Internet Engineering Task Force
INTERNET-DRAFT
draft-ietf-pim-sm-v2-new-02.ps
PIM WG
Bill Fenner/AT&T
Mark Handley/ACIRI
Hugh Holbrook/Cisco
Isidor Kouvelas/Cisco
2 March 2001
Expires: September 2001

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

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.

Table of Contents

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 multicast topology table, which is typically derived from the unicast routing table, or routing protocols such as MBGP that carry multicast-specific topology information. In PIM-SM this 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.

Upstream
: Towards the root of the tree. The root of tree may either be the source or the RP depending on the context.

Downstream
: Away from the root of the tree.

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 topology-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 come and go 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. When all receivers on a leaf-network leave the group, the DR will send a PIM (*,G) Prune message towards the RP for that multicast group. However if the prune message is not sent for any reason, 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 register-encapsulated 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 is currently set aside for source-specific multicast in IPv4. 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, as described in Section 4.8.

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 either through a bootstrap mechanism or through static configuration.

One dynamic way to do this is to use the Bootstrap Router (BSR) mechanism [4]. One router in each PIM domain is elected the Bootstrap Router 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 RP discovery mechanisms.
- The subset of PIM required to support Source-Specific Multicast, PIM-SSM, is described in Section 4.8.
- PIM packet formats are specified in Section 4.9.
- A summary of PIM-SM timers and their default values is given in Section 4.10.

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 implementation would actually implement packet forwarding operations in terms of this state. Most 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 four sections:

(*,*,RP) state
: State that maintains per-RP trees, for all groups served by a given RP.

(*,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:

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

Designated Router state is described in section 4.6.

4.1.2. (*,*,RP) State

For every RP a router keeps the following state:

(*,*,RP) state:

For each interface:

PIM (*,*,RP) Join/Prune State:

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

Not interface specific:

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

PIM (*,*,RP) Join/Prune state is the result of receiving PIM (*,*,RP) Join/Prune messages on this interface, and is specified in section 4.4.1.

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

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(*,*,RP) to the new upstream neighbor and a Prune(*,*,RP) 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(*,*,RP). These mechanisms are specified in Section 4.4.5.

4.1.3. (*,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 4.1.6.

PIM (*,G) Join/Prune state is the result of receiving PIM (*,G) Join/Prune messages on this interface, and is specified in section 4.4.2. The state is used by the macros that calculate the outgoing interface list in section 4.1.6, and in the `JoinDesired(*,G)` macro (defined in section 4.4.6) 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) 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 4.4.6.

4.1.4. (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 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 4.1.6.

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 4.4.2. The state is used by the macros that calculate the outgoing interface list in section 4.1.6, and in the `JoinDesired(S,G)` macro (defined in section 4.4.7) 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 4.4.7.

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.

4.1.5. (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 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 4.1.6.

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 4.4.4. The state is used by the macros that calculate the outgoing interface list in section 4.1.6, and in the rules for adding Prune(S,G,rpt) messages to Join(*,G) messages specified in section 4.4.8.

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 4.4.9.

4.1.6. 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 in the (*,G) olist from the olist that is actually used to forward traffic. 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(*,*,RP)=
    joins(*,*,RP)

immediate_olist(*,G) =
    joins(*,G) (+) pim_include(*,G) (-) lost_assert(*,G)

immediate_olist(S,G) =
```

```
    joins(S,G) (+) pim_include(S,G) (-) lost_assert(S,G)

inherited_olist(S,G,rpt) =
        ( joins(*,*,RP(G)) (+) joins(*,G) (-) prunes(S,G,rpt) )
    (+) ( pim_include(*,G) (-) pim_exclude(S,G))
    (-) ( lost_assert(*,G) (+) lost_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 lost_assert(*,G,I) == FALSE )
       OR AssertWinner(*,G,I) == me )
     AND  local_receiver_include(*,G,I) }

pim_include(S,G) =
    { all interfaces I such that:
      ( (I_am_DR( I ) AND lost_assert(S,G,I) == FALSE )
        OR AssertWinner(S,G,I) == me )
       AND  local_receiver_include(S,G,I) }

pim_exclude(S,G) =
    { all interfaces I such that:
      ( (I_am_DR( I ) AND lost_assert(S,G,I) == FALSE )
        OR AssertWinner(S,G,I) == me )
       AND  local_receiver_exclude(S,G,I) }
```

The clause "local_receiver_include(S,G,I)" is true if the IGMP module or other local membership mechanism has determined that there are local members on interface I that desire to receive traffic sent specifically by S to G. "local_receiver_include(*,G,I)" is true if the IGMP module or other local membership mechanism has determined that there are local members on interface I that desire to receive all traffic sent to G. "local_receiver_exclude(S,G,I) is true if local_receiver_include(*,G,I) is true but none of the local members desire to receive traffic from S.

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

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

DownstreamJPState(*,*,RP,I) is the state of the finite state machine in section 4.4.1.

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
      DownstreamJPState(*,G,I) is either Joined or PrunePending }
```

DownstreamJPState(*,G,I) is the state of the finite state machine in section 4.4.2.

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
      DownstreamJPState(S,G,I) is either Joined or PrunePending }
```

DownstreamJPState(S,G,I) is the state of the finite state machine in section 4.4.3.

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.

```
prunes(S,G,rpt) =
    { all interfaces I such that
      DownstreamJPState(S,G,I) is Pruned or PruneTmp }
```

The set "lost_assert(*,G)" is the set of all interfaces on which the router has received (*,G) joins but has lost a (*,G) assert. The macro lost_assert(*,G,I) is defined in Section 4.5.5.

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

The set "lost_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 lost_assert(S,G,rpt,I) is defined in Section 4.5.5.

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

The set "lost_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 lost_assert(S,G,I) is defined in Section 4.5.5.

```
lost_assert(S,G) =
    { all interfaces I such that
      lost_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). 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 4.4.8).

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 NULL. At the RP for G, `MRIB.next_hop( RP(G ))` returns NULL.

I_Am_Assert_loser(S, G, I) is true if the Assert start machine (in section 4.5.1) for (S,G) on Interface I is in "I am Assert Loser" state.

I_Am_Assert_loser(*, G, I) is true if the Assert start machine (in section 4.5.2) for (*,G) on Interface I is in "I am Assert Loser" state.

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 should be accepted based on TIB state and the interface that the packet arrived on.

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 (*,*,RP) or (*,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.

On receipt on a data from S to G on interface iif:

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

Update_SPTbit(S,G,iif)

if( iif == RPF_interface(S) AND UpstreamJPState(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 iif is in inherited_olist(S,G) ) {
      send Assert(S,G) on iif
   } else if ( SPTbit(S,G) == FALSE AND
               iif is in inherited_olist(S,G,rpt) {
      send Assert(*,G) on iif
   }
}

oiflist = oiflist (-) iif
forward packet on 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 (*,*,RP) state, (*,G) state, asserts, etc.

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

Update_SPTbit(S,G,iif) is defined in section 4.2.1.

UpstreamJPState(S,G) is the state of the finite state machine in section 4.4.7.

Keepalive_Period is defined in Section 4.10.

