INTERNET-DRAFT
IETF MANET Working Group
1 March 2002

Richard G. Ogier
Fred L. Templin
Bhargav Bellur
Mark G. Lewis
SRI International

Topology Broadcast Based on Reverse-Path Forwarding (TBRPF)

<draft-ietf-manet-tbrpf-05.txt>

Status of This Memo

Abstract

TBRPF is a proactive, link-state routing protocol designed for mobile ad-hoc networks, which provides hop-by-hop routing along shortest paths to each destination. Each node running TBRPF computes a source tree (providing paths to all reachable nodes) based on partial topology information stored in its topology table, using a modification of Dijkstra's algorithm. To minimize overhead, each node reports only *part* of its source tree to neighbors. This is in contrast to other protocols (e.g., STAR) in which each node reports its *entire* source tree to neighbors. TBRPF uses a combination of periodic and differential updates to keep all neighbors informed of the reportable part of its source tree. Each node also has the option to report additional topology information (up to the full topology), to provide improved robustness in highly mobile networks. TBRPF performs neighbor discovery using "differential" HELLO messages which report only *changes* in the status of neighbors. This results in HELLO messages that are much smaller than those of other link-state routing protocols such as OSPF and OLSR.

Contents

1. Introduction

TBRPF is a proactive, link-state routing protocol that provides hop-by-hop routing along shortest paths to each destination. Each node running TBRPF computes a source tree (providing shortest paths to all reachable nodes) based on partial topology information stored in its topology table, using a modification of Dijkstra's algorithm. To minimize overhead, each node reports only *part* of its source tree to neighbors. This is in contrast to other protocols (e.g., FTSP [7] and STAR [4]) in which each node reports its *entire* source tree to neighbors.

TBRPF uses a combination of periodic and differential updates to keep all neighbors informed of the reportable part of its source tree. Each node also has the option to report addition topology information (up to the full topology), to provide improved robustness in highly mobile networks.

TBRPF performs neighbor discovery using "differential" HELLO messages which report only *changes* in the status of neighbors. This results in HELLO messages that are much smaller than those of other link-state routing protocols such as OSPF [17] and OLSR [18].

TBRPF consists of two modules: the neighbor discovery module and the routing module (which performs topology discovery and route computation). An overview of these modules is given in Section 5.

2. Changes

Major changes from version 04 to version 05:

- Introduction of support for multiple interfaces, associated hosts, and network prefixes.

- Introduction of support for link metrics.

- Introduction of a flooding mechanism.

- Clarification of IPv6 operation.

Major changes from version 03 to version 04:

- "Status of This Memo" has been updated.

- A simple packet header format is specified for use when header extensions are not required.

- Each node has the option to report its entire source tree (instead of part of its source tree), to provide increased robustness when

sufficient bandwidth is available.

Major changes from version 02 to version 03:

- TBRPF no longer uses ACKs or NACKs to provide reliable delivery of messages. Instead, it uses a combination of periodic and differential topology updates.

- NEW PARENT messages are no longer used for the selection of parents. Instead, parents select themselves, thus avoiding the overhead of NEW PARENT messages.

- The full topology (FT) mode of TBRPF has been merged with the partial topology (PT) mode. Each node is required to report the minimum topology information (as in PT), but has the option of reporting additional topology information.

- The neighbor discovery module has been modified slightly so that it is completely modular and performs ONLY neighbor sensing.

Major changes from version 01 to version 02:

- The neighbor discovery protocol has been modified so that it is simpler and is independent of the routing protocol.

- Sections 9 and 10 have been shortened considerably.

- The IANA considerations section has been revised to bring it up to date with recent discussions on the MANET email list.

Major changes from version 00 to version 01:

- A partial-topology mode (TBRPF-PT) has been introduced to achieve a further reduction in control traffic in large, dense networks.

- The neighbor discovery protocol has been improved significantly.

- The unicast mode for transmitting TBRPF messages has been eliminated: each TBRPF packet is now broadcast to all neighbors. If a message is to be processed by only a few neighbor nodes, their identities are included in the message.

- All topology updates are now sent reliably to all neighbors using NACKs.

- Two new configurable parameters, MIN_UPDATE_INTERVAL and

MIN_FORW_UPDATE_INTERVAL, have been introduced to minimize frequent generation and forwarding of topology updates.

- TBRPF-FT includes a new message type NEW PARENT REPLY, which is sent in response to one or more NEW PARENT messages. A single message is sent in response to multiple NEW PARENT messages that are received at about the same time.

- A new packet formatting scheme is used that provides optimal packing of TBRPF protocol elements with minimal insertion of header and null padding octets. A packet compression mechanism is included that eliminates all null padding octets when enabled.

- Formats have been added for new TBRPF-FT and TBRPF-PT message types. Formats have been revised for some existing TBRPF-FT message types.

- An implicit NARP mechanism has been provided for binding one or more IP addresses to a Router ID.

- Optional checksum facilities have been provided for data integrity.

3. TBRPF Terminology

The keywords "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC2119 [2]. The following terms are also used to describe TBRPF:

node
: A router that implements TBRPF.

interface
: A network device that connects a node to the MANET. A node can have multiple interfaces. An interface can be wireless or wired, and can be broadcast (e.g., Ethernet) or point-to-point. Each interface is identified by an IP address (unless the interface is to an unnumbered point-to-point link).

Router ID
: Each node is identified by a unique 32-bit Router ID (RID), also called a node ID, which for IPv4 is equal to the IP address of one of its interfaces.

link
: A logical connection from one node to another, identified by a pair (u,v), where u and v represent nodes. Nodes u and v are called the "tail" and "head" of the link, respectively.

bidirectional link
A link between two nodes u and v is said to be bidirectional, or 2-way, if node u has an interface I and node v has an interface J such that interface I can hear interface J and vice versa.

neighbor node
A node j is said to be a neighbor of node i if node i can hear node j on some interface. Node j is said to be a 2-way neighbor if there is a bidirectional link between i and j.

ad-hoc interface
Any interface such that two neighbor nodes on the interface need not be neighbors of each other. MANET nodes typically have at least one ad-hoc interface, but this is not a requirement.

topology
The topology of the network is described by a graph G = (V, E), where V is the set of nodes u and E is the set of links (u,v) in the network.

directed tree
A subset of (directed) links (u,v) that does not contain any loops. The root of a directed tree is the only node u such that the tree contains no link whose tail is u.

source tree
The directed tree computed by each node that provides shortest paths to all other nodes. Not the same as a broadcast tree.

topology update
A message or part of a message that reports a state change for one or more links.

parent
The parent of a node i for an update source u is the next node on the computed shortest path to node u.

4. Applicability Section

This section describes the networking context and protocol characteristics of the TBRPF protocol as specified in the MANET Routing Protocol Applicability Statement [16].

4.1. Networking Context

TBRPF is appropriate for any MANET network having up to a few hundred nodes, especially when nodes are highly mobile and bandwidth is lim-

ited.

4.2. Protocol Characteristics and Mechanisms

* Does the protocol provide shortest path routes?

  Yes.

* Does the protocol provide support for unidirectional links? (if so, how?)

  No. It uses only bidirectional links (as in 802.11).

* Does the protocol require the use of tunneling? (if so, how?)

  No.

* Does the protocol require using some form of source routing? (if so, how?)

  No. The protocol uses hop-by-hop routing. However, since the entire path is known, source routing can be used as an option.

* Does the protocol require the use of periodic messaging? (if so, how?)

  Each node transmits HELLO and topology update messages periodically.

* Does the protocol require the use of reliable or sequenced packet delivery? (if so, how?)

  No.

* Does the protocol provide support for routing through a multi-technology routing fabric? (if so, how?)

  Yes. Each network interface is assigned a unique IP address.

* Does the protocol provide support for multiple hosts per router? (if so, how?)

  Yes. The concept of a Router ID (RID) [3,14,17] can be used to associate multiple hosts with a single RID.

* Does the protocol support the IP addressing architecture? (if so, how?)

  Yes.

* Does the protocol require link or neighbor status sensing (if so,

how?)

Yes. A new, efficient neighbor discovery protocol is presented, in which each HELLO message contains only the IDs of nodes that have recently been heard but with which a 2-way link has not yet been established.

* Does the protocol depend on a central entity? (if so, how?)

  No.

* Does the protocol function reactively? (if so, how?)

  The protocol reacts to topology changes, but not to traffic demand.

* Does the protocol function proactively? (if so, how?)

  Yes. It maintains optimal paths to all reachable destinations at all times.

* Does the protocol provide loop-free routing? (if so, how?)

  Yes. Since routing is based on link states, it provides loop-free routing when the topology is stable.

* Does the protocol provide for sleep period operation? (if so, how?)

  TBRPF operates correctly even if nodes can go into and out of sleep mode at arbitrary times. Other features can be added, such as assigning awake routers to store packets for sleeping nodes.

* Does the protocol provide some form of security? (if so, how?)

  TBRPF can be extended in a straightforward manner to incorporate authentication of TBRPF protocol packets in a manner similar to OSPF version 2 [RFC 2328]. (See Section 12.)

* Does the protocol provide support for utilizing multi-channel, link-layer technologies? (if so, how?)

  Yes. TBRPF supports multiple interfaces.

5. TBRPF Overview

TBRPF consists of two main modules: the neighbor discovery module, and the routing module (which performs topology discovery and route computation).

5.1. Overview of Neighbor Discovery

The TBRPF Neighbor Discovery (TND) protocol allows each node to quickly detect the neighbor nodes with which the node has a direct, bidirectional link, i.e., such that the node and the neighbor node each have an interface that can hear the other interface. The protocol also detects when a bidirectional link to some neighbor no longer exists.

The key feature of TND is that it uses "differential" HELLO messages which report only *changes* in the status of neighbors. This results in HELLO messages that are much smaller than those of other link-state routing protocols such as OSPF and OLSR, in which each HELLO message includes the IDs of *all* neighbors. As a result, HELLO messages can be sent more frequently in highly mobile networks without increasing overhead significantly.

TND is designed to be fully modular and independent of the routing module. TND performs ONLY neighbor sensing (unlike OLSR's neighbor sensing module, which also discovers 2-hop neighbors and selects multipoint relays). As a result, TND can be used by other routing protocols, and TBRPF can use another neighbor discovery protocols in place of TND, e.g., one provided by the link layer.

Nodes with multiple interfaces run TND separately on each interface, similar to OSPF. Thus, a neighbor table is maintained for each interface, and a HELLO sent on a particular interface contains only the RIDs of neighbors for that interface. (This differs from OLSR, in which each HELLO is sent on all interfaces.)

Each node sends a HELLO message every HELLO_INTERVAL seconds, with a small jitter. Each HELLO message contains three (possibly empty) lists of router IDs, formatted as the following three message subtypes: NEIGHBOR REQUEST, NEIGHBOR REPLY, and NEIGHBOR LOST. A NEIGHBOR REQUEST message is always included in a HELLO, even if its list of router IDs is empty. However, a NEIGHBOR REPLY or NEIGHBOR LOST message is included only if its list of router IDs is nonempty. Each HELLO message also contains the current HELLO sequence number (HSEQ), which is incremented with each transmitted HELLO. For convenience, we say that "node i sends a NEIGHBOR REQUEST message for node j" if node i sends such a message that includes the ID of node j, and similarly for NEIGHBOR REPLY and NEIGHBOR LOST messages.

