Internet Engineering Task Force **PIM WG**
INTERNET-DRAFT
draft-ietf-pim-sm-v2-new-00.ps

Bill Fenner/AT&T
Mark Handley/ACIRI
Hugh Holbrook/Cisco
Isidor Kouvelas/Cisco

13 July 2000
Expires: January 2001

Protocol Independent Multicast - Sparse Mode (PIM-SM): Protocol Specification (Revised)

Status of this Document

This document is an Internet-Draft and is in full conformance with all provisions of Section 10 of RFC2026.

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

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

The list of current Internet-Drafts can be accessed at http://www.ietf.org/ietf/1id-abstracts.txt

The list of Internet-Draft Shadow Directories can be accessed at http://www.ietf.org/shadow.html.

This document is a product of the IETF PIM WG. Comments should be addressed to the authors, or the WG's mailing list at pim@catarina.usc.edu.

Abstract

This document specifies Protocol Independent Multicast - Sparse Mode (PIM-SM). PIM-SM is a multicast routing protocol that can use the underlying unicast routing information base or a separate multicast-capable routing information base. It builds unidirectional shared trees rooted at a Rendezvous Point (RP) per group, and optionally creates shortest-path trees per source.

Note on PIM-SM status

PIM-SM v2 is currently widely implemented and deployed, but the existing specification in RFC 2362 is insufficient to implement from, and is incorrect in a number of aspects. This document is a complete re-write from RFC 2362, and is intended to obsolete RFC 2362. The authors have attempted to document current practice as far as possible, but a number of cases have arisen where current practice is clearly incorrect, typically leading to traffic being black-holed. In these cases we diverge from current practice, but always in a way that will interoperate successfully with the legacy PIM v2 implementations that we are aware of.

1. Introduction

This document specifies a protocol for efficiently routing multicast groups that may span wide-area (and inter-domain) internets. This protocol is called Protocol Independent Multicast - Sparse Mode (PIM-SM) because, although it may use the underlying unicast routing to provide reverse-path information for multicast tree building, it is not dependent on any particular unicast routing protocol.

PIM-SM version 2 was originally specified in RFC 2117, and revised in RFC 2362. This document is intended to obsolete RFC 2362, and to correct a number of deficiencies that have been identified with the way PIM-SM was previously specified. As far as possible, this document specifies the same protocol as RFC 2362, and only diverges from the behavior intended by RFC 2362 when the previously specified behavior was clearly incorrect. Routers implemented according to the specification in this document will be able to successfully interoperate with routers implemented according to RFC 2362.

2. Terminology

In this document, the key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" are to be interpreted as described in RFC 2119 and indicate requirement levels for compliant PIM-SM implementations.

2.1. Definitions

This specification uses a number of terms to refer to the roles of routers participating in PIM-SM. The following terms have special significance for PIM-SM:

Rendezvous Point (RP):
: An RP is a router that has been configured to be used as the root of the non-source-specific distribution tree for a multicast group. Join messages from receivers for a group are sent towards the RP, and data from senders is sent to the RP so that receivers can discover who the senders are, and start to receive traffic destined for the group.

Designated Router (DR):
: A shared-media LAN like Ethernet may have multiple PIM-SM routers connected to it. If the LAN has directly connected hosts, then a single one of these routers, the DR, will act on behalf of those hosts with respect to the PIM-SM protocol. A single DR is elected per LAN using a simple election process.

MRIB
: Multicast Routing Information Base. This is the routing table, typically created using MBGP, that is used to make decisions regarding where to forward Join/Prune messages.

RPF Neighbor
: RPF stands for "Reverse Path Forwarding". The RPF Neighbor of a router with respect to an address is the neighbor that the MRIB indicates should be used to forward packets to that address. In the case of a PIM-SM multicast group, the RPF neighbor is the router that a Join message for that group would be directed to, in the absence of modifying Assert state.

TIB Tree Information Base. This is the collection of state at a PIM router that has been created by receiving PIM Join/Prune messages, PIM Assert messages, and IGMP information from local hosts. It essentially stores the state of all multicast distribution trees at that router.

MFIB Multicast Forwarding Information Base. The TIB holds all the state that is necessary to forward multicast packets at a router. However, although this specification defines

forwarding in terms of the TIB, to actually forward packets using the TIB is very inefficient. Instead a real router implementation will normally build an efficient MFIB from the TIB state to perform forwarding. How this is done is implementation-specific, and is not discussed in this document.

2.2. Pseudocode Notation

We use set notation in several places in this specification.

`A (+) B`
: is the union of two sets A and B.

`A (-) B`
: is the elements of set A that are not in set B.

`NULL`
: is the empty set or list.

In addition we use C-like syntax:

= denotes assignment of a variable.

== denotes a comparison for equality.

!= denotes a comparison for inequality.

Braces { and } are used for grouping.

3. PIM-SM Protocol Overview

This section provides an overview of PIM-SM behavior. It is intended as an introduction to how PIM-SM works, and is not definitive. For the definitive specification, see Section 4.

PIM relies on an underlying toppology-gathering protocol to populate a routing table with routes. This routing table is called the MRIB or Multicast Routing Information Base. The routes in this table may be taken directly from the unicast routing table, or it may be different and provided by a separate routing protocol such as MBGP [1]. In any event, the routes in the MRIB must represent a multicast-capable path to each subnet. The MRIB is used to determine the path that PIM control messages such as Join messages take to get to the source subnet, and data flows along the reverse path of the Join messages. Thus, in contrast to the unicast RIB where the routes give a path that data packets take to get *to* each subnet, the MRIB gives reverse-path information, and indicates the path that data packets would take *from* each subnet to the router that has the MRIB.

Like all multicast routing protocols that implement the service model from RFC 1112 [2], PIM-SM must be able to route data packets from sources to receivers without either the sources or receivers knowing a-priori of the existence of the others. This is essentially done in three phases, although as senders and receivers may join and leave at any time, all three phases may be occur simultaneously.

Phase One: RP Tree

In phase one, a multicast receiver expresses its interest in receiving traffic destined for a multicast group. Typically it does this using IGMP [3], but other mechanisms might also serve this purpose. One of the receiver's local routers is elected as the Designated Router (DR) for that subnet. On receiving the receiver's expression of interest, the DR then sends a PIM Join message towards the RP for that multicast group. This Join message is known as a (*,G) Join because it joins group G for all sources to that group. The (*,G) Join travels hop-by-hop towards the RP for the group, and in each router it passes through, multicast tree state for group G is instantiated.

Eventually the (*,G) Join either reaches the RP, or reaches a router that already has (*,G) Join state for that group. When many receivers join the group, their Join messages converge on the RP, and form a distribution tree for group G that is rooted at the RP. This is known as the RP Tree (RPT), and is also known as the shared tree because it is shared by all sources sending to that group. Join messages are resent periodically so long as the receiver remains in the group. After they stop being sent, the state will eventually time out.

A multicast data sender just starts sending data destined for a multicast group. The sender's local router (DR) takes those data packets, unicast-encapsulates them, and sends them directly to the RP. The RP receives these encapsulated data packets, decapsulates them, and forwards them onto the shared tree. The packets then follow the (*,G) multicast tree state in the routers on the RP Tree, being replicated wherever the RP Tree branches, and eventually reaching all the receivers for that multicast group. The process of encapsulating data packets to the RP is called registering, and the encapsulation packets are known as PIM Register packets.

At the end of phase one, multicast traffic is flowing encapsulated to the RP, and then natively over the RP tree to the multicast receivers.

Phase Two: Register Stop

Register-encapsulation of data packets is inefficient for two reasons:

- Encapsulation and decapsulation may be relatively expensive operations for a router to perform, depending on whether or not the router has appropriate hardware for these tasks.
- Traveling all the way to the RP, and then back down the shared tree may entail the packets traveling a relatively long distance to reach receivers that are close to the sender. For some applications, this increased latency is undesirable.

Although Register-encapsulation may continue indefinitely, for these reasons, the RP will normally choose to switch to native forwarding. To do this, when the RP receives a data packet from source S on group G, it will normally initiate an (S,G) source-specific Join towards S. This join message travels hop-by-hop towards S, instantiating (S,G) multicast tree state in the routers along the path. (S,G) multicast tree state is used only to forward packets for group G if those packets come from source S. Eventually the Join message reaches S's subnet or a router that already has (S,G) multicast tree state, and then packets from S start to flow following the (S,G) tree state towards the RP. These data packets may also reach routers with (*,G) state along the path towards the RP - if so, they can short-cut onto the RP tree at this point.

While the RP is in the process of joining the source-specific tree for S, the data packets will continue being encapsulated to the RP. When packets from S also start to arrive natively at the the RP, the RP will be receiving two copies of each of these packets. At this point, the RP starts to discard the encapsulated copy of these packets, and it sends a Register-Stop message back to S's DR to prevent the DR unnecessarily encapsulating the packets.

At the end of phase 2, traffic will be flowing natively from S along a source-specific tree to the RP, and from there along the shared tree to the receivers. Where the two trees intersect, traffic may transfer from the source-specific tree to the RP tree, and so avoid taking a long detour via the RP.

It should be noted that a sender may start sending before or after a receiver joins the group, and thus phase two may happen before the shared tree to the receiver is built.

Phase 3: Shortest-Path Tree

Although having the RP join back towards the source removes the encapsulation overhead, it does not completely optimize the forwarding paths. For many receivers the route via the RP may involve a significant detour when compared with the shortest path from the source to the receiver.

To obtain lower latencies, a receiver's DR may optionally initiate a transfer from the shared tree to a source-specific shortest-path tree (SPT). To do this, it issues an (S,G) Join towards S. This instantiates state in the routers along the path to S. Eventually this join either reaches S's subnet, or reaches a router that already has (S,G) state. When this happens, data packets from S start to flow following the (S,G) state until they reach the receiver.

At this point the receiver (or a router upstream of the receiver) will be receiving two copies of the data - one from the SPT and one from the RPT. When the first traffic starts to arrive from the SPT, the DR or upstream router starts to drop the packets for G from S that arrive via the RP tree. In addition, it sends an (S,G) prune message towards the RP. This is known as an (S,G,rpt) Prune. The prune message travels hop-by-hop, instantiating state along the path towards the RP indicating that traffic from S for G should NOT be forwarded in this direction. The prune is propagated until it reaches the RP or a router that still needs the traffic from S for other receivers.

By now, the receiver will be receiving traffic from S along the shortest-path tree between the receiver and S. In addition, the RP is receiving the traffic from S, but this traffic is no longer reaching the receiver along the RP tree. As far as the receiver is concerned, this is the final distribution tree.

Source-specific Joins

IGMPv3 permits a receiver to join a group and specify that it only wants to receive traffic for a group if that traffic comes from a particular source. If a receiver does this, and no other receiver on the LAN requires all the traffic for the group, then the DR may omit performing a (*,G) join to set up the shared tree, and instead issue a source-specific (S,G) join only.

The range of multicast addresses from 232.0.0.0 to 232.255.255.255 has been set aside for source-specific multicast. For groups in this range, receivers should only issue source-specific IGMPv3 joins. If a PIM router receives a non-source-specific join for a group in this range, it should ignore it.

Source-specific Prunes

IGMPv3 also permits a receiver to join a group and specify that it only wants to receive traffic for a group if that traffic *does not* come from a specific source or sources. In this case, the DR will perform a (*,G) join as normal, but may combine this with an (S,G,rpt) prune for each of the sources the receiver does not wish to receive.

Multi-access Transit LANs

The overview so far has concerned itself with point-to-point links. However, using multi-access LANs such as Ethernet for transit is not uncommon. This can cause complications for three reasons:

- Two or more routers on the LAN may issue (*,G) Joins to different upstream routers on the LAN because they have inconsistent MRIB entries regarding how to reach the RP. Both paths on the RP tree will be set up, causing two copies of all the shared tree traffic to appear on the LAN.

- Two or more routers on the LAN may issue (S,G) Joins to different upstream routers on the LAN because they have inconsistent MRIB entries regarding how to reach source S. Both paths on the source-specific tree will be set up, causing two copies of all the traffic from S to appear on the LAN.
- A router on the LAN may issue a (*,G) Join to one upstream router on the LAN, and another router on the LAN may issue an (S,G) Join to a different upstream router on the same LAN. Traffic from S may reach the LAN over both the RPT and the SPT. If the receiver behind the downstream (*,G) router doesn't issue an (S,G,rpt) prune, then this condition would persist.

All of these problems are caused by there being more than one upstream router with join state for the group or source-group pair. PIM does not prevent such duplicate joins from occurring - instead when duplicate data packets appear on the LAN from different routers, these routers notice this, and then elect a single forwarder. This election is performed using PIM Assert messages, which resolve the problem in favor of the upstream router which has (S,G) state, or if neither or both router has (S,G) state, then in favor of the router with the best metric to the RP for RP trees, or the best metric to the source to source-specific trees.

These Assert messages are also received by the downstream routers on the LAN, and these cause subsequent join messages to be sent to the upstream router that won the Assert.

RP Discovery

PIM-SM routers need to know the address of the RP for each group for which they have (*,G) state. This address is obtained through a bootstrap mechanism.

One router in each PIM domain is elected the Bootstrap Router (BSR) through a simple election process. All the routers in the domain that are configured to be candidates to be RPs periodically unicast their candidacy to the BSR. From the candidates, the BSR picks an RP-set, and periodically announces this set in a bootstrap message. Bootstrap messages are flooded hop-by-hop throughout the domain until all routers in the domain know the RP-Set.

To map a group to an RP, a router hashes the group address into the RP-set using an order-preserving hash function (one that minimizes changes if the RP set changes). The resulting RP is the one that it uses as the RP for that group.

4. Protocol Specification

The specification of PIM-SM is broken into several parts:

- Section 4.1 details the protocol state stored.
- Section 4.2 specifies the data packet forwarding rules.
- Section 4.3 specifies the PIM Register generation and processing rules.
- Section 4.4 specifies the PIM Join/Prune generation and processing rules.
- Section 4.5 specifies the PIM Assert generation and processing rules.
- Designated Router (DR) election is specified in Section 4.6.
- Section 4.7 specifies the Bootstrap and RP discovery mechanisms.
- PIM packet formats are specified in Section 4.8.
- A summary of PIM-SM timers and their default values is given in Section 4.9.

4.1. PIM Protocol State

This section specifies all the protocol state that a PIM implementation should maintain in order to function correctly. We term this state the *Tree Information Base* or TIB, as it holds the state of all

the multicast distribution trees at this router. In this specification we define PIM mechanisms in terms of the TIB. However, only a very simple router implementation would actually implement packet forwarding operations in terms of this state. Most real router implementations will use this state to build a multicast forwarding table, which would then be updated when the relevant state in the TIB changes.

Although we specify precisely the state to be kept, this does not mean that an implementation of PIM-SM needs to hold the state in this form. This is actually an abstract state definition, which is needed in order to specify the router's behavior. A PIM-SM implementation is free to hold whatever internal state it requires, and will still be conformant with this specification so long as it results in the same externally visible protocol behavior as an abstract router that holds the following state.

We divide TIB state into three sections:

(*,G) state
: State that maintains the RP tree for G.

(S,G) state
: State that maintains a source-specific tree for source S and group G.

(S,G,rpt) state
: State that maintains source-specific information about source S on the RP tree for G. For example, if a source is being received on the source-specific tree, it will normally have been pruned off the RP tree. This prune state is (S,G,rpt) state.

The state that should be kept is described below. Of course, implementations will only maintain state when it is relevant to forwarding operations - for example, the "NoInfo" state might be assumed from the lack of other state information, rather than being held explicitly.

4.1.1. General Purpose State

A router holds the following non-group-specific state:

Bootstrap State:

- Bootstrap Router's IP Address
- BSR Priority
- Bootstrap Timer (BST)

RP Set

For each interface:

Neighbor State:

For each neighbor:

- Information from neighbor's Hello
- Neighbor's Gen ID.
- Neighbor liveness timer (NLT)

Designated Router (DR) State:

- Designated Router's IP Address
- DR's DR Priority

Bootstrap state is described in section 4.7, the RP Set is described in section 4.7.5, and Designated Router state is described in section 4.6.

4.1.2. (*,G) State

For every group G a router keeps the following state:

(*,G) state:

For each interface:

Local Membership:

State: One of {"NoInfo", "Include"}

PIM (*,G) Join/Prune State:

- State: One of {"NoInfo" (NI), "Join" (J), "PrunePending" (PP)}
- Prune Pending Timer (PPT)
- Join/Prune Expiry Timer (ET)

(*,G) Assert Winner State

- State: One of {"NoInfo" (NI), "I lost Assert" (L), "I won Assert" (W)}
- Assert Timer (AT)
- Assert winner's IP Address
- Assert winner's Assert Metric

Not interface specific:

- Upstream Join/Prune Timer (JT)
- Last RP Used
- Last RPF Neighbor towards RP that was used

Local membership is the result of the local membership mechanism (such as IGMP) running on that interface. It need not be kept if this router is not the DR on that interface unless this router won a (*,G) assert on this interface for this group, although implementations may optionally keep this state in case they become the DR or assert winner. This information is used by the `pim_include(*,G)` macro described in section 4.1.5.

PIM (*,G) Join/Prune state is the result of receiving PIM (*,G) Join/Prune messages on this interface, and is specified in section 4.4.1. The state is used by the macros that calculate the outgoing interface list in section 4.1.5, and in the `JoinDesired(*,G)` macro (defined in section 4.4.4) that is used in deciding whether a Join(*,G) should be sent upstream.

(*,G) Assert Winner state is the result of sending or receiving (*,G) assert messages on this interface. It is specified in section 4.5.2.

The upstream (*,G) Join/Prune timer is used to send out periodic Join(*,G) messages, and to override Prune(*,G) messages from peers on an upstream LAN interface.

The last RP used must be stored because if the RP Set changes (section 4.7.5) then state must be torn down and rebuilt for groups whose RP changes.

The last RPF neighbor towards the RP is stored because if the MRIB changes then the RPF neighbor towards the RP may change. If it does so, then we need to trigger a new Join(*,G) to the new upstream neighbor and a Prune(*,G) to the old upstream neighbor. Similarly, if a router detects through a changed GenID in a Hello message that the upstream neighbor towards the RP has rebooted, then it should re-instantiate state by sending a Join(*,G). These mechanisms are specified in Section 4.4.4.

4.1.3. (S,G) State

For every source/group pair (S,G) a router keeps the following state:

(S,G) state:

For each interface:

Local Membership:

State: One of {"NoInfo", "Include"}

PIM (S,G) Join/Prune State:

- State: One of {"NoInfo" (NI), "Join" (J), "PrunePending" (PP)}
- Prune Pending Timer (PPT)
- Join/Prune Expiry Timer (ET)

(S,G) Assert Winner State

- State: One of {"NoInfo" (NI), "I lost Assert" (L), "I won Assert" (W)}
- Assert Timer (AT)
- Assert winner's IP Address
- Assert winner's Assert Metric

Not interface specific:

- Upstream (S,G) Join/Prune Timer (JT)
- Last RPF Neighbor towards S that was used
- SPT bit (indicates (S,G) state is active)
- (S,G) KeepAlive Timer (KAT)

Local membership is the result of the local source-specific membership mechanism (such as IGMP version 3) running on that interface and specifying that this particular source should be included. As stored here, this state is the resulting state after any IGMPv3 inconsistencies have been resolved. It need not be kept on point-to-point links. It also need not be kept if this router is not the DR on that interface unless this router won a (S,G) assert on this interface for this group. This information is used by the `pim_include(S,G)` macro described in section 4.1.5.

PIM (S,G) Join/Prune state is the result of receiving PIM (S,G) Join/Prune messages on this interface, and is specified in section 4.4.1. The state is used by the macros that calculate the outgoing interface list in section 4.1.5, and in the `JoinDesired(S,G)` macro (defined in section 4.4.5) that is used in deciding whether a Join(S,G) should be sent upstream.

(S,G) Assert Winner state is the result of sending or receiving (S,G) assert messages on this interface. It is specified in section 4.5.1.

The upstream (S,G) Join/Prune timer is used to send out periodic Join(S,G) messages, and to override Prune(S,G) messages from peers on an upstream LAN interface.

The last RPF neighbor towards S is stored because if the MRIB changes then the RPF neighbor towards S may change. If it does so, then we need to trigger a new Join(S,G) to the new upstream neighbor and a Prune(S,G) to the old upstream neighbor. Similarly, if the router detects through a changed GenID in a Hello message that the upstream neighbor towards S has rebooted, then it should re-instantiate state by sending a Join(S,G). These mechanisms are specified in Section 4.4.5.

The SPTbit is used to indicate whether forwarding is taking place on the (S,G) Shortest Path Tree (SPT) or on the (*,G) tree. A router can have (S,G) state and still be forwarding on (*,G) state during the interval when the source-specific tree is being constructed. When SPTbit is FALSE, only (*,G) forwarding state is used to forward packets from S to G. When SPTbit is TRUE, both (*,G) and (S,G) forwarding state are used.

The (S,G) Keepalive Timer is updated by data being forwarded using this (S,G) forwarding state. It is used to keep (S,G) state alive in the absence of explicit (S,G) Joins. Amongst other things, this is necessary for the so-called "turnaround rules" - when the RP uses (S,G) joins to stop encapsulation, and then (S,G) prunes to prevent traffic from unnecessarily reaching the RP.