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 (*,*,RP)/(*,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 (*,*,RP)/(*,G) state while upstream (S,G) state is being established during which time a router should continue to forward only on (*,*,RP)/(*,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
  }
}
```

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

`JoinDesired(S,G)` is defined in Section 4.4.7, 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 appropriate (S,G) join state and the packet arrived on the correct upstream interface for S, and one or more of the following conditions applies:

1. The source is directly connected, in which case the switch to the SPT is a no-op.
2. 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.
3. No-one wants the packet on the RP tree.
4. 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. However item (3) above is needed because there may not be any (*,G) state to trigger an Assert(S,G) to happen.

The SPT bit is cleared in the (S,G) upstream state machine (see Section 4.4.7) when JoinDesired(S,G) becomes FALSE.

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 RP.

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.

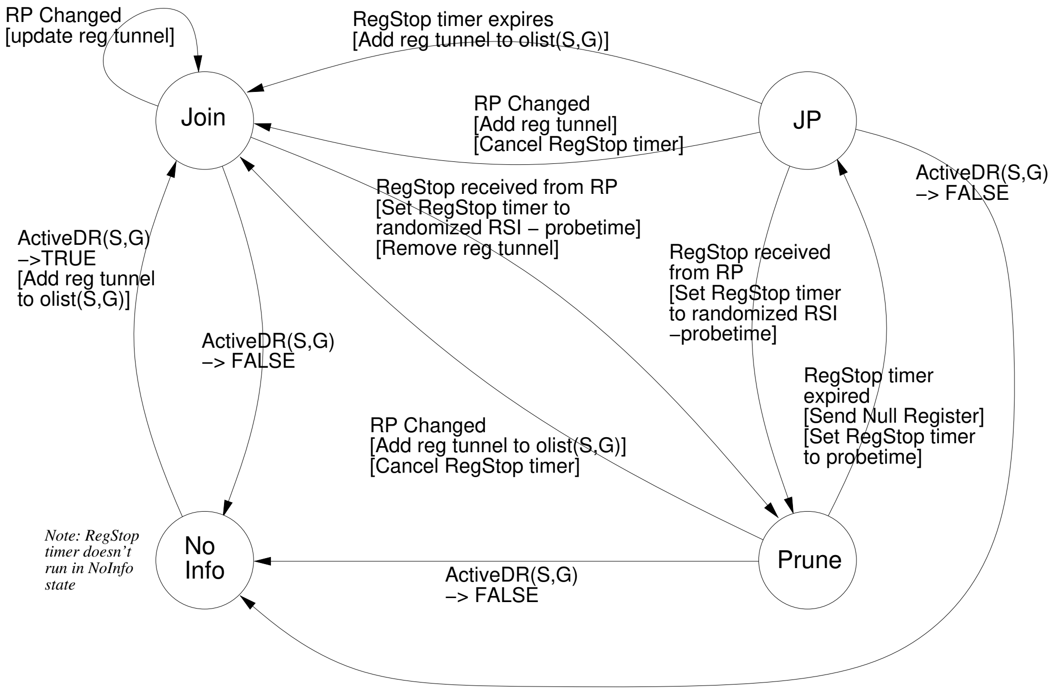

Figure 1: Per-(S,G) register state-machine at a DR

In tabular form, the state-machine is:

Prev State	Event				
	Register-Stop Timer expires	**CouldRegister –>True**	**CouldRegister –>False**	**Register-Stop received**	**RP changed**
No Info (NI)	-	-> J state add tunnel	-	-	-
Join (J)	-	-	-> NI state remove tunnel	-> P state remove tunnel; set Register-Stop timer(*)	-> J state redirect tunnel to new RP; stop Register-Stop timer
Join Pending (JP)	-> J state add tunnel	-	-> NI state remove tunnel	-> P state set Register-Stop timer(*)	-> J state redirect tunnel to new RP; stop Register-Stop timer
Prune (P)	-> JP state set Register-Stop timer(**); send null register	-	-> NI state remove tunnel	-	-> J state redirect tunnel to new RP; stop Register-Stop 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 "CouldRegister" in the state machine is defined as:

```
Bool CouldRegister(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 `CouldRegister(S,G)` in the register state machine, or the first packet from a source won't be registered.

Encapsulating data packets in the Register Tunnel

Conceptually, the Register Tunnel is an interface with a smaller MTU than the underlying IP interface towards the RP. IP fragmentation on packets forwarded on the Register Tunnel is performed based upon this smaller MTU. The encapsulating DR may perform Path-MTU Discovery to the RP to determine the effective MTU of the tunnel.

In IPv6, an ICMP Fragmentation Required message may be sent by the encapsulating DR.

Just like any forwarded packet, the TTL of the original data packet is decremented before it is encapsulated in the Register Tunnel.

Handling RegisterStop(*,G) Messages at the DR

An old RP might 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 {
            if( ! pkt.NullRegisterBit ) {
                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 * 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, at the RP, KeepaliveTimer(S,G) should be restarted to (3 * Register_Suppression_Time + Register_Probe_Time).

Just like any forwarded packet, the TTL of the original data packet is decremented after it is decapsulated from the Register Tunnel.

4.4. PIM Join/Prune Messages

A PIM Join/Prune message consists of a list of groups and a list of Joined and Pruned sources for each group. When processing a received Join/Prune message, each Joined or Pruned source for a Group is effectively considered individually, and applies to one or more of the following state machines. When considering a Join/Prune message whose PIM Destination field addresses this router, (*,*,RP) Joins and Prunes can affect the (*,*,RP) and (S,G,rpt) downstream state machines, (*,G) Joins and Prunes can affect both the (*,G) and (S,G,rpt) downstream state machines, while (S,G) and (S,G,rpt) Joins and Prunes can only affect their respective downstream state machines. When considering a Join/Prune message whose PIM Destination field addresses another router, most Join or Prune messages could affect each upstream state machine. *(... it's possible to enumerate this ...) XXX*

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

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

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

Join (J)
The interface has (*,*,RP) Join state which will cause us to forward packets destined for any group handled by RP from this interface except if there is also (S,G,rpt) prune information (see Section 4.4.4) or the router lost an assert on this interface.

PrunePending (PP)
The router has received a Prune(*,*,RP) 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 the state-machine uses two timers:

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

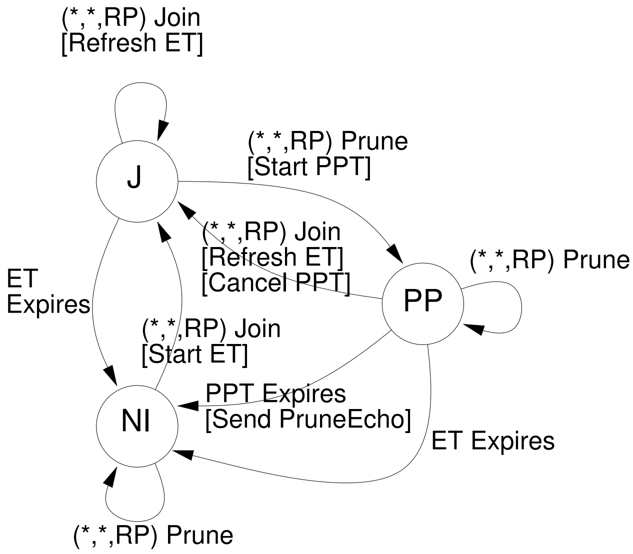

Figure 2: Downstream (*,*,RP) per-interface state-machine

In tabular form, the per-interface (*,*,RP) state-machine is:

<table>
<tr><th rowspan="2">Prev State</th><th colspan="4">Event</th></tr>
<tr><th>Receive Join(*,*,RP)</th><th>Receive Prune(*,*,RP)</th><th>Prune Pending Timer Expires</th><th>Expiry Timer Expires</th></tr>
<tr><td>NoInfo (NI)</td><td>-> J state
start Expiry Timer</td><td>-> NI state</td><td>-</td><td>-</td></tr>
<tr><td>Join (J)</td><td>-> J state
restart Expiry Timer</td><td>-> PP state
start Prune Pending Timer</td><td>-</td><td>-> NI state</td></tr>
<tr><td>Prune Pending (PP)</td><td>-> J state
restart Expiry Timer; stop Prune Pending Timer</td><td>-> PP state</td><td>-> NI state
Send Prune-Echo(*,*,RP)</td><td>-> NI state</td></tr>
</table>

The transition events "Receive Join(*,*,RP)" and "Receive Prune(*,*,RP)" 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.

Transitions from NoInfo State

When in NoInfo state, the following event may trigger a transition:

Receive Join(*,*,RP)

A Join(*,*,RP) is received on interface I with its IP destination set to the router's address on I.

The (*,*,RP) downstream state machine on interface I transitions to the Join state. The Expiry Timer (ET) is started, and set to the HoldTime from the triggering Join/Prune message.

Transitions from Join State

When in Join state, the following events may trigger a transition:

Receive Join(*,*,RP)

A Join(*,*,RP) is received on interface I with its IP destination set to the router's address on I.

The (*,*,RP) downstream state machine on interface I remains in Join state, and the Expiry Timer (ET) is restarted, set to maximum of its current value and the HoldTime from the triggering Join/Prune message.

Receive Prune(*,*,RP)

A Prune(*,*,RP) is received on interface I with its IP destination set to the router's address on I.

The (*,*,RP) downstream state machine on interface I transitions to the PrunePending state. The PrunePending timer 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.

Expiry Timer Expires

The Expiry Timer for the (*,*,RP) downstream state machine on interface I expires.

The (*,*,RP) downstream state machine on interface I transitions to the NoInfo state.

Transitions from PrunePending State

When in PrunePending state, the following events may trigger a transition:

Receive Join(*,*,RP)

A Join(*,*,RP) is received on interface I with its IP destination set to the router's address on I.

The (*,*,RP) downstream state machine on interface I transitions to the Join state. The PrunePending timer is canceled (without triggering an expiry event). The Expiry Timer is restarted, set to maximum of its current value and the HoldTime from the triggering Join/Prune message.

Expiry Timer Expires

The Expiry Timer for the (*,*,RP) downstream state machine on interface I expires.

The (*,*,RP) downstream state machine on interface I transitions to the NoInfo state.

PrunePending Timer Expires

The PrunePending Timer for the (*,*,RP) downstream state machine on interface I expires.

The (*,*,RP) downstream state machine on interface I transitions to the NoInfo state. A PruneEcho(*,*,RP) is sent onto the subnet connected to interface I.

The action "Send PruneEcho(*,*,RP)" is triggered when the router stops forwarding on an interface as a result of a prune. A PruneEcho(*,*,RP) is simply a Prune(*,*,RP) message sent by the upstream router to itself on a LAN. By "to itself" we mean that the Upstream Neighbor Address field of the enclosing Join/Prune message is set to the address of the sending router. 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(*,*,RP) need not be sent on a point-to-point interface.

4.4.2. 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 per-interface 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

4.4.4) 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 the state-machine uses 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.