Each node maintains a neighbor table for each interface, which stores the state information for neighbors heard on that interface. The state (or level) of each node can be 1-WAY, 2-WAY, or LOST. When node i changes the state of a neighbor j, it sends the appropriate message (NEIGHBOR REQUEST/UP/LOST) in at most NBR_HOLD_COUNT (typi-

cally 3) consecutive HELLOs. This ensures that node j will either receive the message, or will miss NBR_HOLD_COUNT HELLOs and thus declare node i to be LOST. This technique also makes it unnecessary

for a node to include each 1-WAY neighbor in HELLOs indefinitely, unlike OSPF and OLSR.

The NEIGHBOR REQUEST message includes a list of neighbors from which HELLO messages have recently been heard but for which a 2-WAY link is not currently established. To avoid establishing a link that is likely to be short lived (i.e., to employ hysteresis), a node i must receive at least HELLO_ACQUIRE_COUNT (e.g., 2) of the last HELLO_ACQUIRE_WINDOW (e.g., 3) HELLOs from another node j before declaring the node to be 1-WAY. In this case, node i sends a NEIGHBOR REQUEST message for node j in each of its next NBR_HOLD_COUNT HELLO messages, or until a NEIGHBOR REPLY message for node i is received from node j.

If node j receives a NEIGHBOR REQUEST from node i, then node j declares the link to node i to be 2-WAY (if it is not already 2-WAY), and includes a NEIGHBOR REPLY message for node i in its next NBR_HOLD_COUNT transmitted HELLO messages. Upon receiving the NEIGHBOR REPLY message, node i declares the link to node j to be 2-WAY.

If node i receives a HELLO from a 1-WAY or 2-WAY neighbor j whose HSEQ indicates that at least NBR_HOLD_COUNT HELLOs were missed, or if node i receives no HELLO from node j within NBR_HOLD_TIME seconds, then node i changes the state of node j to LOST, and sends a NEIGHBOR LOST message for node i in its next NBR_HOLD_COUNT transmitted HELLO messages (unless the link changes state before these transmissions are complete). Node j will either receive the message or will miss NBR_HOLD_COUNT HELLOs; in either case it will declare node i to be LOST. In this manner, both nodes will agree that the link is no longer bidirectional, even if node j can still hear HELLOs from node i.

Each node MAY maintain and update one or more link metrics to each neighbor j for each interface I, representing the quality of the link. As described in Section 7.7, such link metrics can be used as additional conditions for changing the state of a neighbor, based on the link metric going above or below some threshold. TBRPF also allows link metrics to be advertised in topology updates and used for computing shortest paths.

5.2. Overview of the Routing Module

Each node running TBRPF computes a source tree (providing shortest paths to all reachable nodes) based on partial topology information stored in its topology table, using a modification of Dijkstra's algorithm. To minimize overhead, each node reports only part of its source tree to neighbors. This is in contrast to FTSP [7] and STAR

[4], in which each node reports its *entire* source tree to neighbors, which is redundant since the source trees of different neighbors often overlap considerably. The main idea behind the current

version of TBRPF came from PTSP [7], another protocol in which each node reports only part of its source tree. However, PTSP differs in many ways from TBRPF: e.g., PTSP uses NEW PARENT messages and reliable updates, unlike the current version of TBRPF (but like previous versions of TBRPF [1]).

To decide which part of its source tree T to report to neighbors, a node running TBRPF first computes its "reportable node set" RN. Node i includes node u in RN if it determines that some neighbor j (of node i) may select node i to be the next hop (or parent) on the shortest path to destination u. (See the pseudocode procedure Update_RN() in Section 8.4 for details.) This technique avoids the need for the NEW PARENT messages that were used in PTSP and previous versions of TBRPF.

The part of T that a node reports to neighbors is called the "reportable subtree" and is denoted RT. RT consists of links (u,v) of T such that u is in RN. To provide increased robustness when sufficient bandwidth is available, each node has the option to report its entire source tree T, by setting the configurable parameter REPORT_FULL_TREE to 1. This results in all reachable nodes u being included in RN, so that RT = T.

Each node reports RT to neighbors in *periodic* topology updates (e.g., every 5 seconds), and reports changes (additions and deletions) to RT in more frequent *differential* updates (e.g., every 1 second). Periodic updates inform new neighbors of RT, and ensure that each neighbor eventually learns RT even if it does not receive all updates. Differential updates ensure the fast propagation of each topology update to all nodes that are affected by the update. A received topology update is not forwarded, but *may* result in a change to RT, which will be reported in the next differential update. Whenever possible, topology updates are included in the same packet as a HELLO message, to minimize the number of control packets sent. TBRPF does not require reliable or sequenced delivery of messages, and does not use ACKs or NACKs.

TBRPF does not use sequence numbers for topology updates, thus reducing message overhead and avoiding wraparound problems. Instead, a technique similar to SPTA [20] is used in which, for each link (u,v) reported by one or more neighbors, only the parent p(u) for u is believed regarding the the state of the link. (However, in SPTA each node reports the full topology.) Using this technique, each node maintains a topology graph TG, consisting of "believable" links that are reported by neighbors, and computes T as the shortest-path tree within TG. To allow immediate rerouting, the restriction that each link (u,v) in TG must be reported by p(u) is relaxed temporarily if p(u) changes to a neighbor that is not reporting the link.

Each node is required to report RT, but MAY report additional links, e.g., to provide increased robustness in highly mobile networks.

More precisely, a node may maintain any subgraph H of TG that contains T, and report the reportable subgraph RH, which consists of links (u,v) of H such that u is in RN. For example, H can equal TG, which would provide each node with the full network topology if this is done by all nodes. H can also be a biconnected subgraph that contains T, which would provide each node with two disjoint paths to each other node, if this is done by all nodes.

Nodes MAY report a metric for each link in topology updates, and to compute paths that are shortest with respect to the metric. This allows packets to be sent along paths that are higher quality than minimum-hop paths.

TBRPF allows path optimality to be traded off in order to reduce the amount of control traffic in networks with a large diameter, where the degree of approximation is determined by the configurable parameter NON_TREE_PENALTY.

In a MANET, some nodes, called non-relay nodes, may choose not to forward data packets for other nodes. Non-relay nodes are supported by TBRPF in a very simple manner: they run TBRPF (and transmit HELLOs) but do not do not transmit topology update messages. As a result, no node will compute a path that uses a non-relay node as an intermediate node.

6. TBRPF Packets

Nodes send TBRPF protocol data in contiguous units known as *packets*. Each packet includes a header, optional header extensions, and a body comprising one or more *message(s)* and padding options as needed. The total length of all packet elements MUST be less than the maximum length represented by an unsigned 16-bit integer, i.e. 64KB. To facilitate efficient receiver processing, senders SHOULD insert padding options as necessary to align multi-octet words within the TBRPF packet on "natural" boundaries (i.e. modulo-8/4/2 addresses for 64/32/16-bit words, respectively). Receivers MUST be capable of processing multi-octet words whether/not aligned on natural boundaries, and SHOULD do so as efficiently as possible. The following sections specify elements of the TBRPF packet in more detail.

6.1. TBRPF Packet Header

TBRPF packet headers are variable-length (minimum one octet), and MUST begin on a modulo-8 boundary to provide a base for multi-octet word alignment. The format for the first octet of the header is given below:

```
 0 1 2 3 4 5 6 7
+-+-+-+-+-+-+-+-+
|E|Vers | BITS  |
+-+-+-+-+-+-+-+-+
```

Bits (4 bits):
Interpreted based on the sense of the 'E' flag.

Version (3 bits):
A 3-bit unsigned integer value that identifies the TBRPF protocol version; the following values are defined:

TBRPFVERSION_2	2
TBRPFVERSION_1	1

Implementations of this protocol description MUST encode the value TBRPFVERSION_3 in the version field. (Implementations MAY provide backwards-compatibility for older protocol versions.)

Mode (1 bit):
Specifies simple mode (E = 0) or extended mode (E = 1) for the current TBRPF packet header as described in the following subsections:

6.1.1. Simple-mode Packet Headers (E = 0)

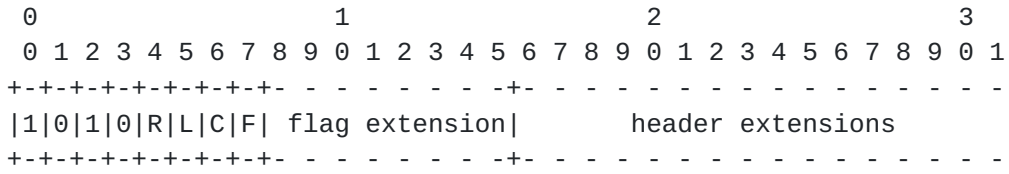
Senders MAY encode TBRPF packet headers in simple-mode when no header extensions are required, such as when the lower-level delivery service provides requisite fields (e.g. length; checksum). Simple-mode packet headers are 1-octet in length and *coincide* with the TYPE field of the first message in the TBRPF packet body (see below). Simple-mode packet headers MAY ONLY be used when 0 <= TYPE <= 4 for the first message of the packet body. (This is usually the case, since the first message is usually a HELLO message.) Simple-mode packet headers are specified as follows:

```
 0 1 2 3 4 5 6 7
+-+-+-+-+-+-+-+-+
|0|  3  | TYPE  |
+-+-+-+-+-+-+-+-+
```

TYPE (4 bits):
A 4-bit unsigned integer with value 0 - 15 that encodes the low-order 4 bits of the TYPE field in the first element of the TBRPF packet body.

6.1.2. Extended-mode Packet Headers (E = 1)

Senders MUST encode TBRPF packet headers in extended-mode when header extensions are required, such as when the lower-level delivery service omits requisite fields. Extended-mode packet headers are specified as follows:



Flags (4 bits):
One bit (F) specifies whether a 1-octet flag extension field follows. Three bits (C, L, R) specify which header extension fields (if any) follow. Any extension fields specified by these bits MUST occur in canonical order (i.e. first F, then C, then L, then R) as follows:

F - Flag extension field included:
When F = '1', a 1-octet flag extension field immediately follows the first octet of the TBRPF packet header. Currently, all bits in the flag extension field are RESERVED for future use; senders MAY set F = '1' to insert a single octet of padding for alignment purposes.

C - Checksum field included:
If the underlying delivery service provides a checksum facility the sender MAY set C = '0' and omit the checksum extension field. Otherwise, the sender SHOULD set C = '1' and include a 16-bit checksum field beginning in the first octet of the header extension. The checksum is calculated as described in [11] and written into the checksum field. The checksum is calculated across *all* data bytes in the packet header and body, but DOES NOT include a pseudo-header of the underlying delivery service. If other header extension fields are also included (see below), the sender MUST fill them in *before* calculating the checksum.