4.1.4. (S,G,rpt) State

For every source/group pair (S,G) for which a router also has (*,G) state, it also keeps the following state:

(S,G,rpt) state:

For each interface:

Local Membership:

State: One of {"NoInfo", "Exclude"}

PIM (S,G,rpt) Join/Prune State:

- State: One of {"NoInfo", "Pruned", "PrunePending"}
- Prune Pending Timer (PPT)
- Join/Prune Expiry Timer (ET)

Not interface specific:

Upstream (S,G,rpt) Join/Prune State:

- State: One of {"NotJoined(*,G)", "NotPruned(S,G,rpt)", "Pruned(S,G,rpt)"}
- Override Timer (OT)

Local membership is the result of the local source-specific membership mechanism (such as IGMPv3) running on that interface and specifying that although there is (*,G) Include state, this particular source should be excluded. As stored here, this state is the resulting state after any IGMPv3 inconsistencies between LAN members have been resolved. It need not be kept on point-to-point links. It also need not be kept if this router is not the DR on that interface unless this router won a (*,G) assert on this interface for this group. This information is used by the `pim_exclude(S,G)` macro described in section 4.1.5.

PIM (S,G,rpt) Join/Prune state is the result of receiving PIM (S,G,rpt) Join/Prune messages on this interface, and is specified in section 4.4.3. The state is used by the macros that calculate the outgoing interface list in section 4.1.5, and in the rules for adding Prune(S,G,rpt) messages to Join(*,G) messages specified in section 4.4.6.

The upstream (S,G,rpt) Join/Prune state is used along with the Override Timer to send the correct override messages in response to Join/Prune messages sent by upstream peers on a LAN. This state and behavior are specified in section 4.4.7.

4.1.5. State Summarization Macros

Using this state, we define the following "macro" definitions which we will use in the descriptions of the state machines and pseudocode in the following sections.

The most important macros are those that define the outgoing interface list (or "olist") for the relevant state. An olist can be "immediate" if it is built directly from the state of the relevant type. For example, the immediate_olist(S,G) is the olist that would be built if the router only had (S,G) state and no (*,G) or (S,G,rpt) state. In contrast, the "inherited" olist inherits state from other types. For example, the inherited_olist(S,G) is the olist that is relevant for forwarding a packet from S to G using both source-specific and group-specific state.

There is no immediate_olist(S,G,rpt) as (S,G,rpt) state is negative state - it removes interfaces from the olist. The inherited_olist(S,G,rpt) is therefore the olist that would be used for a packet from S to G forwarding on the RP tree. It is a strict subset of immediate_olist(*,G).

Generally speaking, the inherited olists are used for forwarding, and the immediate_olists are used to make decisions about state maintenance.

```
immediate_olist(*,G) =
    joins(*,G) (+) pim_include(*,G) (-) assert(*,G)

immediate_olist(S,G) =
    joins(S,G) (+) pim_include(S,G) (-) assert(S,G)

inherited_olist(*,G) =
    immediate_olist(*,G)

inherited_olist(S,G,rpt) =
        ( joins(*,G) (-) prunes(S,G,rpt) )
    (+) ( pim_include(*,G) (-) pim_exclude(S,G))
    (-) ( assert(*,G) (+) assert(S,G,rpt) )

inherited_olist(S,G) =
    inherited_olist(S,G,rpt) (+) immediate_olist(S,G)
```

The macros pim_include(*,G) and pim_include(S,G) indicate the interfaces to which traffic might be forwarded because of hosts that are local members on that interface. Note that normally only the DR cares about local membership, but when an assert happens, the assert winner takes over responsibility for forwarding traffic to local members that have requested traffic on a group or source/group pair.

```
pim_include(*,G) =
   { all interfaces I such that:
     ( ( I_am_DR( I ) AND assert(*,G,I) == NULL )
       OR AssertWinner(*,G,I) == me )
     AND  igmp_desired(*,G,I) }

pim_include(S,G) =
    { all interfaces I such that:
      ( (I_am_DR( I ) AND assert(S,G,I) == NULL )
        OR AssertWinner(S,G,I) == me )
       AND  igmp_desired(S,G,I) }
```

The clause "igmp_desired(S,G,I)" is true if the IGMP module has determined that there are local members on interface I that desire to receive traffic sent specifically by S to G. "igmp_desired(*,G,I)" is true if the IGMP module has determined that there are local members on interface I that desire to receive all traffic sent to G.

The set "joins(*,G)" is the set of all interfaces on which the router has received (*,G) Joins:

```
joins(*,G) =
    { all interfaces I such that
      DownstreamState(*,G,I) is either Joined or PrunePending }
```

The set "joins(S,G)" is the set of all interfaces on which the router has received (S,G) Joins:

```
joins(S,G) =
    { all interfaces I such that
      DownstreamState(S,G,I) is either Joined or PrunePending }
```

The set "prunes(S,G,rpt)" is the set of all interfaces on which the router has received (*,G) joins and (S,G,rpt) prunes. The macro prune(S,G,rpt,I) is defined in Section 4.4.3.

```
prunes(S,G,rpt) =
    { all interfaces I such that
      prune(S,G,rpt,I) == TRUE }
```

The set "assert(*,G)" is the set of all interfaces on which the router has received (*,G) joins but has lost a (*,G) assert. The macro assert(*,G,I) is defined in Section 4.5.3.

```
assert(*,G) =
    { all interfaces I such that
      assert(*,G,I) == TRUE }
```

The set "assert(S,G,rpt)" is the set of all interfaces on which the router has received (*,G) joins but has lost an (S,G) assert. The macro assert(S,G,rpt,I) is defined in Section 4.5.3.

```
assert(S,G,rpt) =
    { all interfaces I such that
      assert(S,G,rpt,I) == TRUE }
```

The set "assert(S,G)" is the set of all interfaces on which the router has received (S,G) joins but has lost an (S,G) assert. The macro assert(S,G,I) is defined in Section 4.5.3.

```
assert(S,G) =
   { all interfaces I such that
    assert(S,G,I) == TRUE }
```

The following pseudocode macro definitions are also used in many places in the specification. Basically RPF' is the RPF neighbor towards an RP or source unless a PIM-Assert has overridden the normal choice of neighbor.

```
  neighbor RPF'(*,G) {
      if ( I_Am_Assert_Loser(*,G,RPF_interface(RP(G))) ) {
           return AssertWinner(*, G, RPF_interface(RP(G)) )
      } else {
           return mrib.next_hop( RP(G) )
      }
  }

  neighbor RPF'(S,G,rpt) {
      if( I_Am_Assert_Loser(S, G, RPF_interface(RP(G)) ) ) {
           return AssertWinner(S, G, RPF_interface(RP(G)) )
      } else {
           return RPF'(*,G)
      }
  }
```

```
neighbor RPF'(S,G) {
    if ( I_Am_Assert_loser(S, G, RPF_interface(S) )) {
         return AssertWinner(S, G, RPF_interface(S) )
    } else {
         return mrib.next_hop( S )
    }
}
```

RPF'(*,G) and RPF'(S,G) indicate the neighbor from which data packets should be coming and to which joins should be sent on the RP tree and SPT respectively.

RPF'(S,G,rpt) is basically RPF'(*,G) modified by the result of an Assert(S,G) on RPF_interface(RP(G)). In such a case, packets from S will be originating from a different router than RPF'(*,G), and this router may be a different router (or on a different interface) from RPF'(S,G). If we only have active (*,G) Join state, we need to accept packets from RPF'(S,G,rpt), and add a Prune(S,G,rpt) to the periodic Join(*,G) messages that we send to RPF'(*,G) (See Section 4.4.6).

The function `mrib.next_hop( S )` returns the next-hop PIM neighbor toward the host S, as indicated by the current MRIB. If S is directly adjacent, then mrib.next_hop(S) returns S itself.

4.2. Data Packet Forwarding Rules

The PIM-SM packet forwarding rules are defined below in pseudocode.

`iif` is the incoming interface of the packet.
S is the source address of the packet.
G is the destination address of the packet (group address).
RP is the address of the Rendezvous Point for this group.
`RPF_interface(S)` is the interface the MRIB indicates would be used to route packets to S.
`RPF_interface(RP)` is the interface the MRIB indicates would be used to route packets to RP, except at the RP when it is the decapsulation interface (the "virtual" interface on which register packets are received).

First, we restart (or start) the Keepalive timer if the source is on a directly connected subnet.

Second, we check to see if the SPT bit should be set because we've now switched from the RP tree to the SPT.

Next we check to see whether the packet arrived on an interface that should be forwarded, either using (S,G) or (*,G) state.

If the packet should be forwarded using (S,G) state, we then build an outgoing interface list for the packet. If this list is not empty, then we refresh the (S,G) state keepalive timer.

If the packet should be forwarded using (*,G) state, then we just build an outgoing interface list for the packet.

Finally we remove the incoming interface from the outgoing interface list we've created, and if the resulting outgoing interface list is not empty, we forward the packet out of those interfaces.

```
if( DirectlyConnected(S) == TRUE ) {
      restart KeepaliveTimer(S,G)
      #  Note: register state transition may happen as a result
      #  of restarting KeepaliveTimer, and must be dealt with here.
}

Update_SPTbit(S,G,iif)
# See section 4.2.1

if( iif == RPF_interface(S) AND UpstreamState(S,G) == Joined ) {

   oiflist = inherited_olist(S,G)

   if( oiflist != NULL ) {
       restart KeepaliveTimer(S,G)
   }

} else if( iif == RPF_interface(RP) AND SPTbit(S,G) == FALSE) {

  oiflist = inherited_olist(S,G,rpt)

} else {
   #  Note: RPF check failed
   if ( SPTbit(S,G) == TRUE AND inherited_olist(S,G) != NULL ) {
      send Assert(S,G) on iif
   } else if ( SPTbit(S,G) == FALSE AND
               iif is an element of inherited_olist(S,G,rpt) {
      send Assert(*,G) on iif
   }
}

oiflist = oiflist (-) iif
forward packet from all interfaces in oiflist
```

This pseudocode employs several "macro" definitions:

`directly_connected(S)` is TRUE if the source S is on any subnet that is directly connected to this router (or for packets originating on this router).

`inherited_olist(S,G)` and `inherited_olist(S,G,rpt)` are defined in Section 4.1.

Basically `inherited_olist(S,G)` is the outgoing interface list for packets forwarded on (S,G) state taking into account (*,G) state, asserts, etc.

`inherited_olist(S,G,rpt)` is the outgoing interface for packets forwarded on (*,G) state taking into account (S,G) prune state on the RP tree, and asserts, etc.

In the state-machine, when KeepaliveTimer(S,G) is restarted, it is set to Keepalive_Period (see Section 4.9).

Data triggered PIM-Assert messages sent from the above forwarding code should be rate-limited in a implementation-dependent manner.

4.2.1. Setting and Clearing the (S,G) SPT bit

The (S,G) SPTbit is used to distinguish whether to forward on (*,G) or on (S,G) state. When switching from the RP tree to the source tree, there is a transition period when data is arriving due to upstream (*,G) state while upstream (S,G) state is being established during which time a router should continue to forward only on (*,G) state. This prevents temporary black-holes that would be caused by sending a Prune(S,G,rpt) before the upstream (S,G) state has finished being established.

Thus, when a packet arrives, the (S,G) SPTbit is updated as follows:

```
void
Update_SPTbit(S,G,iif) {
  if ( iif == RPF_interface(S)
        AND JoinDesired(S,G) == TRUE
        AND ( DirectlyConnected(S) == TRUE
              OR RPF_interface(S) != RPF_interface(RP)
              OR inherited_olist(S,G,rpt) == NULL
              OR RPF'(S,G) == RPF'(*,G) ) ) {
     Set SPTbit(S,G) to TRUE
  } else if ( JoinDesired(S,G)==FALSE ) {
     Set SPTbit(S,G) to FALSE
  }
}
```

Additionally a router sets SPTbit(S,G) to TRUE it sees an Assert(S,G) on RPF_interface(S).

`JoinDesired(S,G)` is defined in Section 4.4.5, and indicates whether we have the appropriate (S,G) Join state to wish to send a Join(S,G) upstream.

Basically Update_SPTbit will set the SPT bit if we have the appropiate (S,G) join state and the packet arrived on the correct incoming interface to have come from S, and one or more of the following conditions applies:

- The source is directly connected, and so the switch to the SPT is a no-op.
- The RPF interface to S is different from the RPF interface to the RP. The packet arrived on RPF_interface(S), and so the SPT must have been completed.
- No-one wants the packet on the RP tree, so we're not going to confuse any downstream routers into thinking the SPT has been completed.
- RPF'(S,G) == RPF'(*,G). In this case the router will never be able to tell if the SPT has been completed, so it should just switch immediately.

In the case where the RPF interface is the same for the RP and for S, but RPF'(S,G) and RPF'(*,G) differ, then we wait for an Assert(S,G) which indicates that the upstream router with (S,G) state believes the SPT has been completed.

4.3. PIM Register Messages

Overview

The Designated Router (DR) on a LAN or point-to-point link encapsulates multicast packets from local sources to the RP for the relevant group unless it recently received a Register Stop message for that (S,G) or (*,G) from the RP. When the DR receives a Register Stop message from the RP, it starts a Register Stop timer to maintain this state. Just before the Register Stop timer expires, the DR sends a Null-Register Message to the RP to allow the RP to refresh the Register Stop

information at the DR. If the Register Stop timer actually expires, the DR will resume encapsulating packets to the source.

4.3.1. Sending Register Messages from the DR

Every PIM-SM router has the capability to be a DR. The state machine below is used to implement Register functionality. For the purposes of specification, we represent the mechanism to encapsulate packets to the RP as a Register-Tunnel interface, which is added to or removed from the (S,G) olist. The tunnel interface then takes part in the normal packet forwarding rules specified in Section 4.2.

If register state is maintained, it is maintained only for directly connected sources, and is per-(S,G). There are four states in the DR's per-(S,G) Register state-machine:

Join (J)
The register tunnel is "joined" (the join is actually implicit, but the DR acts as if the RP has joined the DR on the tunnel interface).

Prune (P)
The register tunnel is "pruned" (this occurs when a Register Stop is received).

Join Pending (JP)
The register tunnel is pruned but the DR is contemplating adding it back.

No Info (NI)
No information. This is the initial state, and the state when the router is not the DR.

In addition, a RegisterStop timer (RST) is kept if the state machine in not in the No Info state.

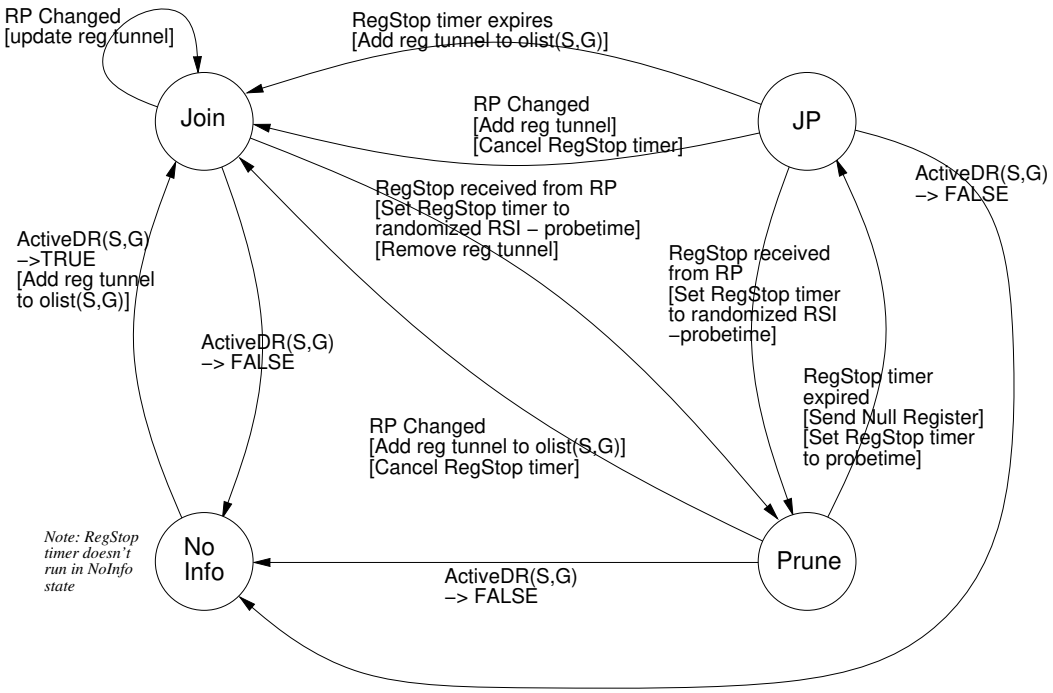


Figure 1: Per-(S,G) register state-machine at a DR

In tabular form, the state-machine is:

PrevState	RegisterStop Timer expires	ActiveDR->True	ActiveDR->False	RegisterStop received	RP changed
No Info (NI)	-	**-> J state** add tunnel	-	-	-
Join (J)	-	-	**-> NI state** remove tunnel	**-> P state** remove tunnel; set RST(*)	**-> J state** redirect tunnel to new RP; stop RST
JP	**-> J state** add tunnel	-	**-> NI state** remove tunnel	**-> P state** set RS timer(*)	**-> J state** redirect tunnel; stop RST
Prune (P)	**-> JP state** set RS timer(**); send null register	-	**-> NI state** remove tunnel	-	**-> J state** redirect tunnel; stop RS timer

Notes:

(*) The RegisterStopTimer is set to a random value chosen uniformly from the interval (0.5 * Register_Suppression_Time, 1.5 * Register_Suppression_Time) minus Register_Probe_Time;

Subtracting off register_probe_time is a bit unnecessary because it is really small compared to register suppression timeout, but was in the old spec and is kept for compatibility.

(**) The RegisterStopTimer is set to register_probe_time.

The macro "ActiveDR" in the state machine is defined as:

```
Bool ActiveDR(S,G) {
   return ( I_am_DR( RPF_interface(S) ) AND
            KeepaliveTimer(S,G) is running AND
            DirectlyConnected(S) == TRUE )
}
```

Note that on reception of a packet at the DR from a directly connected source, `KeepaliveTimer(S,G)` needs to be set by the packet forwarding rules *before* computing `ActiveDR(S,G)` in the register state machine, or the first packet from a source won't be registered.

Handling RegisterStop(*,G) Messages at the DR

An RP MAY send a RegisterStop message with the source address set to all-zeros. This was the normal course of action in RFC 2326 when the register message matched against (*,G) state at the RP, and was defined as meaning "stop encapsulating all sources for this group". However, the behavior of such a RegisterStop(*,G) is ambiguous or incorrect in some circumstances.

We specify that an RP should not send RegisterStop(*,G) messages, but for compatibility, a DR should be able to accept one if it is received.

A RegisterStop(*,G) should be treated as a RegisterStop(S,G) for all existing (S,G) Register state machines. A router should not apply a RegisterStop(*,G) to sources that become active after the

RegisterStop(*,G) was received.

4.3.2. Receiving Register Messages at the RP

When an RP receives a Register message, the course of action is decided according to the following pseudocode:

```
packet_arrives_on_rp_tunnel( pkt ) {
    if( outer.dst is not one of my addresses )
        drop the packet silently.
        # note that this should not happen if the lower layer is working
    }
    if( I_am_RP( G ) && outer.dst == RP(G) ) {
        restart KeepaliveTimer(S,G)
        if(( inherited_olist(S,G) == NULL ) OR SPTbit(S,G)) {
             send RegisterStop(S,G) to outer.src
        } else {
             decapsulate and pass the inner packet to the normal
             forwarding path for forwarding on the (*,g) tree.
        }
    } else {
        send RegisterStop(S,G) to outer.src
        # Note (*)
    }
}
```

`outer.dst` is the IP destination address of the encapsulating header.

`outer.src` is the IP source address of the encapsulating header, i.e., the DR's address.

`I_am_RP(G)` is true if the group-to-RP mapping indicates that this router is the RP for the group.

Note (*): This may block traffic from S for Register_Suppression_Time if the DR learned about a new group-to-RP mapping before the RP did. However, this doesn't matter unless we figure out some way for the RP to also accept (*,G) joins when it doesn't yet realize that it is about to become the RP for G. This will all get sorted out once the RP learns the new group-to-rp mapping. We decided to do nothing about this and just accept the fact that PIM may suffer interrupted (*,G) connectivity following an RP change.

KeepaliveTimer(S,G) is restarted at the RP when packets arrive on the proper tunnel interface. This may cause the upstream (S,G) state machine to trigger a join if the inherited_olist(S,G) is not NULL;

An RP should preserve (S,G) state that was created in response to a Register message for at least (3/2 * Register_Suppression_Time), otherwise the RP may stop joining (S,G) before the DR for S has restarted sending registers. Traffic would then be interrupted until the Register-Stop timer expires at the DR.

Thus KeepaliveTimer(S,G) should be restarted to (1.5 * Register_Suppression_Time + Register_Probe_Time).

4.3.3. RP Joining to the Source

An RP will normally send a Join(S,G) immediately it receives a valid Register(S,G) from S's DR. However, it may optionally decide to continue to receive traffic from S via register encapsulation. Such a decision should normally be consistent within a domain as otherwise the RP cannot tell if the encapsulation overhead at the DR is tolerable.

Join(S,G) messages originated in response to register messages should be rate-limited in an implementation-dependent manner.

4.4. PIM Join/Prune Messages

4.4.1. Receiving (*,G) Join/Prune Messages

When a router receives a Join(*,G) or Prune(*,G) it must first check to see whether the RP in the message matches RP(G) (the router's idea of who the RP is). If the RP in the message does not match RP(G) the Join or Prune should be silently dropped. If a router has no RP information (e.g. has not recently received a BSR message) then it may choose to accept Join(*,G) or Prune(*,G) and treat the RP in the message as RP(G).

The state machine for receiving (*,G) Join/Prune Messages is given below. There are three states:

NoInfo (NI)
: The interface has no (*,G) Join state and no timers running.

Join (J)
: The interface has (*,G) Join state which will cause us to forward packets destined for G from this interface except if there is also (S,G,rpt) prune information (see Section 4.4.3) or the router lost an assert on this interface.

PrunePending (PP)
: The router has received a Prune(*,G) on this interface from a downstream neighbor and is waiting to see whether the prune will be overridden by another downstream router. For forwarding purposes, the PrunePending state functions exactly like the Join state.