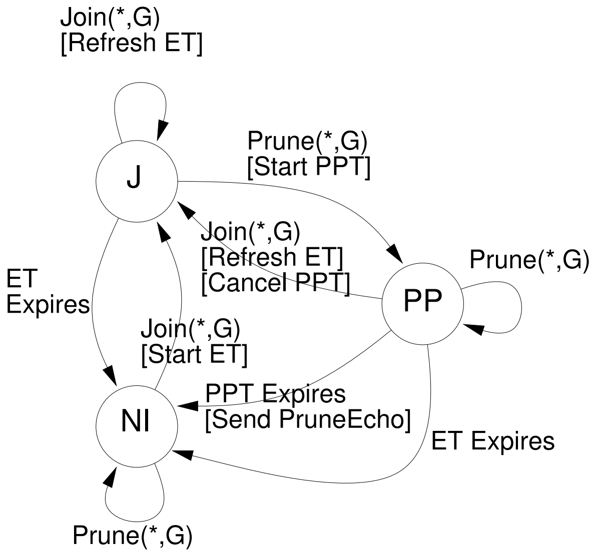

Figure 3: Downstream (*,G) per-interface state-machine

In tabular form, the per-interface (*,G) state-machine is:

Prev State	**Event**			
	Receive Join(*,G)	**Receive Prune(*,G)**	**Prune Pending Timer Expires**	**Expiry Timer Expires**
NoInfo (NI)	-> J state start Expiry Timer	-> NI state	-	-
Join (J)	-> J state restart Expiry Timer	-> PP state start Prune Pending Timer	-	-> NI state
Prune Pending (PP)	-> J state restart Expiry Timer; stop Prune Pending Timer	-> PP state	-> NI state Send Prune-Echo(*,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.

Transitions from NoInfo State

When in NoInfo state, the following event may trigger a transition:

Receive Join(*,G)

A Join(*,G) is received on interface I with its IP destination set to the router's address on I.

The (*,G) downstream state machine on interface I transitions to the Join state. The Expiry Timer (ET) is started, and set to the HoldTime from the triggering Join/Prune message.

Transitions from Join State

When in Join state, the following events may trigger a transition:

Receive Join(*,G)

A Join(*,G) is received on interface I with its IP destination set to the router's address on I.

The (*,G) downstream state machine on interface I remains in Join state, and the Expiry Timer (ET) is restarted, set to maximum of its current value and the HoldTime from the triggering Join/Prune message.

Receive Prune(*,G)

A Prune(*,G) is received on interface I with its IP destination set to the router's address on I.

The (*,G) downstream state machine on interface I transitions to the PrunePending state. The PrunePending timer 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.

Expiry Timer Expires

The Expiry Timer for the (*,G) downstream state machine on interface I expires.

The (*,G) downstream state machine on interface I transitions to the NoInfo state.

Transitions from PrunePending State

When in PrunePending state, the following events may trigger a transition:

Receive Join(*,G)

A Join(*,G) is received on interface I with its IP destination set to the router's address on I.

The (*,G) downstream state machine on interface I transitions to the Join state. The PrunePending timer is canceled (without triggering an expiry event). The Expiry Timer is restarted, set to maximum of its current value and the HoldTime from the triggering Join/Prune message.

Expiry Timer Expires

The Expiry Timer for the (*,G) downstream state machine on interface I expires.

The (*,G) downstream state machine on interface I transitions to the NoInfo state.

PrunePending Timer Expires

The PrunePending Timer for the (*,G) downstream state machine on interface I expires.

The (*,G) downstream state machine on interface I transitions to the NoInfo state. A PruneEcho(*,G) is sent onto the subnet connected to interface I.

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. By "to itself" we mean that the Upstream Neighbor Address field of the enclosing Join/Prune message is set to the address of the sending router. 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.

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

The per-interface 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.

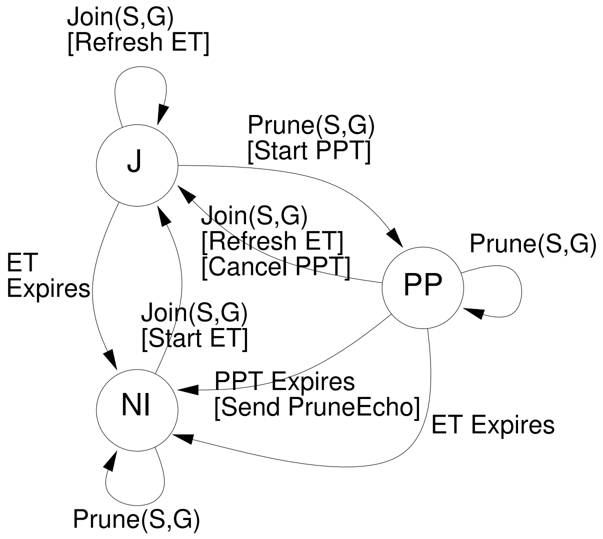

Figure 4: Downstream per-interface (S,G) state-machine

In tabular form, the state machine is:

Prev State	Event			
	Receive Join(S,G)	**Receive Prune(S,G)**	**Prune Pending Timer Expires**	**Expiry Timer Expires**
NoInfo (NI)	-> J state start Expiry Timer	-> NI state	-	-
Join (J)	-> J state restart Expiry Timer	-> PP state start Prune Pending Timer	-	-> NI state
Prune Pending (PP)	-> J state restart Expiry Timer; stop Prune Pending Timer	-> PP state	-> NI state Send Prune-Echo(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 PIM messages with a 0.0.0.0 destination address are also accepted.

Transitions from NoInfo State

When in NoInfo state, the following event may trigger a transition:

Receive Join(S,G)

A Join(S,G) is received on interface I with its IP destination set to the router's address on I.

The (S,G) downstream state machine on interface I transitions to the Join state. The Expiry Timer (ET) is started, and set to the HoldTime from the triggering Join/Prune message.

Transitions from Join State

When in Join state, the following events may trigger a transition:

Receive Join(S,G)

A Join(S,G) is received on interface I with its IP destination set to the router's address on I.

The (S,G) downstream state machine on interface I remains in Join state, and the Expiry Timer (ET) is restarted, set to maximum of its current value and the HoldTime from the triggering Join/Prune message.

Receive Prune(S,G)

A Prune(S,G) is received on interface I with its IP destination set to the router's address on I.

The (S,G) downstream state machine on interface I transitions to the PrunePending state. The PrunePending timer 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.

Expiry Timer Expires

The Expiry Timer for the (S,G) downstream state machine on interface I expires.

The (S,G) downstream state machine on interface I transitions to the NoInfo state.

Transitions from PrunePending State

When in PrunePending state, the following events may trigger a transition:

Receive Join(S,G)

A Join(S,G) is received on interface I with its IP destination set to the router's address on I.

The (S,G) downstream state machine on interface I transitions to the Join state. The PrunePending timer is canceled (without triggering an expiry event). The Expiry Timer is restarted, set to maximum of its current value and the HoldTime from the triggering Join/Prune message.

Expiry Timer Expires

The Expiry Timer for the (S,G) downstream state machine on interface I expires.

The (S,G) downstream state machine on interface I transitions to the NoInfo state.

PrunePending Timer Expires

The PrunePending Timer for the (S,G) downstream state machine on interface I expires.

The (S,G) downstream state machine on interface I transitions to the NoInfo state. A PruneEcho(S,G) is sent onto the subnet connected to interface I.

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. By "to itself" we mean that the Upstream Neighbor Address field of the enclosing Join/Prune message is set to the address of the sending router. 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.

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

The per-interface 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.

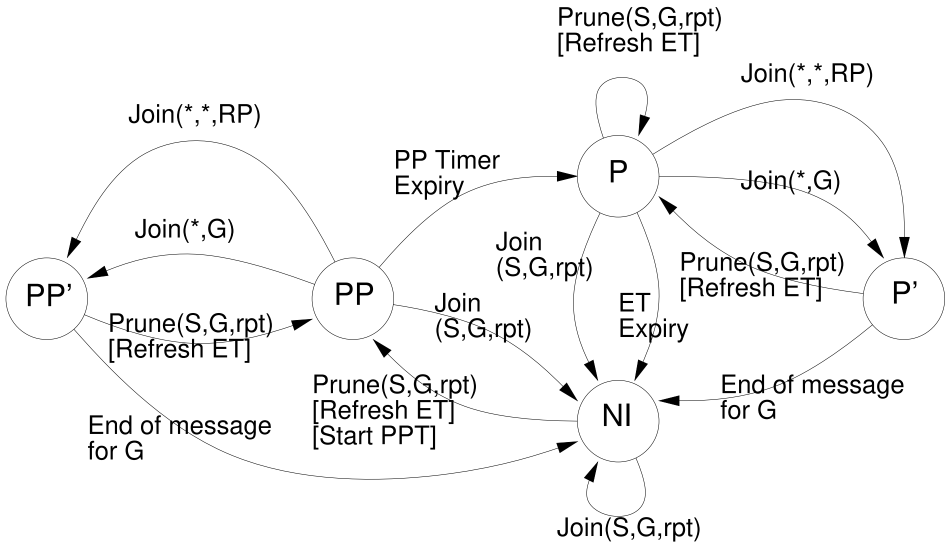

Figure 5: Downstream per-interface (S,G,rpt) state-machine

In tabular form, the state machine is:

Prev State	Event					
	Receive Join(*,G) or Join(*,*,RP(G))	**Receive Join (S,G,rpt)**	**Receive Prune (S,G,rpt)**	**End of Message**	**Prune Pending Timer Expires**	**Expiry Timer Expires**
No Info (NI)	-	-	-> PP state start Prune Pending Timer; start Expiry Timer Timer	-	n/a	n/a
Pruned (P)	-> P' state	-> NI state	-> P state restart Expiry Timer	-	n/a	**-> NI state**
Prune Pending (PP)	-> PP' state	-> NI state	-	-	-> P state	n/a
Temp. Pruned (P')	error	error	-> P state restart Expiry Timer	-> NI state	n/a	n/a
Temp. Prune Pending (PP')	error	error	-> PP state restart Expiry Timer	-> NI state	n/a	n/a

The transition events "Receive Join(S,G,rpt)", "Receive Prune(S,G,rpt)", "Receive Join(*,G)" and "Receive Join(*,*,RP(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.

