        Topology Dissemination Based on Reverse-Path Forwarding (TBRPF)

Status of this Memo

Copyright Notice

Abstract

   Topology Dissemination Based on Reverse-Path Forwarding (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. TBRPF uses a combination of
   periodic and differential updates to keep all neighbors informed of
   the reported 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.

Table of Contents

1. Introduction

   Topology Dissemination Based on Reverse-Path Forwarding (TBRPF) is a
   proactive, link-state routing protocol designed for mobile ad-hoc
   networks (MANETs), which 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.

   TBRPF uses a combination of periodic and differential updates to keep
   all neighbors informed of the reported 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 [6].

   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. Requirements

   The keywords "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT",
   "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL", when
   they appear in this document, are to be interpreted as described in
   BCP 14, RFC 2119 [1].

   This document also makes use of internal conceptual variables to
   describe protocol behavior and external variables that an
   implementation must allow system administrators to change. The
   specific variable names, how their values change, and how their
   settings influence protocol behavior are provided to demonstrate
   protocol behavior. An implementation is not required to have them in
   the exact form described here, so long as its external behavior is
   consistent with that described in this document.

3. Terminology

   The following terms are used to describe TBRPF:

   node
      A router that implements TBRPF.

   router ID
      Each node is identified by a unique 32-bit router ID (RID), which
      for IPv4 is typically equal to the IP address of one of its
      interfaces. The term "node u" denotes the node whose RID is equal
      to u.

   interface
      A node's attachment to a communication facility or medium through
      which it can communicate with other nodes. 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 its IP address. The term "interface I"
      denotes the interface whose IP address is I.

   link
      A link is an ordered pair of interfaces (I,J) where I and J are on
      two different nodes, and where interface I has recently received
      packets sent from interface J. A link (i,j) from node i to node j
      is said to exist if node i has an interface I and node j has an
      interface J such that (I,J) is a link. Nodes i and j are called
      the "tail" and "head" of the link, respectively.

   bidirectional link
      A link (I,J) such that interfaces I and J can both hear each
      other. Also called a 2-way link.

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.

MANET interface
    Any wireless interface such that two neighbor nodes on the
    interface need not be neighbors of each other. MANET nodes
    typically have at least one MANET 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.

source tree
    The directed tree (denoted T) computed by each node that provides
    shortest paths to all other reachable nodes.

topology update
    A message that reports the state of one or more links.

parent
    The parent of node i for node u is the next node on the computed
    shortest path from node i to node u.

predecessor
    The predecessor of a node v on the source tree is the node u such
    that the link (u,v) is in the source tree.

leaf node
    A leaf node of the source tree is a node on the source tree that
    is not the predecessor of any other node on the source tree.

proactive routing protocol
    A routing protocol in which each node maintains routes to all
    reachable destinations at all times, whether or not there is
    currently any need to deliver packets to those destinations. In
    contrast, an "on-demand" routing protocol discovers and maintains
    routes only when they are needed.

4. Applicability Section

    TBRPF is a proactive routing protocol designed for mobile ad-hoc
    networks (MANETs). It can support networks with up to a few hundred
    nodes, and can be combined with hierarchical routing techniques to
    support much larger networks. Because it employs techniques to

greatly reduce control traffic, TBRPF can support much larger and
denser networks than routing protocols based on the classical link-
state algorithm (e.g., OSPF).

The number of nodes that can be supported depends on several factors,
including the MAC data rate, the rate of topology changes, and the
network density (average number of neighbors). Simulations have been
reported in which TBRPF has supported as many as 500 nodes. In
simulations with 100 nodes and 20 traffic streams (sources), using
IEEE 802.11 with a data rate of 2 Mbps, TBRPF was found to generate
approximately 80-120 kb/s of routing control traffic for the
scenarios considered, which compared favorably with other MANET
routing protocols [7][8]. A proof of correctness for TBRPF can be
found in references [8] and [9].

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 i to
   quickly detect the neighbor nodes j such that a bidirectional link
   (I,J) exists between an interface I of node i and an interface J of
   node j. The protocol also quickly detects when a bidirectional link
   breaks or becomes unidirectional.

   The key feature of TND is that it uses "differential" HELLO messages
   which report only *changes* in the status of links. This results in
   HELLO messages that are much smaller than those of other link-state
   routing protocols such as OSPF, in which each HELLO message includes
   the IDs of *all* neighbors. As a result, HELLO messages can be sent
   more frequently, which allows faster detection of topology changes.

   TND is designed to be fully modular and independent of the routing
   module. TND performs ONLY neighbor sensing, i.e., it determines
   which nodes are (1-hop) neighbors. In particular, it does not
   discover 2-hop neighbors (which is handled by the routing module).
   As a result, TND can be used by other routing protocols, and TBRPF
   can use another neighbor discovery protocol 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 local
   interface, and a HELLO sent on a particular interface contains only
   information regarding neighbors heard on that interface.

We note that, in wireless networks, it is possible for a single
interface I to receive packets from multiple interfaces J associated
with the same neighbor node. This could happen, for example, if the
neighbor uses a directional antenna with different interfaces
representing different beams. For this reason, TBRPF includes
neighbor interface addresses in HELLO messages, unlike OSPF, which
includes only router IDs in HELLO packets.

Each TBRPF node maintains a neighbor table for each local interface
I, which stores state information for each neighbor interface J heard
on that interface, i.e., for each link (I,J) between interface I and
a neighbor interface J. The status of each link can be 1-WAY, 2-WAY,
or LOST. The neighbor table for interface I determines the contents
of HELLO messages sent on interface I, and is updated based on HELLO
messages received on interface I (and possibly on link-layer
notifications).

Each TBRPF node sends (on each interface) at least one HELLO message
per HELLO_INTERVAL. Each HELLO message contains three (possibly
empty) lists of neighbor interface addresses (which are formatted as
three message subtypes): NEIGHBOR REQUEST, NEIGHBOR REPLY, and
NEIGHBOR LOST. Each HELLO message also contains the current HELLO
sequence number (HSEQ), which is incremented with each transmitted
HELLO.

In the following overview of the operation of TND, we assume that
interface I belongs to node i, and interface J belongs to node j.
When a node i changes the status of a link (I,J), it includes the
neighbor interface address J in the appropriate list (NEIGHBOR
REQUEST/REPLY/LOST) in at most NBR_HOLD_COUNT (typically 3)
consecutive HELLOs sent on interface I. This ensures that node j
will either receive one of these HELLOs on interface J, or will miss
NBR_HOLD_COUNT HELLOs and thus declare the link (J,I) to be LOST.
This technique makes it unnecessary for a node to include each 1-WAY
or 2-WAY neighbor in HELLOs indefinitely, unlike OSPF.

To avoid establishing a link that is likely to be short lived (i.e.,
to employ hysteresis), node i must receive (on interface I) at least
HELLO_ACQUIRE_COUNT (e.g., 2) of the last HELLO_ACQUIRE_WINDOW (e.g.,
3) HELLOs sent from a neighbor interface J, before declaring the link
(I,J) to be 1-WAY. When this happens, node i includes J in the
NEIGHBOR REQUEST list in each of its next NBR_HOLD_COUNT HELLO
messages sent on interface I, or until a NEIGHBOR REPLY message
containing I is received on interface I from neighbor interface J.

If node j receives (on interface J) one of the HELLOs sent from
interface I that contains J in the NEIGHBOR REQUEST list, then node j
declares the link (J,I) to be 2-WAY (unless it is already 2-WAY), and

includes I in the NEIGHBOR REPLY list in each of its next
NBR_HOLD_COUNT HELLO messages sent on interface J. Upon receiving
one of these HELLOs on interface I, node i declares the link (I,J) to
be 2-WAY.