Receivers MUST examine the C bit to determine whether the checksum field is present. If C = '1', a receiver MUST verify the checksum encoded in the 16-bit message checksum field as described in [11] *regardless* of whether the underlying delivery service also performed a checksum. Receivers MUST discard TBRPF packets that contain an incorrect checksum.

L - Length field included:
If the underlying delivery service provides a length field, the

sender MAY set L = '0' and omit the length extension field. Otherwise, the sender MUST set L = '1' and include a 16-bit unsigned integer length field immediately after any previous header fields.

The length includes all header and data bytes and is written into the length field in network byte order.

Receivers MUST examine the L bit to determine whether the length field is present. If L = '1', a receiver MUST convert the length field to host byte order to determine the length of the TBRPF packet, including the TBRPF packet header. Receivers MUST discard any TBRPF packet if neither the underlying delivery service nor the TBRPF packet header provide packet length.

R - Router ID (RID) included:
If the underlying delivery service encodes the sender's RID, the sender MAY set R = '0' and omit the RID field. Otherwise, the sender MUST set R = '1' and include a 32-bit unsigned integer RID immediately after any previous header fields. The RID option provides a mechanism for implicit Network-level Address Resolution (NARP) as described in [14]. A receiver that detects an RID option MUST create a NARP binding between the RID and the source address that appears in the network-level header.

6.2. TBRPF Packet Body

The TBRPF packet body consists of the concatenation of one or more TBRPF message(s) (and padding options where necessary) encoded using a method similar to the type-length-value (TLV) encoding method described in [15]. Messages and padding options within the TBRPF packet body are encoded using the following format:

```
+-+-+-+-+-+-+-+-+- - - - -
|OPTIONS| TYPE  | VALUE
+-+-+-+-+-+-+-+-+- - - - -
```

TYPE (4 bits):
A 4-bit identifier with value 0 - 15 that identifies the element.

VALUE
Variable-length field. (Format and length depend on TYPE, as described in the following sections.)

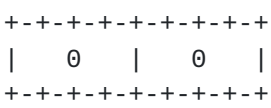
OPTIONS
Four option bits that depend on TYPE.

As specified in [15], the sequence of elements MUST be processed strictly in the order they appear within the TBRPF packet body; a receiver must not, for example, scan through the packet body looking for a particular type of element prior to processing all preceding elements. TBRPF packet elements include *padding options* and *messages* as described below.

6.2.1. Padding Options (TYPE = 0 thru 1)

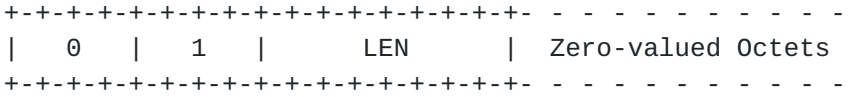
As specified in [15], senders may insert two types of padding options where necessary to satisfy alignment requirements for other elements. Padding options may occur anywhere within the TBRPF packet body. The following two padding options are defined:

Pad1 option (TYPE = 0)



The Pad1 option inserts one octet of padding into the TBRPF packet body; the VALUE field is omitted. If more than one octet of padding is required, the PadN option (described next) should be used, rather than multiple Pad1 options.

PadN option (TYPE = 1)



The PadN option inserts two or more octets of padding into the TBRPF packet body. The first octet of the VALUE field contains an 8-bit unsigned integer length containing a value between 0 - 253 which specifies the number of zero-valued octets that immediately follow, yielding a maximum total of 255 padding octets.

6.2.2. Messages (TYPE = 2 thru 15)

Message are described as they occur in the TBRPF protocol specification in the following sections, including the message name, type code and a detailed message format diagram. Senders MUST encode messages as specified by the individual message formats. Receivers MUST detect errors in message construction, such as messages with a non-integral number of elements or with fewer elements than indicated. In all cases, upon detecting an error receivers MUST discontinue processing the current TBRPF packet and discard any unprocessed elements.

7. TBRPF Neighbor Discovery

This section describes the TBRPF Neighbor Discovery (TND) protocol, which allows each node to quickly detect the neighboring nodes with which the node has a direct, bidirectional (2-WAY) link, and to

quickly detect the loss of such a link. TND is run separately on each interface.

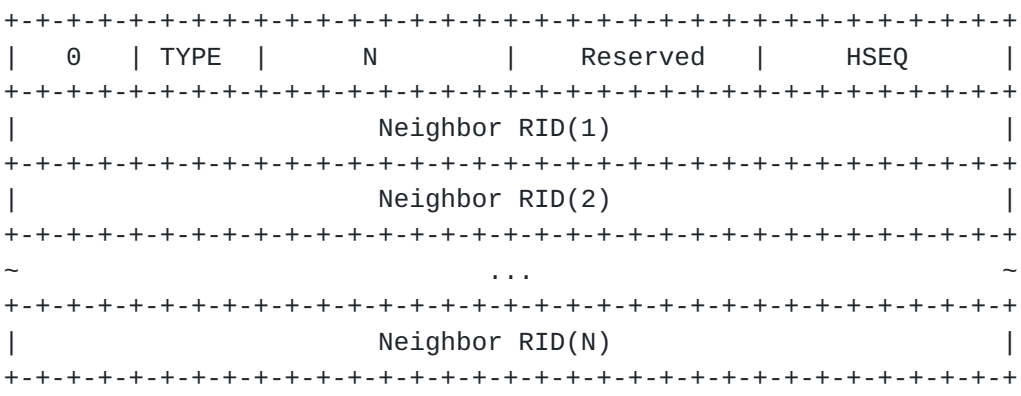
TND is designed to be independent of the TBRPF routing module. The interface between these two modules is defined simply by the three functions Link_Up(j, I, metric), Link_Down(j, I), and Link_Change(j, I, metric), which are called by TND to report a new neighbor, the loss of a neighbor, or a change in the link metric on a given interface.

7.1. HELLO Message Format

The HELLO message has the following three subtypes:

- NEIGHBOR REQUEST (TYPE = 2)
- NEIGHBOR REPLY (TYPE = 3)
- NEIGHBOR LOST (TYPE = 4)

Each HELLO subtype has the following format:



- The message body contains N 4-octet router IDs.
- HSEQ is the 8-bit HELLO sequence number.

A HELLO message is the concatenation of a NEIGHBOR REQUEST message, a NEIGHBOR REPLY message, and a NEIGHBOR LOST message, where each of the last two messages is omitted if its list of router IDs is empty. Thus, a HELLO message always includes a (possibly empty) NEIGHBOR REQUEST.

7.2. Neighbor Table

Each node maintains a neighbor table for each interface, which stores state information for each neighbor that has recently been heard on that interface. The entry for neighbor j contains the following variables:

nbr_rid(j) - The Router ID of node j.

nbr_if_addr(j) - The interface IP address of node j.

nbr_level(j) - The current level (or status) of the link to node j, which can be LOST, 1-WAY, or 2-WAY.

nbr_life(j) - The amount of time (in seconds) remaining before nbr_level(j) must be changed to LOST if no further HELLO message from node j is received. Set to NBR_HOLD_TIME whenever a HELLO is received from node j.

nbr_hseq(j) - The last value of HSEQ received from node j. Used to determine the number of HELLOs have been missed.

nbr_count(j) - The number of times a NEIGHBOR REQUEST/UP/LOST message has been sent for node j since nbr_level(j) has changed.

hello_history(j) - A list of the sequence numbers of the last HELLO_ACQUIRE_WINDOW HELLO messages received from node j.

nbr_metric(j) - An optional measure of the quality of the link to node j.

The table entry for a neighbor j may be deleted if no HELLO has been received from node j within the last 2*NBR_HOLD_TIME seconds. (It is kept while the NEIGHBOR LOST for node j is being transmitted.) The absence of an entry for a given node j is equivalent to an entry with nbr_level(j) = LOST and hello_history(j) = NULL.

The three possible values of nbr_level(j) at node i have the following meaning:

LOST
A neighbor is declared to be LOST if no HELLO message has been received from node j within the last NBR_HOLD_TIME seconds, and remains LOST until HELLO_ACQUIRE_COUNT of the last HELLO_ACQUIRE_WINDOW HELLOs from node j are received.

1-WAY
HELLO_ACQUIRE_COUNT of the last HELLO_ACQUIRE_WINDOW HELLOs from node j have been received, but the link is not 2-WAY.

2-WAY
Node i has received from node j either a NEIGHBOR REQUEST or a NEIGHBOR REPLY from node j, and no event has since occurred to change this status. A link that is 2-WAY is considered to be "up".

7.3. Sending HELLO Messages

Each node sends a HELLO message periodically every HELLO_INTERVAL

seconds, with a random jitter selected from the interval [0, MAX_JITTER]. Each HELLO message always includes a NEIGHBOR REQUEST

message, even if its router ID list is empty. The NEIGHBOR REQUEST message includes the sequence number HSEQ, which is incremented each time a HELLO is sent. The HELLO message also includes a NEIGHBOR REPLY message if its router ID list is nonempty, and a NEIGHBOR LOST message if its router ID list is nonempty. The contents of these three messages are determined by the following steps at node i:

1. For each node j such that nbr_level(j) = LOST and nbr_count(j) > 0, include node j in the NEIGHBOR LOST message and decrement nbr_count(j).

2. For each node j such that nbr_level(j) = 1-WAY and nbr_count(j) > 0, include node j in the NEIGHBOR REQUEST message and decrement nbr_count(j).

3. For each node j such that nbr_level(j) = 2-WAY and nbr_count(j) > 0, include node j in the NEIGHBOR REPLY message and decrement nbr_count(j).

7.4. Processing a Received HELLO Message

Upon receiving a HELLO message with sequence number HSEQ from node j on interface I, node i performs the following steps:

1. If the neighbor table for interface I does not contain an entry for node j, create one with nbr_rid(j) equal to the Router ID of node j, nbr_if_addr(j) equal to the interface address from which the message was sent, nbr_level(j) = LOST (temporarily), and nbr_count(j) = 0. Update hello_history(j) to reflect the received HELLO message.

2. If nbr_level(j) = LOST and hello_history(j) indicates that HELLO_ACQUIRE_COUNT of the last HELLO_ACQUIRE_WINDOW HELLO messages from node j have been received:

   a. If node i does not appear in the NEIGHBOR REQUEST list, set nbr_level(j) = 1-WAY and nbr_count(j) = NBR_HOLD_COUNT (a NEIGHBOR REQUEST will be sent).

   b. If node i appears in the NEIGHBOR REQUEST list, set nbr_level(j) = 2-WAY and nbr_count(j) = NBR_HOLD_COUNT (a NEIGHBOR REPLY will be sent). Call Link_Up(j, I, metric).