In addition there are two timers:

ExpiryTimer (ET)
: This timer is restarted when a valid Join(*,G) is received. Expiry of the ExpiryTimer causes the interface state to revert to NoInfo for this group.

PrunePendingTimer (PPT)
: This timer is set when a valid Prune(*,G) is received. Expiry of the PrunePendingTimer causes the interface state to revert to NoInfo for this group.

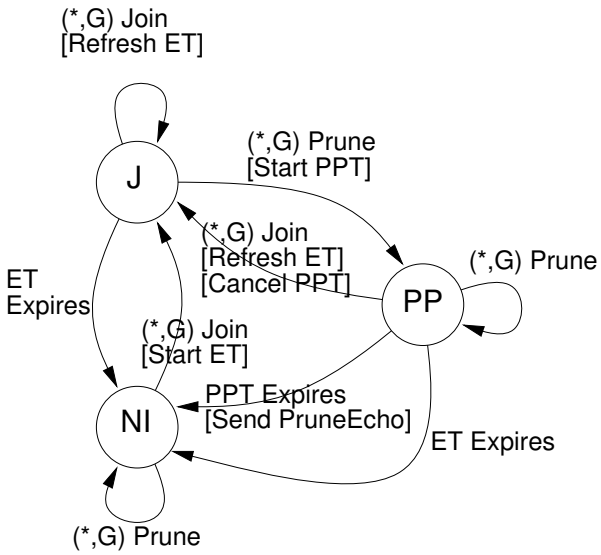


Figure 2: Downstream (*,G) per-interface state-machine

In tabular form, the state machine is:

Prev State	Receive Join(*,G)	Receive Prune(*,G)	PrunePending Timer Expires	Expiry Expires
NoInfo (NI)	**-> J state** start ExpiryTimer	**-> NI state**	**-**	**-**
Join (J)	**-> J state** restart ExpiryTimer	**-> PP state** start PrunePendingTimer	**-**	**-> NI state**
PrunePending (PP)	**-> J state** restart ExpiryTimer stop PrunePendingTimer	**-> PP state**	**-> NI state** Send PruneEcho(*,G)	**-> NI state**

The transition events "Receive Join(*,G)" and "Receive Prune(*,G)" imply receiving a Join or Prune targeted to this router's address on the received interface. If the destination address is not correct, these state transitions in this state machine must not occur, although seeing such a packet may cause state transitions in other state machines.

On unnumbered interfaces on point-to-point links, the router's address should be the same as the source address it chose for the hello packet it sent over that interface. However on point-to-point links we also recommend that PIM messages with a 0.0.0.0 destination address are also accepted.

When ExpiryTimer is started or restarted, it is set to the HoldTime from the triggering Join/Prune message.

When PrunePendingTimer is started, it is set to the J/P_Override_Interval if the router has more than one neighbor on that interface; otherwise it is set to zero causing it to expire immediately.

The action "Send PruneEcho(*,G)" is triggered when the router stops forwarding on an interface as a result of a prune. A PruneEcho(*,G) is simply a Prune(*,G) message sent by the upstream router to itself on a LAN. Its purpose is to add additional reliability so that if a Prune that should have been overridden by another router is lost locally on the LAN, then the PruneEcho may be

received and cause the override to happen. A PruneEcho(*,G) need not be sent on a point-to-point interface.

4.4.2. Receiving (S,G) Join/Prune Messages

The state machine for receiving (S,G) Join/Prune messages is given below, and is almost identical to that for (*,G) messages. There are three states:

NoInfo (NI)
The interface has no (S,G) Join state and no (S,G) timers running.

Join (J)
The interface has (S,G) Join state which will cause us to forward packets from S destined for G from this interface if the (S,G) state is active (the SPTbit is set) except if the router lost an assert on this interface.

PrunePending (PP)
The router has received a Prune(S,G) on this interface from a downstream neighbor and is waiting to see whether the prune will be overridden by another downstream router. For forwarding purposes, the PrunePending state functions exactly like the Join state.

In addition there are two timers:

ExpiryTimer (ET)
This timer is set when a valid Join(S,G) is received. Expiry of the ExpiryTimer causes the interface state to revert to NoInfo for this group.

PrunePendingTimer (PPT)
This timer is set when a valid Prune(S,G) is received. Expiry of the PrunePendingTimer causes the interface state to revert to NoInfo for this group.

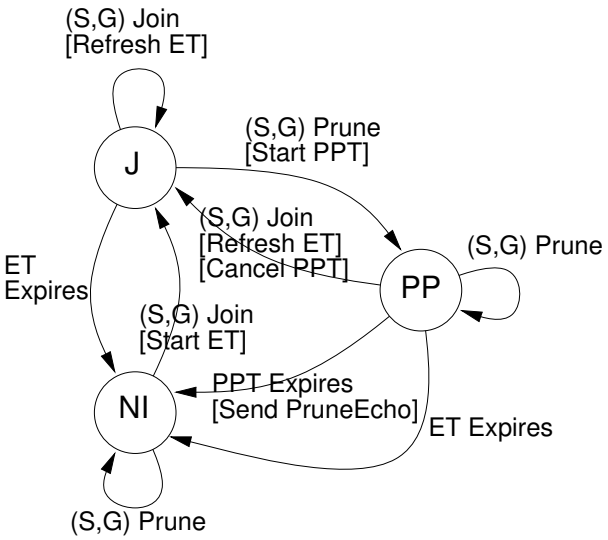


Figure 3: Downstream (S,G) state-machine

In tabular form, the state machine is:

Prev State	Receive Join(S,G)	Receive Prune(S,G)	PrunePending Timer Expires	Expiry Expires
NoInfo (NI)	**-> J state** start ExpiryTimer	**-> NI state**	**-**	**-**
Join (J)	**-> J state** restart ExpiryTimer	**-> PP state** start PrunePendingTimer	**-**	**-> NI state**
PrunePending (PP)	**-> J state** restart ExpiryTimer stop PrunePendingTimer	**-> PP state**	**-> NI state** Send PruneEcho(S,G)	**-> NI state**

The transition events "Receive Join(S,G)" and "Receive Prune(S,G)" imply receiving a Join or Prune targeted to this router's address on the received interface. If the destination address is not correct, these state transitions in this state machine must not occur, although seeing such a packet may cause state transitions in other state machines.

On unnumbered interfaces on point-to-point links, the router's address should be the same as the source address it chose for the hello packet it sent over that interface. However on point-to-point links we also recommend that messages with a 0.0.0.0 destination address are also accepted.

When ExpiryTimer is started or restarted, it is set to the HoldTime from the triggering Join/Prune message.

When PrunePendingTimer is started, it is set to the J/P_Override_Interval if the router has more than one neighbor on that interface; otherwise it is set to zero causing it to expire immediately.

The action "Send PruneEcho(S,G)" is triggered when the router stops forwarding on an interface as a result of a prune. A PruneEcho(S,G) is simply a Prune(S,G) message sent by the upstream router to itself on a LAN. Its purpose is to add additional reliability so that if a Prune that should have been overridden by another router is lost locally on the LAN, then the PruneEcho may be received and cause the override to happen. A PruneEcho(S,G) need not be sent on a point-to-point interface.

4.4.3. Receiving (S,G,rpt) Join/Prune Messages

The state machine for receiving (S,G,rpt) Join/Prune messages is given below. There are five states:

NoInfo (NI)
: The interface has no (S,G,rpt) Prune state and no (S,G,rpt) timers running.

Prune (P)
: The interface has (S,G,rpt) Prune state which will cause us not to forward packets from S destined for G from this interface even though the interface has active (*,G) Join state. When interface I is in this state, the macro `prune(S,G,rpt,I)` returns true.

PrunePending (PP)
: The router has received a Prune(S,G,rpt) on this interface from a downstream neighbor and is waiting to see whether the prune will be overridden by another downstream router. For forwarding purposes, the PrunePending state functions exactly like the NoInfo state.

PruneTmp (P')
: This state is a transient state which for forwarding purposes behaves exactly like the

Prune state. A (*,G) Join has been received (which may cancel the (S,G,rpt) Prune). As we parse the Join/Prune message from top to bottom, we first enter this state if the message contains a (*,G) Join. Later in the message we will normally encounter an (S,G,rpt) prune to re-instate the Prune state. However if we reach the end of the message without encountering such a (S,G,rpt) prune, then we will revert to NoInfo state in this state machine.

As no time is spent in this state, no timers can expire.

PrunePendingTmp (PP')

This state is a transient state which is identical to P' except that it is associated with the PP state rather than the P state. For forwarding purposes, PP' behaves exactly like PP state.

In addition there are two timers:

ExpiryTimer (ET)

This timer is set when a valid Prune(S,G,rpt) is received. Expiry of the ExpiryTimer causes this state machine to revert to NoInfo state.

PrunePendingTimer (PPT)

This timer is set when a valid Prune(S,G,rpt) is received. Expiry of the PrunePendingTimer causes this state machine to move on to Prune state.

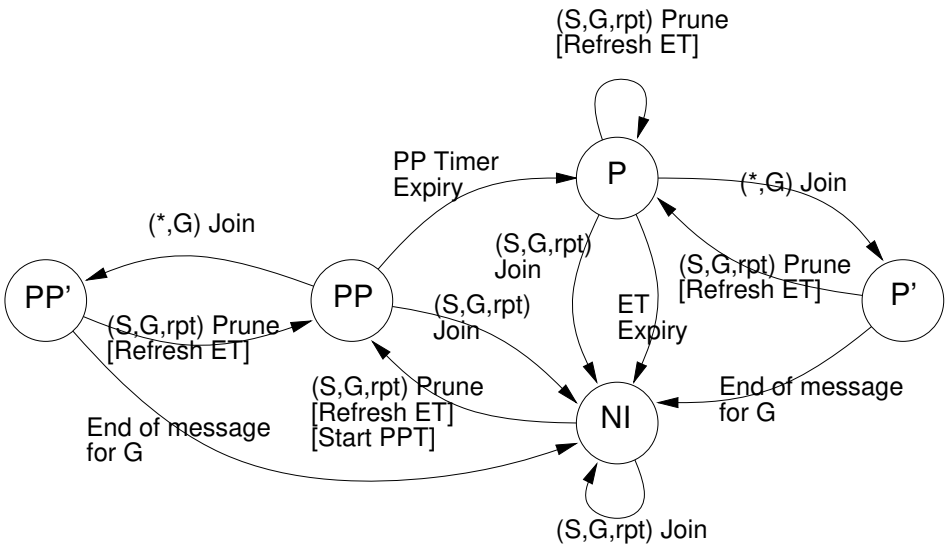


Figure 4: Downstream (S,G,rpt) state-machine

In tabular form, the state machine is:

Prev State	Receive Join(*,G)	Receive Join(S,G,rpt)	Receive Prune(S,G,rpt)	End Of Message	PPT Expires	ET Expires
NI	**-**	**-**	**-> PP state** start PPT start ET	**-**	n/a	n/a
P	**P'**	**-> NI state**	**-> P state** restart ET	**-**	n/a	**-> NI state**
PP	**PP'**	**-> NI state**	**-**	**-**	**-> P state**	n/a
P'	**error**	**error**	**-> P state** restart ET	**-> NI state**	n/a	n/a
PP'	**error**	**error**	**-> PP state** restart ET	**-> NI state**	n/a	n/a

The transition events "Receive Join(S,G,rpt)","Receive Prune(S,G,rpt)" and "Receive Join(*,G)" imply receiving a Join or Prune targeted to this router's address on the received interface. If the destination address is not correct, these state transitions in this state machine must not occur, although seeing such a packet may cause state transitions in other state machines.

On unnumbered interfaces on point-to-point links, the router's address should be the same as the source address it chose for the hello packet it sent over that interface. However on point-to-point links we also recommend that messages with a 0.0.0.0 destination address are also accepted.

Receiving a Prune(*,G) does not affect the (S,G,rpt) state machine.

When ExpiryTimer is started or restarted, it is set to the HoldTime from the Join/Prune message.

When PrunePendingTimer is started, it is set to the J/P_Override_Interval if the router has more than one neighbor on that interface; otherwise it is set to zero causing it to expire immediately.

The HoldTime from the Join/Prune message must be larger than the J/P_Override_Interval.

4.4.4. Sending (*,G) Join/Prune Messages

The per-interface state-machines for (*,G) hold join state from downstream PIM routers. This state then determines whether a router needs to propagate a Join(*,G) upstream towards the RP.

If a router wishes to propagate a Join(*,G) upstream, it must also watch for messages on it's upstream interface from other routers on that subnet, and these may modify its behavior. If it sees a Join(*,G) to the correct upstream neighbor, it should suppress its own Join(*,G). If it sees a Prune(*,G) to the correct upstream neighbor, it should be prepared to override that prune by sending a Join(*,G) almost immediately. Finally, if it sees the Generation ID (see Section 4.6) of the correct upstream neighbor change, it knows that the upstream neighbor has lost state, and it should be prepared to refresh the state by sending a Join(*,G) almost immediately.

In addition if the MRIB changes to indicate that the next hop towards the RP has changed, the router should prune off from the old next hop, and join towards the new next hop.

The upstream (*,G) state-machine only contains two states:

Not Joined

The downstream state-machines indicate that the router does not need to join the RP tree for this group.

Joined

The downstream state-machines indicate that the router would like to join the RP tree for this group.

In addition, one timer JT(*,G) is kept which is used to trigger the sending of a Join(*,G) to the upstream next hop towards the RP, RPF'(RP).

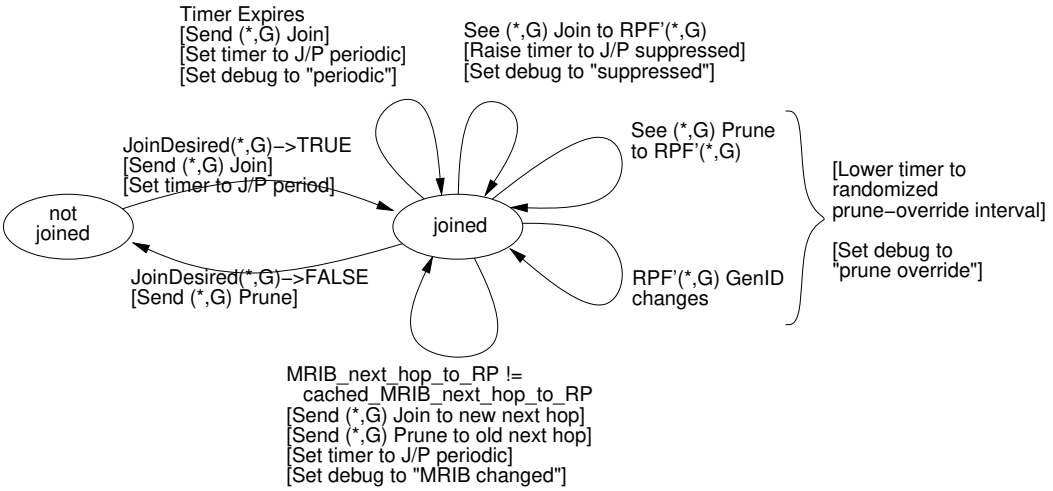


Figure 5: Upstream (*,G) state-machine

In tabular form, the state machine is:

Prev State	JoinDesired (*,G)->True	JoinDesired (*,G)->False
NotJoined	**-> J state** Send Join(*,G) Set Timer to t_periodic	**-**
Joined	**-**	**-> NJ state** Send Prune(*,G)

In addition, we have the following transitions which occur within the Joined state:

Prev State	Timer Expires	See Join(*,G) to RPF'(*,G)	See Prune(*,G) to RPF'(*,G)	topology change wrt MRIB.next_hop(RP)	RPF'(*,G) GenID changes
Joined	Send Join(*,G) Set Timer to t_periodic	Increase Timer to t_suppressed	Decrease Timer to t_override	Send Join(*,G) to new next hop Send Prune(*,G) to old next hop Set Timer to t_periodic	Decrease Timer to t_override

This state machine uses the following macro:

```
bool JoinDesired(*,G) {
   if inherited_olist(*,G) != NULL
       return TRUE
   else
       return FALSE
}
```

4.4.5. Sending (S,G) Join/Prune Messages

The per-interface state-machines for (S,G) hold join state from downstream PIM routers. This state then determines whether a router needs to propagate a Join(S,G) upstream towards the source.

If a router wishes to propagate a Join(S,G) upstream, it must also watch for messages on its upstream interface from other routers on that subnet, and these may modify its behavior. If it sees a Join(S,G) to the correct upstream neighbor, it should suppress its own Join(S,G). If it sees a Prune(S,G), Prune(S,G,rpt), or Prune(*,G) to the correct upstream neighbor towards S, it should be prepared to override that prune by scheduling a Join(S,G) to be sent (almost) immediately. Finally, if it sees the Generation ID of its upstream neighbor change, it knows that the upstream neighbor has lost state, and it should refresh the state by scheduling a Join(S,G) to be sent (almost) immediately.

In addition if MRIB changes cause the next hop towards the source to change, the router should send a prune to the old next hop, and a join to the new next hop.

The upstream (S,G) state-machine only contains two states:

Not Joined

The downstream state machines and IGMP information do not indicate that the router needs to join the shortest-path tree for this (S,G).

Joined

The downstream state machines and IGMP information indicate that the router should join the shortest-path tree for this (S,G).

In addition, one timer JT(S,G) is kept which is used to trigger the sending of a Join(S,G) to the upstream next hop toward S, RPF'(S).

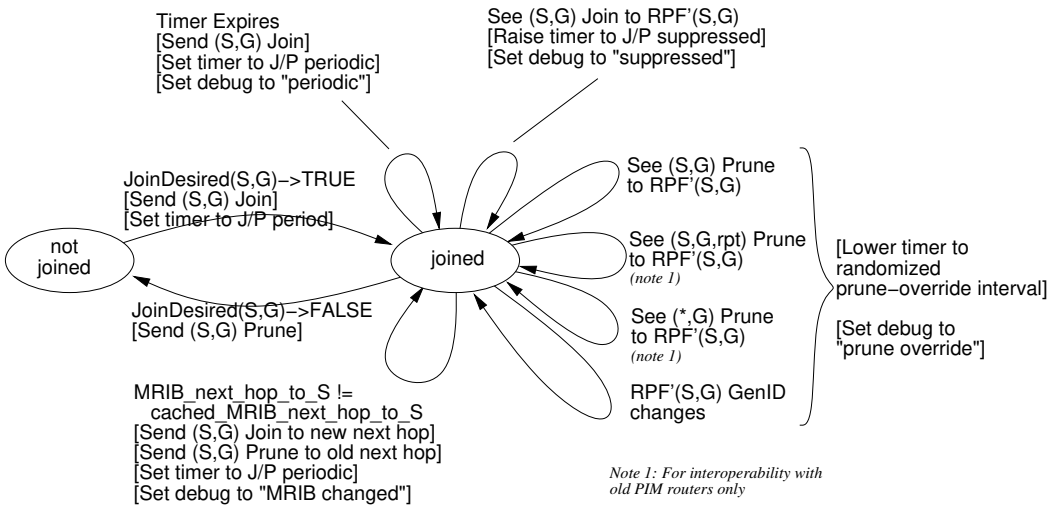


Figure 6: Upstream (S,G) state-machine

In tabular form, the state machine is:

Prev State	JoinDesired (S,G)->True	JoinDesired (S,G)->False
NotJoined	**-> J state** Send Join(S,G) Set Timer to t_periodic	**-**
Joined	**-**	**-> NJ state** Send Prune(S,G)

In addition, we have the following transitions which occur within the Joined state:

Prev State	Timer Expires	See Join(S,G) to RPF'(S,G)	See Prune(S,G) to RPF'(S,G)	See Prune(S,G,rpt) to RPF'(S,G)	See (*,G) Prune to RPF'(S,G)
Joined	Send Join(S,G) Set Timer to t_periodic	Increase Timer to t_suppressed	Decrease Timer to t_override	Decrease Timer to t_override	Decrease Timer to t_override

Prev State	topology change wrt MRIB.next_hop(S)	RPF'(S,G) GenID changes
Joined	Send Join(S,G) to new next hop Send Prune(S,G) to old next hop Set Timer to t_periodic	Decrease Timer to t_override

This state machine uses the following macro:

```
bool JoinDesired(S,G) {
    return( immediate_olist(S,G) != NULL
            OR ( KeepaliveTimer(S,G) is running
                 AND inherited_olist(S,G) != NULL ) )
}
```

4.4.6. (S,G,rpt) Periodic Messages

(S,G,rpt) Joins and Prunes are (S,G) Joins or Prunes sent on the RP tree with the RPT bit set, either to modify the results of (*,G) Joins, or to override the behavior of other upstream LAN peers. The next section describes the rules for sending triggered messages. This section describes the rules for including an Prune(S,G,rpt) message with a Join(*,G).

When a router is going to send a Join(*,G), it should use the following pseudocode, for each (S,G) for which it has state, to decide whether to include a Prune(S,G,rpt) in the compound Join/Prune message:

```
if( SPTbit(S,G) == TRUE ) {
     # Note: If receiving (S,G) on the SPT, we only prune off the
     # shared tree if the rpf neighbors differ.
      if( RPF'(*,G) != RPF'(S,G) ) {
          add Prune(S,G,rpt) to compound message
      }
} else if ( inherited_olist(S,G,rpt) == NULL ) {
  #  Note: all (*,G) olist interfaces sent rpt prunes for (S,G).
  add Prune(S,G,rpt) to compound message
} else if ( RPF'(*,G) != RPF'(S,G,rpt) {
  # Note: we joined the shared tree, but there was an (S,G) assert and
  # the source tree RPF neighbor is different.
  add Prune(S,G,rpt) to compound message
}
```

Note that Join(S,G,rpt) is not normally sent as a periodic message, but only as a triggered message.