Transitions from NoInfo State

When in NoInfo (NI) state, the following event may trigger a transition:

Receive Prune(S,G,rpt)

A Prune(S,G,rpt) is received on interface I with its IP destination set to the router's address on I.

The (S,G,rpt) downstream state machine on interface I transitions to the PrunePending state. The Expiry Timer (ET) is started, and set to the HoldTime from the triggering Join/Prune message. The PrunePending timer 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.

Transitions from PrunePending State

When in PrunePending (PP) state, the following events may trigger a transition:

Receive Join(*,G) or Join(*,*,RP(G))

A Join(*,*,RP(G)) or Join(*,G) is received on interface I with its IP destination set to the router's address on I.

The (S,G,rpt) downstream state machine on interface I transitions to Temp. PrunePending state whilst the remainder of the compound Join/Prune message containing the Join(*,*,RP(G)) or Join(*,G) is processed.

Receive Join(S,G,rpt)

A Join(S,G,rpt) is received on interface I with its IP destination set to the router's address on I.

The (S,G,rpt) downstream state machine on interface I transitions to NoInfo state. ET and PPT are canceled.

PrunePending Timer Expires

The PrunePending Timer for the (S,G,rpt) downstream state machine on interface I expires.

The (S,G,rpt) downstream state machine on interface I transitions to the Pruned state.

Transitions from Pruned State

When in Pruned (P) state, the following events may trigger a transition:

Receive Join(*,G) or Join(*,*,RP(G))

A Join(*,*,RP(G)) or Join(*,G) is received on interface I with its IP destination set to the router's address on I.

The (S,G,rpt) downstream state machine on interface I transitions to Temp. Pruned state whilst the remainder of the compound Join/Prune message containing the Join(*,*,RP(G)) or Join(*,G) is processed.

Receive Join(S,G,rpt)

A Join(S,G,rpt) is received on interface I with its IP destination set to the router's address on I.

The (S,G,rpt) downstream state machine on interface I transitions to NoInfo state. ET and PPT are canceled.

Receive Prune(S,G,rpt)

A Prune(S,G,rpt) is received on interface I with its IP destination set to the router's address on I.

The (S,G,rpt) downstream state machine on interface I remains in Pruned state. The Expiry Timer (ET) is restarted, set to maximum of its current value and the HoldTime from the triggering Join/Prune message.

Expiry Timer Expires

The Expiry Timer for the (S,G,rpt) downstream state machine on interface I expires.

The (S,G,rpt) downstream state machine on interface I transitions to the NoInfo state. ET and PPT are canceled.

Transitions from Temp. PrunePending State

When in Temp. PrunePending (PP') state and processing a compound Join/Prune message, the following events may trigger a transition:

Receive Prune(S,G,rpt)

The compound Join/Prune message contains a Prune(S,G,rpt).

The (S,G,rpt) downstream state machine on interface I transitions back to the PrunePending state. The Expiry Timer (ET) is restarted, set to maximum of its current value and the HoldTime from the triggering Join/Prune message.

End of Message

The end of the compound Join/Prune message is reached.

The (S,G,rpt) downstream state machine on interface I transitions to the NoInfo state. ET and PPT are canceled.

Transitions from Temp. Pruned State

When in Temp. Pruned (P') state and processing a compound Join/Prune message, the following events may trigger a transition:

Receive Prune(S,G,rpt)

The compound Join/Prune message contains a Prune(S,G,rpt).

The (S,G,rpt) downstream state machine on interface I transitions back to the Pruned state. The Expiry Timer (ET) is restarted, set to maximum of its current value and the HoldTime from the triggering Join/Prune message.

End of Message

The end of the compound Join/Prune message is reached.

The (S,G,rpt) downstream state machine on interface I transitions to the NoInfo state. ET and PPT are canceled.

Notes:

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

4.4.5. Sending (*,*,RP) Join/Prune Messages

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

If a router wishes to propagate a Join(*,*,RP) 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(*,*,RP) to the correct upstream neighbor, it should suppress its own Join(*,*,RP). If it sees a Prune(*,*,RP) to the correct upstream neighbor, it should be prepared to override that prune by sending a Join(*,*,RP) 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(*,*,RP) 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 (*,*,RP) 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(*,*,RP) is kept which is used to trigger the sending of a Join(*,*,RP) to the upstream next hop towards the RP, MRIB.next_hop(RP).

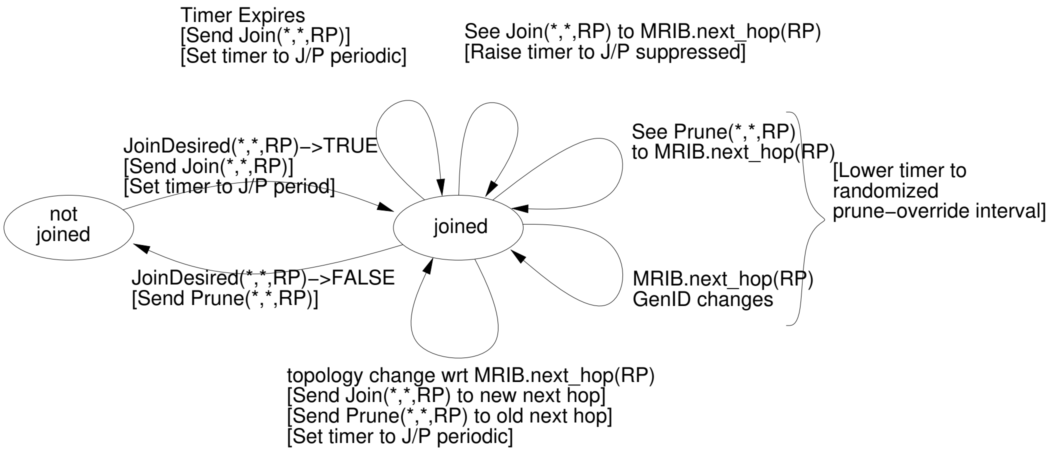

Figure 6: Upstream (*,*,RP) state-machine

In tabular form, the state machine is:

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

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

In Joined (J) State		
Timer Expires	**See Join(*,*,RP) to MRIB.next_hop(RP)**	**See Prune(*,*,RP) to MRIB.next_hop(RP)**
Send Join(*,*,RP); Set Timer to t_periodic	Increase Timer to t_suppressed	Decrease Timer to t_override

In Joined (J) State	
topology change wrt MRIB.next_hop(RP)	**MRIB.next_hop(RP) GenID changes**
Send Join(*,*,RP) to new next hop; Send Prune(*,*,RP) to old next hop; set Timer to t_periodic	Decrease Timer to t_override

This state machine uses the following macro:

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

JoinDesired(*,*,RP) is true when the router has received (*,*,RP) Joins from any downstream interface. Note that although JoinDesired is true, the router's sending of a Join(*,*,RP) message may be suppressed by another router sending a Join(*,*,RP) onto the upstream interface.

Transitions from NotJoined State

When the upstream (*,*,RP) state-machine is in NotJoined state, the following event may trigger a state transition:

JoinDesired(*,*,RP) becomes True

The downstream state for (*,*,RP) has changed so that at least one interface is in immediate_olist(*,*,RP), making JoinDesired(*,*,RP) become True.

The upstream (*,*,RP) state machine transitions to Joined state. Send Join(*,*,RP) to the appropriate upstream neighbor, which is MRIB.next_hop(RP). Set the Join Timer

(JT) to expire after t_periodic seconds.

Transitions from Joined State

When the upstream (*,*,RP) state-machine is in Joined state, the following events may trigger state transitions:

JoinDesired(*,*,RP) becomes False

The downstream state for (*,*,RP) has changed so no interface is in immediate_olist(*,*,RP), making JoinDesired(*,*,RP) become False.