3. Else if nbr_level(j) = 1-WAY:

   a. If node i appears in the NEIGHBOR REQUEST list, then set nbr_level(j) = 2-WAY, nbr_count(j) = NBR_HOLD_COUNT (a NEIGHBOR

REPLY will be sent). Call Link_Up(j, I, metric).

b. Else if node i appears in the NEIGHBOR REPLY list, then set nbr_level(j) = 2-WAY and nbr_count(j) = 0. Call Link_Up(j, I, metric).

c. Else if HSEQ - nbr_hseq(j) > NBR_HOLD_COUNT, then set nbr_level(j) = LOST and nbr_count(j) = NBR_HOLD_COUNT (a NEIGHBOR LOST will be sent).

4. Else if nbr_level(j) = 2-WAY:

a. If node i appears in the NEIGHBOR LOST list, set nbr_level(j) = LOST and nbr_count(j) = 0. Call Link_Down(j).

b. Else if HSEQ - nbr_hseq(j) > NBR_HOLD_COUNT, set nbr_level(j) = LOST and nbr_count(j) = NBR_HOLD_COUNT (a NEIGHBOR LOST will be sent).

c. Else if node i appears in the NEIGHBOR REQUEST list and nbr_count(j) = 0, then set nbr_count(j) = NBR_HOLD_COUNT (a NEIGHBOR REPLY will be sent).

5. Set nbr_life(j) = NBR_HOLD_TIME and nbr_hseq(j) = HSEQ.

7.5. Expiration of Timer nbr_life

Upon expiration of the timer nbr_life(j), node i performs the following step:

If nbr_level(j) = 1-WAY or 2-WAY, set nbr_level(j) = LOST and nbr_count(j) = NBR_HOLD_COUNT (a NEIGHBOR LOST will be sent). Call Link_Down(j).

7.6. Link-Layer Failure Indication

Some link-layer protocols (e.g., IEEE 802.11) provide an indication that the link to a particular neighbor has failed, e.g., after attempting a maximum number of retransmissions. If such an indication is provided by the link layer, then node i SHOULD perform the following step upon receipt of a link-layer failure indication for the link to node j:

If nbr_level(j) = 2-WAY, set nbr_level(j) = LOST and nbr_count(j) = NBR_HOLD_COUNT (a NEIGHBOR LOST will be sent). Call Link_Down(j).

7.7. Optional Link Metrics

Each node MAY maintain and update one or more link metrics to each neighbor j for each interface I, representing the quality of the link, e.g., signal strength, number of HELLOs received over some time interval, reliability, stability, bandwidth, etc. Each node MUST declare a neighbor to be LOST if either NBR_HOLD_COUNT HELLOs are missed or if no HELLO is received within NBR_HOLD_TIME seconds; however, a node MAY also declare a neighbor to be LOST based on a link metric being above or below some threshold. Each node MUST receive at least HELLO_ACQUIRE_COUNT of the last HELLO_ACQUIRE_WINDOW HELLOs from a neighbor before declaring the neighbor 1-WAY or 2-WAY; however, a node MAY require an additional condition based on a link metric being above or below some threshold, before declaring the neighbor 1-WAY or 2-WAY. This document does not specify any particular link metric, but an implementation of TBRPF that uses such metrics is considered to be compliant with this draft.

If USE_METRICS = 1, one of the link metrics computed by TND, denoted nbr_metric(j,I) for neighbor j and interface I, is reported to the routing module via the functions Link_Up(j,I,metric) and Link_Change(j,I,metric). The latter function is called whenever nbr_metric(j,I) changes significantly. As described in Section 8, if USE_METRICS = 1, this link metric is advertised in topology updates and used for computing shortest paths.

7.8. Configurable Parameters

This section lists the parameters used in the description of the neighbor discovery protocol, and their proposed default values.

HELLO_INTERVAL (1 second)

MAX_JITTER (0.1 second)

NBR_HOLD_TIME (3 seconds)

NBR_HOLD_COUNT (3)

HELLO_ACQUIRE_COUNT (2)

HELLO_ACQUIRE_WINDOW (3)

8. TBRPF Routing Module

This section describes the TBRPF routing module (which performs

topology discovery and route computation). The interface of this module with the neighbor discovery module (TND) is defined by the

three functions Link_Up(), Link_Down(), and Link_Change(), which are called by TND. Therefore, it is possible to use another neighbor discovery mechanism in place of TND, e.g., one that is provided by the link layer.

8.1. Conceptual Data Structures

In addition to the information required by the neighbor discovery protocol, each node running TBRPF contains a topology table TT, which stores information for each known link and node in the network. The following information is stored at node i for each known link (u,v) and node u:

T(u,v) - Equal to 1 if (u,v) is in node i's source tree T, and 0 otherwise. The previous source tree is also maintained as old_T.

RN(u) - Equal to 1 if u is in node i's reportable node set RN, and 0 otherwise. The previous reportable node set is also maintained as old_RN.

RT(u,v) - Equal to 1 if (u,v) is in node i's reportable subtree RT, and 0 otherwise. Since RT is defined as the set of links (u,v) in T such that u is in RN, this variable need not be maintained explicitly.

TG(u,v) = Equal to 1 if (u,v) is in node i's topology graph TG, and 0 otherwise.

N_I - The set of 2-way neighbors for interface I.

N - The set of 2-way neighbors of node i, equal to the union of N_I for all interfaces.

r(u,v) - The list of neighbors that are reporting link (u,v) in their reportable subtree RT. The set of links (u,v) reported by neighbor j is denoted RT_j.

r(u) - The list of neighbors that are reporting node u in their reportable node set RN.

p(u) - The current parent for node u, equal to the next node on the shortest path to u.

nbr_if(j) - The ID of the preferred interface for forwarding packets to neighbor j.

pred(u) - The node preceding node u in node i's source tree T. Equal to NULL if node u is not reachable.

pred(j,u) - The node preceding node u in the subtree RT_j reported

by neighbor j.

d(u) - The length of the shortest path to node u. If USE_METRICS = 0, d(u) is the number of hops to node u.

reported(u,v) - Equal to 1 if link (u,v) in TG is reported by p(u).

tg_expire(u) - Expiration time for links (u,v) in TG.

rt_expire(j,u) - Expiration time for links (u,v) in RT_j.

nr_expire(u,v) - Expiration time for a link (u,v) in TG such that reported(u,v) = 0. Such non-reported links can be used temporarily during rerouting and usually have an earlier expiration time than tg_expire(u).

The following metric variables are used only if USE_METRICS = 1:

if_metric(j,I) - The metric (or cost) for reaching neighbor j through interface I.

metric(j,u,v) - The metric for link (u,v) reported by neighbor j.

metric(u,v) - The metric for link (u,v) in TG. For a neighbor j, metric(i,j) is the minimum of if_metric(j,I) over all interfaces I such that j is in N_I.

The routing table consists of a list of tuples of the form (rt_dest, rt_next, rt_dist, rt_if_id), where rt_dest is the destination IP address or prefix, rt_next is the interface address of the next hop of the route, rt_dist is the length of the route, and rt_if_id is the ID of the local interface through which the next hop can be reached.

Each node also maintains three tables that describe associated IP addresses or prefixes: the "interface table", which associates interface IP addresses with router IDs, the "host table", which associates host IP addresses with router IDs, and the "network prefix table", which associates network prefixes with router IDs.

The "interface table" consists of tuples of the form (if_addr, if_rid, if_expire), where if_addr is an interface IP address associated with the router with RID = if_rid, and if_expire is the time at which the tuple expires and MUST be removed. The interface table at a node does NOT contain an entry in which if_addr equals the node's own RID; thus, a node does not advertise its own RID as an associated interface.

The "host table" consists of tuples of the form (h_addr, h_rid,

h_expire), where h_addr is a host IP address associated with the router with RID = h_rid, and h_expire is the time at which the tuple

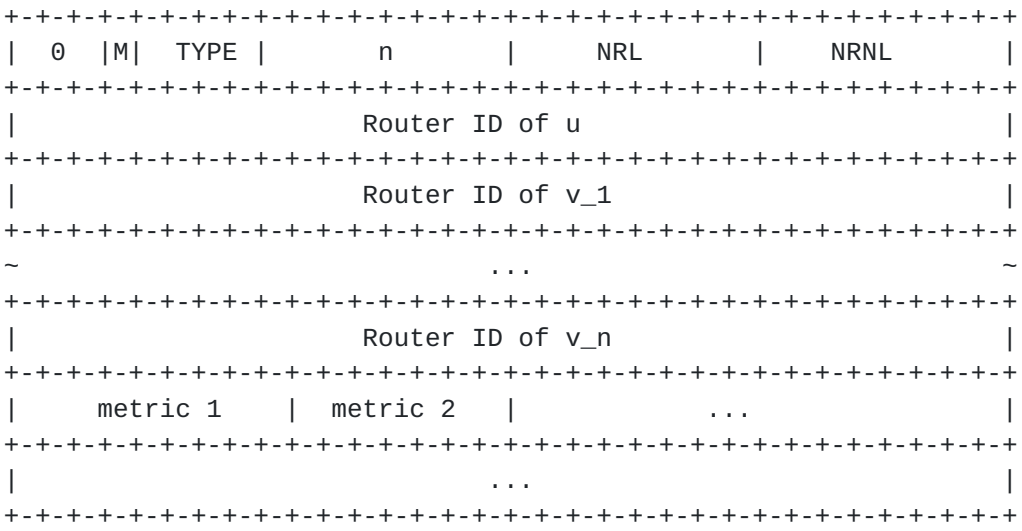
expires and MUST be removed.

The "network prefix table" consists of tuples of the form (net_prefix, net_length, net_rid, net_expire), where net_prefix and net_length describe a network prefix associated with the router with RID = net_rid, and net_expire is the time at which the tuple expires and MUST be removed. A MANET may be configured as a "stub" network, in which case one or more gateway routers may announce a default prefix such that net_prefix = net_length = 0.

Two copies of each table are kept: an "old" copy that was last reported to neighbors, and the current copy that is updated when association messages are received. We note that the "interface table" is equivalent to the "interface association information base" of OLSR, and that the "host table" and "network prefix table", if combined, would be equivalent to the "host and network association information base" of OLSR [18].

8.2. TOPOLOGY UPDATE Message Format

The TOPOLOGY UPDATE message has the following format:



The message body contains the N+1 router IDs for nodes u, v_1,...,v_n, which represent the links (u,v_1),..., (u,v_n). The first NRL of the v_k are "reported leaf nodes", the next NRNL of the v_k are "non-reported leaf nodes", and the last n - (NRL+NRNL) of the v_k are "non-reported non-leaf nodes". (The meanings of these terms are defined in the pseudocode.) The M bit indicates whether or not link metrics are included in the message. If M = 1, then a 1-octet metric is included for each of the links (u,v_1),..., (u,v_n), following the last router ID.