4.4.7. State Machine for (S,G,rpt) Triggered Messages

The state machine for (S,G,rpt) triggered messages is required per-(S,G) when there is (*,G) join state at a router, and the router or any of its upstream LAN peers wishes to prune S off the RP tree.

There are three states in the state-machine. One of the states is when there is no (*,G) join state at this router. If there is (*,G) join state at the router, then the state machine must be at one of the other two states:

Pruned(S,G,rpt)
(*,G) Joined, but (S,G,rpt) pruned

NotPruned(S,G,rpt)
(*,G_ Joined, and not (S,G,rpt) pruned

NotJoined(*,G)
(*,G) has not been joined.

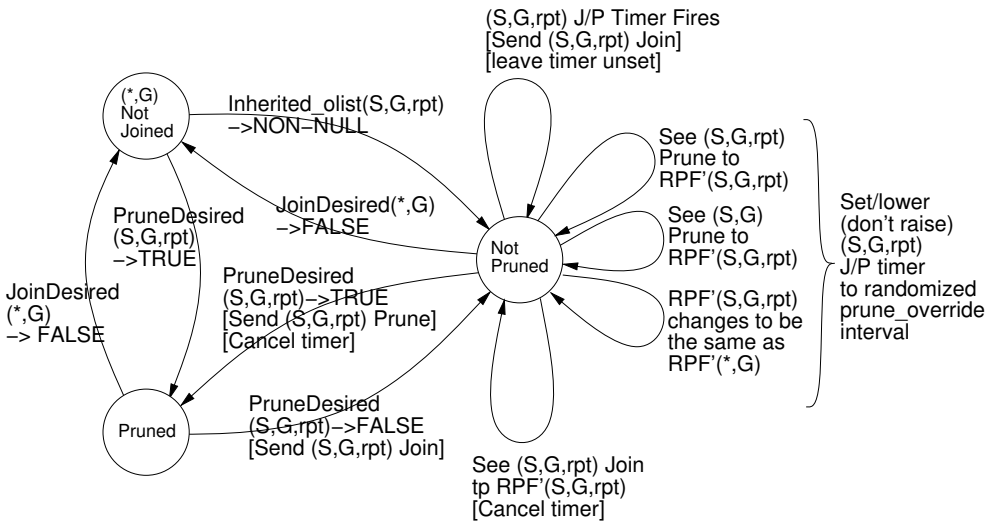


Figure 7: Upstream (S,G,rpt) state-machine for triggered messages

In tabular form, the state machine is:

Prev State	PruneDesired (S,G,rpt)->True	PruneDesired (S,G,rpt)->False	JoinDesired (*,G)->False	JoinDesired (*,G)->True	inherited_olist (S,G,rpt)->non-NULL
NotJoined (*,G)	**-> P state**	**-**	**-**	**-**	**-> NP state**
Pruned - (S,G,rpt)	**-**	**-> NP state** Send Join(S,G,rpt)	**-> NJ state**	-	-
NotPruned (S,G,rpt)	**-> P state** Send Prune(S,G,rpt) Stop timer	**-**	**-> NJ state**	-	-

Additionally, we have the following transitions within the NotPruned(S,G,rpt) state which are all used for join override behavior.

Prev State	timer expires	See Prune(S,G,rpt) to RPF'(S,G,rpt)	See Join(S,G,rpt) to RPF'(S,G,rpt)	See Prune(S,G) to RPF'(S,G,rpt)	RPF'(S,G,rpt) -> RPF'(*,G)
NotPruned (S,G,rpt)	**-> NP state** Send Join(S,G,rpt) Stop timer	**-> NP state** timer =min(timer,t_po)	**-> NP state** stop timer	**-> NP state** timer =min(timer,t_po)	**-> NP state** timer =min(timer,t_po)

This state machine uses the following macro:

```
bool PruneDesired(S,G,rpt) {
     return ( JoinDesired(*,G) AND
              inherited_olist(S,G,rpt) == NULL )
}
```

The state machine contains the following transition events:

See Join(S,G,rpt) to RPF'(S,G,rpt)

This event is only relevant in the "Not Pruned" state.

The router sees a Join(S,G,rpt) from someone else to RPF'(S,G,rpt), which is the correct upstream neighbor. If we're in "NotPruned" state and the (S,G,rpt) timer is running, then this is because we have been triggered to send our own Join(S,G,rpt) to RPF'(S,G,rpt). Someone else beat us to it, so there's no need to send our own Join.

The action is to cancel the timer.

See Prune(S,G,rpt) to RPF'(S,G,rpt)

This event is only relevant in the "NotPruned" state.

The router sees a Prune(S,G,rpt) from someone else to to RPF'(S,G,rpt), which is the correct upstream neighbor. If we're in the "NotPruned" state, then we want to continue to receive traffic from S destined for G, and that traffic is being supplied by RPF'(S,G,rpt). Thus we need to override the Prune.

The action is to set the (S,G,rpt) time to the randomized prune-override interval. However if the timer is already running, we only set the timer if doing so would set it to a lower value. At the end of this interval, if no-one else has sent a Join, then we will do so.

See Prune(S,G) to RPF'(S,G,rpt)

This event is only relevant in the "NotPruned" state.

This transition and action are the same as the above transition and action, except that the Prune does not have the RPT bit set. This transition is necessary to be compatible with existing routers that don't maintain separate (S,G) and (S,G,rpt) state.

The (S,G,rpt) prune override timer expires

This event is only relevant in the "NotPruned" state.

When the prune override timer expires, we must send a Join(S,G,rpt) to RPF'(S,G,rpt) to override the Prune message that caused the timer to be running. We only send this if RPF'(S,G,rpt) equals RPF'(*,G) - if this were not the case, then the Join might be sent to a router that does not have (*,G) Join state, and so the behavior would not be well defined. If RPF'(S,G,rpt) is not the same as RPF'(*,G), then it may stop forwarding S. However, if this happens, then the router will send an AssertCancel(S,G), which would then cause RPF'(S,G,rpt) to become equal to RPF'(*,G) (see below).

RPF'(S,G,rpt) changes to become equal to RPF'(*,G)

This event is only relevant in the "NotPruned" state.

RPF'(S,G,rpt) can only be different from RPF'(*,G) if an (S,G) Assert has happened, which means that traffic from S is arriving on the SPT, and so Prune(S,G,rpt) will have been sent to RPF'(*,G). When RPF'(S,G,rpt) changes to become equal to RPF'(*,G), we need to trigger a Join(S,G,rpt) to RPF'(*,G) to cause that router to start forwarding S again.

The action is to set the (S,G,rpt) time to the randomized prune-override interval. However if the timer is already running, we only set the timer if doing so would set it to a lower value. At the end of this interval, if no-one else has sent a Join, then we will do so.

PruneDesired(S,G,rpt)->TRUE

See macro above.

The router wishes to receive traffic for G, but does not wish to receive traffic from S destined for G. This causes the router to transition into the Pruned state.

If the router was previously in NotPruned state, then the action is to send a Prune(S,G,rpt) to RPF'(S,G,rpt). If the router was previously in NotJoined(*,G) state, then there is no need to

trigger an action in this state machine because sending a Prune(S,G,rpt) is handled by the rules for sending the Join(*,G).

PruneDesired(S,G,rpt)->FALSE

See macro above. This transition is only relevant in the "Pruned" state.

If the router is in the Pruned(S,G,rpt) state, and PruneDesired(S,G,rpt) changes to FALSE, this could be because the router no longer is in the Joined(*,G) state, or now wishes to receive traffic from S again. If it is the former, then this transition should not happen, but instead the "JoinDesired(*,G)->FALSE" transition should happen. Thus this transition should be interpreted as "PruneDesired(S,G,rpt)->FALSE AND JoinDesired(*,G)==TRUE"

The action is to send a Join(S,G,rpt) to RPF'(S,G,rpt).

JoinDesired(*,G)->FALSE

The router no longer wishes to receive any traffic destined for G on the RP Tree. This causes a transition to the NotJoined(*,G) state. Any actions are handled by the (*,G) upstream state machine.

inherited_olist(S,G,rpt) becomes non-NULL

This transition is only relevant in the NotJoined(*,G) state.

The router has joined the RP tree (handled by the (*,G) upstream state machine), and wants to receive traffic from S. This does not trigger any events in this state machine, but causes a transition to the NotPruned(S,G,rpt) state.

4.5. PIM Assert Messages

4.5.1. (S,G) Assert Message State Machine

The (S,G) Assert state machine for interface I is shown in Figure 8. There are three states:

NoInfo (NI)
This router has no (S,G) assert state on interface I.

I am Assert Winner (W)
This router has won an (S,G) assert on interface I. It is now responsible for forwarding traffic from S destined for G onto interface I. Irrespective of whether it is the DR for I, while a router is the assert winner, it is also responsible for forwarding traffic onto I on behalf of local hosts on I that have made membership requests that specifically refer to S (and G).

I am Assert Loser (L)
This router has lost an (S,G) assert on interface I. It must not forward packets from S destined for G onto interface I. If it is the DR on I, it is no longer responsible for forwarding traffic onto I to satisfy local hosts with membership requests that specifically refer to S and G.

In addition there is also a assert timer (AT) that is used to time out asserts on the assert losers and to resend asserts on the assert winner.

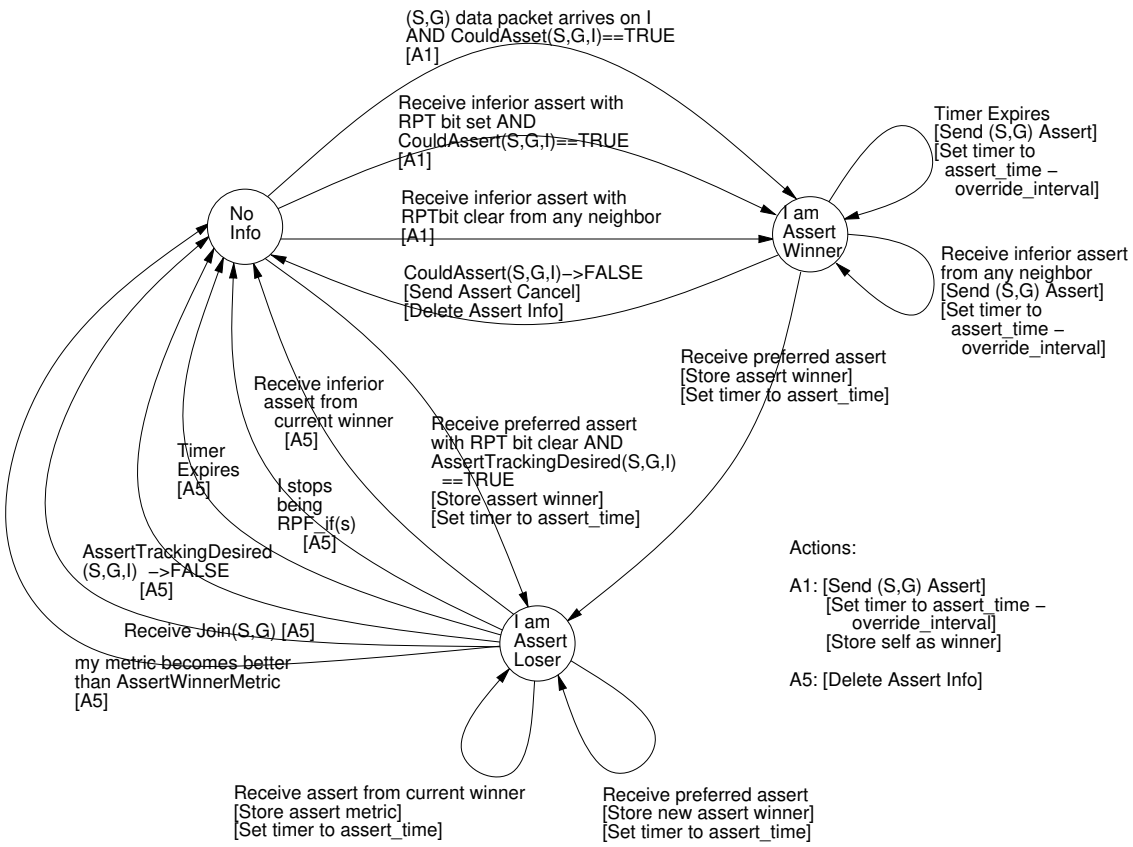


Figure 8: (S,G) Assert State-machine

In tabular form the state machine is:

Prev State	Rx Inferior Assert with RPTbit clear	Rx Assert with RPTbit set and CouldAssert(S,G,I)	Data arrives from S to G and CouldAssert(S,G,I)	Rx Preferred Assert with RPTbit clear and AssTrDes(S,G,I)
NoInfo (NI)	**-> Winner state** [Actions A1]	**-> Winner state** [Actions A1]	**-> Winner state** [Actions A1]	**-> Loser state** [Actions A2]

Prev State	Timer Expires	Rx Inferior Assert	Receive Preferred Assert	CouldAssert(S,G,I) -> FALSE
Winner (W)	**-> Winner state** [Actions A3]	**-> Winner state** [Actions A3]	**-> Loser state** [Actions A2]	**-> NoInfo state** [Actions A4]

Prev State	Rx Preferred Assert	Rx Inferior Assert from Current Winner	Timer Expires	AssTrDes (S,G,I) -> FALSE
Loser (L)	**-> L state** [Actions A2]	**-> NI state** [Actions A5]	**-> NI state** [Actions A5]	**-> NI state** [Actions A5]

Prev State	my_metric -> better than winner's metric	RPF interface stops being I	Receive Join(S,G) on interface I
Loser (L)	**-> NI state** [Actions A5]	**-> NI state** [Actions A5]	**-> NI State** [Actions A5]

Note that for reasons of compactness, "AssTrDes(S,G,I)" is used in the state-machine table to refer to AssertTrackingDesired(S,G,I).

Terminology:

A "preferred assert" is one with a better metric than the current winner.

An "inferior assert" is one with a worse metric than me.

The state machine uses the following macros:

```
CouldAssert(S,G,I) =
     I in (join(S,G) (+) pim_include(S,G))
     AND (RPF_interface(S) != I)
```

CouldAssert(S,G,I) is true on downstream interfaces for which we have (S,G) join state, or local members that explicitly requested traffic from S destined for G.

```
AssertTrackingDesired(S,G,I) =
     ((((I in joins(*,G)) AND (I not in prunes(S,G,rpt)))
       OR (pim_include(*,G,I) AND (pim_exclude(S,G,I)==FALSE)))
      AND (assert(*,G,I)==FALSE) AND (RPF_interface(S) != I))
      OR CouldAssert(S,G,I)==TRUE
      OR (RPF_interface(S) == I AND JoinSesired(S,G)==TRUE)
```

AssertTrackingDesired(S,G,I) is true on any interface in which an (S,G) assert might affect our behavior.

The first 3 lines of AssertTrackingDesired account for (*,G) join information received on I that might cause the router to be interested in asserts on I.

The 4th line accounts for (S,G) join information received on I that might cause the router to be interested in asserts on I.

The 5th line accounts for the fact that a router must keep track of assert information on the upstream interface in order to send joins to the proper neighbor.

Transitions from NoInfo State

When in NoInfo state, the following transitions are relevant:

Receive Inferior Assert with RPTbit cleared

An assert is received for (S,G) with the RPT bit cleared that is inferior to our own assert metric. The RPT bit cleared indicates that the sender of the assert had (S,G) forwarding state on this interface. If the assert is inferior to our metric, then we must also have (S,G) forwarding state as (S,G) asserts beat (*,G) asserts, and so we should be the assert winner. We transition to the "I am Assert Winner" state, and perform Actions A1 (below).

Receive Assert with RPTbit set AND CouldAssert(S,G,I)==TRUE

An assert is received for (S,G) on I with the RPT bit set (it's a (*,G) assert). CouldAssert(S,G,I) is TRUE only if we have (S,G) forwarding state on this interface, so we should be the assert winner. We transition to the "I am Assert Winner" state,

and perform Actions A1 (below).

An (S,G) data packet arrives on interface I, AND CouldAssert(S,G,I)==TRUE

An (S,G) data packet arrived on an downstream interface which is in our (S,G) outgoing interface list. We optimistically assume that we will be the assert winner for this (S,G), and so we transition to the "I am Assert Winner" state, and perform Actions A1 (below) which will initiate the asesrt negotiation for (S,G).

Receive Preferred Assert with RPT bit clear AND AssertTrackingDesired(S,G,I)==TRUE

We're interested in (S,G) Asserts, either because I is a downstream interface for which we have (S,G) or (*,G) forwarding state, or because I is the upstream interface for S and we have (S,G) forwarding state. The received assert that has a better metric than our own, so we do not win the Assert. We transition to "I am Assert Loser" and perform actions S2 (below).

Transitions from Winner State

When in "I am Assert Winner" state, the following transitions are relevant:

Timer Expires

The (S,G) assert timer expires. As we're in the Winner state, then we must still have (S,G) forwarding state that is actively being kept alive. We re-send the (S,G) Assert and restart the timer (Action A3 below). Note that the assert winner's timer is engineered to expire shortly before timers on assert losers; this prevents unnecessary thrashing of the forwarder and periodic flooding of duplicate packets.

Receive Inferior Assert

We receive an (S,G) assert or (*,G) assert mentioning S that has a worse metric than our own. Whoever sent the assert is in error, and so we re-send an (S,G) Assert, and restart the timer (Action A3 below).

Receive Preferred Assert

We receive an (S,G) assert that has a better metric than our own. We transition to "I am Assert Loser" state and perform actions A2 (below). Note that this may affect the value of joinDesired(S,G) which could cause transitions in the upstream (S,G) state machine.

CouldAssert(S,G,I) -> FALSE

Our (S,G) forwarding state or RPF interface changed so as to make CouldAssert(S,G,I) become false. We can no longer perform the actions of the assert winner, and so we transition to NoInfo state and perform actions A4 (below). This includes sending a "cancelling assert" with an infinite metric.

Transitions from Loser State

When in "I am Assert Loser" state, the following transitions can occur:

Receive Preferred Assert

We receive an assert that is better than that of the current assert winner. We stay in Loser state, and perform actions A2 below.

Receive Inferior Assert from Current Winner

We receive an assert from the current assert winner that is worse than our own metric for this group (typically the winner's metric became worse). We transition to NoInfo state, deleting the (S,G) assert information and allowing the normal PIM Join/Prune mechanisms to operate. Usually we will eventually re-assert and win when data packets from S have started flowing again.

Timer Expires
The (S,G) assert timer expires. We transition to NoInfo state, deleting the (S,G) assert information.

AssertTrackingDesired(S,G,I)->FALSE
AssertTrackingDesired(S,G,I) becomes FALSE. Our forwarding state has changed so that (S,G) Asserts on interface I are no longer of interest to us. We transition to the NoInfo state, deleting the (S,G) assert information.

My metric becomes better than the assert winner's metric
My routing metrics have changed so that now my assert metric for (S,G) is better than the metric we have stored for current assert winner. We transition to NoInfo state, delete this (S,G) assert state, and allow the normal PIM Join/Prune mechanisms to operate. Usually we will eventually re-assert and win when data packets from S have started flowing again.

RPF interface changed away from interface I
Interface I used to be the RPF interface for S, and now it is not. We transition to NoInfo state, delete this (S,G) assert state.

Receive Join(S,G)fR
We receive a Join(S,G) directed to my IP address in interface I. The action is to transition to NoInfo state, and delete this (S,G) assert state, and allow the normal PIM Join/Prune mechanisms to operate. If whoever sent the Join was in error, then the normal assert mechanism will eventually re-apply and we will lose the assert again. However whoever sent the assert may know that the previous assert winner has died, and so we may end up being the new forwarder.

(S,G) Assert State-machine Actions

A1: Send Assert(S,G)
Set timer to (Assert_Time - Assert_Override_Interval)
Store self as AssertWinner

A2: Store new assert winner (if different from previous winner)
Set timer to Assert_Time

A3: Send Assert(S,G)
Set timer to (Assert_Time - Assert_Override_Interval)

A4: Send AssertCancel(S,G)
Delete assert info

A5: Delete assert info

4.5.2. (*,G) Assert Message State Machine

The (*,G) Assert state-machine for interface I is shown in Figure 9. There are three states:

NoInfo (NI)
This router has no (*,G) assert state on interface I.

I am Assert Winner (W)
This router has won an (*,G) assert on interface I. It is now responsible for forwarding traffic destined for G onto interface I with the exception of traffic for which it has (S,G) "I am Assert Loser" state. Irrespective of whether it is the DR for I, it is also responsible for handling the membership requests for G from local hosts on I.

I am Assert Loser (L)
This router has lost an (*,G) assert on interface I. It must not forward packets for G onto interface I with the exception of traffic from sources for which is has (S,G) "I am Assert Winner" state. If it is the DR, it is no longer responsible for handling the membership requests for group G from local hosts on I.

In addition there is also an assert timer (AT) that is used to time out asserts on the assert losers and to resend asserts on the assert winner.

It is important to note that no transition occurs in this state machine as a result of receiving an assert message if the (S,G) assert state machine for the relevant S and G is not in the "NoInfo" state.