The upstream (*,*,RP) state machine transitions to NotJoined state. Send Prune(*,*,RP) to the appropriate upstream neighbor, which is MRIB.next_hop(RP). Cancel the Join Timer (JT).

Join Timer Expires

The Join Timer (JT) expires, indicating time to send a Join(*,*,RP)

Send Join(*,*,RP) to the appropriate upstream neighbor, which is MRIB.next_hop(RP). Restart the Join Timer (JT) to expire after t_periodic seconds.

See Join(*,*,RP) to MRIB.next_hop(RP)

This event is only relevant if RPF_interface(RP) is a shared medium. This router sees another router on RPF_interface(RP) send a Join(*,*,RP) to MRIB.next_hop(RP). This causes this router to suppress its own Join.

The upstream (*,*,RP) state machine remains in Joined state. If the Join Timer is set to expire in less than t_suppressed seconds, reset it so that it expires after t_suppressed seconds. If the Join Timer is set to expire in more than t_suppressed seconds, leave it unchanged.

See Prune(*,*,RP) to MRIB.next_hop(RP)

This event is only relevant if RPF_interface(RP) is a shared medium. This router sees another router on RPF_interface(RP) send a Prune(*,*,RP) to MRIB.next_hop(RP). As this router is in Joined state, it must override the Prune after a short random interval.

The upstream (*,*,RP) state machine remains in Joined state. If the Join Timer is set to expire in more than t_override seconds, reset it so that it expires after t_override seconds. If the Join Timer is set to expire in less than t_override seconds, leave it unchanged.

Topology Change wrt MRIB.next_hop(RP)

A route changed in the routing base stored in the MRIB so that the next hop towards the RP is a different neighbor from before.

The upstream (*,*,RP) state machine remains in Joined state. Send Prune(*,*,RP) to the old upstream neighbor, which is the old value of MRIB.next_hop(RP). Send Join(*,*,RP) to the new upstream neighbor which is the new value of MRIB.next_hop(RP). Set the Join Timer (JT) to expire after t_periodic seconds.

MRIB.next_hop(RP) GenID changes

The Generation ID of the router that is MRIB.next_hop(RP) changes. This normally means that this neighbor has lost state, and so the state must be refreshed.

The upstream (*,*,RP) state machine remains in Joined state. If the Join Timer is set to expire in more than t_override seconds, reset it so that it expires after t_override seconds.

4.4.6. 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 its 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'(*,G).

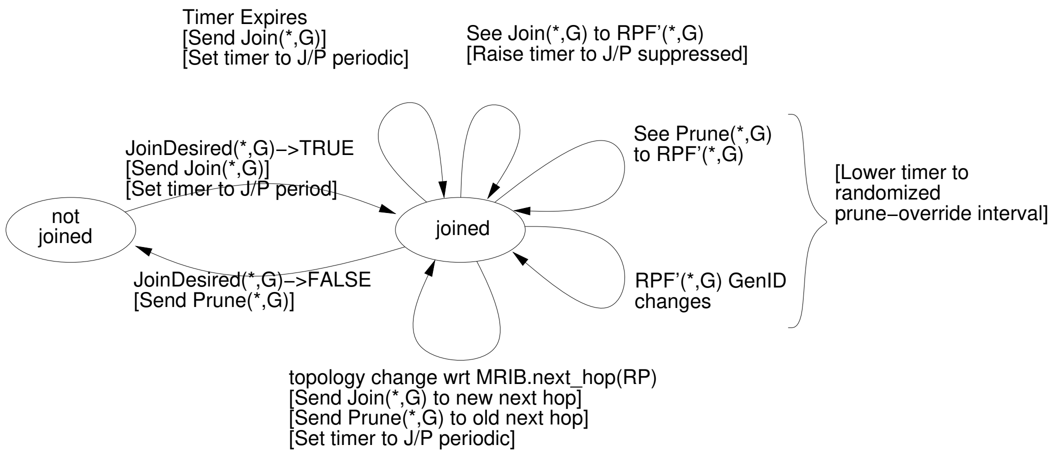

Figure 7: Upstream (*,G) state-machine

In tabular form, the state machine is:

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

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

In Joined (J) State			
Timer Expires	**See Join(*,G) to RPF'(*,G)**	**See Prune(*,G) to RPF'(*,G)**	**RPF'(*,G) changes**
Send Join(*,G); Set Timer to t_periodic	Increase Timer to t_suppressed	Decrease Timer to t_override	Decrease Timer to t_override

In Joined (J) State	
topology change wrt MRIB.next_hop(RP)	**RPF'(*,G) GenID changes**
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 immediate_olist(*,G) != NULL
       return TRUE
   else
       return FALSE
}
```

JoinDesired(*,G) is true when the router has forwarding state that would cause it to forward traffic for G using shared tree state. Note that although JoinDesired is true, the router's sending of a Join(*,G) message may be suppressed by another router sending a Join(*,G) onto the upstream interface.

Transitions from NotJoined State

When the upstream (*,G) state-machine is in NotJoined state, the following event may trigger a state transition:

JoinDesired(*,G) becomes True

The downstream state for (*,G) has changed so that at least one interface is in immediate_olist(*,G), making JoinDesired(*,G) become True.

The upstream (*,G) state machine transitions to Joined state. Send Join(*,G) to the appropriate upstream neighbor, which is RPF'(*,G). Set the Join Timer (JT) to expire after t_periodic seconds.

Transitions from Joined State

When the upstream (*,G) state-machine is in Joined state, the following events may trigger state transitions:

JoinDesired(*,G) becomes False

The downstream state for (*,G) has changed so no interface is in immediate_olist(*,G), making JoinDesired(*,G) become False.

The upstream (*,G) state machine transitions to NotJoined state. Send Prune(*,G) to the appropriate upstream neighbor, which is RPF'(*,G). Cancel the Join Timer (JT).

Join Timer Expires

The Join Timer (JT) expires, indicating time to send a Join(*,G)

Send Join(*,G) to the appropriate upstream neighbor, which is RPF'(*,G). Restart the Join Timer (JT) to expire after t_periodic seconds.

See Join(*,G) to RPF'(*,G)

This event is only relevant if RPF_interface(RP(G)) is a shared medium. This router sees another router on RPF_interface(RP(G)) send a Join(*,G) to RPF'(*,G). This causes this router to suppress its own Join.

The upstream (*,G) state machine remains in Joined state. If the Join Timer is set to expire in less than t_suppressed seconds, reset it so that it expires after t_suppressed seconds. If the Join Timer is set to expire in more than t_suppressed seconds, leave it unchanged.

See Prune(*,G) to RPF'(*,G)

This event is only relevant if RPF_interface(RP(G)) is a shared medium. This router sees another router on RPF_interface(RP(G)) send a Prune(*,G) to RPF'(*,G). As this router is in Joined state, it must override the Prune after a short random interval.

The upstream (*,G) state machine remains in Joined state. If the Join Timer is set to expire in more than t_override seconds, reset it so that it expires after t_override seconds. If the Join Timer is set to expire in less than t_override seconds, leave it unchanged.

RPF'(*,G) changes

The current net hop towards the RP changes due an Assert(*,G) on the RPF_interface(RP(G)).

The upstream (*,G) state machine remains in Joined state. If the Join Timer is set to expire in more than t_override seconds, reset it so that it expires after t_override seconds. If the Join Timer is set to expire in less than t_override seconds, leave it unchanged.

Topology Change wrt MRIB.next_hop(RP)

A route changed in the routing base stored in the MRIB so that the next hop towards the RP is a different neighbor from before.

The upstream (*,G) state machine remains in Joined state. Send Prune(*,G) to the old upstream neighbor, which is the old value of RPF'(*,G). Send Join(*,G) to the new upstream neighbor which is the new value of RPF(*,G). Note that the Join goes to RPF(*,G) and not RPF'(*,G) even if the new neighbor is on the same interface as the old one because the routing change may cause the assert state to be incorrect. Set the Join Timer (JT) to expire after t_periodic seconds.

RPF'(*,G) GenID changes

The Generation ID of the router that is RPF'(*,G) changes. This normally means that this neighbor has lost state, and so the state must be refreshed.

The upstream (*,G) state machine remains in Joined state. If the Join Timer is set to expire in more than t_override seconds, reset it so that it expires after t_override seconds.

4.4.7. 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,G).

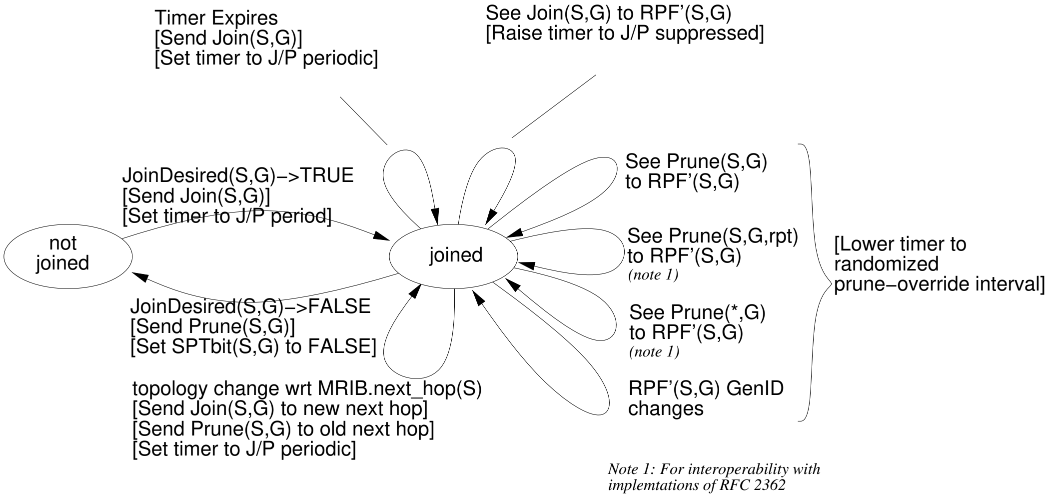

Figure 8: Upstream (S,G) state-machine

In tabular form, the state machine is:

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

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

In Joined (J) 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)**
Send Join(S,G); Set Timer to t_periodic	Increase Timer to t_suppr	Decrease Timer to t_override	Decrease Timer to t_override

In Joined (J) State		
See Prune(*,G) to RPF'(S,G)	**topology change wrt MRIB.next_hop(S)**	**RPF'(S,G) GenID changes**
Decrease Timer to t_override	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 ) )
}
```