If node i receives a HELLO on interface I, sent from neighbor
interface J, whose HSEQ indicates that at least NBR_HOLD_COUNT HELLOs
were missed, or if node i receives no HELLO on interface I sent from
interface J within NBR_HOLD_TIME seconds, then node i changes the
status of link (I,J) to LOST (unless it is already LOST), and
includes J in the NEIGHBOR LOST list in each of its next
NBR_HOLD_COUNT HELLO messages sent on interface I (unless the link
changes status before these transmissions are complete). Node j will
either receive one of these HELLOs on interface J or will miss
NBR_HOLD_COUNT HELLOs; in either case, node j will declare the link
(J,I) to be LOST. In this manner, both nodes will agree that the
link between I and J 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 for each
link (I,J) from a local interface I to a neighbor interface J,
representing the quality of the link. Such link metrics can be used
as additional conditions for changing the status 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 to be
used for computing shortest paths.

5.2. Overview of the Routing Module

   Each node running TBRPF maintains a source tree, denoted T, which
   provides shortest paths to all reachable nodes. Each node computes
   and updates its source tree 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. The main idea behind the current version
   of TBRPF came from PTSP [10], another protocol in which each node
   reports only part of its source tree. (However, TBRPF differs from
   PTSP in several ways.) The current version of TBRPF should not be
   confused with its previous version [11], which is a full-topology
   routing protocol.

   The part of T that a node reports to neighbors is called the
   "reported subtree" and is denoted RT. 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 or periodic 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 supports multiple interfaces, associated hosts, and network
prefixes. Information regarding associated interfaces, hosts, and
prefixes is disseminated efficiently in periodic and differential
updates, similar to the dissemination of topology updates.

The reported subtree RT consists of links (u,v) of T such that u is
in the "reported node set" RN, which is computed as follows. Node i
includes a neighbor j in RN if and only if node i determines that one
of its neighbors may select i to be its next hop on its shortest path
to j. To make this determination, node i computes the shortest
paths, up to 2 hops, from each neighbor to each other neighbor, using
only neighbors (or node i itself) as an intermediate node, and using
relay priority (included in HELLO messages) and router ID to break
ties. After a node determines which neighbors are in RN, each
reachable node u is included in RN if and only if the next hop on the
shortest path to u is in RN. A node also includes itself in RN. As
a result, the reported subtree RT includes the subtrees of T that are
rooted at neighbors in RN, and also includes all local links to
neighbors.

We note that neighbors in RN are analogous to multipoint relay (MPR)
selectors [12]. Thus, if node i selects neighbor j to be in RN, then
node i effectively selects itself to be an MPR of node j. This is
quite different from [12], in which a node does not select itself to
be an MPR, but selects a subset of its neighbors to be MPRs.

A node with a larger relay priority reports a larger part of its
source tree (on average), and is more likely to be selected as a
next-hop relay by its neighbors. A node with relay priority equal to
0 is called a non-relay node, and never forwards packets originating
from other nodes.

TBRPF does not use sequence numbers for topology updates, thus
reducing message overhead and avoiding wraparound problems. Instead,
a technique similar to SPTA [13] is used in which, for each link
(u,v) reported by one or more neighbors, only the next hop p(u) to u
is believed regarding 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 links that are believed

to be up, 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 reported 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.

TBRPF allows the option to include link metrics 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.

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 messages and padding options as needed. 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 or not aligned on natural
   boundaries. 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). The
   format for the packet header is as follows:

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| Vers |L|I|R|R|   Reserved    |      Header Extensions ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```
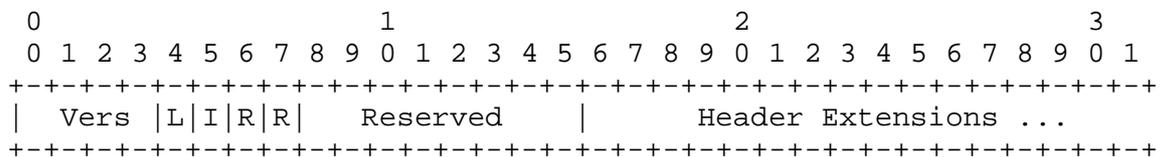
Version (4 bits)
   The TBRPF version number. This specification documents version 4
   of the protocol.

Flags (4 bits)
   Two bits (L,I) specify which header extensions (if any) follow.
   Two bits (R) are reserved for future use, and MUST be zero. Any
   extensions specified by these bits MUST appear in the same order
   as the bits (i.e., first L, then I) as follows:

L - Length included
   If the underlying delivery service provides a length field, the
   sender MAY set L = '0' and omit the length extension. Otherwise,
   the sender MUST set L = '1' and include a 16-bit unsigned integer
   length immediately after any previous header field. The length
   includes all header and data bytes and is written into the length
   field in network byte order.

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

I - Router ID (RID) included
   If the underlying delivery service encodes the sender's RID, the
   sender MAY set I = '0' and omit the RID field. Otherwise, the
   sender MUST set I = '1' and include a 4-octet RID in network byte
   order immediately after any previous header fields. The RID
   option provides a mechanism for implicit network-level address
   resolution. A receiver that detects a RID option SHOULD create a
   binding between the RID and the source address that appears in the
   network-level header.

Reserved
   Reserved for future use; MUST be zero.

6.2. TBRPF Packet Body

The TBRPF packet body consists of the concatenation of one or more
TBRPF messages (and padding options where necessary). Messages and
padding options within the TBRPF packet body are encoded using the
following format:

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

OPTIONS (4 bits)
   Four option bits that depend on TYPE.

TYPE (4 bits)
   Identifier for message type or padding option.

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

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 [2]. TBRPF
packet elements include padding options and messages as described
below.

6.2.1. Padding Options (TYPE = 0 thru 1)

Senders MAY insert two types of padding options where necessary,
e.g., to satisfy alignment requirements for other elements [2].
Padding options may occur anywhere within the TBRPF packet body. The
following two padding options are defined:

 Pad1 option (TYPE = 0)

```
+-+-+-+-+-+-+-+-+
|   0   |   0   |
+-+-+-+-+-+-+-+-+
```
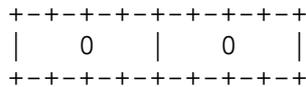
The Pad1 option inserts one octet of padding into the TBRPF packet
body; the VALUE field is omitted. If more than one octet of padding
is required, the PadN option (described next) should be used, rather
than multiple Pad1 options.

 PadN option (TYPE = 1)

```
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+- - - - - - - - - - -
|   0   |   1   |      LEN      | Zero-valued Octets
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+- - - - - - - - - - -
```
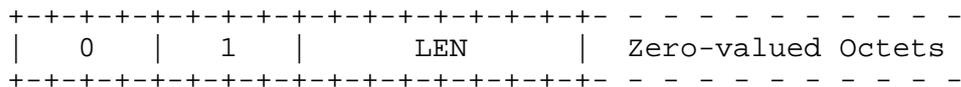
The PadN option inserts two or more octets of padding into the TBRPF
packet body. The first octet of the VALUE field contains an 8-bit
unsigned integer length containing a value between 0 - 253 which
specifies the number of zero-valued octets that immediately follow,
yielding a maximum total of 255 padding octets.

6.2.2. Messages (TYPE = 2 thru 10)

   Additional message types are described as they occur in the following
   sections. Senders encode messages as specified by the individual
   message formats. Receivers detect errors in message construction,
   e.g., messages with unrecognized types, messages with a non-integral
   number of elements, or with fewer elements than indicated, etc. In
   all cases, upon detecting an error, the receiver 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 bidirectional links (I,J)
   between a local interface I and a neighbor interface J, and to
   quickly detect the loss of such links. The interface between TND and
   the routing module is defined by the neighbor table maintained by TND
   and the three procedures Link_Up(I,J), Link_Down(I,J), and
   Link_Change(I,J), which are called by TND to announce a new link, the
   loss of a link, and a change in the metric of a link, respectively.