The TOPOLOGY UPDATE message has the following three subtypes:

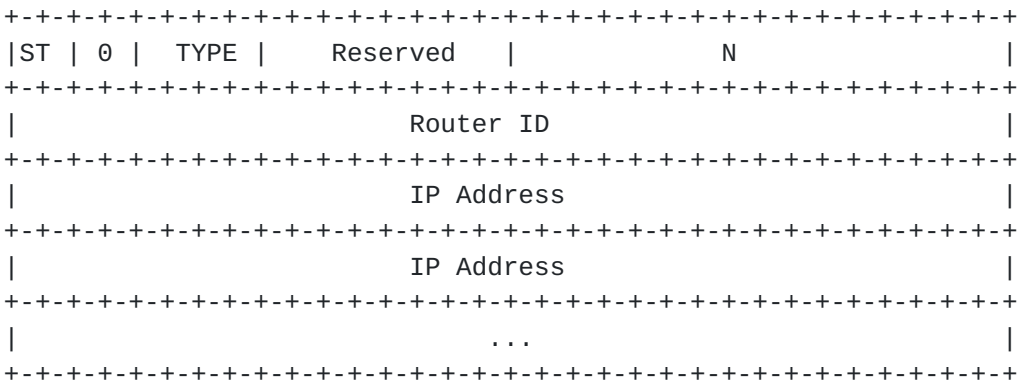
FULL (TYPE = 5)
A FULL update (FULL, n, NRL, NRNL, u, v_1,..., v_n) reports that the links (u,v_1),..., (u,v_n) belong to the sending router's reportable subtree RT, and that RT contains no other links with tail u.

ADD (TYPE = 6)
An ADD update (ADD, n, NRL, NRNL, u, v_1,..., v_n) reports that the links (u,v_1),..., (u,v_n) have been added to the sending router's reportable subtree RT.

DELETE (TYPE = 7)
A DELETE update (DELETE, n, NRL, NRNL, u, v_1,..., v_n) reports that the links (u,v_1),..., (u,v_n) have been deleted from the sending router's reportable subtree RT.

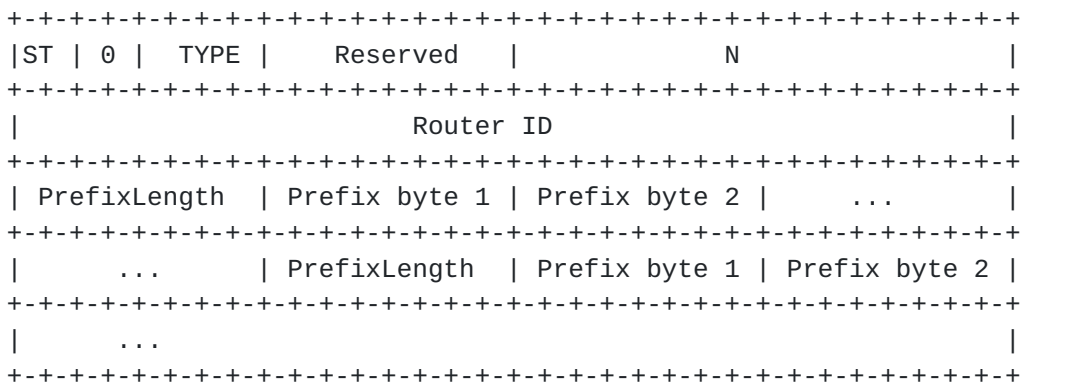
8.3. Interface, Host, and Network Prefix Association Message Formats

The INTERFACE ASSOCIATION (TYPE = 8) and HOST ASSOCIATION (TYPE = 9) messages have the following format:



The message body contains the router ID of the originating node, and N IP addresses of interfaces (TYPE = 8) or hosts (TYPE = 9) that are associated with the router ID. The ST field is defined below.

The NETWORK PREFIX ASSOCIATION message (TYPE = 10) has the following format:



The message body contains the router ID of the originating node, and N network prefixes, each specified by a 1-octet prefix length followed immediately by the prefix, using the minimum number of whole octets required. To minimize overhead, the prefix lengths and prefixes are NOT aligned along word boundaries.

The INTERFACE ASSOCIATION, HOST ASSOCIATION, and NETWORK PREFIX ASSOCIATION messages each have the following three subtypes (similar to those for the TOPOLOGY UPDATE message):

FULL (ST = 0)
: Indicates that this is a FULL update that includes all interface addresses, host addresses, or network prefixes associated with the given router ID.

ADD (ST = 1)
: Indicates that the included IP addresses or network prefixes are associated with the router ID, but may not include all such IP addresses or network prefixes.

DELETE (ST = 2)
: Indicates that the included IP addresses or network prefixes are no longer associated with the router ID.

8.4. TBRPF Operation

This section describes the operation of TBRPF topology discovery and route computation, with the aid of the pseudocode given in the next section.

The procedure Update_All() is called periodically every MIN_UPDATE_INTERVAL seconds, which is typically equal to HELLO_INTERVAL. Update_All() first generates a (different) HELLO message for each interface.

If USE_METRICS = 1 and it is time for a periodic update, link costs and metrics are then updated as described in Update_Link_Costs().

Link metrics are updated only for periodic updates, to avoid having to report changes to link metrics in differential updates (thus reducing overhead). If a given neighbor j can be reached through more than one interface, metric(i,j) is set to the minimum metric over all interfaces.

Update_Routes() is then called, which updates T, TG, and the routing table. Update_RN() is then called to update the reportable node set RN. (Recall that the reportable subtree RT is defined to be the set of links (u,v) in T such that u is in RN.) If it is time for a periodic update, Generate_Periodic_Update() is called; otherwise, Generate_Diff_Update() is called, which generates differential topology updates if RT has changed.

Generate_Association_Messages() is then called to generate INTERFACE ASSOCIATION, HOST ASSOCIATION, and NETWORK PREFIX ASSOCIATION messages. All addresses or prefixes in the interface table, host table, and network prefix table are reported periodically every IA_INTERVAL, HA_INTERVAL, and NPA_INTERVAL seconds, respectively. In addition, differential changes to the tables are reported every MIN_UPDATE_INTERVAL seconds if it is not time for a periodic update (similar to differential topology updates).

The generated messages are then sent on each interface. Whenever possible, these messages are combined into the same packet, in order to minimize the number of control packets transmitted. Although a different HELLO message is sent on each interface, identical copies of each other message are sent on all interfaces.

When a packet containing TOPOLOGY UPDATE messages is received, Process_Updates() is called to update the topology table TT, in particular to update TG and the reporting neighbor lists r(u) and r(u,v). If any link in T has been deleted from TG, then Update_Routes() is called to provide immediate rerouting.

When a packet containing one or more association messages is received, Process_Association_Messages() is called, which updates the interface table, host table, and/or network prefix table.

Link_Up(j, I) adds node j to the neighbor lists N_I (for interface I) and to N if it does not already belong, sets nbr_if(j) to I if it is NULL, and adds the link (i,j) to TG if it does not already belong, which may result in a change to the source tree T when Update_Routes() is next called.

Link_Down(j, I) removes node j from N_I, and also from N if j does not belong to N_J for any interface J. If j is removed from N, then link (i,j) is removed from TG and Update_Routes() is called to pro-

vide immediate rerouting. If the link loss is the result of a link-layer failure indication, a differential update is generated immediately.

Each node is required to report its reportable subtree RT to neighbors. However, each node (independently of the other nodes) MAY report additional links, e.g., to provide increased robustness in highly mobile networks. More precisely, a node may compute any subgraph H of TG that contains T. It would then report the "reportable subgraph" RH which consists of links (u,v) of H such that u is in RN. RH is then be reported instead of RT to all neighbors, in periodic and differential updates. Additional details for this option will be described in a future version of this draft.

Non-relay nodes are implemented simply by not generating or transmitting any TOPOLOGY UPDATE messages or NETWORK PREFIX ASSOCIATION messages. (Only HELLO, HOST ASSOCIATION, and INTERFACE ASSOCIATION messages are transmitted.) Therefore, the following functions can be omitted in non-relay nodes: Update_RN(), Generate_Periodic_Update(), and Generate_Diff_Update().