JoinDesired(S,G) is true when the router has forwarding state that would cause it to forward traffic for G using source tree state. The source tree state can either be as a result of active source-specific join state, or the (S,G) keepalive timer and active non-source-specific state. Note that although JoinDesired is true, the router's sending of a Join(S,G) message may be suppressed by another router sending a Join(S,G) onto the upstream interface.

Transitions from NotJoined State

When the upstream (S,G) state-machine is in NotJoined state, the following event may trigger a state transition:

JoinDesired(S,G) becomes True

The downstream state for (S,G) has changed so that at least one interface is in inherited_olist(S,G), making JoinDesired(S,G) become True.

The upstream (S,G) state machine transitions to Joined state. Send Join(S,G) to the appropriate upstream neighbor, which is RPF'(S,G). Set the Join Timer (JT) to expire after t_periodic seconds.

Transitions from Joined State

When the upstream (S,G) state-machine is in Joined state, the following events may trigger state transitions:

JoinDesired(S,G) becomes False

The downstream state for (S,G) has changed so no interface is in inherited_olist(S,G), making JoinDesired(S,G) become False.

The upstream (S,G) state machine transitions to NotJoined state. Send Prune(S,G) to the appropriate upstream neighbor, which is RPF'(S,G). Cancel the Join Timer (JT).

Join Timer Expires

The Join Timer (JT) expires, indicating time to send a Join(S,G)

Send Join(S,G) to the appropriate upstream neighbor, which is RPF'(S,G). Restart the Join Timer (JT) to expire after t_periodic seconds.

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

This event is only relevant if RPF_interface(S) is a shared medium. This router sees another router on RPF_interface(S) send a Join(S,G) to RPF'(S,G). This causes this router to suppress its own Join.

The upstream (S,G) state machine remains in Joined state. If the Join Timer is set to expire in less than t_suppressed seconds, reset it so that it expires after t_suppressed seconds. If the Join Timer is set to expire in more than t_suppressed seconds, leave it unchanged.

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

This event is only relevant if RPF_interface(S) is a shared medium. This router sees another router on RPF_interface(S) send a Prune(S,G) to RPF'(S,G). As this router is in Joined state, it must override the Prune after a short random interval.

The upstream (S,G) state machine remains in Joined state. If the Join Timer is set to expire in more than t_override seconds, reset it so that it expires after t_override seconds.

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

This event is only relevant if RPF_interface(S) is a shared medium. This router sees another router on RPF_interface(S) send a Prune(S,G,rpt) to RPF'(S,G). If the upstream router is an RFC 2362 compliant PIM router, then the Prune(S,G,rpt) will cause it to stop forwarding. For backwards compatibility, this router should override the prune so that forwarding continues.

The upstream (S,G) state machine remains in Joined state. If the Join Timer is set to expire in more than t_override seconds, reset it so that it expires after t_override seconds.

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

This event is only relevant if RPF_interface(S) is a shared medium. This router sees another router on RPF_interface(S) send a Prune(*,G) to RPF'(S,G). If the upstream router is an RFC 2362 compliant PIM router, then the Prune(*,G) will cause it to stop forwarding. For backwards compatibility, this router should override the prune so that forwarding continues.

The upstream (S,G) state machine remains in Joined state. If the Join Timer is set to expire in more than t_override seconds, reset it so that it expires after t_override seconds.

RPF'(S,G) changes

The current net hop towards the RP changes due an Assert(S,G) on the RPF_interface(S).

The upstream (S,G) state machine remains in Joined state. If the Join Timer is set to expire in more than t_override seconds, reset it so that it expires after t_override seconds. If the Join Timer is set to expire in less than t_override seconds, leave it unchanged.

Topology Change wrt MRIB.next_hop(S)

A route changed in the routing base stored in the MRIB so that the next hop towards S is a different neighbor from before.

The upstream (S,G) state machine remains in Joined state. Send Prune(S,G) to the old upstream neighbor, which is the old value of RPF'(S,G). Send Join(S,G) to the new upstream neighbor which is the new value of RPF(S,G). Note that the Join goes to RPF(S,G) and not RPF'(S,G) even if the new neighbor is on the same interface as the old one because the routing change may cause the assert state to be incorrect. Set the Join Timer (JT) to expire after t_periodic seconds.

RPF'(S,G) GenID changes

The Generation ID of the router that is RPF'(S,G) changes. This normally means that this neighbor has lost state, and so the state must be refreshed.

The upstream (S,G) state machine remains in Joined state. If the Join Timer is set to expire in more than t_override seconds, reset it so that it expires after t_override seconds.

4.4.8. (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.

4.4.9. 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) or (*,*,RP) 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 neither (*,G) nor (*,*,RP(G)) join state at this router. If there is (*,G) or (*,*,RP(G)) join state at the router, then the state machine must be at one of the other two states:

Pruned(S,G,rpt)
: (*,G) or (*,*,RP(G)) Joined, but (S,G,rpt) pruned

NotPruned(S,G,rpt)
: (*,G) or (*,*,RP(G)) Joined, and (S,G,rpt) not pruned

RPTNotJoined(G)
: neither (*,G) nor (*,*,RP(G)) has not been joined.

In addition there is an (S,G,rpt) Override Timer, OT(S,G,rpt), which is used to delay triggered Join(S,G,rpt) messages to prevent implosions of triggered messages.

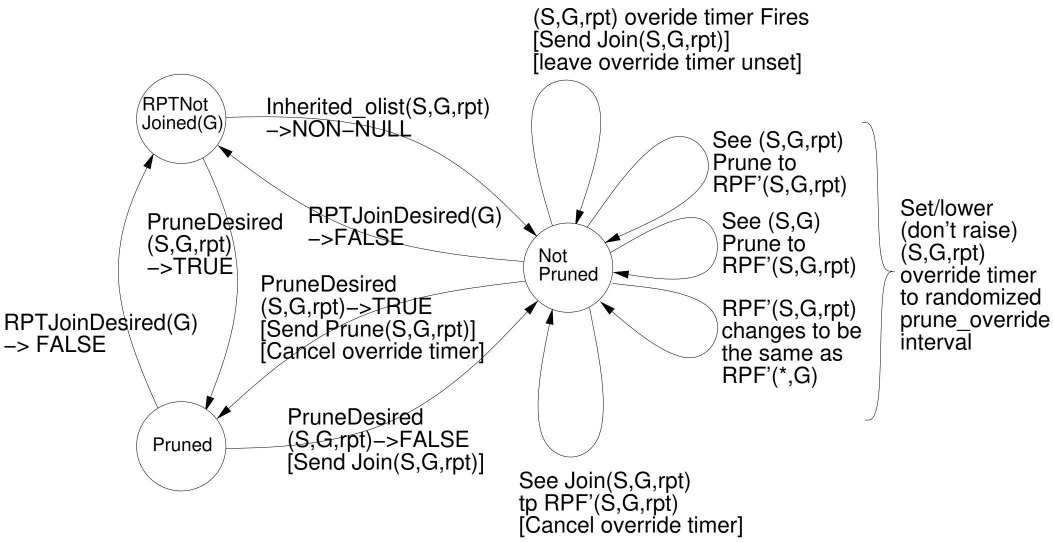

Figure 9: Upstream (S,G,rpt) state-machine for triggered messages

In tabular form, the state machine is:

Prev State	Event			
	PruneDesired (S,G,rpt) –>True	**PruneDesired (S,G,rpt) –>False**	**RPTJoinDesired(G) –>False**	**inherited_olist (S,G,rpt) –>non-NULL**
RPTNotJoined (G) (NJ)	-> P state	-	-	-> NP state
Pruned (S,G,rpt) (P)	-	-> NP state Send Join (S,G,rpt)	-> NJ state	-
NotPruned (S,G,rpt) (NP)	-> P state Send Prune (S,G,rpt); Stop OT timer	-	-> NJ state	-

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

In NotPruned(S,G,rpt) State				
OT 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)**
Send Join (S,G,rpt); Stop OT timer	OT timer = min(timer, t_po)	stop OT timer	OT timer = min(timer, t_po)	OT timer = min(timer, t_po)

Note that the `min` function in the above state machine considers a non-running timer to have an infinite value (e.g. `min(not-running, t_po)` = `t_po`).