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:

```
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |  0  | TYPE  |     HSEQ      |  Pri  |            n             |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |                Neighbor Interface Address (1)                |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |                Neighbor Interface Address (2)                |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   ~                             ...                              ~
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |                Neighbor Interface Address (n)                |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```
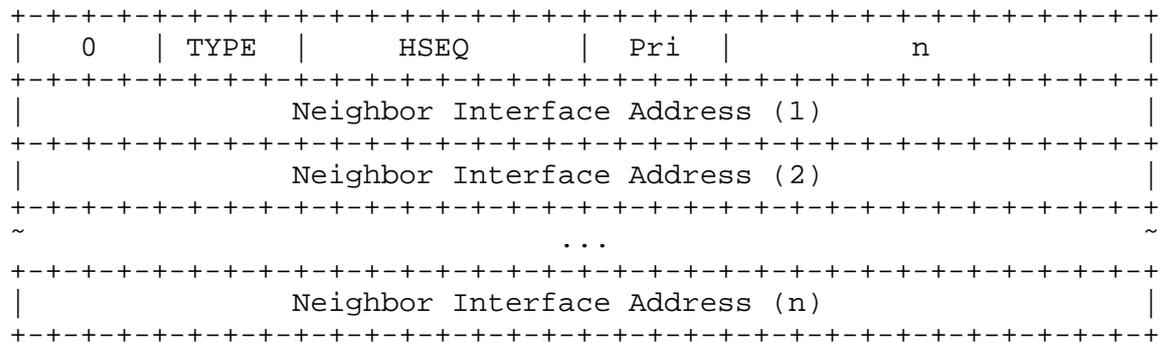
   HSEQ (8 bits)
      The HELLO sequence number.

Pri (4 bits)
   This field indicates the sending node's relay priority, which is
   an integer between 0 and 15. A node with a higher relay priority
   is more likely to be selected as the next hop on a route. The
   value 0 is reserved for non-relay nodes, i.e., nodes that should
   never forward packets originating from other nodes. A router in
   normal operation SHOULD have a relay priority equal to 7. A
   router can change its relay priority dynamically, e.g., when its
   power supply becomes critical.

n (12 bits)
   The number of 32-bit neighbor interface addresses in the message.

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 neighbor interface
addresses is empty. Thus, a HELLO message always includes a
(possibly empty) NEIGHBOR REQUEST.

7.2. Neighbor Table

Each node maintains, for each of its local interfaces I, a neighbor
table, which stores state information for each neighbor interface J
from which HELLO messages have recently been received by interface I.
The entry for neighbor interface J, in the neighbor table for I,
contains the following variables:

   nbr_rid(I,J) - The router ID of the node associated with neighbor
   interface J.

   nbr_status(I,J) - The current status of the link (I,J), which can
   be LOST, 1-WAY, or 2-WAY.

   nbr_life(I,J) - The amount of time (in seconds) remaining before
   nbr_status(I,J) must be changed to LOST if no further HELLO
   message from interface J is received. Set to NBR_HOLD_TIME
   whenever a HELLO is received on interface I from interface J.

   nbr_hseq(I,J) - The value of HSEQ in the last HELLO message
   received on interface I from interface J. Used to determine the
   number of HELLOs that have been missed.

   nbr_count(I,J) - The remaining number of times a NEIGHBOR REQUEST/
   REPLY/LOST message containing J must be sent on interface I.

   hello_history(I,J) - A list of the sequence numbers of the last
   HELLO_ACQUIRE_WINDOW HELLO messages received on interface I from
   interface J.

nbr_metric(I,J) - An optional measure of the quality of the link
(I,J), represented by an integer between 1 and 255, where smaller
values indicate better quality. Defaults to 1 if not used.

nbr_pri(I,J) - The relay priority of the node associated with
interface J.

The entry for interface J in the neighbor table for interface I may
be deleted if no HELLO has been received on interface I from
interface J within the last 2*NBR_HOLD_TIME seconds. (It is kept
while NEIGHBOR LOST messages containing J are being transmitted.)
The absence of an entry for a given interface J is equivalent to an
entry with nbr_status(I,J) = LOST and hello_history(I,J) = NULL.

The three possible values of nbr_status(I,J) have the following
informal meanings (the exact meanings are defined by the protocol):

LOST
    Interface I has not received a sufficient number of HELLO messages
    recently from Interface J.

1-WAY
    Interface I has received a sufficient number of HELLO messages
    recently from Interface J, but the link is not 2-WAY.

2-WAY
    Interfaces I and J have both received a sufficient number of HELLO
    messages recently from each other.

7.3. Sending HELLO Messages

Each node MUST send, on each local interface, at least one HELLO
message per HELLO_INTERVAL. HELLO messages MAY be sent more
frequently than this (e.g., for faster detection of topology
changes). However, to avoid the possibility that HSEQ wraps around
to the same number before a neighbor that stops receiving HELLO
messages changes the status of the link to LOST, the time between two
consecutive HELLO messages (sent on a given interface) MUST be
greater than NBR_HOLD_TIME/128 second.

To avoid synchronization of control messages, which can result in
collisions, HELLO messages SHOULD NOT be transmitted at equal
intervals. To achieve this, a node MAY choose the interval between
consecutive HELLO messages to be HELLO_INTERVAL - jitter, where
jitter is selected randomly from the interval [0, MAX_JITTER].

Each HELLO message always includes a NEIGHBOR REQUEST message, even
if its list of neighbor addresses is empty. The NEIGHBOR REQUEST
message includes the sequence number HSEQ, which is incremented by 1
(modulo 256) each time a HELLO is sent. The HELLO message also
includes a NEIGHBOR REPLY message if its list of neighbor addresses
is nonempty, and a NEIGHBOR LOST message if its list of neighbor
addresses is nonempty. The contents of these three messages are
determined by the following steps at node i for each interface I:

1. For each interface J such that nbr_status(I,J) = LOST and
   nbr_count(I,J) > 0, include J in the NEIGHBOR LOST message and
   decrement nbr_count(I,J).

2. For each interface J such that nbr_status(I,J) = 1-WAY and
   nbr_count(I,J) > 0, include J in the NEIGHBOR REQUEST message and
   decrement nbr_count(I,J).

3. For each interface J such that nbr_status(I,J) = 2-WAY and
   nbr_count(I,J) > 0, include J in the NEIGHBOR REPLY message and
   decrement nbr_count(I,J).

If a node restarts, so that all entries are removed from the neighbor
table, then the node MUST ensure that (for each interface) at least
one of the following two conditions is satisfied:

1. The difference between the transmission times of the first HELLO
   sent after restarting and the last HELLO sent before restarting is
   at least 2*NBR_HOLD_TIME.

2. Letting HSEQ_LAST denote the sequence number of the last HELLO
   that was sent before restarting, the sequence number of the first
   HELLO sent after restarting is set to HSEQ_LAST + NBR_HOLD_COUNT +
   1 (modulo 256).

Either of these conditions ensures that, if node i with interface I
restarts, then each neighbor of node i that has a link (J,I) to
interface I will set the status of the link to LOST.

7.4. Processing a Received HELLO Message

When a node receives a HELLO message, it obtains the IP address of
the sending interface from the IP header. If the TBRPF packet header
of the received HELLO contains the RID option, then the RID of the
sending node is obtained from the TBRPF packet header; otherwise it
is equal to the IP address of the sending interface. If node i (with
RID equal to i) receives a HELLO message on interface I, sent by node
j (with RID equal to j) on interface J, with sequence number HSEQ and
relay priority PRI, then node i performs the following steps:

1. If the neighbor table for interface I does not contain an entry
   for interface J, create one with nbr_rid(I,J) = j, nbr_status(I,J)
   = LOST (temporarily), nbr_count(I,J) = 0, and nbr_hseq(I,J) =
   HSEQ.

2. Update hello_history(I,J) to reflect the received HELLO message.
   If nbr_hseq(I,J) > HSEQ (due to wraparound), set nbr_hseq(I,J) =
   nbr_hseq(I,J) - 256.

3. If nbr_status(I,J) = LOST and hello_history(I,J) indicates that
   HELLO_ACQUIRE_COUNT of the last HELLO_ACQUIRE_WINDOW HELLO
   messages from interface J have been received:

   a. If interface I does not appear in the NEIGHBOR REQUEST list or
      the NEIGHBOR REPLY list, set nbr_status(I,J) = 1-WAY and
      nbr_count(I,J) = NBR_HOLD_COUNT.

   b. Else, set nbr_status(I,J) = 2-WAY and nbr_count(I,J) =
      NBR_HOLD_COUNT. Call Link_Up(I,J).

4. Else, if nbr_status(I,J) = 1-WAY:

   a. If HSEQ - nbr_hseq(I,J) > NBR_HOLD_COUNT, then set
      nbr_status(I,J) = LOST and nbr_count(I,J) = NBR_HOLD_COUNT.

   b. Else, if interface I appears in the NEIGHBOR REQUEST list, set
      nbr_status(I,J) = 2-WAY and nbr_count(I,J) = NBR_HOLD_COUNT.
      Call Link_Up(I,J).

   c. Else, if interface I appears in the NEIGHBOR REPLY list, set
      nbr_status(I,J) = 2-WAY and nbr_count(I,J) = 0. Call
      Link_Up(I,J).

5. Else, if nbr_status(I,J) = 2-WAY:

   a. If interface I appears in the NEIGHBOR LOST list, set
      nbr_status(I,J) = LOST and nbr_count(I,J) = 0. Call
      Link_Down(I,J).

   b. Else, if HSEQ - nbr_hseq(I,J) > NBR_HOLD_COUNT, set
      nbr_status(I,J) = LOST and nbr_count(I,J) = NBR_HOLD_COUNT.
      Call Link_Down(I,J).

   c. Else, if interface I appears in the NEIGHBOR REQUEST list and
      nbr_count(I,J) = 0, set nbr_count(I,J) = NBR_HOLD_COUNT.

6. Set nbr_life(I,J) = NBR_HOLD_TIME, nbr_hseq(I,J) = HSEQ, and
   nbr_pri(I,J) = PRI.

7.5. Expiration of Timer nbr_life

   Upon expiration of the timer nbr_life(I,J) in the neighbor table for
   interface I, node i performs the following step:

      If nbr_status(I,J) = 1-WAY or 2-WAY, set nbr_status(I,J) = LOST
      and nbr_count(I,J) = NBR_HOLD_COUNT. Call Link_Down(I,J).

7.6. Link-Layer Failure Notification

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

      If nbr_status(I,J) = 2-WAY, set nbr_status(I,J) = LOST and
      nbr_count(I,J) = NBR_HOLD_COUNT. Call Link_Down(I,J).

7.7. Optional Link Metrics

   Each node MAY maintain and update one or more link metrics for each
   link (I,J), 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 specification.

   The function Link_Change(I,J) is called to alert the routing module
   whenever nbr_metric(I,J) changes significantly. If the configurable
   parameter USE_METRICS is equal to 1, then the metrics nbr_metric(I,J)
   are used by the routing module for route computation, as described in
   Section 8.

7.8. Configurable Parameters

   This section lists the parameters used by the neighbor discovery
   protocol, and their proposed default values. All nodes MUST be
   configured to have the same value for all of the following
   parameters.

       Parameter Name   Default Value
       --------------   -------------
       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.

8.1. Conceptual Data Structures

   In addition to the information required by the neighbor discovery
   protocol, each node running TBRPF maintains a topology table TT,
   which stores information for each known node and link in the network.
   Nodes are identified by their RIDs, i.e., node u is the node whose
   RID is u. The following information is stored in the topology table
   at node i for each node u and link (u,v):

      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 reported node set RN, and 0
      otherwise. The previous reported node set is also maintained as
      old_RN.

      RT(u,v) - Equal to 1 if (u,v) is in node i's reported 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 - The set of 2-way neighbors of node i.

r(u,v) - The list of neighbors that are reporting link (u,v) in
their reported 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
reported node set RN.

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

pred(u) - The node that is the predecessor of node u in the source
tree T. Equal to NULL if node u is not reachable.

pred(j,u) - The node that is the predecessor of 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), and 0 otherwise.

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.

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 nbr_metric(I,J) over all 2-WAY links
(I,J) from i to j.

cost(u,v) - The cost for link (u,v), equal to metric(u,v) if
USE_METRICS = 1, and otherwise equal to 1.

local_if(j) - The address of the preferred local interface for
forwarding packets to neighbor j.

nbr_if(j) - The address of the preferred interface of neighbor j.

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.

8.2. TOPOLOGY UPDATE Message Format

The TOPOLOGY UPDATE message has the two formats, depending on the size of the message. The normal format is as follows, and is used whenever n, NRL, and NRNL all do not exceed 255:

```
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|M|D|0|0| TYPE  |       n       |      NRL      |     NRNL      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                       Router ID of u                          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                      Router ID of v_1                         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
~                              ...                              ~
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                      Router ID of v_n                         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|    metric 1   |    metric 2   |             ...               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                              ...                              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

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 reported non-leaf nodes, and the last n - (NRL+NRNL) of the
v_k are not reported (not in RN).

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 D bit indicates whether or not implicit deletion is used, and
must be set to 1 if and only if IMPLICIT_DELETION = 1.

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
    reported 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 reported 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 reported subtree RT.

   If n, NRL, or NRNL is larger than 255, then the long format of the
   TOPOLOGY UPDATE message is used, in which the first 4 octets of the
   normal format are replaced by the following 8 octets:

```
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |M|D|1|0|  TYPE |        0        |                n              |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |             NRL              |              NRNL              |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

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:

```
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |ST | 0 |  TYPE |    Reserved    |                n              |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |                          Router ID                            |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |                          IP Address                           |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |                          IP Address                           |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |                             ...                               |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

   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:

```
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |ST | 0 |  TYPE |    Reserved    |                n              |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |                          Router ID                            |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   | PrefixLength  | Prefix byte 1 | Prefix byte 2 |      ...       |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |      ...      | PrefixLength  | Prefix byte 1 | Prefix byte 2 |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |      ...                                                      |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

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 Routing Operation

This section describes the operation of the TBRPF routing module.
The operation is divided into the following subsections: periodic
processing, updating the source tree and topology graph, updating the
routing table, updating the reported node set, generating periodic
updates, generating differential updates, processing topology
updates, expiring topology information, optional reporting of
redundant topology information, local topology changes, generating
association messages, processing association messages, and non-relay
operation. The operation is described in terms of procedures (e.g.,
Update_All), which may be executed periodically or in response to
some event, and may be called by other procedures. In all
procedures, node i is the node executing the procedure.

8.4.1. Periodic Processing

Each node executes the procedure Update_All() periodically, at least
once every DIFF_UPDATE_INTERVAL seconds, which is typically equal to
HELLO_INTERVAL. This procedure is defined as follows:

```
Update_All()
   1. For each interface I, create empty message list msg_list(I).
   2. For each interface I, generate a HELLO message for
      interface I and add it to msg_list(I).
   3. Expire_Links().
   4. Update_Source_Tree().
   5. Update_Routing_Table().
   6. If REPORT_FULL_TREE = 0, execute Update_RN(); otherwise (the
      full source tree is reported) Update_RN_Simple().
   7. If current_time >= next_periodic:
      7.1. Generate_Periodic_Update().
      7.2. Set next_periodic = current_time + PER_UPDATE_INTERVAL.
   8. Else, Generate_Diff_Update().
   9. Generate_Association_Messages().
  10. For each interface I, send the msg_list(I) on interface I.
  11. Set old_T = T and old_RN = RN.
```

8.4.2. Updating the Source Tree and Topology Graph

   The procedure Update_Source_Tree() is a variant of Dijkstra's
   algorithm, which is called periodically and in response to topology
   changes, to update the source tree T and the topology graph TG. This
   algorithm computes shortest paths subject to two link cost penalties.
   The penalty NON_REPORT_PENALTY is added to the cost of links (u,v)
   that are not currently reported by the parent p(u) so that, whenever
   possible, a link (u,v) is included in T only if it is currently
   reported by the parent. To allow immediate rerouting when p(u)
   changes, it may be necessary to temporarily use a link (u,v) that is
   not currently reported by the new parent. The penalty
   NON_TREE_PENALTY is added to the cost of links (u,v) that are not
   currently in T, to reduce the number of changes to T. When there
   exist multiple paths of equal cost to a given node, router ID is used
   to break ties.

   The algorithm is defined as follows (where node i is the node
   executing the procedure):

```
Update_Source_Tree()
   1. For each node v in TT, set d(v) = INFINITY, pred(v) = NULL,
      old_p(v) = p(v), and p(v) = NULL.