8.5. Pseudocode

The following pseudocode describes the procedures of the TBRPF routing module, as performed by node i.

```
Update_All() {
// Called periodically every MIN_UPDATE_INTERVAL seconds
  For each interface I, set msg_list(I) = empty.
  For each interface I, generate a HELLO message for
    interface I and add it to msg_list(I).
  Expire_Links().
  If (USE_METRICS = 1 AND current_time >= next_periodic)
    Update_Link_Costs().
  Update_Routes().
  If (REPORT_FULL_TREE = 0) Update_RN().
  Else Update_RN_Simple().
  If (current_time >= next_periodic) {
    Generate_Periodic_Update().
    Set next_periodic = current_time + PER_UPDATE_INTERVAL.}
  Else Generate_Diff_Update().
  Generate_Association_Messages().
  For each interface I, send msg_list(I) on interface I.
  Set old_T = T and old_RN = RN.
}

Update_Link_Costs() {
// Called only when it is time to send a periodic update,
// and only if USE_METRICS = 1.
  // Update metrics of non-adjacent links.
  For each link (u,v) in TT s.t. u != i {
```

```
          For each neighbor j in r(u,v) {
            Set metric(j,u,v) = new_metric(j,u,v).
            If (j = p(u)) {
```

```
            // Set metric(u,v) to that reported by p(u).
            Set metric(u,v) = metric(j,u,v).
            // Set link cost as a function of metric(u,v).
            Set c(u,v) = 1 + METRIC_COEFF * metric(u,v). }}
        }
        // Update metrics of adjacent links.
        For each neighbor j in N {
          Set nbr_if(j) to the ID of an interface I such that
            j is in N_I and if_metric(j,I) is minimum.
          Set metric(i,j) = if_metric(j,I), where I = nbr_if(j).
          Set c(i,j) = 1 + METRIC_COEFF * metric(i,j). }}
        }
      }

      Update_Routes() {
        For each node v in TT {
          Set d(v) = INFINITY, pred(v) = NULL.
          Set old_p(v) = p(v), p(v) = NULL. }
        Set d(i) = 0, p(i) = i, pred(i) = i.
        Set S = {i} (labeled nodes).
        For each node j in N {
          Set d(j) = c(i,j), pred(j) = i, p(j) = j. }
        // If USE_METRICS = 0, then all link costs are 1.
        While there exists a node u in TT - S s.t. d(u) < INFINITY {
          Let u be a node in TT - S that minimizes d(u).
          // A heap should be used to find u efficiently.
          Add u to S.
          If p(u) != old_p(u) {  // parent has changed
            For each link (u,v) in TG {
              If (reported(u,v) = 1) {
                Set reported(u,v) = 0. // Mark old links as not reported.
                Set nr_expire(u,v) = current_time + PER_UPDATE_INTERVAL.}}
            If (p(u) is in r(u)) {  // p(u) is reporting u
              Set tg_expire(u) = rt_expire(p(u),u).
              If (p(u) = u)
                // If u is a neighbor, remove old links.
                Remove all links (u,v) from TG.  // Remove old links.
              For each link (u,v) such that p(u) is in r(u,v) {
                Add (u,v) to TG.  // Add new links reported by p(u).
                Set reported(u,v) = 1. // (u,v) is reported by p(u)
                If (USE_METRICS = 1) {
                  // Set link metric and cost based on metric
                  // reported by p(u).
                  Set metric(u,v) = metric(p(u),u,v).
                  Set c(u,v) = 1 + METRIC_COEFF * metric(u,v). }}}
          }
          For each node v s.t. (u,v) is in TG {
            // Penalize links (u,v) not reported by p(u).
```

```
            If (reported(u,v) = 0)
              Set cost = c(u,v) + NON_REPORT_PENALTY.
            // Penalize (u,v) if u is a neighbor and is not reporting v.
```

```
          Else if (p(u) = u AND u is not in r(v))
            Set cost = c(u,v) + NON_REPORT_PENALTY.
          // Penalize (u,v) if it is not in old_T and u is not a neighbor
          If ((u,v) is not in old_T AND p(u) != u)
            Set cost = cost + NON_TREE_PENALTY.
          If (d(u) + cost < d(v) OR
              [d(u) + cost = d(v) AND RID(u) < RID(pred(v))]) {
            Set d(v) = d(u) + c(u,v).
            Set pred(v) = u.
            Set p(v) = p(u). }
        }
      }
      // Update source tree T.
      Set T = empty set.
      For each node u in TT - i, add link (pred(u), u) to T.
      Update_Routing_Table().
    }

    Update_Routing_Table() {
      Remove all tuples from the routing table.
      For each node u in TT - i such that p(u) != NULL {
        Add the tuple (rt_dest, rt_next, rt_dist, rt_if_id)
        to the routing table, where:
          rt_dest = RID(u),
          rt_if_id = nbr_if(p(u)),
          rt_next = nbr_if_addr(p(u)), obtained from the
            neighbor table for rt_if_id,
          rt_dist = d(u).
      }
      For each tuple (if_addr, if_rid, if_expire)
      in the interface table{
        If there exists a routing table entry
        (rt_dest, rt_next, rt_dist, rt_if_id)
        such that rt_dest = if_rid, add the tuple
        (if_addr, rt_next, rt_dist, rt_if_id) to the routing table.
      }
      For each tuple (h_addr, h_rid, h_expire)
      in the host table{
        If there exists a routing table entry
        (rt_dest, rt_next, rt_dist, rt_if_id)
        such that rt_dest = h_rid, add the tuple
        (h_addr, rt_next, rt_dist, rt_if_id) to the routing table,
        unless an entry already exists with the same value for
        h_addr and a smaller value for rt_dist.
      }
      For each tuple (net_prefix, net_length, net_rid, net_expire)
      in the network prefix table {
        If there exists a routing table entry
```

```
          (rt_dest, rt_next, rt_dist, rt_if_id)
          such that rt_dest = net_rid, add the tuple
          (net_prefix/net_length, rt_next, rt_dist, rt_if_id)
```

```
      to the routing table, unless an entry already exists with
      the same value for net_prefix/net_length and a smaller value
      for rt_dist.
    }
  }

  Update_RN() {
    // RN is the set of nodes reported by node i.
    Set RN = empty.
    // A neighbor j is in RN if some other neighbor s
    // would select node i as p(j).
    For each neighbor s in N s.t. j is in r(s) {
      // Initialize to run Dijkstra for source s, for 2 hops
      For each node j in N+{i},
        Set d(j) = INFINITY, par(j) = NULL.
      Set d(s) = 0, par(s) = s.
      For each link (s,j) s.t. s is in r(s,j) and j is in N+i {
        Set d(j) = 1, par(j) = j.
        For each link (j,k) s.t. j is in r(j,k) and k in in N {
          Set cost = 1.
          If (1 + cost < d(k) OR
              (1 + cost = d(k) AND RID(j) < RID(par(k)))) {
            Set d(k) = 1 + cost, par(k) = j. }}}
      // End of Dijkstra for source s
      For each neighbor j in N {
        // Add neighbor j to RN if its parent is i.
        If par(j) = i, add j to RN. }
    } // End of selection of neighbors in RN
    Add i to RN. // Node i is always in RN
    // A non-neighbor node u is in RN if p(u) is in RN.
    For each node u in TT - i,
      If p(u) is in RN, add u to RN.
  }

  Update_RN_Simple() {
  // Called only if REPORT_FULL_TREE = 1. Includes all reachable
  // nodes u in RN, so that the full source tree T is reported.
     Set RN = empty.
     Add i to RN.
     For each node u in TT - i,
        If pred(u) is not NULL, add u to RN.
  }

  Generate_Periodic_Update() {
  // Generates updates describing the reportable subtree RT.
    For each node u in RN that is not a leaf of T,
    add the update (FULL, n, NRL, NRNL, u, v_1,..., v_n)
    to msg_list(I) for each I, where:
```

(a) v_1,..., v_n are the nodes v such that (u,v) is in T,
the first NRL of these are nodes in RN that are leaves of T,
the next NRNL of these are nodes in RN that are not leaves

```
        of T, and the last n-(NRL+NRNL) of these are not in RN.
    (b) If USE_METRICS = 1, then the METRICS bit is set to 1 and the
        link metrics metric(u,v_1),..., metric(u,v_n) are included
        in the message.
  }

  Generate_Diff_Update() {
  // Generates updates reporting changes to the reportable
  // subtree RT.
    For each node u in RN {
      If u is not in old_RN and is not a leaf of T {
        // u was added to RN
        Add the update (FULL, n, NRL, NRNL, u, v_1,..., v_n)
        to msg_list(I) for each I, where:
        (a) v_1,..., v_n, NRL, and NRNL are defined as above for
            periodic updates.
        (b) If USE_METRICS = 1, then the METRICS bit is set to 1 and
            the link metrics metric(u,v_1),..., metric(u,v_n) are
            included in the message.
      }
      Else if u is in old_RN and is not a leaf of T {
        Let v_1,..., v_n be the nodes v s.t. (u,v) is in T AND {
          [(u,v) is not in old_T] OR
          [v is in RN - old_RN AND v is a leaf] OR
          [v is in old_RN - RN] }
        If this set of nodes is nonempty, add the update (ADD, n, NRL,
        NRNL, u, v_1,..., v_n) to msg_list(I) for each I, where:
        (a) NRL and NRNL are defined as above.
        (b) If USE_METRICS = 1, then the METRICS bit is set to 1 and
            the link metrics metric(u,v_1),..., metric(u,v_n) are
            included in the message.
      }
      If (u is in old_RN) {
        Let v_1,..., v_n be the nodes v s.t. (u,v) is in old_T - TG
        AND [pred(v) is NULL or is not in RN].
        // If pred(v) is in RN, the delete is implied by an add.
        If this set of nodes is nonempty, add the update
        (DELETE, n, u, v_1,..., v_n) to msg_list(I) for each I.
      }
    }
  }

  Process_Updates(j, msg_list) {
  // Processes a list of update messages from node j.
    Set update_routes_flag = 0.
    // Flag will be set to 1 if a link in T is deleted.
    For each update = (subtype, n, NRL, NRNL, u, v_1,..., v_n)
    in msg_list {
```

```
        Create an entry for u in TT if it does not exist.
        Create an entry for u in TT_j if it does not exist.
        If (subtype = FULL)
```

```
          Process_Full_Update(j, update).
        If (subtype = ADD)
          Process_Add_Update(j, update).
        If (subtype = DELETE)
          Process_Delete_Update(j, update).
        // Set update_routes_flag.
        If update_routes_flag = 0 {
          For each link (u,v) in TT(u) {
            If (u,v) is in T but not in TG {
              Set update_routes_flag = 1.
              Break. }}}
      } // End for each update
      If (update_routes_flag = 1) Update_Routes().
    }

    Process_Full_Update(j, update) {
      // update = (FULL, n, NRL, NRNL, u, v_1,..., v_n)
      Add u to RN.
      Set rt_expire(j,u) = current_time + TOP_HOLD_TIME.
      For each link (u,v) s.t. j is in r(u,v) {
        Remove j from r(u,v).
        If pred(j,v) = u, set pred(j,v) = NULL. }
      If (j = p(u) OR p(u) = NULL) {
        Set tg_expire(u) = current_time + TOP_HOLD_TIME.
        For each v s.t. (u,v) is in TG,
          // Delete old reported links before adding new ones.
          If reported(u,v) = 1, remove (u,v) from TG.
      }
      Process_Add_Update(j, update).
    }

    Process_Add_Update(j, update) {
      // update = (subtype, ADD, n, NRL, NRNL, u, v_1,..., v_n)
      For m = 1,..., n {
        Let v = v_m.
        Create an entry for v in TT if it does not exist.
        Create an entry for v in TT_j if it does not exist.
        Add j to r(u,v).
        If (j = p(u) OR p(u) = NULL) {
          Add (u,v) to TG.
          If (USE_METRICS = 1 and the METRICS bit is 1)
            Set new_metric(u,v) to the m-th metric in the update.
          Set reported(u,v) = 1.
        }
        // Process implicit delete
        Set w = pred(j,v).
        Set pred(j,v) = u.
        If (w != NULL AND w != u) {
```

```
      Remove j from r(w,v).
      If (j = p(w)) remove (w,v) from TG.
    }
```

```
      If (m <= NRL) { // v is a reported leaf
        Set leaf_update = (FULL, 0, 0, 0, u)
        Process_Full_Update(j, leaf_update).
      }
      If (m > NRL + NRNL) { // v is not reported by j
         Remove j from r(v).
         Set rt_expire(j,v) = 0.
         For each node w s.t. j is in r(v,w)
           Remove j from r(v,w).
         If (j = p(v))
           For each node w s.t. (v,w) is in TG {
             Set reported(v,w) = 0. // (v,w) is not reported by p(v)
             Set nr_expire(u,v) = current_time + PER_UPDATE_INTERVAL.}
      }
    }
  }

  Process_Delete_Update(j, update) {
    // update = (DELETE, n, NRL, NRNL, u, v_1,..., v_n)
    For m = 1,..., n {
      Let v = v_m.
      Remove j from r(u,v).
      If pred(j,v) = u, set pred(j,v) = NULL.
      If (j = p(u)) remove (u,v) from TG.
    }
  }

  Link_Up(j,I,metric) {
  // Called when a link to j is discovered on interface I
    Add j to N_I.
    Set if_metric(j,I) = metric.
    If j is not in N {
      Add j to N.
      Set nbr_if(j) = I.
      Add (i,j) to TG.
      Set report(i,j) = 1.
    }
  }

  Link_Change(j,I,metric) {
  // Called when the metric to j via I changes
    Set if_metric(j,I) = metric.
  }

  Link_Down(j,I) {
  // Called when the link to neighbor j is lost on interface I
    If j is in N_I { remove j from N_I.
    If j is in N but does not belong to N_J for any interface J {
```

```
        Remove j from N.
        Remove (i,j) from TG.
        If lost link is due to link-layer failure indication {
```

```
          Update_Routes().
          Update_RN().
          Set msg_list = empty.
          Generate_Diff_Update().
          Send msg_list on all interfaces.
          Set old_T = T and old_RN = RN.
        }
        Else Update_Routes().
      }
    }

    Expire_Links() {
      For each node u in TT - i {
        If (tg_expire(u) < current_time) {
          For each v s.t. (u,v) is in TG,
            Remove (u,v) from TG. }
        Else for each v s.t. (u,v) is in TG {
          If (report(u,v) = 0 AND nr_expire(u,v) < current_time)
            Remove (u,v) from TG. }
        For each node j in r(u) {
          If (rt_expire(j,u) < current_time) {
            Remove j from r(u).
            For each link (u,v) s.t. j is in r(u,v)
              Remove j from r(u,v). }}
      }
    }

    Generate_Association_Messages() {
      Generate_Interface_Association_Messages().
      Generate_Host_Association_Messages().
      Generate_Network_Prefix_Association_Messages().
    }

    Generate_Interface_Association_Messages() {
      If (current_time > next_ia_time) {
        next_ia_time = current_time + IA_INTERVAl.
        For each node u in RN {
          Let addr_1,..., addr_n be the interface IP addresses
          associated with RID u in the current interface table.
          If this list is nonempty, add the INTERFACE ASSOCIATION
          message (FULL, n, u, addr_1,..., addr_n) to msg_list(I)
          for each I. }
      }
      Else {
        For each node u in RN {
          Add the INTERFACE ASSOCIATION message
          (ADD, n, u, addr_1,..., addr_n) to msg_list(I) for each I,
          where addr_1,..., addr_n are the interface IP addresses
```