This state machine uses the following macros:

```
bool RPTJoinDesired(G) {
  return (JoinDesired(*,G) || JoinDesired(*,*,RP(G)))
}
```

RPTJoinDesired(G) is true when the router has forwarding state that would cause it to forward traffic for G using either (*,G) or (*,*,RP) shared tree state.

```
bool PruneDesired(S,G,rpt) {
     return ( RPTJoinDesired(G) AND
                ( inherited_olist(S,G,rpt) == NULL
                  OR (SPTbit(S,G)==TRUE
                       AND (RPF'(*,G) != RPF'(S,G)) )))
}
```

PruneDesired(S,G,rpt) can only be true if RPTJoinDesired(G) is true. If RPTJoinDesired(G) is true, then PruneDesired(S,G,rpt) is true if either there are no outgoing interfaces that S would be forwarded on, or if the router has active (S,G) forwarding state but RPF'(*,G) != RPF'(S,G).

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) override 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 override 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 override 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 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) or (*,*,RP(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) override timer 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 RPTNotJoined(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) or Join(*,*,RP).

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 has RPTJoinDesired(G) true, or it now wishes to receive traffic from S again. If it is the former, then this transition should not happen, but instead the "RPTJoinDesired(G)->FALSE" transition should happen. Thus this transition should be interpreted as "PruneDesired(S,G,rpt)->FALSE AND RPTJoinDesired(G)==TRUE"

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

RPTJoinDesired(G)->FALSE

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

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

This transition is only relevant in the RPTNotJoined(G) state.

The router has joined the RP tree (handled by the (*,G) or (*,*,RP) upstream state machine as appropriate), 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 10. 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 out of 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.

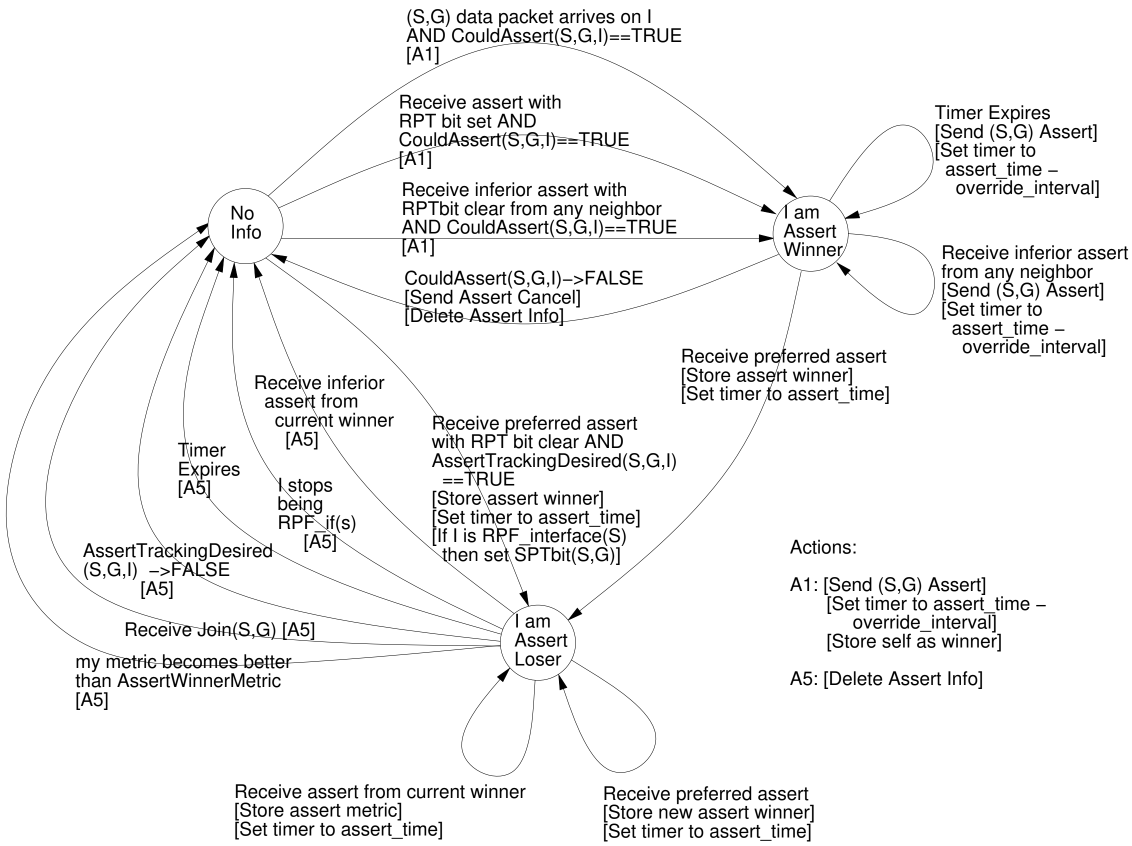

Figure 10: Per-interface (S,G) Assert State-machine

In tabular form the state machine is:

In NoInfo (NI) State			
Receive Inferior Assert with RPTbit clear and CouldAssert (S,G,I)	**Receive Assert with RPTbit set and CouldAssert (S,G,I)**	**Data arrives from S to G on I and CouldAssert (S,G,I)**	**Receive Preferred Assert with RPTbit clear and AssTrDes (S,G,I)**
-> W state [Actions A1]	-> W state [Actions A1]	-> W state [Actions A1]	-> L state [Actions A6]

In I Am Assert Winner (W) State			
Timer Expires	**Receive Inferior Assert**	**Receive Preferred Assert**	**CouldAssert (S,G,I) –> FALSE**
-> W state [Actions A3]	-> W state [Actions A3]	-> L state [Actions A2]	-> NI state [Actions A4]

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

In I Am Assert Loser (L) State			
my_metric -> better than winner's metric	**RPF interface stops being I**	**Receive Join(S,G) on interface I**	**Receive Assert from Current Winner**
-> NI state [Actions A5]	-> NI state [Actions A5]	-> NI State [Actions A5]	-> L state [Actions A2]

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 `my_assert_metric(S,G,I)`.

The state machine uses the following macros:

```
CouldAssert(S,G,I) =
     SPTbit(S,G)==TRUE
     AND (RPF_interface(S) != I)
     AND (I in ( ( joins(*,G) (-) prunes(S,G,rpt) )
                 (+) ( pim_include(*,G) (-) pim_exclude(S,G) )
                 (-) lost_assert(*,G)
                 (+) joins(S,G) (+) pim_include(S,G) ) )
```

CouldAssert(S,G,I) is true for downstream interfaces which would be in the inherited_olist(S,G) if (S,G) assert information was not taken into account.

```
AssertTrackingDesired(S,G,I) =
     (I in ( ( joins(*,G) (-) prunes(S,G,rpt) )
             (+) ( pim_include(*,G) (-) pim_exclude(S,G) )
             (-) lost_assert(*,G)
             (+) joins(S,G) (+) pim_include(S,G) ) )
     OR (RPF_interface(S)==I AND JoinDesired(S,G)==TRUE)
     OR (RPF_interface(RP)==I AND JoinDesired(*,G)==TRUE
         AND SPTbit(S,G)==FALSE)
```

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

The first three 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 last three lines account for the fact that a router must keep track of assert information on upstream interfaces in order to send joins and prunes to the proper neighbor.

Transitions from NoInfo State

When in NoInfo state, the following events may trigger transitions:

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 assert 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 A2 (below).

Transitions from Winner State

When in "I am Assert Winner" state, the following events trigger transitions:

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_assert_metric(S,G,I) has changed so that now my assert metric for (S,G) is better than the metric we have stored for current assert winner. This might happen the underlying routing metric changes, or when when CouldAssert(S,G,I) becomes true; for example, when SPTbit(S,G) becomes true. 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)
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
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

A6: Store new assert winner
Set timer to Assert_Time
If I is RPF_interface(S) Set SPTbit(S,G) to TRUE.

4.5.2. (*,G) Assert Message State Machine

The (*,G) Assert state-machine for interface I is shown in Figure 11. 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 the (*,G) 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.