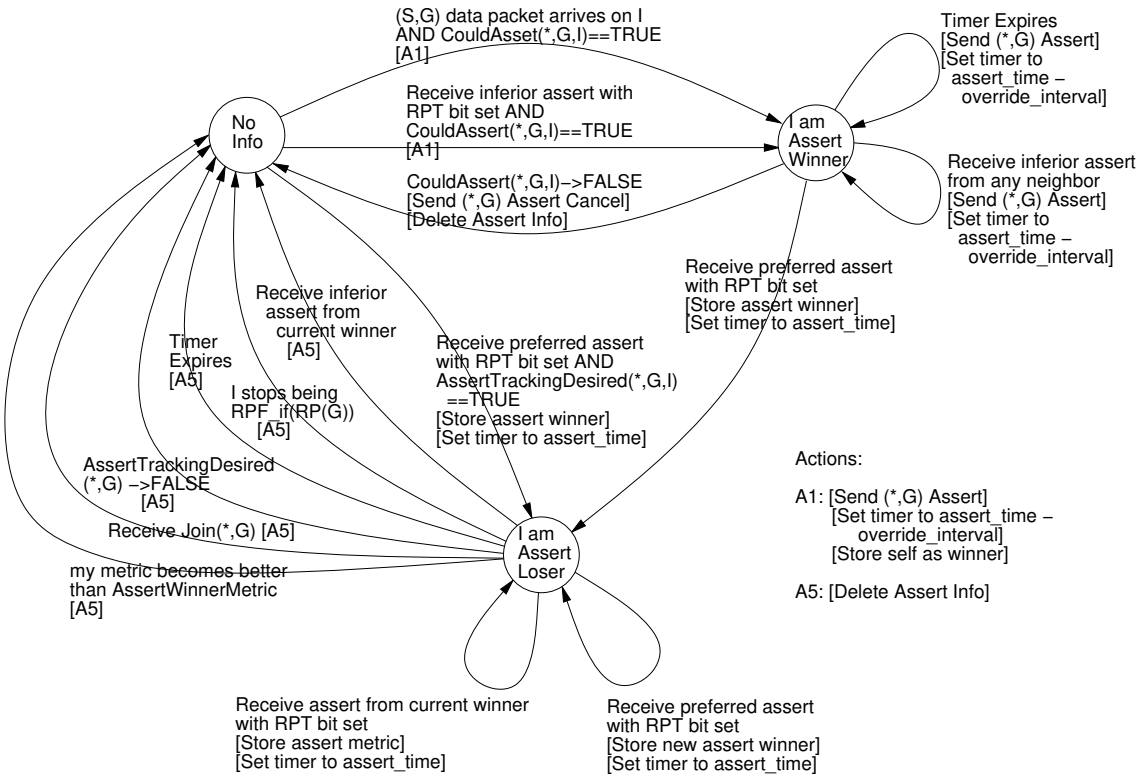


Figure 9: (*,G) Assert State-machine

In tabular form the state machine is:

Prev State	Rx Inferior Assert with RPTbit set	Data arrives for G and CouldAssert(*,G,I)	Rx Preferred Assert with RPTbit set and AssTrDes(*,G,I)
NoInfo (NI)	**-> Winner state** [Actions A1]	**-> Winner state** [Actions A1]	**-> Loser state** [Actions A2]

Prev State	Timer Expires	Rx Inferior Assert	Receive Preferred Assert	CouldAssert(*,G,I) -> FALSE
Winner (W)	**-> Winner state** [Actions A3]	**-> Winner state** [Actions A3]	**-> Loser state** [Actions A2]	**-> NoInfo state** [Actions A4]

Prev State	Rx Preferred Assert	Rx Inferior Assert from Current Winner	Timer Expires	AssTrDes (*,G,I) -> FALSE
Loser (L)	**-> L state** [Actions A2]	**-> NI state** [Actions A5]	**-> NI state** [Actions A5]	**-> NI state** [Actions A5]

Prev State	my_metric -> better than winner's metric	RPF interface stops being I	Receive Join(*,G) on interface I
Loser (L)	**-> NI state** [Actions A5]	**-> NI state** [Actions A5]	**-> NI State** [Actions A5]

The state machine uses the following macros:

```
CouldAssert(*,G,I) =
    ( I in (joins(*,G) (+) pim_include(*,G)) )
    AND RPF_interface(RP(G)) != I
```

CouldAssert(*,G,I) is true on downstream interfaces for which we have (*,G) join state, or local members that requested any traffic destined for G.

```
AssertTrackingDesired(*,G,I) =
    CouldAssert(*,G) OR
    ( RPF_interface(RP(G)) == I AND JoinDesired(*,G) )
```

AssertTrackingDesired(*,G,I) is true on any interface on which an (*,G) assert might affect our behavior.

Note that for reasons of compactness, "AssTrDes(*,G,I)" is used in the state-machine table to refer to AssertTrackingDesired(*,G,I).

Terminology:

A "preferred assert" is one with a better metric than the current winner.

An "inferior assert" is one with a worse metric than me.

Transitions from NoInfo State

When in NoInfo state, the following transitions are relevant only if the (S,G) assert state machine is in NoInfo state:

Receive Assert with RPTbit set AND CouldAssert(*,G,I)==TRUE

An (*,G) assert is received for G on Interface I. If CouldAssert(*,G,I) is TRUE, then I is our downstream interface, and we have (*,G) forwarding state on this interface, so we should be the assert winner. We transition to the "I am Assert Winner" state, and perform Actions A1 (below).

A data packet destined for G arrives on interface I, AND CouldAssert(*,G,I)==TRUE

A data packet destined for G arrived on an downstream interface which is in our (*,G) outgoing interface list. We therefore believe we should be the forwarder for this (*,G), and so we transition to the "I am Assert Winner" state, and perform Actions A1 (below).

Receive Preferred Assert with RPT bit set AND AssertTrackingDesired(*,G,I)==TRUE

We're interested in (*,G) Asserts, either because I is a downstream interface for which we have (*,G) forwarding state, or because I is the upstream interface for RP(G) and we have (*,G) forwarding state. We get a (*,G) Assert that has a better metric than our own, so we do not win the Assert. We transition to "I am Assert Loser" and perform actions S2 (below).

Transitions from Winner State

When in "I am Assert Winner" state, the following transitions are relevant only if the (S,G) assert state machine is in NoInfo state:

Receive Inferior Assert

We receive a (*,G) assert that has a worse metric than our own. Whoever sent the assert is in error, and so we re-send a (*,G) Assert, and restart the timer (Action A3 below).

Receive Preferred Assert

We receive a (*,G) assert that has a better metric than our own. We transition to "I am Assert Loser" state and perform actions A2 (below).

When in "I am Assert Winner" state, the following transitions are always relevant:

Timer Expires

The (*,G) assert timer expires. As we're in the Winner state, then we must still have (*,G) forwarding state that is actively being kept alive. To prevent unnecessary thrashing of the forwarder and periodic flooding of duplicate packets, we re-send the (*,G) Assert, and restart the timer (Action A3 below).

CouldAssert(*,G,I) -> FALSE

Our (*,G) forwarding state or RPF interface changed so as to make CouldAssert(*,G,I) become false. We can no longer perform the actions of the assert winner, and so we transition to NoInfo state and perform actions A4 (below).

Transitions from Loser State

When in "I am Assert Loser" state, the following transitions are relevant only if the (S,G) assert state machine is in NoInfo state:

Receive Preferred Assert

We receive a (*,G) assert that is better than that of the current assert winner. We stay in Loser state, and perform actions A2 below.

Receive Inferior Assert from Current Winner

We receive an assert from the current assert winner that is worse than our own metric

for this group (typically because the winner's metric became worse). We transition to NoInfo state, delete this (*,G) assert state, and allow the normal PIM Join/Prune mechanisms to operate. Usually we will eventually re-assert and win when data packets for G have started flowing again.

When in "I am Assert Loser" state, the following transitions are always relevant:

Timer Expires
The (*,G) assert timer expires. We transition to NoInfo state and delete this (*,G) assert info.

AssertTrackingDesired(*,G,I)->FALSE
AssertTrackingDesired(*,G,I) becomes FALSE. Our forwarding state has changed so that (*,G) Asserts on interface I are no longer of interest to us. We transition to NoInfo state and delete this (*,G) assert info.

My metric becomes better than the assert winner's metric
My routing metrics have changed so that now my assert metric for (*,G) is better than the metric we have stored for current assert winner. We transition to NoInfo state, and delete this (*,G) assert state, and allow the normal PIM Join/Prune mechanisms to operate. Usually we will eventually re-assert and win when data packets for G have started flowing again.

RPF interface changed away from interface I
Interface I used to be the RPF interface for RP(G), and now it is not. We transition to NoInfo state, and delete this (*,G) assert state.

Receive Join(*,G)fR
We receive a Join(*,G) directed to my IP address in interface I. The action is to transition to NoInfo state, and delete this (*,G) assert state, and allow the normal PIM Join/Prune mechanisms to operate. If whoever sent the Join was in error, then the normal assert mechanism will eventually re-apply and we will lose the assert again. However whoever sent the assert may know that the previous assert winner has died, and so we may end up being the new forwarder.

(*,G) Assert State-machine Actions

A1: Send Assert(*,G)
Set timer to (Assert_Time - Assert_Override_Interval)
Store self as AssertWinner(*,G)

A2: Store new AssertWinner(*,G) (if different from previous winner)
Set timer to assert_time

A3: Send Assert(*,G)
Set timer to (Assert_Time - Assert_Override_Interval)

A4: Send AssertCancel(*,G)
Delete assert info

A5: Delete assert info

4.5.3. Assert Metrics

Assert metrics are defined as:

```
struct assert_metric {
  rpt_bit_flag;
  metric_preference;
  route_metric;
  ip_address;
};
```

When comparing assert_metrics, the rpt_bit_flag, metric_preference, and route_metric field are compared lexicographically. If all fields are equal, the IP address is used as a tie-breaker, with the highest IP address winning.

An assert metric for (S,G) to include in (or compare against) an Assert message sent on interface I should be computed using the following pseudocode:

```
assert_metric
my_assert_metric(S,G,I) {
    if( I in immediate_olist(S,G) AND SPTbit(S,G) ) {
        return spt_assert_metric(S,G,I)
    } else if( I in inherited_olist(S,G,rpt) ) {
        # inherited_olist excludes assert(S,G,rpt)
        # interfaces, but it doesn't matter
        # because we *always* lose anyway, if we didn't hit the above
        # if clause
        return rpt_assert_metric(G,I)
    } else {
        return infinite_assert_metric()
    }
}
```

`spt_assert_metric(S,I)` gives the assert metric we use if we're sending an assert based on active (S,G) forwarding state:

```
assert_metric
spt_assert_metric(S,I) {
   return {0,mrib.pref(S),mrib.metric(S),my_ip_address(I)}
}
```

`rpt_assert_metric(G,I)` gives the assert metric we use if we're sending an assert based only on (*,G) forwarding state:

```
assert_metric
rpt_assert_metric(G,I) {
    return {1,mrib.pref(RP(G)),mrib.metric(RP(G)),my_ip_address(I)}
}
```

`mrib.pref(X)` and `mrib.metric(X)` are the routing preference and routing metrics associated with the route to a particular (unicast) destination X, as determined by the MRIB. `my_ip_address(I)` is simply the router's IP address that is associated with the local interface I.

`infinite_assert_metric()` gives the assert metric we need to send an assert but don't match either (S,G) or (*,G) forwarding state:

```
assert_metric
infinite_assert_metric() {
     return {1,infinity,infinity,infinity}
}
```

4.5.4. AssertCancel Messages

An AssertCancel message is simply an RPT Assert message but with infinite metric. It is sent by the assert winner when it deletes the forwarding state that had caused the assert to occur. Other routers will see this metric, and it will cause any other router that has forwarding state to itself assert, and to take over forwarding.

An AssertCancel(S,G) is an infinite metric assert with the RPT bit set that names S as the source.

An AssertCancel(*,G) is an infinite metric assert with the RPT bit set, and typically will name RP(G) as the source as it cannot name an appropriate S.

4.5.5. Assert State Macros

The macros `assert(S,G,rpt,I)`, `assert(S,G,I)`, and `assert(*,G,I)` are used in the olist computations of Section 4.1, and are defined as:

```
bool assert(S,G,rpt,I) {
  if ( RPF_interface(RP) == I ) {
     return FALSE
  } else {
     return ( AssertWinner(S,G,I) != me )
  }
}

bool assert(S,G,I) {
  if ( RPF_interface(S) == I ) {
     return FALSE
  } else {
     return ( AssertWinner(S,G,I) != me  AND
              (AssertWinnerMetric(S,G,I) is worse
                 than spt_assert_metric(S,G,I) )
  }
}

bool assert(*,G,I) {
  if ( RPF_interface(RP) == I ) {
     return FALSE
  } else {
     return ( AssertWinner(*,G,I) != me )
  }
}
```

`AssertWinner(S,G,I)` defaults to Null and `AssertWinnerMetric(S,G,I)` defaults to Infinity when in the NoInfo state.

Rationale for Assert Rules

The following is a summary of the rules for sending and reacting to asserts. It is not intended to be definitive (the state machines and pseudocode provide the definitive behavior). Instead it provides some rationale for the behavior.

1. Downstream neighbors send Join(*,G) and Join(S,G) periodic messages to the appropriate RPF' neighbor, i.e., the RPF neighbor as modified by the assert process. Normal suppression and override rules apply.

   This guarantees that all requested traffic will continue to arrive. This doesn't allow switch back to the "normal" RPF neighbor until the assert times out, which it won't while data is flowing if we are implementing rule 8.

2. The assert winner for (*,G) acts as the local DR for (*,G) on behalf of IGMP members, (and handles (S,G) excludes also)

   This is required to allow a single router to merge PIM and IGMP joins and leaves. Without this, overrides don't work.

3. The assert winner for (S,G) must act as the local DR for (S,G) on behalf of IGMPv3 members.

   Same rationale as (2)

4. (S,G) and (*,G) prune overrides are sent to the RPF' neighbor and not to the regular RPF neighbor.

   Same rationale as (1).

5. An (S,G,rpt) prune override is not sent (at all) if RPF'(S,G,rpt) != RPF'(*,G).

   This avoids keeping state alive on (S,G) tree when only (*,G) downstream members are left. Also, it avoids sending (S,G,rpt) joins to a router that is not on the (*,G) tree. This might be confusing and could be interpreted as being undefined although technically the current spec says to drop such a join.

6. An assert loser that receives a Join(S,G) directed to it cancels the (S,G) assert timer.

7. An assert loser that receives a Join(*,G) directed to it cancels the (*,G) assert timer and all (S,G) assert timers that do not have corresponding Prune(S,G,rpt) messagess in the compound Join/Prune message.

   Rules 7 and 8 help convergence during topology changes.

8. An assert winner for (*,G), (S,G) must send a canceling assert when it is about to stop forwarding a (*,G) or an (S,G). This rule does not apply to (S,G,rpt).

   This allow switching back to the shared tree after the last spt router on the lan leaves. We don't want RPT downstream routers to keep SPT state alive.

9. [Optionally] re-assert before timing out.

   This prevents periodic duplicates.

10. When RPF'(S,G,rpt) changes to be the same as RPF'(*,G) we need to trigger a Join(S,G,rpt) to RPF(*,G).

    This allows switching back to the RPT after the last SPT member leaves.

4.6. Designated Routers (DR) and Hello Messages

4.6.1. Sending Hello Messages

PIM-Hello messages are sent periodically on each PIM-enabled interface. They allow a router to learn about the neighboring PIM routers on each interface. Hello messages are also the mechanism used to elect a Designated Router (DR). A router must record the Hello information received from each PIM neighbor.

Hello messages are sent periodically on each PIM-enabled interface. Hello messages are multicast to address 224.0.0.13 (the ALL-PIM-ROUTERS group). Hello messages must be sent on all active interfaces, including physical point-to-point links. A hello message should be sent immediately whenever PIM is enabled on an interface, including when a router first starts. When a router first starts, the hello timer is set to a random value between 1 and `Hello_Period` to prevent synchronization of Hello messages if multiple routers are powered on simultaneously. After the initial randomized interval, Hello messages must be sent every `Hello_Period` seconds. A single hello timer is used to trigger sending Hello messages on all active interfaces. The hello timer should not be reset except when it expires.

The DR Election Priority Option allows a network administrator to give preference to a particular router in the DR election process by giving it a numerically larger DR Election Priority. The DR Election Priority Option SHOULD be included in every Hello message, even if no DR election priority is explicitly configured on that interface. This is necessary because priority-based DR election is only enabled when all neighbors on an interface advertise that they are capable of using the DR Election Priority Option. The default priority is 1.

The Generation Identifier (GenID) Option SHOULD be included in all Hello messages. The generation ID option contains a randomly generated 32-bit value that is regenerated each time PIM forwarding is started or restarted on the interface, including when the router itself restarts. When a Hello message with a new GenID is received from a neighbor, any old Hello information about that neighbor SHOULD be discarded and superseded by the information from the new Hello message. This may cause a new DR to be chosen on that interface.

4.6.2. DR Election

When a PIM-Hello message is received on interface I the following information about the sending neighbor is recorded:

`neighbor.interface`
: The interface on which the Hello message arrived.

`neighbor.ip_address`
: The IP address of the PIM neighbor.

`neighbor.genid`
: The Generation ID of the PIM neighbor.

`neighbor.dr_priority`
: The DR Priority field of the PIM neighbor if it is present in the Hello message.

`neighbor.dr_priority_present`
: A flag indicating if the DR Priority field was present in the Hello message.

`neighbor.timeout`
: A timer to time out the neighbor state when it becomes stale.
This is reset to `Hello Holdtime` whenever a Hello message is received, or to the value specified in the message, if the hold time option is used.

Neighbor state is deleted when the neighbor timeout expires.

The function for computing the DR on interface I is:

```
host
DR(I) {
    dr = me
    for each neighbor on interface I {
        if ( dr_is_better( neighbor, dr, I ) == TRUE ) {
            dr = neighbor
        }
    }
    return dr
}
```

The function used for comparing DR "metrics" on interface I is:

```
bool
dr_is_better(a,b,I) {
    if( there is a neighbor on I that does not support
        dr priority election ) {
        return a.ip_address > b.ip_address
    } else {
        return ( a.dr_priority > b.dr_priority ) OR
           ( a.dr_priority == b.dr_priority AND
                 a.ip_address > b.ip_address )
    }
}
```

The DR election priority is a 32-bit unsigned number and the numerically larger priority is always preferred. A router's idea of the current DR on an interface can change when a PIM-Hello message is received, when a neighbor times out, or when a router's own dr priority changes. If the router becomes the DR or ceases to be the DR, this will normally cause the DR Register state-machine to change state. Subsequent actions are determined by that state-machine.

4.7. PIM Bootstrap and RP Discovery

To obtain the RP information, all routers within a PIM domain collect Bootstrap messages. Bootstrap messages are sent hop-by-hop within the domain; the domain's bootstrap router (BSR) is responsible for originating the Bootstrap messages. Bootstrap messages are used to carry out a dynamic BSR election when needed and to distribute RP information in steady state.

A domain in this context is a contiguous set of routers that all implement PIM and are configured to operate within a common boundary defined by PIM Multicast Border Routers (PMBRs). PMBRs connect each PIM domain to the rest of the internet.

Routers use a set of available RPs (called the RP-Set) distributed in Bootstrap messages to get the proper Group to RP mapping. The following paragraphs give an overview of this process. The mechanism is specified in Sections 4.7.2 and 4.7.4.

4.7.1. Overview of RP Discovery

A small set of routers from a domain are configured as candidate bootstrap routers (C-BSRs) and, through a simple election mechanism, a single BSR is selected for that domain. A set of routers within a domain are also configured as candidate RPs (C-RPs); typically these will be the same routers that are configured as C-BSRs. Candidate RPs periodically unicast Candidate-RP-Advertisement messages (C-RP-Advs) to the BSR of that domain, advertising their willingness to be an RP. A C-RP-Adv message includes the address of the advertising C-RP, as well as an optional list of group addresses and a mask length fields, indicating the group prefix(es) for which the candidacy is advertised. The BSR then includes a set of these Candidate-RPs (the RP-Set), along with the corresponding group prefixes, in Bootstrap messages it periodically originates. Bootstrap messages are distributed hop-by-hop throughout the domain.

All the PIM routers in the domain receive and store Bootstrap messages originated by the BSR. When a DR gets a indication of local membership from IGMP or a data packet from a directly connected host, for a group for which it has no forwarding state, the DR uses a hash function to map the group address to one of the C-RPs whose group-prefix includes the group (see Section 4.7.5). The DR then sends a Join message towards that RP if the local host joined the group, or it Register-encapsulates and unicasts the data packet to the RP if the local host sent a packet to the group.

A Bootstrap message indicates liveness of the RPs included therein. If an RP is included in the message, then it is tagged as ‘up’ at the routers; while RPs not included in the message are removed from the list of RPs over which the hash algorithm acts. Each router continues to use the contents of the most recently received Bootstrap message from the BSR until it receives a new Bootstrap message.

If a PIM domain becomes partitioned, each area separated from the old BSR will elect its own BSR, which will distribute an RP-Set containing RPs that are reachable within that partition. When the partition heals, another election will occur automatically and only one of the BSRs will continue to send out Bootstrap messages. As is expected at the time of a partition or healing, some disruption in packet delivery may occur. This time will be on the order of the region’s round-trip time and the bootstrap router timeout value.

4.7.2. Bootstrap Router Election and RP-Set Distribution

For simplicity, bootstrap messages (BSMs) are used in both the BSR election and the RP-Set distribution mechanisms.

The state-machine for bootstrap messages depends on whether or not a router has been configured to be a Candidate-BSR. The state-machine for a C-BSR is given below, followed by the state-machine for a router that is not configured to be a C-BSR.

