to the current time. Set HEIGHT[j].oid to the unique id of this node. Set HEIGHT[j].r=0. Set HEIGHT[j].delta=0. Set HEIGHT[j].id to the unique id of this

node. Update LNK_STAT[j][n] for all n. Unset the RT_REQ flag. Set TIME_UPD to the current time. Create an UPD packet and place it in the queue to be sent to all neighbors. Event Processing Complete.

B) Event Processing Complete.

4 Security Considerations

TBD.

References

[1] V. Park and M. S. Corson, A Highly Adaptive Distributed Routing Algorithm for Mobile Wireless Networks, Proc. IEEE INFOCOM '97, Kobe, Japan (1997).
[2] M.S. Corson and A. Ephremides, A distributed routing algorithm for mobile wireless networks, Wireless Networks 1 (1995).
[3] E. Gafni and D. Bertsekas, Distributed algorithms for generating loop-free routes in networks with frequently changing topology, IEEE Trans. Commun. (January 1981).
[4] M.S. Corson and V. Park, An Internet MANET Encapsulation Protocol (IMEP), draft-ietf-manet-imep-spec-00.txt
[5] NAVSTAR GPS user equipment introduction, MZ10298.001 (February 1991).
[6] D. Mills, Network time protocol, specification, implementation and analysis, Internet RFC-1119 (September 1989).
[7] C. Perkins, Mobile Ad Hoc Networking Terminology, draft-ietf-manet-term-00.txt, (October 1997).

Author's Addresses

Vincent D. Park
Information Technology Division
Naval Research Laboratory
Washington, DC 20375
(202) 767-5098
vpark@itd.nrl.navy.mil

M. Scott Corson
Institute for Systems Research
University of Maryland
College Park, MD 20742
(301) 405-6630
corson@isr.umd.edu