   2. Set d(i) = 0, p(i) = i, pred(i) = i.

   3. Set S = {i}. (S is the set of labeled nodes.)

   4. For each node j in N, set d(j) = c(i,j), pred(j) = i,
      and p(j) = j.  (If USE_METRICS = 0, then all link costs
      c(i,j) are 1.)
```

    5. While there exists an unlabeled node u in TT such that
       d(u) < INFINITY:
       5.1. Let u be an unlabeled node in TT with minimum d(u).
            (A heap should be used to find u efficiently.)
       5.2. Add u to S (u becomes labeled).
       5.3. If p(u) is not equal to old_p(u) (parent has changed):
            5.3.1. For each link (u,v) in TG with tail u, if
                   reported(u,v) = 1, set reported(u,v) = 0 and set
                   nr_expire(u,v) = current_time + PER_UPDATE_INTERVAL.
            5.3.2. If p(u) is in r(u) (p(u) is reporting u):
                   5.3.2.1. Set tg_expire(u) = rt_expire(p(u),u).
                   5.3.2.2. If p(u) = u (u is a neighbor), remove all links
                            (u,v) with tail u from TG.
                   5.3.2.3. For each link (u,v) with p(u) in r(u,v):
                            5.3.2.3.1. Add (u,v) to TG and set reported(u,v) = 1.
                            5.3.2.3.2. Set metric(u,v) = metric(p(u),u,v).
                                       If USE_METRICS=1, set c(u,v)=metric(u,v).
       5.4. For each node v such that (u,v) is in TG:
            5.4.1. If reported(u,v) = 0,
                   set cost = c(u,v) + NON_REPORT_PENALTY.
                   (This penalizes (u,v) if not reported by p(u).)
            5.4.2. Else, if p(u) = u AND u is not in r(v),
                   set cost = c(u,v) + NON_REPORT_PENALTY.
                   (This penalizes (u,v) if u is a neighbor and is not
                   reporting v.)
            5.4.3. If (u,v) is not in old_T and p(u) != u,
                   set cost = cost + NON_TREE_PENALTY.
            5.4.4. If (d(u) + cost, u) is lexicographically less
                   than (d(v), pred(v)), set d(v) = d(u) + c(u,v),
                   pred(v) = u, and p(v) = p(u).

    6. Update the source tree T as follows:
       6.1. Remove all links from T.
       6.2. For each node u other than i such that pred(u) is not
            NULL, add the link (pred(u), u) to T.

8.4.3. Updating the Routing Table

   The routing table is updated following any change to the source tree
   or the association tables (interface table, host table, or network
   prefix table). The routing table is updated according to procedure
   Update_Routing_Table(), which is defined as follows:

Update_Routing_Table()

   1. Remove all tuples from the routing table.

   2. For each node u in TT (other than this node) such that p(u) is
      not NULL, add the tuple (rt_dest, rt_next, rt_dist, rt_if_id)
      to the routing table, where:
          rt_dest = u,
          rt_if_id = local_if(p(u)),
          rt_next = nbr_if(p(u)),
          rt_dist = d(u).

   3. For each tuple (if_addr, if_rid, if_expire) in the interface
      table, if a routing table entry (rt_dest, rt_next, rt_dist,
      rt_if_id) exists such that rt_dest = if_rid, add the tuple
      (if_addr, rt_next, rt_dist, rt_if_id) to the routing table.

   4. 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
      lexicographically smaller value for (rt_dist, rt_dest).

   5. 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 lexicographically smaller value
      for (rt_dist, rt_dest).

8.4.4. Updating the Reported Node Set

   Recall that the reported subtree RT is defined to be the set of links
   (u,v) in T such that u is in the reported node set RN. Each node
   updates its RN immediately before generating periodic or differential
   topology updates.

   If REPORT_FULL_TREE = 1 (so that a node reports its entire source
   tree), then RN simply consists of all reachable nodes, i.e., all
   nodes u such that pred(u) is not NULL. The procedure that computes
   RN in this manner is called Update_RN_Simple(). The rest of this
   section describes how RN is computed assuming REPORT_FULL_TREE = 0.

   A node first determines which of its neighbors belong to RN. Node i
   includes a neighbor j in RN if and only if node i determines that one
   of its neighbors may select i to be its next hop on its shortest path
   to j. To make this determination, node i computes the shortest
   paths, up to 2 hops, from each neighbor to each other neighbor, using
   only neighbors (or node i itself) as an intermediate node, and using

relay priority and router ID to break ties. If a link metric is
used, then shortest paths are computed with respect to the link
metric; otherwise min-hop paths are computed.

After a node determines which neighbors are in RN, each node u (other
than node i) in the topology table is included in RN if and only if
the next hop p(u) to u is in RN. Equivalently, node u is included in
RN if and only if u is in the subtree of T rooted at some neighbor j
that is in RN. Thus, the reported subtree RT includes the subtrees
of T that are rooted at neighbors in RN. Node i also includes itself
in RN; thus RT also includes all local links (i,j) to neighbors j.

The precise procedure for updating RN is defined as follows:

Update_RN()
  1. Set RN = empty.
  2. For each neighbor s in N such that s is in r(s), i.e.,
     such that s is reporting itself:
     (Initialize to run Dijkstra for source s, for 2 hops.)
     2.1. For each node j in N+{i}, set dist(j) = INFINITY and
          par(j) = NULL.
     2.2. Set dist(s) = 0 and par(s) = s.
     2.3. For each node j in N+{i} such that (s,j) is in TG:
        2.3.1. Set dist(j) = metric(s,j), par(j) = j.
        2.3.2. For each node k in N such that (j,k) is in TG:
              2.3.2.1. Set cost = metric(j,k).
              2.3.2.2. If (dist(j) + cost, nbr_pri(j), j)
                 is lexicographically less than
                 (dist(k), nbr_pri(par(k)), par(k)),
                 set dist(k) = dist(j) + cost and par(k) = j.
     2.4. For each neighbor j in N, add j to RN if par(j) = i.
  3. Add i to RN. (Node i is always in RN.)
  4. For each node u in the topology table, add u to RN if p(u)
     is in RN.

 In some cases it may be desirable to limit the radius (number of
 hops) that topology information is propagated. Since each TBRPF
 packet is sent only to immediate (1-hop) neighbors, this cannot be
 achieved by using a time-to-live field. Instead, the propagation of
 topology information can be limited to a radius of K hops by limiting
 RN (at all nodes) to include only nodes that are at most K-1 hops
 away. Assuming min-hop routing is used, so that d(u) is the number
 of hops to node u, this can be done by modifying Step 4 of
 Update_RN() as follows:

  4. For each node u in the topology table, add u to RN if p(u)
     is in RN and d(u) <= K-1.

8.4.5. Generating Periodic Updates

   Every PER_UPDATE_INTERVAL seconds, each node generates and transmits,
   on all interfaces, a set of FULL TOPOLOGY UPDATE messages (one
   message for each node in RN that is not a leaf of T), which describes
   the reported subtree RT. Whenever possible, these messages are
   included in a single packet, in order to minimize the number of
   control packets transmitted.

   Each topology update message contains the router IDs for n+1 nodes u,
   v_1,...,v_n, which represent the n links (u,v_1),..., (u,v_n). The n
   head nodes v_1,..., v_n are divided into three lists in order to
   convey additional information and thus reduce the number of messages
   that must be generated. In particular, the first NRL head nodes are
   leaves of T, thus avoiding the need to generate separate topology
   update messages for leaf nodes u. Similarly, the last n-(NRL+NRNL)
   head nodes are not in RN, thus avoiding the need to generate separate
   topology update messages for nodes u that have been removed from RN.

   Periodic update messages are generated according to procedure
   Generate_Periodic_Update(), defined as follows (where node i is the
   node executing the procedure):

   Generate_Periodic_Update()
     For each node u in RN (including node i) 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 interface 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 M (metrics) bit is set to 1 and
         the link metrics metric(u,v_1),..., metric(u,v_n) are
         included in the message.

8.4.6. Generating Differential Updates

   Every DIFF_UPDATE_INTERVAL seconds, if it is not time to generate a
   periodic update, and if RT has changed since the last time a topology
   update was generated, a set of TOPOLOGY UPDATE messages describing
   the changes to RT is generated and transmitted on all interfaces.
   These messages are constructed according to procedure
   Generate_Differential_Update(), defined as follows:

```
Generate_Differential_Update()
  For each node u in RN:
  1. If u is not in old_RN (u was added to RN) and 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, NRL, and NRNL are defined as above for
         periodic updates.
     (b) If USE_METRICS = 1, then the M (metrics) bit is set to 1
         and the link metrics metric(u,v_1),..., metric(u,v_n)
         are included in the message.