that are associated with RID u in the current interface table but not in the old interface table.

```
        Add the INTERFACE ASSOCIATION message
        (DELETE, n, u, addr_1,..., addr_n) to msg_list(I) for each I,
        where addr_1,..., addr_n are the interface IP addresses
        that are associated with RID u in the old interface
        table but not in the current interface table. }
    }
  }

  Generate_Host_Association_Messages() {
    If (current_time > next_ha_time) {
      next_ha_time = current_time + HA_INTERVAl.
      For each node u in RN {
        Let addr_1,..., addr_n be the host IP addresses
        associated with RID u in the current host table.
        If this list is nonempty, add the HOST ASSOCIATION message
        (FULL, n, u, addr_1,..., addr_n) to msg_list(I) for each I.}
    }
    Else {
      For each node u in RN {
        Add the HOST ASSOCIATION message
        (ADD, n, u, addr_1,..., addr_n) to msg_list(I) for each I,
        where addr_1,..., addr_n are the host IP addresses
        that are associated with RID u in the current host
        table but not in the old host table.

        Add the HOST ASSOCIATION message
        (DELETE, n, u, addr_1,..., addr_n) to msg_list(I) for each I,
        where addr_1,..., addr_n are the host IP addresses
        that are associated with RID u in the old host
        table but not in the current host table. }
    }
  }

  Generate_Network_Prefix_Association_Messages() {
    If (current_time > next_npa_time) {
      next_npa_time = current_time + NPA_INTERVAL.
      For each node u in RN {
        Let length_1, prefix_1,..., length_n, prefix_n be the
        network prefix lengths and prefixes associated with RID u
        in the current network prefix table. If this list is
        nonempty, add the NETWORK PREFIX ASSOCIATION message
        (FULL, n, u, length_1, prefix_1,..., length_n, prefix_n)
        to msg_list(I) for each I.
    }
    Else {
      For each node u in RN {
        Add the NETWORK PREFIX ASSOCIATION message
        (ADD, n, u, prefix_1,..., prefix_n) to msg_list(I) for each I,
```

where prefix_1,..., prefix_n are the network prefixes that are associated with RID u in the current prefix table but not in the old prefix table.

```
        Add the NETWORK PREFIX ASSOCIATION message
        (DELETE, n, u, prefix_1,..., prefix_n) to msg_list(I) for
        each I, where prefix_1,..., prefix_n are the network
        prefixes that are associated with RID u in the old prefix
        table but not in the current prefix table. }
    }
  }

  Process_Interface_Association_Messages(j, msg_list) {
  // Processes a list of INTERFACE ASSOCIATION messages from node j.
    For each message (subtype, n, u, addr_1,..., addr_n) in msg_list {
      If (subtype = FULL)
        Remove all entries from the interface table.
      If (subtype = FULL or ADD) {
        For m = 1,..., n {
          Add the tuple (if_addr, if_rid, if_expire) to the
          interface table, where:
            if_addr = addr_m,
            if_rid = u,
            if_expire = current_time + IA_HOLD_TIME.
        }
      }
      If (subtype = DELETE) {
        For m = 1,..., n {
          Remove the tuple (if_addr, if_rid, if_expire) from the
          interface table, where if_addr = addr_m and if_rid = u. }
      }
    }
  }

  Process_Host_Association_Messages(j, msg_list) {
  // Processes a list of HOST ASSOCIATION messages from node j.
    For each message (subtype, n, u, addr_1,..., addr_n) in msg_list {
      If (subtype = FULL)
        Remove all entries from the host table.
      If (subtype = FULL or ADD) {
        For m = 1,..., n {
          Add the tuple (h_addr, h_rid, h_expire) to the
          host table, where:
            h_addr = addr_m,
            h_rid = u,
            h_expire = current_time + HA_HOLD_TIME.
        }
      }
      If (subtype = DELETE) {
        For m = 1,..., n {
          Remove the tuple (h_addr, h_rid, h_expire) from the
          host table, where h_addr = addr_m and h_rid = u. }
```

```
          }
        }
      }
```

```
Process_Network_Prefix_Association_Messages(j, msg_list) {
// Processes a list of NETWORK PREFIX ASSOCIATION messages.
  For each message
  (subtype, n, u, length_1, prefix_1,..., length_n, prefix_n)
  in msg_list {
    If (subtype = FULL)
      Remove all entries from the network prefix table.
    If (subtype = FULL or ADD) {
      For m = 1,..., n {
        Add the tuple (net_prefix, net_length, net_rid, net_expire)
        to the network prefix table, where:
          net_prefix = prefix_m,
          net_length = length_m,
          net_rid = u,
          net_expire = current_time + NPA_HOLD_TIME.
      }
    }
    If (subtype = DELETE) {
      For m = 1,..., n {
        Remove the tuple
        (net_prefix, net_length, net_rid, net_expire)
        from the network prefix table,
        where net_prefix = prefix_m, net_length = length_m,
        and net_rid = u. }
    }
  }
}
```

8.6. Configurable Parameters

This section lists the values of the parameters used in the description of the protocol, and their proposed default values.

MIN_UPDATE_INTERVAL (1 second)

PER_UPDATE_INTERVAL (5 seconds)

TOP_HOLD_TIME (15 seconds)

NON_REPORT_PENALTY (1.01)

NON_TREE_PENALTY (0.01)

IA_INTERVAL (10 seconds)

IA_HOLD_TIME (3 x IA_INTERVAL)

HA_INTERVAL (10 seconds)

HA_HOLD_TIME (3 x HA_INTERVAL)

NPA_INTERVAL (10 seconds)

NPA_HOLD_TIME (3 x NPA_INTERVAL)

USE_METRICS (0)

METRIC_COEFF (TBD)

REPORT_FULL_TREE (0)

9. TBRPF Flooding Mechanism

This section describes a mechanism for the efficient best-effort flooding (or network-wide broadcast) of packets to all nodes of a connected ad-hoc network. These may include TBRPF packets, non-TBRPF packets, and data packets. This mechanism can be considered a special case of the flooding mechanism described in [24], in which information provided by TBRPF is used to decide whether a given received flooded packet should be forwarded, i.e., to perform selective retransmission as discussed in Section 5 of [24]. By performing selective retransmission, each packet is transmitted by only a relatively small subset of nodes, thus consuming much less bandwidth than full flooding.

For the purpose of this description, we call the flooding address ALL_MANET_NODES (TBD). Every node maintains a duplicate cache as described in [24] to keep track of which flooded packets have already been received. When a node receives a packet whose destination IP address is ALL_MANET_NODES, it checks its duplicate cache for an entry that matches the packet. If such an entry exists, the node silently discards the flooded packet since it has already been received. Otherwise, the node retransmits the packet on all interfaces (see the exception below) if and only if the following conditions hold:

1. The TBRPF node associated with the source IP address of the packet belongs to the set RN of reportable nodes computed by TBRPF.

2. When decremented, the 'ip_ttl' in the IPv4 packet header (respectively, the 'hop_count' in the IPv6 packet header) is greater than zero.

If the packet is to be retransmitted, it is sent after a small random time interval in order to avoid collisions. If the interface on which the packet was received is NOT an ad-hoc interface (see the Terminology section), then the packet need not be retransmitted on that

interface.

10. Application of TBRPF In Mobile Ad-Hoc Networks

The TBRPF routing protocols provide efficient proactive topology discovery with dynamic adaptation to link state changes in both fixed and mobile environments whether the topology is relatively static or highly dynamic in nature. TBRPF is particularly well suited to MANETs consisting of mobile nodes with wireless network interfaces operating in peer-to-peer fashion over a multiple access communications channel (e.g. the IEEE 802.11 Distributed Coordination Function (DCF) [13].) Although applicable across a much broader field of use, TBRPF is particularly well suited for supporting the standard DARPA Internet protocols [10] [15] as per current practices advocated by the IETF MANET working group. In the following sections, we discuss practical considerations the operation of TRBPF on MANETs.

10.1. Data Link Layer Assumptions

We assume a MANET data link layer that supports broadcast, multicast and unicast addressing with best-effort (not guaranteed) delivery services between neighbors (i.e. a pair of nodes within operational communications range of one another). We further assume that each interface belonging to a node in the MANET is assigned a unicast data link layer address that is unique within that particular MANET. While uniqueness for data link layer address assignments is not strictly guaranteed, the assumption of uniqueness is consistent with current practices for deployment of the Internet protocols on specific link layers, e.g. [5]. Methods for duplicate data link layer address detection and deconfliction are beyond the scope of this document.

10.2. Network Layer Assumptions

MANETs are formed as collections of MANET routers [14] and non-routing nodes that use network layer addresses when calculating the MANET topology. We assume that each node has at least one data link layer interface (as described above) and that each such interface is assigned a network layer address that is unique within the MANET. (Methods for network layer address assignment and duplicate address detection are beyond the scope of this document.) We further assume that each node will select a unique ID for use in TBRPF protocol messages, which we refer to as the Router ID (RID) whether/not the node acts as a MANET router. Finally, we assume that each MANET router supports the multi-hop relay paradigm at the network layer; i.e. each router provides an inter-node forwarding service via network layer host routes which reflect the current MANET topology as perceived by TBRPF.

10.3. Optional Automatic Address Resolution

TBRPF employs a proactive neighbor discovery protocol at the network layer that maintains bi-directional link state for neighboring nodes through the periodic transmission of messages. Since each neighbor discovery message contains both the data link and network layer address of the sender, implementations MAY employ automatic network-to-data link layer address resolution for the nodes with which they form links. An implementation may use such a mechanism to avoid additional message overhead and potential for packet loss associated with on-demand address resolution mechanisms such as ARP [9] or IPv6 Neighbor Discovery [15]. Since this behavior is optional, however, implementations MUST respond to on-demand address resolution requests in the normal manner.