Candidate-BSR State Machine

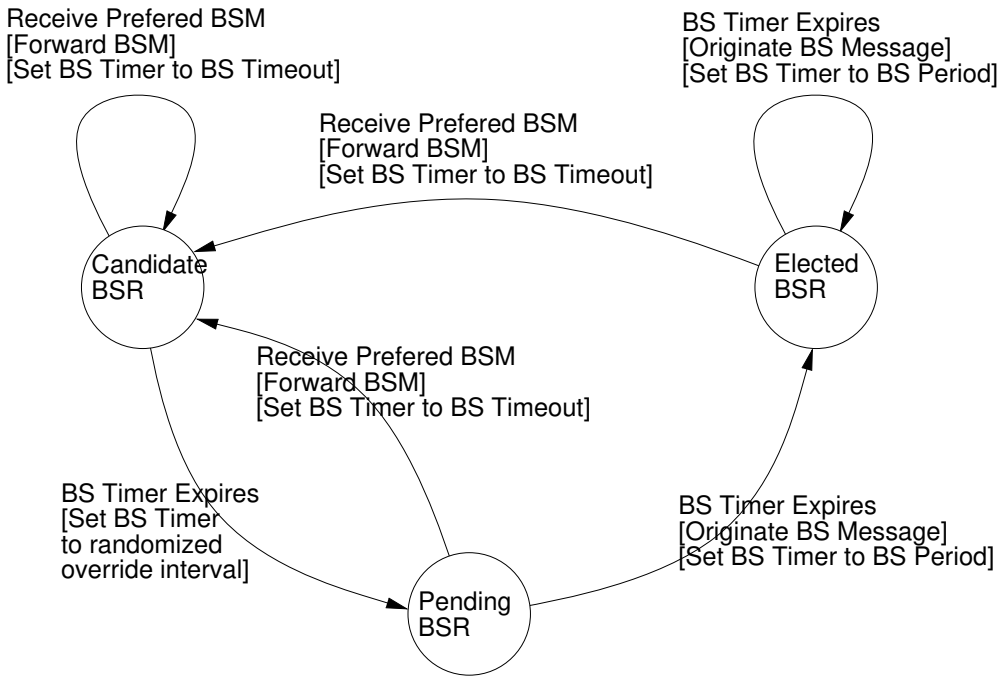


Figure 10: State-machine for a candidate BSR

In tabular form this state machine is:

Prev State	Receive Preferred BSM	BS Timer Expires
Candidate BSR (C-BSR)	**-> C-BSR state** Forward BSM Set BS Timer to BS Timeout	**-> P-BSR state** Set BS Timer to rand_override
Pending BSR (P-BSR)	**-> C-BSR state** Forward BSM Set BS Timer to BS Timeout	**-> E-BSR state** Originate BSM Set BS Timer to BS Period
Elected BSR (E-BSR)	**-> C-BSR state** Forward BSM Set BS Timer to BS Timeout	**-> E-BSR state** Originate BSM Set BS Timer to BS Period

A candidate-BSR may be in one of three states:

Candidate-BSR (C-BSR)

The router is a candidate to be a BSR, but currently another router is the preferred BSR.

Pending-BSR (P-BSR)

The router is a candidate to be a BSR. Currently no other router is the preferred BSR, but this router is not yet the BSR. For comparisons with incoming BS messages, the router treats itself as the BSR. This is a temporary state that prevents rapid thrashing of the choice of BSR during BSR election.

Elected-BSR (E-BSR)

The router is the elected bootstrap router and it must perform all the BSR functions.

On startup, the initial state is "Pending-BSR", and the BS Timer is initialized to the BS Timeout value.

In addition, there is a single timer - the bootstrap timer (BS Timer) - that is used to time out old bootstrap router information, and used in the election process to terminate P-BSR state.

State-machine for Non-Candidate-BSR Routers

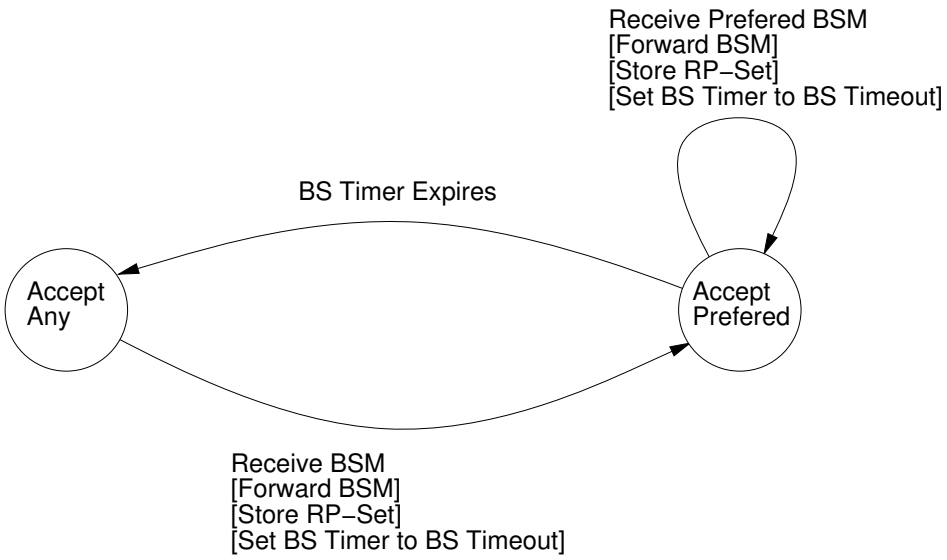


Figure 11: State-machine for a router not configured as C-BSR

In tabular form this state machine is:

Prev State	Receive Preferred BSM	Receive BSM	BS Timer Expires
Accept Any (AA)	**-> AP State** Forward BSM Store RP-Set Set BS Timer to BS Timeout	**-> AP State** Forward BSM Store RP-Set Set BS Timer to BS Timeout	**-**
Accept Preferred (AP)	**-> AP State** Forward BSM Store RP-Set Set BS Timer to BS Timeout	**-**	**-> AA State**

A router that is not a candidate-BSR may be in one of two states:

Accept Any (AA)

The router does not know of an active BSR, and will accept the first bootstrap message it sees as giving the new BSR's identity and the RP-Set. If the router has an RP-Set cached from an obsolete bootstrap message, it continues to use it.

Accept Preferred (AP)

The router knows the identity of the current BSR, and is using the RP-Set provided by that BSR. Only bootstrap messages from that BSR or from a C-BSR with higher weight than the current BSR will be accepted.

On startup, the initial state is "Accept Any".

In addition, there is a single timer - the bootstrap timer (BS Timer) that is used to time out old bootstrap router information.

Bootstrap Message Processing Checks

When a bootstrap message is received, the following initial checks must be performed:

```
if (BSM.dst_ip_address == ALL-PIM-ROUTERS group) {
  if ( BSM.src_ip_address != RPF_neighbor(BSM.BSR_ip_address) ) {
     drop the BS message silently
  }
} else if (BSM.dst_ip_address is one of my addresses) {
  if ( (BSR state != Accept Any)
       OR (DirectlyConnected(BSM.src_ip_address) == FALSE) ) {
     #the packet was unicast, but this wasn't
     #a quick refresh on startup
     drop the BS message silently
  }
} else {
  drop the BS message silently
}
```

Basically, the packet must have been sent to the ALL-PIM-ROUTERS group by the correct upstream router towards the BSR that originated the BS message, or the router must have no BSR state (it just restarted) and have received the BS message by unicast from a directly connected neighbor.

BS State-machine Transition Events

If the bootstrap message passes the initial checks above without being discarded, then it may cause a state transition event in one of the above state-machines. For both candidate and non-candidate BSRs, the following transition events are defined:

Receive Preferred BSM

A bootstrap message is received from a BSR that has greater than or equal weight than the current BSR. In a router is in P-BSR state, then it uses its own weight as that of the current BSR.

The weighting for a BSR is the concatenation in fixed-precision unsigned arithmetic of the BSR priority field from the bootstrap message and the IP address of the BSR from the bootstrap message (with the BSR priority taking the most-significant bits and the IP address taking the least significant bits).

Receive BSM

A bootstrap message is received, regardless of BSR weight.

BS State-machine Actions

The state-machines specify actions that include setting the BS timer to the following values:

BS Period

The periodic interval with which bootstrap messages are normally sent. The default value is 60 seconds.

BS Timeout

The interval after which bootstrap router state is timed out if no bootstrap message from that router has been heard. The default value is 2.5 times the BS Period, which is 150 seconds.

Randomized Override Interval

The randomized interval during which a router avoids sending a bootstrap message while it waits to see if another router has a higher bootstrap weight. This interval is to reduce control message overhead during BSR election. The following pseudocode is proposed as an efficient implementation of this "randomized" value:

```
Delay = 5 + 2 * log_2(1 + bestPriority - myPriority)
        + AddrDelay
```

where `myPriority` is the Candidate-BSR's configured priority, and `bestPriority` equals:

```
bestPriority = Max(storedPriority, myPriority)
```

and `AddrDelay` is given by the following:

```
if ( bestPriority == myPriority) {
    AddrDelay = log_2(bestAddr - myAddr) / 16
} else {
    AddrDelay = 2 - (myAddr / 2^31)
}
```

where `myAddr` is the Candidate-BSR's address, and `bestAddr` is the stored BSR's address.

In addition to setting the timer, the following actions may be triggered by state-changes in the state-machines:

Forward BSM

The bootstrap message is forwarded out of all multicast-capable interfaces except the interface it was received on. The source IP address of the message is the forwarding router's IP address on the interface the message is being forwarded from, the destination address is ALL-PIM-ROUTERS, and the TTL of the message is set to 1.

Originate BSM

A new bootstrap message is constructed by the BSR, giving the BSR's address and BSR priority, and containing the BSR's chosen RP-Set. The message is forwarded out of all multicast-capable interfaces. The IP source address of the message is the forwarding router's IP address on the interface the message is being forwarded from, the destination address is ALL-PIM-ROUTERS, and the TTL of the message is set to 1.

Store RP Set

The RP-Set from the received bootstrap message is stored and used by the router to decide the RP for each group that the router has state for. Storing this RP Set may cause other state-transitions to occur in the router. The BSR's IP address and priority from the received bootstrap message are also stored to be used to decide if future bootstrap messages are preferred.

In addition to the above state-machine actions, a DR also unicasts a stored copy of the Bootstrap message to each new PIM neighbor, i.e., after the DR receives the neighbor's first Hello message. It does so even if the new neighbor becomes the DR.

4.7.3. Sending Candidate-RP-Advertisements

Every C-RP periodically unicasts a C-RP-Adv to the BSR for that domain to inform the BSR of the C-RP's willingness to function as an RP. The interval for sending these messages is subject to local configuration at the C-RP, but must be smaller than the HoldTime in the C-RP-Adv.

A Candidate-RP-Advertisement carries a list of group address and group mask field pairs. This enables the C-RP router to limit the advertisement to certain prefixes or scopes of groups. If the C-RP becomes an RP, it may enforce this scope acceptance when receiving Registers or Join/Prune messages. C-RPs should normally send C-RP-Adv messages with the 'Priority' field set to '0'.

4.7.4. Receiving Candidate-RP-Advertisements at the BSR and Creating the RP-Set

Upon receiving a C-RP-Adv, if the router is not the elected BSR, it silently ignores the message.

If the router is the BSR, then it adds the RP address to its local pool of candidate RPs. For each C-RP, the BSR holds the following information:

IP address
: The IP address of the C-RP.

Group Prefix and Mask list
: The list of group prefixes and group masks from the C-RP advertisement.

HoldTime
: The HoldTime from the C-RP-Adv message. This is included later in the RP-set information in the Bootstrap Message.

C-RP Expiry Timer
: The C-RP-Expiry Timer is used to time out the C-RP when the BSR fails to receive C-RP-Advertisements from it. The expiry timer is initialized to the HoldTime from the RP's C-RP-Adv, and is reset to the HoldTime whenever a C-RP-Adv is received from that C-RP.

C-RP Priority
: *Do we store this?*

When the C-RP Expiry Timer expires, the C-RP is removed from the pool of available C-RPs.

The BSR uses the pool of C-RPs to construct the RP-Set which is included in Bootstrap Messages and sent to all the routers in the PIM domain. The BSR may apply a local policy to limit the number of Candidate RPs included in the Bootstrap message. The BSR may override the prefix indicated in a C-RP-Adv unless the 'Priority' field from the C-RP-Adv is not zero.

The Bootstrap message is subdivided into sets of group-prefix,RP- Count,RP-addresses. For each RP-address, the corresponding HoldTime is included in the "RP-HoldTime" field. The format of the Bootstrap message allows 'semantic fragmentation', if the length of the original Bootstrap message exceeds the packet maximum boundaries. However, we recommend against configuring a large number of routers as C-RPs, to reduce the semantic fragmentation required.

4.7.5. Receiving and Using the RP-Set

When a router receives and stores a new RP-Set, it checks if each of the RPs referred to by existing state (i.e., by (*,G), (*,*,RP), or (S,G,rpt) entries) is in the new RP-Set.

If an RP is not in the new RP-set, that RP is considered unreachable and the hash algorithm (see below) is re-performed for each group with locally active state that previously hashed to that RP. This will cause those groups to be distributed among the remaining RPs.

If the new RP-Set contains a RP that was not previously in the RP-Set, the hash value of the new RP is calculated for each group covered by the new C-RP's Group-prefix. Any group for which the new RP's hash value is greater than hash value of the group's previous RP is switched over to the new RP.

Hash Function

The hash function is used by all routers within a domain, to map a group to one of the C-RPs from the RP-Set. For a particular group, G, the hash function uses only those C-RPs whose Group-prefix covers G. The algorithm takes as input the group address, and the addresses of the Candidate RPs, and gives as output one RP address to be used.

The protocol requires that all routers hash to the same RP within a domain (except for transients). The following hash function must be used in each router:

1 For RP addresses in the RP-Set, whose Group-prefix is the longest that covers G, select the RPs with the highest priority (i.e. lowest 'Priority' value), and compute a value:

Value(G,M,C(i))=
(1103515245 * ((1103515245 * (G&M)+12345) XOR C(i)) + 12345) mod 2^31

where C(i) is the RP address and M is a hash-mask included in Bootstrap messages. The hash-mask allows a small number of consecutive groups (e.g., 4) to always hash to the same RP. For instance, hierarchically-encoded data can be sent on consecutive group addresses to get the same delay and fate-sharing characteristics.

For address families other than IPv4, a 32-bit digest to be used as C(i) must first be derived from the actual RP address. Such a digest method must be used consistently throughout the PIM domain. For IPv6 addresses, we recommend using the equivalent IPv4 address for an IPv4-compatible address, and the CRC-32 checksum [7] of all other IPv6 addresses.

2 From the RPs with the highest priority (i.e. lowest 'Priority' value), the candidate with the highest resulting hash value is then chosen as the RP for that group, and its identity and hash value are stored with the entry created.

Ties between RPs having the same hash value and priority, are broken in advantage of the highest address.

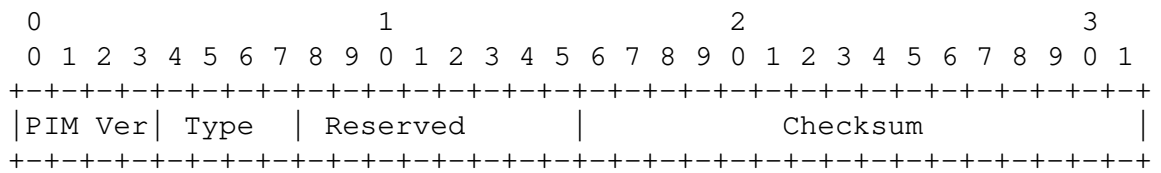
The hash function algorithm is invoked by a DR, upon reception of a packet, or IGMP membership indication, for a group, for which the DR has no entry. It is invoked by any router that has (*,*,RP) state when a packet is received for which there is no corresponding (S,G) or (*,G) entry. Furthermore, the hash function is invoked by all routers upon receiving a (*,G) or (*,*,RP) Join/Prune message.

4.8. PIM Packet Formats

This section describes the details of the packet formats for PIM control messages.

All PIM control messages have protocol number 103.

Basically, PIM messages are either unicast (e.g. Registers and Register-Stop), or multicast with TTL 1 to 'ALL-PIM-ROUTERS' group '224.0.0.13' (e.g. Join/Prune, Asserts, etc.).



PIM Ver

PIM Version number is 2.

Type

Types for specific PIM messages. PIM Types are:

0 = Hello
1 = Register
2 = Register-Stop
3 = Join/Prune
4 = Bootstrap
5 = Assert
6 = Graft (used in PIM-DM only)
7 = Graft-Ack (used in PIM-DM only)
8 = Candidate-RP-Advertisement

Reserved

set to zero. Ignored upon receipt.

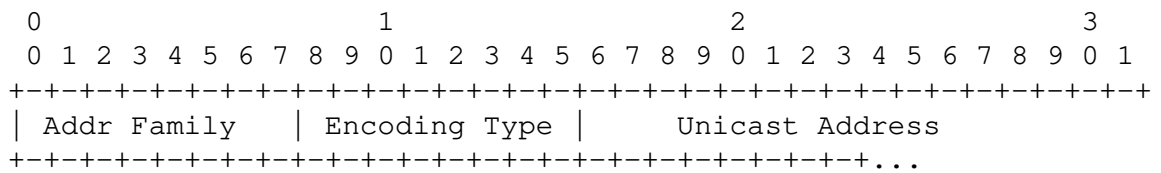
Checksum

The checksum is standard IP checksum, i.e. the 16-bit one's complement of the one's complement sum of the entire PIM message, excluding the data portion in the Register message. For computing the checksum, the checksum field is zeroed.

4.8.1. Encoded Source and Group Address Formats

Encoded-Unicast-address

Encoded-Unicast-address takes the following format:



Addr Family

The PIM address family of the 'Unicast Address' field of this address.

Values of 0-127 are as assigned by the IANA for Internet Address Families in [4]. Values 128-250 are reserved to be assigned by the IANA for PIM-specific Address Families. Values 251 thorugh 255 are designated for private use. As there is no assignment authority for this space, collisions should be expected.

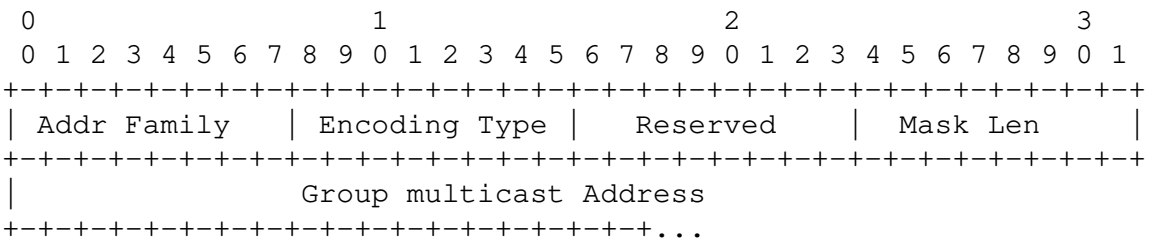
Encoding Type

The type of encoding used within a specific Address Family. The value '0' is reserved for this field, and represents the native encoding of the Address Family.

Unicast Address
The unicast address as represented by the given Address Family and Encoding Type.

Encoded-Group-Address

Encoded-Group-Address takes the following format:



Addr Family
described above.

Encoding Type
described above.

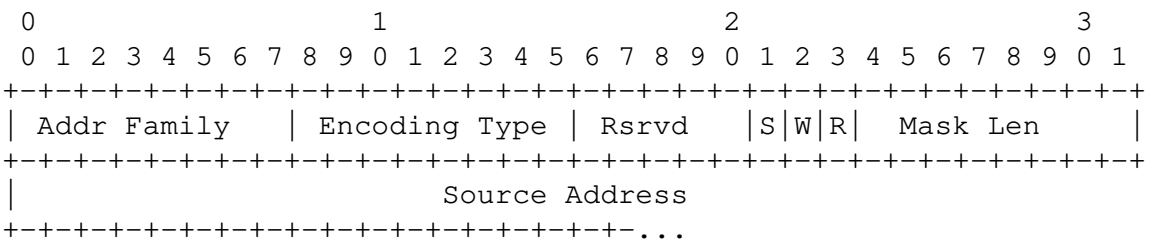
Reserved
Transmitted as zero. Ignored upon receipt.

Mask Len
The Mask length is 8 bits. The value is the number of contiguous bits left justified used as a mask which describes the address. It is less than or equal to the address length in bits for the given Address Family and Encoding Type. If the message is sent for a single group then the Mask length must equal the address length in bits for the given Address Family and Encoding Type. (e.g. 32 for IPv4 native encoding and 128 for IPv6 native encoding).

Group multicast Address
contains the group address.

Encoded-Source-Address

Encoded-Source-Address takes the following format:



Addr Family
described above.

Encoding Type
described above.

Reserved
Transmitted as zero, ignored on receipt.

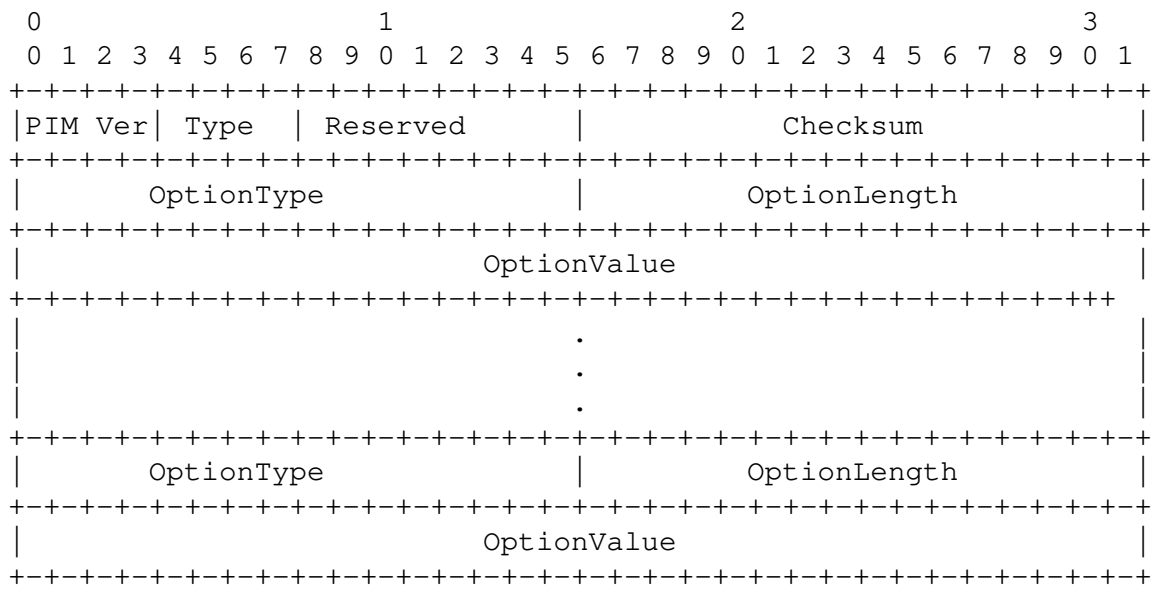
S,W,R
See Section 4.5 for details.