  2. Else, if u is in old_RN and is not a leaf of T:
     2.1. Let v_1,..., v_n be the nodes v such that (u,v) is in T
          AND at least one of the following 3 conditions holds:
              (a) (u,v) is not in old_T, or
              (b) v is in old_RN but not in RN, or
              (c) v is a leaf and is in RN but not in old_RN.
     2.2. 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 interface I, where:
          (a) NRL and NRNL are defined as above.
          (b) If USE_METRICS = 1, then the M (metrics) bit is
              set to 1 and the link metrics metric(u,v_1),...,
              metric(u,v_n) are included in the message.

  3. If u is in old_RN:
     3.1. Let v_1,..., v_n be the nodes v such that (u,v) is in
          old_T but not in TG, and either IMPLICIT_DELETION = 0
          or pred(v) is not in RN (or is NULL).
          (If IMPLICIT_DELETION = 1 and pred(v) is in RN, then
          the deletion of (u,v) is implied by an ADD update for
          another link (w,v).)
       3.2. If this set of nodes is nonempty, add the update
          (DELETE, n, u, v_1,..., v_n) to msg_list(I) for each I.
```

8.4.7. Processing Topology Updates

   When a packet containing a list (msg_list) of TOPOLOGY UPDATE
   messages is received from node j, the list is processed according to
   the procedure Process_Updates(j, msg_list), defined as follows. In
   particular, this procedure updates TT, TG, and the reporting neighbor
   lists r(u) and r(u,v). If any link in T has been deleted from TG,
   then Update_Source_Tree() and Update_Routing_Table() are called to
   provide immediate rerouting.

```
Process_Updates(j, msg_list)
  1. For each update = (subtype, n, NRL, NRNL, u, v_1,..., v_n)
     in msg_list:
```

```
      1.1. Create an entry for u in TT if it does not exist.
      1.2. If subtype = FULL, Process_Full_Update(j, update).
      1.3. If subtype = ADD, Process_Add_Update(j, update).
      1.4. If subtype = DELETE, Process_Delete_Update(j, update).
   2. If there exists any link in T that is not in TG:
      2.1. Update_Source_Tree().
      2.2. Update_Routing_Table().


Process_Full_Update(j, update)
   1. Add j to r(u).
   2. Set rt_expire(j,u) = current_time + TOP_HOLD_TIME.
   3. For each link (u,v) s.t. j is in r(u,v):
      3.1. Remove j from r(u,v).
      3.2. If pred(j,v) = u, set pred(j,v) = NULL.
   4. If j = p(u) OR p(u) = NULL:
      4.1. Set tg_expire(u) = current_time + TOP_HOLD_TIME.
      4.2. For each v s.t. (u,v) is in TG,
           If reported(u,v) = 1, remove (u,v) from TG.
   5. Process_Add_Update(j, update).


Process_Add_Update(j, update)
   For m = 1,..., n:
      ((u,v_m) is the mth link in update.)
      1. Let v = v_m.
      2. Create an entry for v in TT if it does not exist.
      3. Add j to r(u,v).
      4. If j = p(u) OR p(u) = NULL:
         4.1. Add (u,v) to TG.
         4.2. Set reported(u,v) = 1.
      5. If the M (metrics) bit in update is 1:
         5.1. Set metric(j,u,v) to the m-th metric in the update.
         5.2. If j = p(u) OR p(u) = NULL:
            5.2.1. Set metric(u,v) = metric(j,u,v).
            5.2.2. If USE_METRICS = 1, set c(u,v) = metric(u,v).
      6. If the D (implicit deletion) bit in update is 1:
         6.1. Set w = pred(j,v).
         6.2. If (w != NULL AND w != u):
            6.2.1. Remove j from r(w,v).
            6.2.2. If j = p(w), remove (w,v) from TG.
      7. Set pred(j,v) = u.   (Set new predecessor.)
      8. If m <= NRL (v = v_m is a reported leaf):
         8.1. Set leaf_update = (FULL, 0, 0, 0, v).
         8.2. Process_Full_Update(j, leaf_update).
      9. If m > NRL + NRNL (v = v_m is not reported by j):
         9.1. Remove j from r(v).
         9.2. Set rt_expire(j,v) = 0.
         9.3. For each node w s.t. j is in r(v,w),
              remove j from r(v,w).
```

```
      9.4. If j = p(v), then for each node w s.t. (v,w) is in TG
            and reported(v,w) = 1, set reported(v,w) = 0 and set
            nr_expire(v,w) = current_time + PER_UPDATE_INTERVAL.
```

```
Process_Delete_Update(j, update)
  For m = 1,..., n:
     ((u,v_m) is the mth link in update.)
     1. Let v = v_m.
     2. Remove j from r(u,v).
     3. If pred(j,v) = u, set pred(j,v) = NULL.
     4. If j = p(u), remove (u,v) from TG.
```

8.4.8. Expiring Topology Information

   Each node periodically checks for outdated topology information based
   on the expiration timers tg_expire(u), rt_expire(j,u), and
   nr_expire(u,v), and removes any expired entries from TG and from the
   lists r(u) and r(u,v). This is done according to the following
   procedure Expire_Links(), which is called periodically just before
   the source tree is updated.

```
Expire_Links()
  For each node u in TT other than node i:
     1. If tg_expire(u) < current_time, then for each v s.t.
        (u,v) is in TG, remove (u,v) from TG.
     2. Else, for each v s.t. (u,v) is in TG,
        if reported(u,v) = 0 AND nr_expire(u,v) < current_time,
        remove (u,v) from TG.
     3. For each node j in r(u), if rt_expire(j,u) < current_time:
        3.1. Remove j from r(u).
        3.2. For each link (u,v) s.t. j is in r(u,v),
             remove j from r(u,v).
```

   In addition, the following cleanup steps SHOULD be executed
   periodically to remove unnecessary entries from the topology table
   TT. A link (u,v) should be removed from TT if it is not in TG and
   not in old_T. A node u should be removed from TT if all of the
   following conditions hold: r(u) is empty, r(w,u) is empty for all w,
   and no link of TG has u as either the head or the tail.

8.4.9. Optional Reporting of Redundant Topology Information

   Each node is required to report its reported 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. For example, a node may compute any subgraph H of
   TG that contains T, and may report the "reported subgraph" RH which
   consists of links (u,v) of H such that u is in RN. In this case,

each periodic update describes RH instead of RT, and each
differential update describes changes to RH. If this option is used,
then the parameter IMPLICIT_DELETION MUST be set to 0, since the
deletion of a link cannot be implied by the addition of another link
if redundant topology information is reported.

8.4.10. Local Topology Changes

   This section describes the procedures that are followed when the
   neighbor discovery module detects a new link, the loss of a link, or
   a change in the metric for a link.

   When a link (I,J) from a local interface I to a neighbor interface J
   is discovered via the neighbor discovery module, the procedure
   Link_Up(I,J) is executed, as defined below. Letting j be the
   neighbor node associated with interface J, Link_Up(I,J) adds j to N
   (if it does not already belong), updates the preferred local
   interface local_if(j) and neighbor interface nbr_if(j) so that the
   link from local_if(j) to nbr_if(j) has the minimum metric among all
   links from i to j, and updates metric(i,j) to be this minimum metric.

Link_Up(I,J)
   1. Let j = nbr_rid(I,J).
   2. If j is not in N:
      2.1. Add j to N.
      2.2. Add (i,j) to TG.
      2.3. Set reported(i,j) = 1.
   3. If nbr_metric(I,J) < metric(i,j), set local_if(j) = I,
      nbr_if(j) = J, and metric(i,j) = nbr_metric(I,J).
   4. If USE_METRICS = 1, set cost(i,j) = metric(i,j).

   When the loss of a link (I,J) from a local interface I to a neighbor
   interface J is detected via the neighbor discovery module, the
   procedure Link_Down(I,J) is executed, as defined below. Note that
   routes are updated immediately when a link is lost, and if the lost
   link is due to a link-layer failure notification, a differential
   topology update is sent immediately.

Link_Down(I,J)
   1. Let j = nbr_rid(I,J).
   2. If there does not exist a link (K,L) from node i to
      node j with nbr_status(K,L) = 2-WAY:
      2.1. Remove j from N.
      2.2. Remove (i,j) from TG.
   3. If j is in N:
      3.1. Let (K,L) be a link from i to j such that
           nbr_metric(K,L) is the minimum metric among
           all links from i to j.