10.4. Support for Multiple Interfaces and/or Alias Addresses

MANET nodes may comprise multiple interfaces; each with a unique network layer address. Additionally, MANET nodes may wish to publish *alias* addresses such as when multiple network layer addresses are assigned to the same interface or when the MANET node is serving as a Mobile IP [22] home agent for nodes that send binding updates from foreign networks. Multiple interfaces and alias addresses are advertised in INTERFACE ASSOCIATION messages, which bind each such address to the node's RID.

10.5. Support for Network Prefixes

MANET routers may wish to advertise network prefixes which the router discovered via attached networks, route redistributions from other routing protocols, or other means. Network prefixes are advertised in NETWORK PREFIX ASSOCIATION messages, which bind each such prefix to the node's RID.

10.6. Support for non-MANET Hosts

Non-MANET hosts may establish connections to MANET routers through on-demand mechanisms such as ARP or IPv6 Neighbor Discovery. Such connections do not constitute a MANET link and therefore are not reported in TBRPF topology updates. Non-MANET hosts are advertised in HOST ASSOCIATION messages, which bind the IP address of each host to the node's RID.

10.7. Internet Protocol Considerations

As with OSPF [17], TBRPF packets are sent directly over the Internet Protocol's network layer and are encapsulated solely by IP and local

data-link headers. An official IP Protocol number registration [12] for TBRPF will be procured in due course. The same IANA registration will be used whether encapsulation is via IPv4 or IPv6. (IPv6 treats the IP protocol number as a "Next Header" designator). While IP encapsulation is specified, implementations MAY employ alternate data delivery services (e.g. UDP/IP, local data-link encapsulation) in private networks. The selection of an alternate data delivery service MUST be consistent among all MANET routers in the private network.

The following subsections discuss the operation of TBRPF over IPv4 and IPv6, respectively.

10.7.1. IPv4 Operation

When the IPv4 protocol is used, all TBRPF packets are sent to the "All_MANET_Neighbors" multicast address (see: "IANA Considerations") and thus reach all MANET routers within single-hop transmission range of the sender. MANET routers MUST NOT forward packets sent to the "All_MANET_Neighbors" multicast address. Since packet loss due to link failure, interference, etc. can occur, implementations SHOULD split large TBRPF protocol packets into several smaller packets to avoid IPv4 fragmentation/reassembly. Since packets sent to "All_MANET_Neighbors" are delivered only to single-hop neighbors, Path MTU Discovery is not required. Instead, MANET routers can derive the maximum TBRPF packet length(s) directly from the maximum transmission unit(s) (MTUs) of their attached interface(s).

10.7.2. IPv6 Operation

The procedures for TBRPF are the same for IPv6 as for IPv4. (TBRPF for IPv6 will have a different version number.) The only required change for IPv6 is that the 4-octet IPv4 addresses and IPv4 prefixes in the interface, host, and network prefix tables, and in the INTERFACE ASSOCIATION, HOST ASSOCIATION, and NETWORK PREFIX ASSOCIATION messages, must be replaced by 16-octet IPv6 addresses and IPv6 prefixes. All other message types will continue to use only 4-octet router IDs, similar to OSPF for IPv6 [21]. This will help to reduce control traffic, which is an important consideration for MANETs.

Transition mechanisms described in the IETF NGTRANS working group (e.g. ISATAP) enable IPv6 operation over IPv4 routing infrastructures. ISATAP [19] can be used on TBRPF MANETs to enable automatic IPv6-in-IPv4 operation regardless of route changes due to mobility.

11. IANA Considerations

An official IANA IP protocol number assignment is required for TBRPF protocol messages. Additionally, an IANA IPv4 multicast address

assignment for "All_MANET_Neighbors" is required from the range of addresses between 224.0.0.0 and 224.0.0.255 inclusive. This range of addresses is reserved for the use of routing protocols and other low-level topology discovery or maintenance protocols, such as gateway discovery and group membership reporting. Multicast routers should not forward any multicast datagram with destination addresses in this range, regardless of its TTL [23].

Note that such an IPv4 multicast address assignment may have general application for MANET beyond its anticipated use for TBRPF. We therefore propose that MANET consider the procurement of an "All_MANET_Neighbors" IPv4 multicast address from the range of addresses between 224.0.0.0 and 224.0.0.255.

12. Security Considerations

Authentication issues exist for allowing "foreign" nodes to join a MANET via TBRPF neighbor discovery. Additionally, privacy issues exist for any networking protocols run over unencrypted wireless data links such as IEEE 802.11. Finally, denial-of-service attacks are possible if rogue nodes join a TBRPF MANET and offer to provide a multi-hop relay service, but then fail to perform the service when it is required.

The TBRPF protocol can be extended in a straightforward manner to incorporate authentication of TBRPF protocol packets in a manner similar to OSPF version 2 [RFC 2328]. Here, a shared secret key is configured in all TBRPF routers. For each TBRPF protocol packet, the key is used to generate/verify a "message digest" that is appended to the end of the TBRPF packet. The message digest is a one-way function of the TBRPF protocol packet and the secret key [RFC 2104]. For the widely used MD5 message-digest algorithm [RFC 1321], the "message digest" is 16 bytes in length.

Alternatively, the IP Authentication Header algorithm [RFC 2401, 2402] within the IPSec Working Group can be used to provide authentication for TBRPF protocol packets.

13. Implementation Status

TBRPF version 2 (as described in version 3 of the draft) has been implemented in Linux and ns-2.

TBRPF version 1 (as described in version 0 of the draft) has been implemented in the FreeBSD V3.2 operating system with the Merit Multi-Threaded Routing Toolkit daemon (mrtd). The initial FreeBSD V3.2 implementation has been in use for laboratory and fielded exper-

iments since June 1999.

TBRPF version 1 has been ported to Linux. The current port runs on RedHat Version 7.0 and has been tested under multiple Linux kernel releases including 2.2.16 and a 2.4 pre-release. Additionally, the Linux and FreeBSD ports are fully interoperable under TBRPF Version 1.

14. Patent Rights Statement

SRI International has filed one or more patents protecting the inventions described in this submission. If SRI's submission (or portions thereof) is accepted and becomes part of the IETF standard, then SRI will grant royalty-free permission under such patents for both commercial and non-commercial uses, to the extent necessary for compliance with the standard. Any aspects or variants of SRI's submission that are not included in the IETF standard and/or are not necessary for compliance with the standard may require an additional license from SRI under terms to be negotiated by the parties.

15. Acknowledgments

The authors would like to thank ASEO for funding this work, and the following people for helping in the development of TBRPF: Yonael Gorfu and Julie S. Wong.

16. References

[1] B. Bellur and R.G. Ogier. A Reliable, Efficient Topology Broadcast Protocol for Dynamic Networks. Proc. IEEE INFOCOM '99, New York, March 1999.

[2] S. Bradner. Key words for use in RFCs to Indicate Requirement Levels. RFC 2119, March 1997.

[3] M.S. Corson and J. Macker. Mobile Ad hoc Networking (MANET): Routing Protocol Performance Issues and Evaluation Consideration. RFC 2501, 1999.

[4] J.J. Garcia-Luna-Aceves and M. Spohn. Efficient Routing in Packet-Radio Networks Using Link-State Information. Proc. IEEE WCNC '99, September 1999.

[5] C. Hornig. Standard for the Transmission of IP Datagrams over Ethernet Networks. STD0041, April 1984.

[6] D.B. Johnson and D.A. Maltz. Protocols for Adaptive Wireless and Mobile Networking. IEEE Personal Communications, Vol. 3, No. 1,

pp 34-42, February 1996.

[7] R.G. Ogier. Efficient Routing Protocols for Packet-Radio Networks Based on Tree Sharing. Proc. Sixth IEEE Intl. Workshop on Mobile Multimedia Communications (MOMUC'99), November 1999.

[8] C.E. Perkins, E.M. Belding-Royer, and S.R. Das. Ad Hoc On-Demand Distance Vector (AODV) Routing. draft-ietf-manet-aodv-09.txt, November 2001 (work in progress).

[9] D.C. Plummer. An Ethernet address resolution protocol: Or converting network protocol addresses to 48.bit Ethernet addresses for transmission on Ethernet hardware. RFC 826, November 1982.

[10] J. Postel. Internet Protocol. RFC 791, September 1981.

[11] J. Postel. User Datagram Protocol. RFC 768, August 1980.

[12] J. Reynolds and J. Postel. Assigned Numbers. RFC 1700, October 1994.

[13] Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications, ISO/IEC Std. 8802-11, ANSI/IEEE Std 802.11, 1999.

[14] An Internet MANET Encapsulation Protocol (IMEP) Specification, draft-ietf-manet-imep-spec-01.txt

[15] S. Deering and R. Hinden. Internet Protocol Version 6 (IPv6) Specification. RFC 2460, December 1998.

[16] S. Corson. MANET Routing Protocol Applicability Statement, draft-ietf-manet-appl-00.txt, November 1998 (work in progress).

[17] J. Moy. OSPF Version 2. RFC 1583, March 1994.

[18] T. Clausen et al. Optimized Link State Routing Protocol, draft-ietf-manet-olsr-05.txt, October 2001 (work in progress).

[19] F. Templin. "Intra-Site Automatic Tunnel Addressing Protocol (ISATAP)", draft-ietf-ngtrans-isatap-02.txt (work in progress)

[20] D. Bertsekas and R. Gallager. Data Networks, Prentice-Hall, 1987.

[21] R. Coltun, D. Ferguson, and J. Moy. OSPF for IPv6. RFC 2740, December 1999.

[22] C. Perkins. IP Mobility Support. RFC 2002, October 1996.

[23] F. Baker. Requirements for IP Version 4 Routers. RFC 1812, June

1995.

[24] C.E. Perkins, E.M. Belding-Royer, and S.R. Das. IP Flooding in Ad Hoc Mobile Networks, November 2001 (work in progress).

Authors' Addresses

Richard G. Ogier
SRI International
333 Ravenswood Ave.
Menlo Park, CA 94025

Phone: +1 650 859-4216
Fax: +1 650 859-4812
Email: ogier@erg.sri.com

Fred L. Templin
SRI International
333 Ravenswood Ave.
Menlo Park, CA 94025

Phone: +1 650 859-3144
Fax: +1 650 859-4812
Email: templin@erg.sri.com

Bhargav Bellur
SRI International
333 Ravenswood Ave.
Menlo Park, CA 94025

Phone: +1 650 859-6335
Fax: +1 650 859-4812
Email: bhargav@erg.sri.com

Mark G. Lewis
SRI International
333 Ravenswood Ave.
Menlo Park, CA 94025

Phone: +1 650 859-4302
Fax: +1 650 859-4812
Email: lewis@erg.sri.com