Mask Length
Mask length is 8 bits. The value is the number of contiguous bits left justified used as a mask which describes the address. The mask length must be less than or equal to the address length in bits for the given Address Family and Encoding Type. If the message is sent for a single group then the Mask length must equal the address length in bits for the given Address Family and Encoding Type. In version 2 of PIM, it is strongly recommended that this field be set to 32 for IPv4 native encoding.

Source Address
The source address.

4.8.2. Hello Message Format

It is sent periodically by routers on all interfaces.



PIM Version, Type, Reserved, Checksum
Described above.

OptionType
The type of the option given in the following OptionValue field.

OptionLength
The length of the OptionValue field in bytes.

OptionValue
A variable length field, carrying the value of the option.

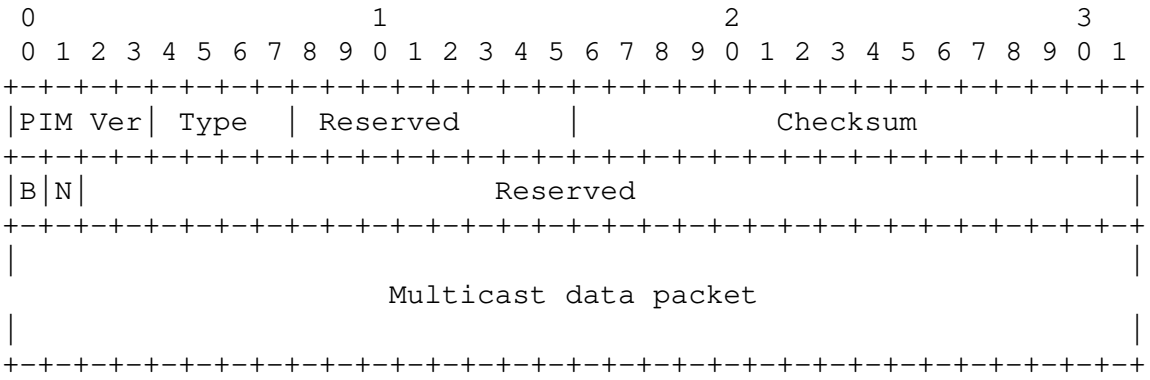
The Option fields may contain the following values:

- OptionType = 1; OptionLength = 2; OptionValue = Holdtime; where Holdtime is the amount of time a receiver must keep the neighbor reachable, in seconds. If the Holdtime is set to '0xffff', the receiver of this message never times out the neighbor. This may be used with ISDN lines, to avoid keeping the link up with periodic Hello messages. Furthermore, if the Holdtime is set to '0', the information is timed out immediately.
- OptionType 2 to 16: reserved to be defined in future versions of this document.
- OptionType 17 and 18: deprecated and should not be used.
- OptionType = 19; OptionLength = 4; OptionValue = DR Priority; where DR Priority is a 32-bit unsigned number and should be considered in the DR election as described in section 4.6.2.
- OptionType = 20; OptionLength = 4; OptionValue = Generation ID; where Generation ID is a random 32-bit value for the interface on which the Hello message is sent. The Generation ID is regenerated whenever PIM forwarding is started or restarted on the interface.
- OptionType = 21; OptionLength = 4; OptionValue = 1; This is the State Refresh capable option for dense mode.

OptionTypes 22 thru 65000 are to be assigned by the IANA. OptionTypes 65001 through 65535 are reserved for Private Use, as defined in [5].
In general, options may be ignored; but a router must not ignore the "Holdtime" OptionType.

4.8.3. Register Message Format

A Register message is sent by the DR or a PMBR to the RP when a multicast packet needs to be transmitted on the RP-tree. Source address is set to the address of the DR, destination address is to the RP's address.



PIM Version, Type, Reserved, Checksum
Described above. Note that the checksum for Registers is done only on first 8 bytes of packet, including the PIM header and the next 4 bytes, excluding the data packet portion.

For interoperability reasons, a message carrying checksum done over the entire PIM register message should be accepted.

B The Border bit. If the router is a DR for a source that it is directly connected to, it sets the B bit to 0. If the router is a PMBR for a source in a directly connected cloud, it sets the B bit to 1.

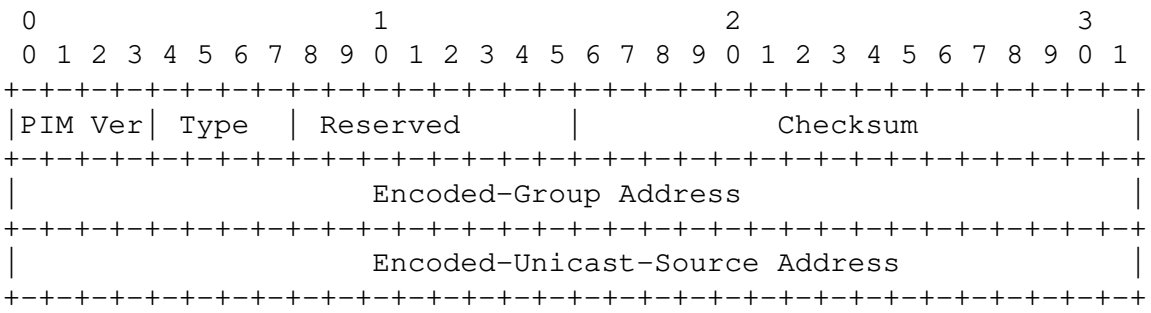
N The Null-Register bit. Set to 1 by a DR that is probing the RP before expiring its local Register-Suppression timer. Set to 0 otherwise.

Multicast data packet
The original packet sent by the source.

For (S,G) null Registers, the Multicast data packet portion contains only a dummy header with S as the source address, G as the destination address, and a data length of zero.

4.8.4. Register-Stop Message Format

A Register-Stop is unicast from the RP to the sender of the Register message. Source address is the address to which the register was addressed. Destination address is the source address of the register message.



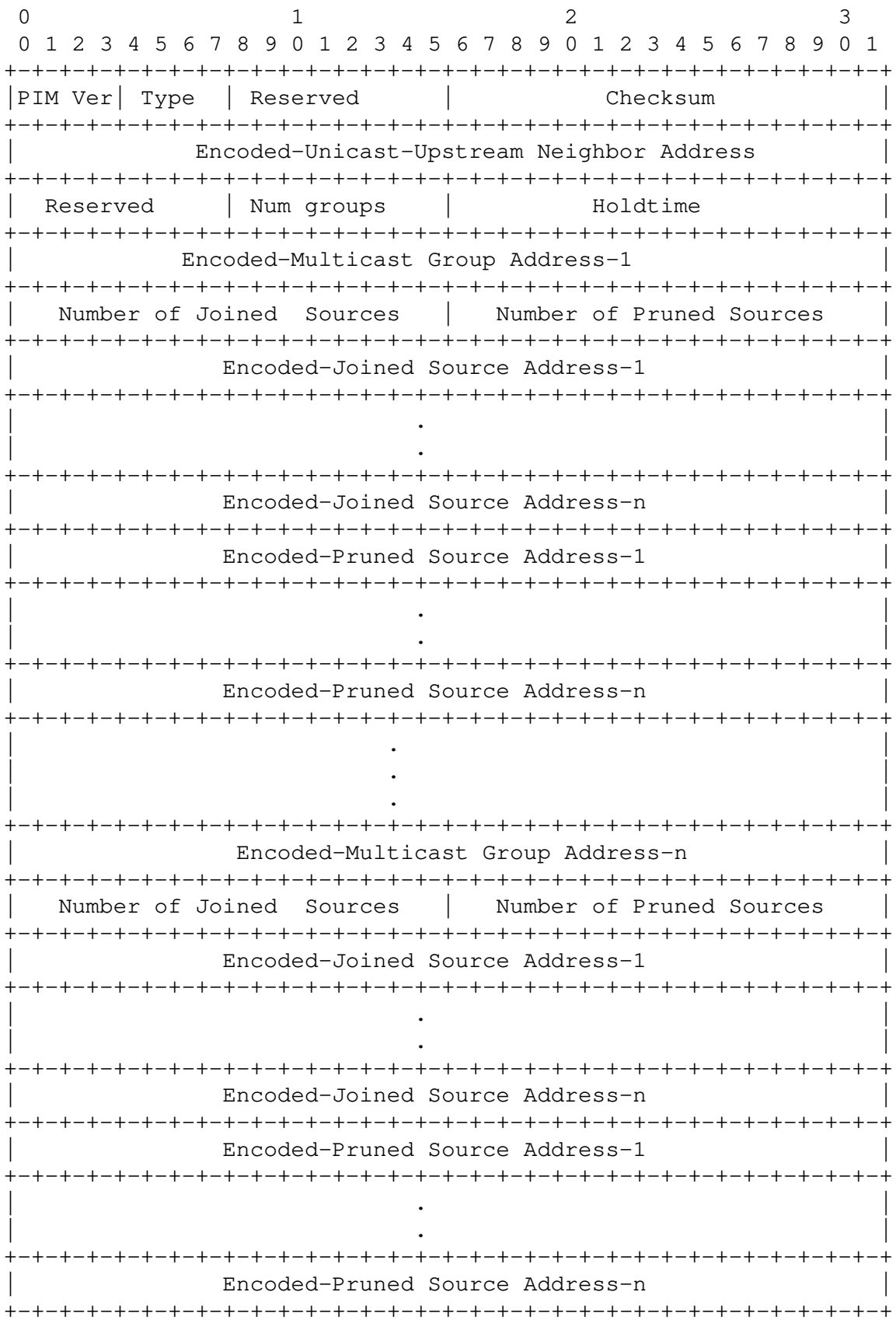
PIM Version, Type, Reserved, Checksum
Described above.

Encoded-Group Address
Format described above. Note that for Register-Stops the Mask Len field contains full address length * 8 (e.g. 32 for IPv4 native encoding), if the message is sent for a single group.

Encoded-Unicast-Source Address
host address of source from multicast data packet in register. The format for this address is given in the Encoded-Unicast-Address in 4.1. A special wild card value (0's), can be used to indicate any source.

4.8.5. Join/Prune Message Format

A Join/Prune message is sent by routers towards upstream sources and RPs. Joins are sent to build shared trees (RP trees) or source trees (SPT). Prunes are sent to prune source trees when members leave groups as well as sources that do not use the shared tree.


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
PIM Ver
Type
Reserved
Checksum
Encoded-Unicast-Upstream Neighbor Address
Reserved
Num groups
Holdtime
Encoded-Multicast Group Address-1
Number of Joined Sources
Number of Pruned Sources
Encoded-Joined Source Address-1
Encoded-Joined Source Address-n
Encoded-Pruned Source Address-1
Encoded-Pruned Source Address-n
Encoded-Multicast Group Address-n
Number of Joined Sources
Number of Pruned Sources
Encoded-Joined Source Address-1
Encoded-Joined Source Address-n
Encoded-Pruned Source Address-1
Encoded-Pruned Source Address-n

PIM Version, Type, Reserved, Checksum
Described above.

Encoded-Unicast Upstream Neighbor Address
The address of the RPF or upstream neighbor. The format for this address is given in the Encoded-Unicast-Address in 4.1. .IP "Reserved" Transmitted as zero, ignored on receipt.

Holdtime
The amount of time a receiver must keep the Join/Prune state alive, in seconds. If the Holdtime is set to '0xffff', the receiver of this message never times out the oif. This may be used with ISDN lines, to avoid keeping the link up with periodical Join/Prune messages. Furthermore, if the Holdtime is set to '0', the information is timed out immediately.

Number of Groups
The number of multicast group sets contained in the message.

Encoded-Multicast group address
For format description see Section 4.1. A wild card group in the (*,*,RP) join is represented by a 224.0.0.0 in the group address field and '4' in the mask length field. A (*,*,RP) join also has the WC-bit and the RPT-bit set.

Number of Joined Sources
Number of join source addresses listed for a given group.

Join Source Address-1 .. n
This list contains the sources that the sending router will forward multicast datagrams for if received on the interface this message is sent on.

See format section 4.1. The fields explanation for the Encoded-Source-Address format follows:

Reserved
Described above.

S The Sparse bit is a 1 bit value, set to 1 for PIM-SM. It is used for PIM v.1 compatibility.

W The WC bit is a 1 bit value. If 1, the join or prune applies to the (*,G) or (*,*,RP) entry. If 0, the join or prune applies to the (S,G) entry where S is Source Address. Joins and prunes sent towards the RP must have this bit set.

R The RPT-bit is a 1 bit value. If 1, the information about (S,G) is sent towards the RP. If 0, the information must be sent toward S, where S is the Source Address.

Mask Length, Source Address
Described above.

Represented in the form of < WC-bit >< RPT-bit ><Mask length >< Source address>:

A source address could be a host IPv4 native encoding address :

< 0 >< 0 >< 32 >< 192.1.1.17 >

A source address could be the RP's IP address :

< 1 >< 1 >< 32 >< 131.108.13.111 >

A source address could be a subnet address to prune from the RP-tree :

< 0 >< 1 >< 28 >< 192.1.1.16 >

A source address could be a general aggregate :

< 0 >< 0 >< 16 >< 192.1.0.0 >

Number of Pruned Sources

Number of prune source addresses listed for a group.

Prune Source Address-1 .. n

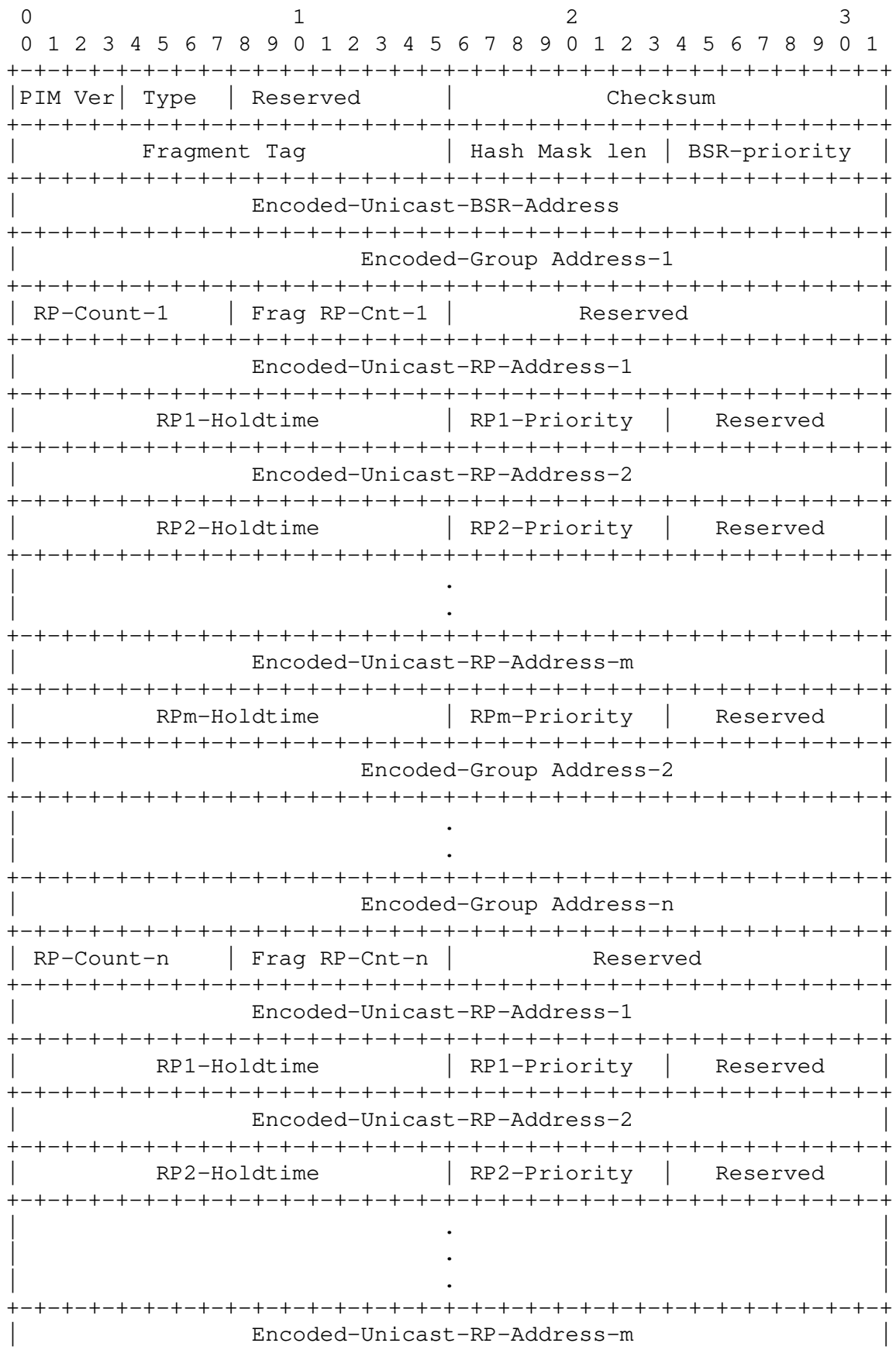
This list contains the sources that the sending router does not want to forward multicast datagrams for when received on the interface this message is sent on. If the Join/Prune message boundary exceeds the maximum packet size, then the join and prune lists for the same group must be included in the same packet.

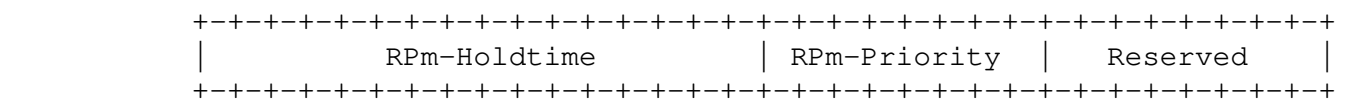
4.8.6. Bootstrap Message Format

The Bootstrap messages are multicast to 'ALL-PIM-ROUTERS' group, out all interfaces having PIM neighbors (excluding the one over which the message was received). Bootstrap messages are sent with TTL value of 1. Bootstrap messages originate at the BSR, and are forwarded by intermediate routers.

Bootstrap message is divided up into 'semantic fragments', if the original message exceeds the maximum packet size boundaries.

The semantics of a single 'fragment' is given below:


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
PIM Ver
Type
Reserved
Checksum
Fragment Tag
Hash Mask len
BSR-priority
Encoded-Unicast-BSR-Address
Encoded-Group Address-1
RP-Count-1
Frag RP-Cnt-1
Reserved
Encoded-Unicast-RP-Address-1
RP1-Holdtime
RP1-Priority
Reserved
Encoded-Unicast-RP-Address-2
RP2-Holdtime
RP2-Priority
Reserved
Encoded-Unicast-RP-Address-m
RPm-Holdtime
RPm-Priority
Reserved
Encoded-Group Address-2
Encoded-Group Address-n
RP-Count-n
Frag RP-Cnt-n
Reserved
Encoded-Unicast-RP-Address-1
RP1-Holdtime
RP1-Priority
Reserved
Encoded-Unicast-RP-Address-2
RP2-Holdtime
RP2-Priority
Reserved
Encoded-Unicast-RP-Address-m



PIM Version, Type, Reserved, Checksum
Described above.

Fragment Tag
A randomly generated number, acts to distinguish the fragments belonging to different Bootstrap messages; fragments belonging to same Bootstrap message carry the same 'Fragment Tag'.

Hash Mask len
The length (in bits) of the mask to use in the hash function. For IPv4 we recommend a value of 30. For IPv6 we recommend a value of 126.

BSR-priority
Contains the BSR priority value of the included BSR. This field is considered as a high order byte when comparing BSR addresses.

Encoded-Unicast-BSR-Address
The address of the bootstrap router for the domain. The format for this address is given in the Encoded-Unicast- Address in 4.1. .IP "Encoded-Group Address-1..n" The group prefix (address and mask) with which the Candidate RPs are associated. Format previously described.

RP-Count-1..n
The number of Candidate RP addresses included in the whole Bootstrap message for the corresponding group prefix. A router does not replace its old RP-Set for a given group prefix until/unless it receives 'RP-Count' addresses for that prefix; the addresses could be carried over several fragments. If only part of the RP-Set for a given group prefix was received, the router discards it, without updating that specific group prefix's RP-Set.

Frag RP-Cnt-1..m
The number of Candidate RP addresses included in this fragment of the Bootstrap message, for the corresponding group prefix. The 'Frag RP-Cnt' field facilitates parsing of the RP-Set for a given group prefix, when carried over more than one fragment.