```
     3.2. Set local_if(j) = K, nbr_if(j) = L, and
          metric(i,j) = nbr_metric(K,L).
     3.3. If USE_METRICS = 1, set cost(i,j) = metric(i,j).
  5. Update_Source_Tree().
  6. Update_Routing_Table().
  7. If j is not in N and lost link is due to link-layer failure
     notification:
     7.1. If (REPORT_FULL_TREE = 0) Update_RN().
     7.2. Else, Update_RN_Simple().
     7.3. Set msg_list = empty.
     7.4. Generate_Diff_Update().
     7.5. Send msg_list on all interfaces.
     7.6. Set old_T = T and old_RN = RN.
```

   If the metric of a link (I,J) from a local interface I to a neighbor
   interface J changes via the neighbor discovery module, the following
   procedure Link_Change(I,J) is executed.

```
Link_Change(I,J)
   1. Let j = nbr_rid(I,J).
   2. Let (K,L) be a link from i to j such that
      nbr_metric(K,L) is the minimum metric among
      all links from i to j.
   3. Set local_if(j) = K, nbr_if(j) = L, and
      metric(i,j) = nbr_metric(K,L).
   4. If USE_METRICS = 1, set cost(i,j) = metric(i,j).
```

8.4.11. Generating Association Messages

   This section describes the procedures used to generate INTERFACE
   ASSOCIATION, HOST ASSOCIATION, and NETWORK PREFIX ASSOCIATION
   messages. Addresses or prefixes in the interface table, host table,
   and network prefix table are reported to neighbors periodically every
   IA_INTERVAL, HA_INTERVAL, and NPA_INTERVAL seconds, respectively. In
   addition, differential changes to the tables are reported every
   DIFF_UPDATE_INTERVAL seconds if it is not time for a periodic update
   (similar to differential topology updates). Each node reports only
   addresses or prefixes that are associated with nodes in the reported
   node set RN; this ensures the efficient broadcast of all associated
   addresses and prefixes to all nodes in the network.

   The generated messages are sent on each interface. Whenever
   possible, these messages are combined into the same packet, in order
   to minimize the number of control packets transmitted.

```
Generate_Association_Messages()
   1. Generate_Interface_Association_Messages().
   2. Generate_Host_Association_Messages().
```

     3. Generate_Network_Prefix_Association_Messages().

Generate_Interface_Association_Messages()
    1. If current_time > next_ia_time:
       1.1. Set next_ia_time = current_time + IA_INTERVAL.
       1.2. For each node u in RN:
            1.2.1. Let addr_1,..., addr_n be the interface IP
                   addresses associated with RID u in the current
                   interface table.
            1.2.2. If this list is nonempty, add the INTERFACE
                   ASSOCIATION message (FULL, n, u, addr_1,..., addr_n)
                   to msg_list(I) for each I.

    2. Else, for each node u in RN:
       2.1. 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.
       2.2. 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()
    1. If current_time > next_ha_time:
       1.1. Set next_ha_time = current_time + HA_INTERVAL.
       1.2. For each node u in RN:
            1.2.1. Let addr_1,..., addr_n be the host IP addresses
                   associated with RID u in the current host table.
            1.2.2. If this list is nonempty, add the HOST ASSOCIATION
                   message (FULL, n, u, addr_1,..., addr_n) to
                   msg_list(I) for each I.

    2. Else, for each node u in RN:
       2.1. 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.
       2.2. 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()
    1. If current_time > next_npa_time:
       1.1. Set next_npa_time = current_time + NPA_INTERVAL.
       1.2. For each node u in RN:
            1.2.1. 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.
            1.2.2. 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.

    2. Else, for each node u in RN:
       2.1. 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.

       2.1. 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.
```

8.4.12. Processing Association Messages

   When an INTERFACE ASSOCIATION, HOST ASSOCIATION, or NETWORK PREFIX
   ASSOCIATION message is received from node j, the interface table,
   host table, or network prefix table, respectively, is updated as
   described in the following three procedures.

```
Process_Interface_Association_Messages(j, msg_list)
  For each message (subtype, n, u, addr_1,..., addr_n) in msg_list
  such that j = p(u):
     1. If subtype = FULL, remove all entries with if_rid = u
        from the interface table.
     2. If subtype = FULL or ADD, then 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.
     3. If subtype = DELETE, then 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)
  For each message (subtype, n, u, addr_1,..., addr_n) in msg_list
  such that j = p(u):
      1. If subtype = FULL, remove all entries with h_rid = u
         from the host table.
      2. If subtype = FULL or ADD, then 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.
      3. If subtype = DELETE, then 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)
  For each message (subtype, n, u, length_1, prefix_1, ...,
  length_n, prefix_n) in msg_list such that j = p(u):
      1. If subtype = FULL, remove all entries with net_rid = u
         from the prefix table.
      2. If subtype = FULL or ADD, then 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.
      3. If subtype = DELETE, then 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.
```

8.4.13. Non-Relay Operation

   Nodes with relay priority equal to zero are called non-relay nodes,
   and do not forward packets (of any type) that are received from other
   nodes. A non-relay node is implemented simply by not generating or
   transmitting any TOPOLOGY UPDATE messages. A non-relay node may
   report (in association messages) addresses or prefixes that are
   associated with itself, but not those associated with other nodes.
   HELLO messages must be transmitted in order to establish links with
   neighbor nodes. The following procedures can be omitted in non-relay
   nodes: Update_RN(), Generate_Periodic_Update(), and
   Generate_Diff_Update().

8.5. Configurable Parameters

   This section lists the configurable parameters used by the routing
   module, and their proposed default values. All nodes MUST have the
   same value for all of the following parameters except
   REPORT_FULL_TREE and IMPLICIT_DELETION.

       Parameter Name   Default Value
       --------------   -------------
       DIFF_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 * IA_INTERVAL
       HA_INTERVAL   10 seconds
       HA_HOLD_TIME   3 * HA_INTERVAL
       NPA_INTERVAL   10 seconds
       NPA_HOLD_TIME   3 * NPA_INTERVAL
       USE_METRICS   0
       REPORT_FULL_TREE   0
       IMPLICIT_DELETION   1

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. This mechanism can be considered an
   optimization of the classical flooding algorithm in which each packet
   is transmitted by every node of the network. In TBRPF flooding,
   information provided by TBRPF is used to decide whether a given
   received flooded packet should be forwarded. As a result, each
   packet is transmitted by only a relatively small subset of nodes,
   thus consuming much less bandwidth than classical flooding.

   This document specifies that the flooding mechanism use the IPv4
   multicast address 224.0.1.20 (currently assigned by IANA for "any
   private experiment"). Every node maintains a duplicate cache to keep
   track of which flooded packets have already been received. The
   duplicate cache contains, for each received flooded packet, the
   flooded packet identifier (FPI), which for IPv4 is composed of the
   source IP address, the IP identification, and the fragment offset
   values obtained from the IP header [14].

   When a node receives a packet whose destination IP address is the
   flooding address (224.0.1.20), 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 reported 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 a MANET interface (see the
Terminology section), then the packet should not be retransmitted on
that interface.