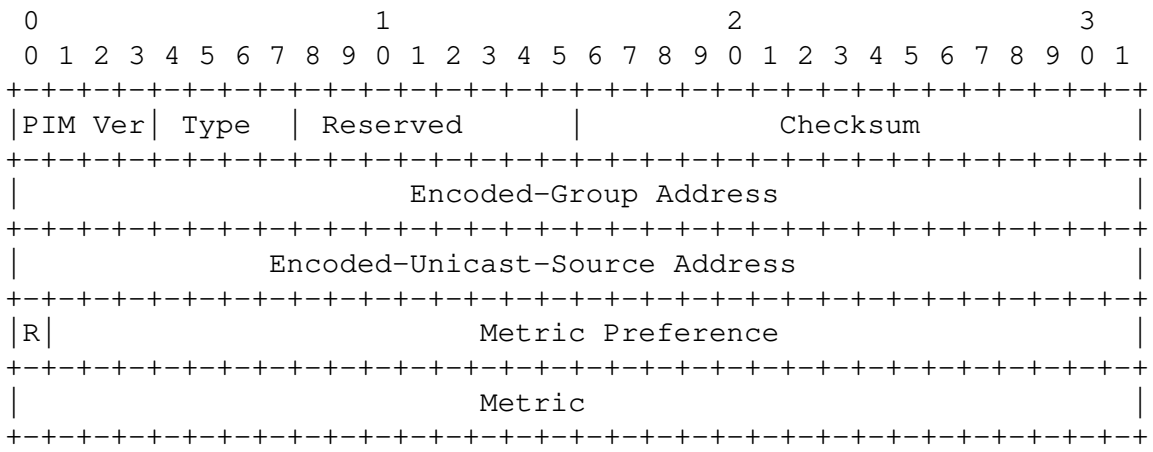
Encoded-Unicast-RP-address-1..m
The address of the Candidate RPs, for the corresponding group prefix. The format for this address is given in the Encoded-Unicast-Address in 4.1. .IP "RP1..m-Holdtime" The Holdtime for the corresponding RP. This field is copied from the 'Holdtime' field of the associated RP stored at the BSR.

RP1..m-Priority
The 'Priority' of the corresponding RP and Encoded-Group Address. This field is copied from the 'Priority' field stored at the BSR when receiving a Candidate-RP- Advertisement. The highest priority is '0' (i.e. the lower the value of the 'Priority' field, the higher). Note

that the priority is per RP per Encoded-Group Address.

4.8.7. Assert Message Format

The Assert message is sent when a multicast data packet is received on an outgoing interface corresponding to the (S,G) or (*,G) associated with the source.



PIM Version, Type, Reserved, Checksum
Described above.

Encoded-Group Address
The group address to which the data packet was addressed, and which triggered the Assert. Format previously described.

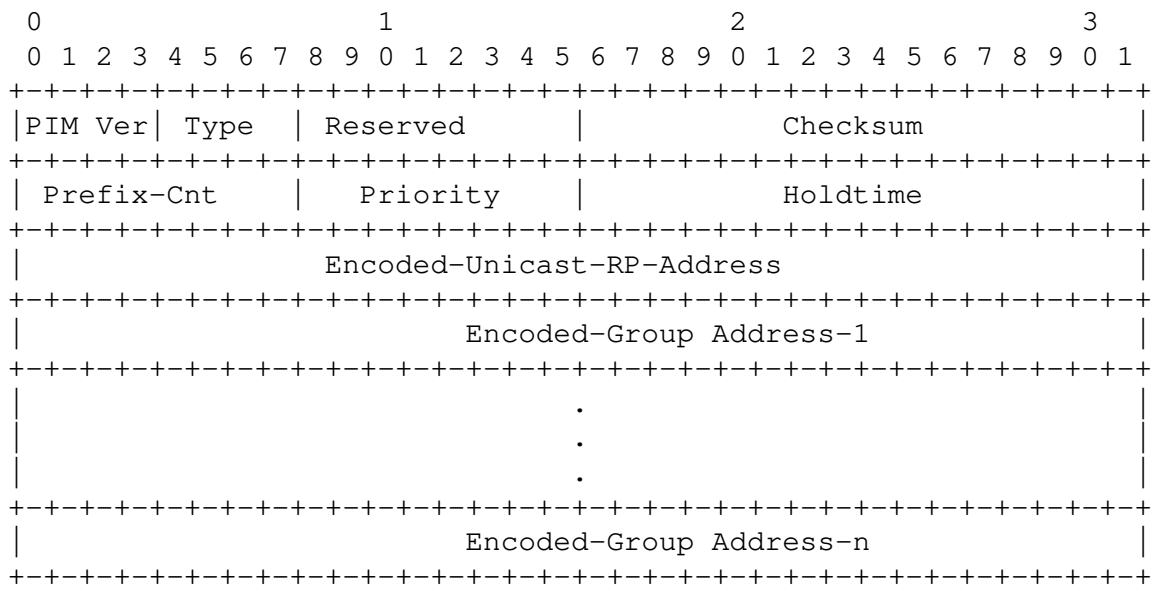
Encoded-Unicast-Source Address
Source address from multicast datagram that triggered the Assert packet to be sent. The format for this address is given in the Encoded-Unicast-Address in 4.1. .IP "R" RPT-bit is a 1 bit value. If the multicast datagram that triggered the Assert packet is routed down the RP tree, then the RPT-bit is 1; if the multicast datagram is routed down the SPT, it is 0.

Metric Preference
Preference value assigned to the unicast routing protocol that provided the route to Host address.

Metric
The unicast routing table metric. The metric is in units applicable to the unicast routing protocol used.

4.8.8. Candidate-RP-Advertisement Format

Candidate-RP-Advertisements are periodically unicast from the C-RPs to the BSR.



PIM Version, Type, Reserved, Checksum
Described above.

Prefix-Cnt
The number of encoded group addresses included in the message; indicating the group prefixes for which the C-RP is advertising. A Prefix-Cnt of '0' implies a prefix of 224.0.0.0 with mask length of 4; i.e. all multicast groups. If the C-RP is not configured with Group-prefix information, the C-RP puts a default value of '0' in this field.

Priority
The 'Priority' of the included RP, for the corresponding Encoded-Group Address (if any). highest priority is '0' (i.e. the lower the value of the 'Priority' field, the higher the priority). This field is stored at the BSR upon receipt along with the RP address and corresponding Encoded-Group Address.

Holdtime
The amount of time the advertisement is valid. This field allows advertisements to be aged out.

Encoded-Unicast-RP-Address
The address of the interface to advertise as a Candidate RP. The format for this address is given in the Encoded- Unicast-Address in 4.1. .IP "Encoded-Group Address-1..n" The group prefixes for which the C-RP is advertising. Format previously described.

4.9. PIM Timers

PIM-SM maintains the following timers, as discussed in section 4.1. All timers are countdown timers - they are set to a value and count down to zero, at which point they typically trigger an action. Of course they can just as easily be implemented as count-up timers, where the absolute expiry time is stored and compared against a real-time clock, but the language in this specification assumes that they count downwards to zero.

Global Timers

Bootstrap Timer: BST

Hello Timer: HT

Per interface (I):

Per neighbor (N):

Neighbor liveness Timer: NLT(N,I)

Per Group (G):

(*,G) Join Expiry Timer: ET(*,G,I)

(*,G) PrunePending Timer: PPT(*,G,I)

(*,G) Assert Timer: AT(*,G,I)

Per Source (S):

(S,G) Join Expiry Timer: ET(S,G,I)

(S,G) PrunePending Timer: PPT(S,G,I)

(S,G) Assert Timer: AT(S,G,I)

(S,G,rpt) Prune Expiry Timer: ET(S,G,rpt,I)

(S,G,rpt) PrunePending Timer: PPT(S,G,rpt,I)

Per Group (G):

(*,G) Upstream Join Timer: JT(*,G)

Per Source,Group pair (S,G):

(S,G) Upstream Join Timer: JT(S,G)

(S,G) Keepalive Timer: KAT(S,G)

(S,G,rpt) Upstream Override Timer: OT(S,G,rpt)

At the Bootstrap Router only:

Per Candidate RP (C):

C-RP Expiry Timer: CET(C)

At the C-RPs only:

C-RP Advertisement Timer: CRPT

At the DRs or relevant Assert Winners only:

Per Source,Group pair (S,G):

Register Stop Timer: RST(S,G)

4.10. Timer Values

When timers are started or restarted, they are set to default values. This section summarizes those default values.

Timer Name: BST

Value Name	Default Value	Explanation
BS Period	60 secs	Period between bootstrap messages
BS Timeout	2*BS_Period +10 seconds	Period after last BS message before BSR is timed out and election begins
BS randomized override interval	rand(0, 5.0 secs)	Suppression period in BSR election to prevent thrashing

Timer Name: HT

Value Name	Default Value	Explanation
Hello_Period	30 sec	Periodic interval for hello messages.

Hello messages are sent on every active interface once every Hello_Period seconds. Hello_Period defaults to 30 secs. At system power-up, the timer is initialized to rand(1,Hello_Period) to prevent synchronization.

Timer Name: NLT(N,I)

Value Name	Default Value	Explanation
Hello Holdtime	from message	Hold Time from Hello Message

The Holdtime in a Hello Message should be set to (3.5 * Hello_Period), giving a default value of 105 seconds.

Timer Names: ET(*,G,I), ET(S,G,I), ET(S,G,rpt,I)

Value Name	Default Value	Explanation
J/P HoldTime	from message	Hold Time from Join/Prune Message

See details of JT(*,G) for the Hold Time that is included in Join/Prune Messages.

Timer Name: PPT(*,G,I), PPT(S,G,I), PPT(S,G,rpt,I)

Value Name	Default Value	Explanation
J/P Override Interval	5 secs	Short period after a join or prune to allow other routers on the LAN to override the join or prune

Timer Names: AT(*,G,I), AT(S,G,I)

Value Name	Default Value	Explanation
Assert Override Interval	5 secs	Short interval before an assert times out where the assert winner resends an assert message
Assert Time	180 secs	Period after last assert before assert state is timed out

Note that for historical reasons, the Assert message lacks a Holdtime field. Thus changing the Assert Time from the default value is not recommended.

Timer Names: JT(*,G), JT(S,G)

Value Name	Default Value	Explanation
t_periodic	60 secs	Period between Join/Prune Messages
t_suppressed	rand(1.1 * t_suppressed, 1.4*t_suppressed)	Suppression period when someone else sends a J/P message so we don't need to do so.
t_override	rand(0, 4.5 secs)	Randomized delay to prevent response implosion when sending a join message to override someone else's prune message.

t_periodic may be set to take into account such things as the configured bandwidth and expected average number of multicast route entries for the attached network or link (e.g., the period would be longer for lower-speed links, or for routers in the center of the network that expect to have a larger number of entries). If the Join/Prune-Period is modified during operation, these changes should be made relatively infrequently and the router should continue to refresh at its previous Join/Prune-Period for at least Join/Prune-Holdtime, in order to allow the upstream router to adapt.

The holdtime specified in a Join/Prune message should be set to (3.5 * t_periodic).

Timer Names: KAT(S,G)

Value Name	Default Value	Explanation
Keepalive_Period	210 secs	Period after last (S,G) data packet during which (S,G) Join state will be maintained even in the absence of (S,G) Join state.
RP_Keepalive_Period	(1.5 * Register Period Suppression) + Register Probe Time	As Keepalive_Period, but at the RP when a RegisterStop is sent.

The normal keepalive period for the KAT(S,G) defaults to 210 seconds. However at the RP, the keepalive period must be at least the Register_Suppression_Time or the RP may time out the (S,G) state before the next Null-Register arrives. Thus the KAT(S,G) is set to max(Keepalive_Period, RP_Keepalive_Period).

Timer Names: OT(S,G,rpt)

Value Name	Default Value	Explanation
t_po	rand(0, 4.5 secs)	Randomized delay to prevent response implosion when sending a join message to override someone else's prune message.

Timer Names: CET(R)

Value Name	Default Value	Explanation
C-RP Timeout	from message	Hold time from C-RP-Adv message

C-RP Advertisement messages are sent periodically with period C-RP-Adv-Period. C-RP-Adv-Period defaults to 60 seconds. The holdtime to be specified in a C-RP-Adv message should be set to (2.5 * C-RP-Adv-Period).

Timer Name: CRPT

Value Name	Default Value	Explanation
C-RP-Adv-Period	60 seconds	Period with which periodic C-RP Advertisements are sent to BSR

Timer Name: RST(S,G)

Value Name	Default Value	Explanation
Register Suppression Time	60 seconds	Period during which a DR stops sending Register-encapsulated data to the RP after receiving a RegisterStop
Register Probe Time	5 seconds	Time before RST expires when a DR may send a Null-Register to the RP to cause it to resend a RegisterStop message.

5. IANA Considerations

5.1. PIM Address Family

The PIM Address Family field was chosen to be 8 bits as a tradeoff between packet format and use of the IANA assigned numbers. Since when the PIM packet format was designed only 15 values were assigned for Address Families, and large numbers of new Address Family values were not envisioned, 8 bits seemed large enough. However, the IANA assigns Address Families in a 16-bit field. Therefore, the PIM Address Family is allocated as follows:

> Values 0 through 127 are designated to have the same meaning as IANA-assigned Address Family Numbers [4].

Values 128 through 250 are designated to be assigned by the IANA based upon IESG Approval, as defined in [5]. XXX note: is the IESG OK with this?

Values 251 through 255 are designated for Private Use, as defined in [5].

5.2. PIM Hello Options

Values 22 through 65000 are to be assigned by the IANA. Since the space is large, they may be assigned as First Come First Served as defined in [5]. Such assignments are valid for one year, and may be renewed. Permanent assignments require a specification (see "Specification Required" in [5].)

6. Security Considerations

All PIM control messages MAY use IPsec [6] to address security concerns. The authentication methods are addressed in a companion document [7]. Keys may be distributed as described in [8].

XXX This probably needs more.

7. Authors' Addresses

Bill Fenner
AT&T Labs - Research,
75 Willow Road,
Menlo Park, CA 94025,
fenner@research.att.com

Mark Handley
ACIRI/ICSI
1947 Center St, Suite 600
Berkeley, CA 94708
mjh@aciri.org

Hugh Holbrook
Cisco Systems
holbrook@cisco.com

Isidor Kouvelas
Cisco Systems
kouvelas@cisco.com

8. Acknowledgments

PIM-SM was designed over many years by a large group of people, including ideas from Deborah Estrin, Dino Farinacci, Ahmed Helmy, David Thaler, Steve Deering, Van Jacobson, C. Liu, Puneet Sharma, Liming Wei, Tom Pusateri, Tony Ballardie, Scott Brim, Jon Crowcroft, Paul Francis, Joel Halpern, Horst Hodel, Polly Huang, Stephen Ostrowski, Lixia Zhang and Girish Chandranmenon.

Thanks are due to the American Licorice Company, for its obscure but possibly essential role in the creation of this document.

9. References

[1] T. Bates , R. Chandra , D. Katz , Y. Rekhter, "Multiprotocol Extensions for BGP-4", RFC 2283

[2] S.E. Deering, "Host extensions for IP multicasting", RFC 1112, Aug 1989.

[3] W. Fenner, "Internet Group Management Protocol, Version 2", RFC 2236.

[4] IANA, "Address Family Numbers", linked from http://www.iana.org/numbers.html

[5] T. Narten , H. Alvestrand, "Guidelines for Writing an IANA Considerations Section in RFCs", RFC 2434.

[6] S. Kent, R. Atkinson, "Security Architecture for the Internet Protocol.", RFC 2401.

[7] L. Wei, "Authenticating PIM version 2 messages", draft-ietf-pim-v2-auth-01.txt, work in progress.

[8] T. Hardjono, B. Cain, "Simple Key Management Protocol for PIM", draft-ietf-pim-simplekmp-01.txt, work in progress.

Table of Contents

1. Introduction . . . 2
2. Terminology . . . 2
2.1. Definitions . . . 2
2.2. Pseudocode Notation . . . 3
3. PIM-SM Protocol Overview . . . 3
4. Protocol Specification . . . 6
4.1. PIM Protocol State . . . 6
4.2. Data Packet Forwarding Rules . . . 13
4.2.1. Setting and Clearing the (S,G) SPT bit . . . 15
4.3. PIM Register Messages . . . 15
4.3.1. Sending Register Messages from the DR . . . 16
4.3.2. Receiving Register Messages at the RP . . . 18
4.3.3. RP Joining to the Source . . . 19
4.4. PIM Join/Prune Messages . . . 19
4.4.1. Receiving (*,G) Join/Prune Messages . . . 19
4.4.2. Receiving (S,G) Join/Prune Messages . . . 21
4.4.3. Receiving (S,G,rpt) Join/Prune Messages . . . 22
4.4.4. Sending (*,G) Join/Prune Messages . . . 25
4.4.5. Sending (S,G) Join/Prune Messages . . . 27
4.4.6. (S,G,rpt) Periodic Messages . . . 29
4.4.7. State Machine for (S,G,rpt) Triggered Messages . . . 29
4.5. PIM Assert Messages . . . 33
4.5.1. (S,G) Assert Message State Machine . . . 33
4.5.2. (*,G) Assert Message State Machine . . . 38
4.5.3. Assert Metrics . . . 41
4.5.4. AssertCancel Messages . . . 43
4.5.5. Assert State Macros . . . 43
4.6. Designated Routers (DR) and Hello Messages . . . 45
4.6.1. Sending Hello Messages . . . 45
4.6.2. DR Election . . . 45
4.7. PIM Bootstrap and RP Discovery . . . 46
4.7.1. Overview of RP Discovery . . . 47
4.7.2. Bootstrap Router Election and RP-Set Distribution . . . 47
4.7.3. Sending Candidate-RP-Advertisements . . . 52
4.7.4. Receiving Candidate-RP-Advertisements at the BSR and Creating the RP-Set . . . 52
4.7.5. Receiving and Using the RP-Set . . . 52
4.8. PIM Packet Formats . . . 53
4.8.1. Encoded Source and Group Address Formats . . . 54
4.8.2. Hello Message Format . . . 56
4.8.3. Register Message Format . . . 57
4.8.4. Register-Stop Message Format . . . 58
4.8.5. Join/Prune Message Format . . . 58
4.8.6. Bootstrap Message Format . . . 61
4.8.7. Assert Message Format . . . 64
4.8.8. Candidate-RP-Advertisement Format . . . 64
4.9. PIM Timers . . . 66

5. IANA Considerations 69
6. Security Considerations 70
8. Acknowledgments 70
9. References 71

Index

ActiveDR(S,G) . . . 18
assert(*,G) . . . 13
assert(*,G,I) . . . 12,13,36,44
assert(S,G) . . . 13
assert(S,G,I) . . . 12,13,44
assert(S,G,rpt) . . . 13
assert(S,G,rpt,I) . . . 13,44
AssertCancel(*,G) . . . 42
AssertTimer(*,G,I) . . . 8,13,39,69
AssertTimer(S,G,I) . . . 9,13,34,69
AssertTrackingDesired(*,G,I) . . . 40
AssertTrackingDesired(S,G,I) . . . 36
AssertWinner(*,G,I) . . . 12,13
AssertWinner(S,G,I) . . . 12,13,44
assert_metric . . . 42
Assert_Override_Interval . . . 38,42,69
Assert_Time . . . 38,42,69
AT(*,G,I) . . . 8,13,39,69
AT(S,G,I) . . . 9,13,34,69
BST . . . 8
BS_Period . . . 68
BS_randomized_override_interval . . . 68
BS_Timeout . . . 68
BS_Timer . . . 68
C-RP-Adv-Period . . . 70
C-RP-Timer . . . 70
C-RP_Timeout . . . 70
CET(R) . . . 70
CouldAssert(*,G,I) . . . 40
CouldAssert(S,G,I) . . . 36
DirectlyConnected(S) . . . 15,16,18
DownstreamState(*,G,I) . . . 12
DownstreamState(S,G,I) . . . 12
DR(I) . . . 47
dr_is_better(a,b,I) . . . 47
ET(*,G,I) . . . 8,20,68
ET(S,G,I) . . . 9,22,68
ET(S,G,rpt,I) . . . 10,24,68
Hash_Function . . . 54
Hello_Holdtime . . . 68
Hello_Period . . . 68
HT . . . 68
igmp_desired(*,G,I) . . . 12
igmp_desired(S,G,I) . . . 12
immediate_olist(*,G) . . . 11
immediate_olist(S,G) . . . 11,29,43
infinite_assert_metric() . . . 44
inherited_olist(*,G) . . . 11,27
inherited_olist(S,G) . . . 12,15,19,29
inherited_olist(S,G,rpt) . . . 12,15,16,30,31,33,43
I_am_DR(I) . . . 12,18

I_am_RP(G) 19
J/P_HoldTime 68
J/P_Override_Interval 21,23,25,68
Join(*,G) 4
join(S,G) 36
JoinDesired(*,G) 27,31,33,40
JoinDesired(S,G) 16,29
joins(*,G) 12,36,40
joins(S,G) 12
JoinSesired(S,G) 36
JT(*,G) 8,26,69
JT(S,G) 9,28,69
KAT(S,G) 9,15,18,19,29,69
KeepaliveTimer(S,G) 9,15,18,19,29,69
Keepalive_Period 69
MBGP 3
MRIB 3
mrib.next_hop(host) 13
my_assert_metric(S,G,I) 43
NLT(N,I) 8,68
OT(S,G,rpt) 11,70
packet_arrives_on_rp_tunnel(pkt) 19
pim_exclude(S,G,I) 36
pim_include(*,G) 12,40
pim_include(*,G,I) 36
pim_include(S,G) 12,36
PPT(*,G,I) 8,20,68
PPT(S,G,I) 9,22,68
PPT(S,G,rpt,I) 10,24,68
prune(S,G,rpt,I) 12
PruneDesired(S,G,rpt) 31,32
prunes(S,G,rpt) 12,36
RegisterStop 4,17
RegisterStop(*,G) 18
RegisterStop(S,G) 19
Register_Probe_Time 18,19,70
Register_Suppression_Time 18,19,69,70
RP(G) 13,40,40
RPF'(*,G) 13,16,30
RPF'(S,G) 13,16,30
RPF'(S,G,rpt) 13,30,32
RPF_interface 40
RPF_interface(host) 13,15,16,18,36,40,44
rpt_assert_metric(G,I) 43
RST(S,G) 17,70
SPTbit(S,G) 15,16,19,30,43
spt_assert_metric(S,I) 43
t_override 27,69
t_periodic 27,69
t_po 31,70
t_suppressed 27,69
Update_SPTbit(S,G) 16
UpstreamState(S,G) 15