10. Operation of TBRPF in Mobile Ad-Hoc Networks

   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. Although
   applicable across a much broader field of use, TBRPF is particularly
   well suited for supporting the standard DARPA Internet protocols
   [3][2]. In the following sections, we discuss practical
   considerations for the operation of TBRPF 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 the MANET's scope. While
   such uniqueness is not strictly guaranteed, the assumption of
   uniqueness is consistent with current practices for deployment of the
   Internet protocols on specific link layers. Methods for duplicate
   link layer address detection and deconfliction are beyond the scope
   of this document.

10.2. Network Layer Assumptions

   MANETs are formed as collections of routers 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
   (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 Router ID (RID) for use in TBRPF protocol messages,
whether or 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 TBRPF neighbor
   discovery messages contain both the data link and network layer
   address of the sender, implementations MAY perform 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
   [15] or IPv6 Neighbor Discovery [16]. 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 [17] home agent. 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 advertise network prefixes which the router
   discovered via attached networks, external routes advertised by other
   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

   TBRPF packets are communicated using UDP/IP. Port 712 has been
   assigned by IANA for exclusive use by TBRPF. Implementations in
   private networks MAY employ alternate data delivery services (i.e.,
   raw IP or local data-link encapsulation). The selection of an
   alternate data delivery service MUST be consistent among all MANET
   routers in the private network. In all implementations, the data
   delivery service MUST provide a checksum facility.

   The following sections specify the operation of TBRPF over UDP/IP.

10.7.1. IPv4 Operation

   When IPv4 is used, TBRPF nodes obey IPv4 host and router requirements
   [4][5]. TBRPF packets are sent to the multicast address 224.0.0.2
   (All Routers) and thus reach all TBRPF routers within single-hop
   transmission range of the sender. TBRPF routers MUST NOT forward
   packets sent to this multicast address.

   Since non-negligible packet loss due to link failure, interference,
   etc. can occur, implementations SHOULD avoid IPv4 fragmentation/
   reassembly whenever possible, by splitting large TBRPF protocol
   packets into multiple smaller packets at the application layer. When
   fragmentation is unavoidable, senders SHOULD NOT send TBRPF packets
   that exceed the minimum reassembly buffer size ([4], section 3.3.2)
   for all receivers in the network.

10.7.2. IPv6 Operation

   The specification of TBRPF for IPv6 is the same as for IPv4, except
   that 32-bit IPv4 addresses are replaced by 128-bit IPv6 addresses.
   However, to minimize overhead, router IDs remain at 32 bits, similar
   to OSPF for IPv6 [18].

11. IANA Considerations

   The IANA has assigned port number 712 for TBRPF.

   The TBRPF flooding mechanism specified in this document uses the IPv4
   multicast address 224.0.1.20, which is currently assigned by IANA for
   "any private experiment". In the event that this specification is
   advanced to standards track, a new multicast address assignment would
   be requested for this purpose.

12. Security Considerations

   Wireless networks are vulnerable to a variety of attacks, including
   denial-of-service attacks (e.g., flooding and jamming), man-in-the-
   middle attacks (e.g., interception, insertion, deletion,
   modification, replaying) and service theft. To counter such attacks,
   it is important to prevent the spoofing (impersonation) of TBRPF
   nodes, and to prevent unauthorized nodes from joining the network via
   neighbor discovery. To achieve this, TBRPF packets can be
   authenticated using the IP Authentication Header [19][20]. In
   addition, the Encapsulating Security Payload (ESP) header [21] can be
   used to provide confidentiality (encryption) of TBRPF packets.

   The IETF SEcuring Neighbor Discovery (SEND) Working Group analyzes
   trust models and threats for ad hoc networks [22]. TBRPF can be
   extended in a straightforward manner to use SEND mechanisms, e.g.,
   [23].

13. Acknowledgements

   The authors would like to thank the Army Systems Engineering Office
   (ASEO) for funding part of this work.

   The authors would like to thank several members of the MANET working
   group for many helpful comments and suggestions, including Thomas
   Clausen, Philippe Jacquet, and Joe Macker.

   The authors would like to thank Bhargav Bellur for major
   contributions to the original (full-topology) version of TBRPF,
   Ambatipudi Sastry for his support and advice, and Julie S. Wong for
   developing a new implementation of TBRPF and suggesting several
   clarifications to the TBRPF Routing Operation section.

14. References

14.1. Normative References

   [1] Bradner, S., "Key words for use in RFCs to Indicate Requirement
        Levels", BCP 14, RFC 2119, March 1997.

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

   [3] Postel, J., "Internet Protocol", STD 5, RFC 791, September 1981.

   [4] Braden, R., Ed., "Requirements for Internet Hosts -
        Communication Layers", STD 3, RFC 1122, October 1989.

   [5]   Baker, F., Ed., "Requirements for IP Version 4 Routers", RFC
         1812, June 1995.

14.2.   Informative References

   [6]   Moy, J., "OSPF Version 2", STD 54, RFC 2328, April 1998.

   [7]   Ogier, R., Message in IETF email archive for MANET,
         ftp://ftp.ietf.org/ietf-mail-archive/manet/2002-02.mail,
         February 2002.

   [8]   Ogier, R., "Topology Dissemination Based on Reverse-Path
         Forwarding (TBRPF): Correctness and Simulation Evaluation",
         Technical Report, SRI International, October 2003.

   [9]   Ogier, R., Message in IETF email archive for MANET,
         ftp://ftp.ietf.org/ietf-mail-archive/manet/2002-03.mail, March
         2002.

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

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

   [12] Clausen, T. and P. Jacquet, Eds., "Optimized Link State Routing
         Protocol (OLSR)", RFC 3626, October 2003.

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

   [14] Perkins, C., Belding-Royer, E. and S. Das, "IP Flooding in Ad
         Hoc Mobile Networks", Work in Progress, November 2001.

   [15] Plummer, D., "Ethernet Address Resolution Protocol: Or
         converting network protocol addresses to 48.bit Ethernet address
         for transmission on Ethernet hardware", STD 37, RFC 826,
         November 1982.

   [16] Narten, T., Nordmark, E. and W. Simpson, "Neighbor Discovery for
         IP Version 6 (IPv6)", RFC 2461, December 1998.

   [17] Perkins, C., Ed., "IP Mobility Support for IPv4", RFC 3344,
         August 2002.

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

   [19] Kent, S. and R. Atkinson, "Security Architecture for the
        Internet Protocol", RFC 2401, November 1998.

   [20] Kent, S. and R. Atkinson, "IP Authentication Header", RFC 2402,
        November 1998.

   [21] Kent, S. and R. Atkinson, "IP Encapsulating Security Payload
        (ESP)", RFC 2406, November 1998.

   [22] Nikander, P., "IPv6 Neighbor Discovery Trust Models and
        Threats", Work in Progress, April 2003.

   [23] Arkko, J., "SEcure Neighbor Discovery (SEND)", Work in Progress,
        June 2003.

Authors' Addresses

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

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

   Fred L. Templin
   Nokia
   313 Fairchild Drive
   Mountain View, CA 94043
   USA

   Phone: +1 650 625 2331
   Fax: +1 650 625 2502
   EMail: ftemplin@iprg.nokia.com

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

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

Intellectual Property

   The IETF takes no position regarding the validity or scope of any
   Intellectual Property Rights or other rights that might be claimed
   to pertain to the implementation or use of the technology
   described in this document or the extent to which any license
   under such rights might or might not be available; nor does it
   represent that it has made any independent effort to identify any
   such rights. Information on the procedures with respect to
   rights in RFC documents can be found in BCP 78 and BCP 79.

   Copies of IPR disclosures made to the IETF Secretariat and any
   assurances of licenses to be made available, or the result of an
   attempt made to obtain a general license or permission for the use
   of such proprietary rights by implementers or users of this
   specification can be obtained from the IETF on-line IPR repository
   at http://www.ietf.org/ipr.

   The IETF invites any interested party to bring to its attention
   any copyrights, patents or patent applications, or other
   proprietary rights that may cover technology that may be required
   to implement this standard. Please address the information to the
   IETF at ietf-ipr@ietf.org.

Acknowledgement