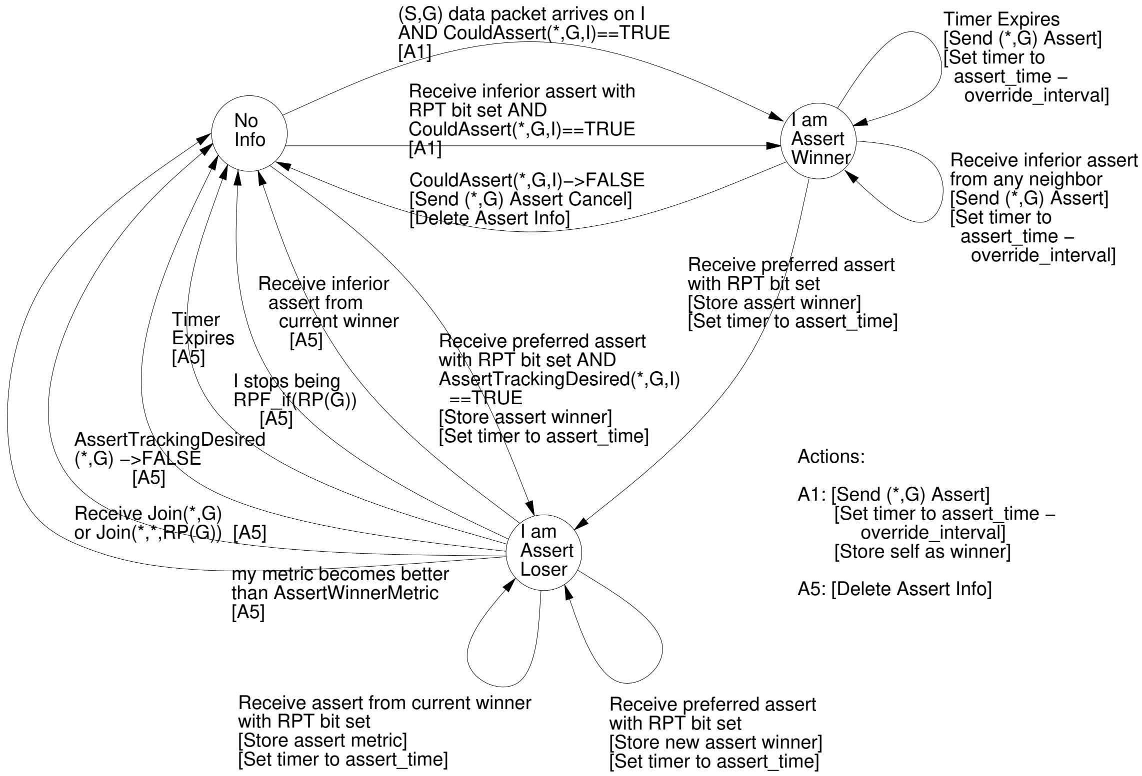

Figure 11: (*,G) Assert State-machine

In tabular form the state machine is:

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

In I Am Assert Winner (W) State			
Timer Expires	**Receive Inferior Assert**	**Receive Preferred Assert**	**CouldAssert (*,G,I) –> FALSE**
-> W state [Actions A3]	-> W state [Actions A3]	-> L state [Actions A2]	-> NI state [Actions A4]

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

In I Am Assert Loser (L) State		
my_metric -> better than Winner's metric	**RPF interface stops being I**	**Receive Join(*,G) or Join(*,*,RP(G)) on Interface I**
-> 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) (+) joins(*,*,RP(G))
             (+) pim_include(*,G)) )
    AND RPF_interface(RP(G)) != I
```

CouldAssert(*,G,I) is true on downstream interfaces for which we have (*,G) or (*,*,RP(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 RPTJoinDesired(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 my_assert_metric(S,G).

Transitions from NoInfo State

When in NoInfo state, the following events trigger transitions, but only if the (S,G) assert state machine is in NoInfo state:

Receive Inferior Assert with RPTbit set AND CouldAssert(*,G,I)==TRUE
An Inferior (*,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 a 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 A2 (below).

Transitions from Winner State

When in "I am Assert Winner" state, the following events trigger transitions, but 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 has lost, 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 events trigger transitions:

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 events trigger transitions, but 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 events trigger transitions:

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 metric, rpt_assert_metric(G,I), has 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) or Join(*,*,RP(G))
We receive a Join(*,G) or a Join(*,*,RP(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)
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 in order, where the first lower value wins. If all fields are equal, the IP address of the router that sourced the Assert message 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( CouldAssert(S,G,I) == TRUE ) {
        return spt_assert_metric(S,G,I)
    } else if( CouldAssert(*,G,I) == TRUE ) {
        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 send its own 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.

AssertCancel messages are simply an optimization. The original Assert timeout mechanism will allow a subnet to eventually become consistent; the AssertCancel mechanism simply causes faster convergence. No special processing is required for an AssertCancel message, since it is simply an Assert message from the current winner.

4.5.5. Assert State Macros

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

```
bool lost_assert(S,G,rpt,I) {
  if ( RPF_interface(RP) == I  OR
       ( RPF_interface(S) == I AND SPTbit(S,G) == TRUE ) ) {
     return FALSE
  } else {
     return ( AssertWinner(S,G,I) != me )
  }
}

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

bool lost_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 switching 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.

   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) or a Join(*,*,RP(G)) directed to it cancels the (*,G) assert timer and all (S,G) assert timers that do not have corresponding Prune(S,G,rpt) messages in the compound Join/Prune message.

   Rules 7 and 8 help convergence during topology changes.

8. An assert winner for (*,G) or (S,G) sends a canceling assert when it is about to stop forwarding on a (*,G) or an (S,G) entry. 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. When PIM is enabled on an interface or a router first starts, the hello timer is set to a random value between 0 and `Triggered_Hello_Delay` 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.

To allow new or rebooting routers to learn of PIM neighbors quickly, when a Hello message is received from a new neighbor, or a Hello message with a new GenID is received from an existing neighbor, a new Hello message should be sent on this interface after a randomized delay between 0 and `Triggered_Hello_Delay`. This triggered message need not change the timing of the scheduled periodic message.

When an interface goes down or changes IP address, a Hello message with a zero Hold Time should be sent immediately (with the old IP address if the IP address changed). This will cause PIM neighbors to remove this neighbor (or its old IP address) immediately.

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 n on I for which n.dr_priority_present
            is false ) {
        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

For correct operation, every PIM router within a PIM domain must be able to map a particular multicast group address to the same RP. If this is not the case then black holes may appear, where some receivers in the domain cannot receive some groups. 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.

A notable exception to this is where a PIM domain is broken up into multiple administrative scope regions - these are regions where a border has been configured so that a range of multicast groups will not be forwarded across that border. For more information on Administratively Scoped IP Multicast, see RFC 2365. The modified criteria for admin-scoped regions are that the region is convex with respect to forwarding based on the MRIB, and that all PIM routers within the scope region map scoped groups to the same RP within that region.

This specification does not mandate the use of a single mechanism to provide routers with the information to perform the group-to-RP mapping. Currently three mechanisms are possible, and all three have associated problems:

Static Configuration
A PIM router MUST support the static configuration of group-to-RP mappings. Such a mechanism is not robust to failures, but does at least provide a basic interoperability mechanism.

Cisco's Auto-RP
Auto-RP uses a PIM Dense-Mode multicast group to announce group-to-RP mappings from a central location. This mechanism is not useful if PIM Dense-Mode is not being run in parallel with PIM Sparse-Mode, and was only intended for use with PIM Sparse-Mode Version 1. No standard specification currently exists.

BootStrap Router (BSR)
RFC 2362 specifies a bootstrap mechanism based around the automatic election of a bootstrap router (BSR). Any router in the domain that is configured to be a possible RP reports its candidacy to the BSR, and then a domain-wide flooding mechanism distributes the BSR's chosen set of RPs throughout the domain. As specified in RFC 2362, BSR is flawed in its handling of admin-scoped regions that are smaller than a PIM domain, but the mechanism does work for global-scoped groups.

As far as PIM-SM is concerned, the only important requirement is that all routers in the domain (or admin scope zone for scoped regions) receive the same set of group-range-to-RP mappings. This may be achieved through the use of any of these mechansms, or through alternative mechanisms not currently specified.

Any RP address configured or learned MUST be a domain-wide reachable address.

4.7.1. Group-to-RP Mapping

Using one of the mechanisms described above, a PIM router receives one or more possible group-range-to-RP mappings. Each mapping specifies a range of multicast groups (expressed as a group and mask) and the RP to which such groups should be mapped. Each mapping may also have an associated priority. It is possible to receive multiple mappings all of which might match the same multicast group - this is the common case with BSR. The algorithm for performing the group-to-RP mapping is as follows:

1 Perform longest match on group-range to obtain a list of RPs.

2 From this list of matching RPs, find the one with highest priority. Eliminate any RPs from the list that have lower priorities.

3 If only one RP remains in the list, use that RP.

4 If multiple RPs are in the list, use the PIM hash function to choose one.

Thus if two or more group-range-to-RP mappings cover a particular group, the one with the longest mask is the mapping to use. If the mappings have the same mask length, then the one with the highest priority is chosen. If there is more than one matching entry with the same longest mask and the priorities are identical, then a hash function (see Section 4.7.2) is applied to choose the RP.

This algorithm is invoked by a DR when it needs to determine an RP for a given group, e.g. upon reception of a packet or IGMP membership indication for a group for which the DR does not know the RP. 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 mapping function is invoked by all routers upon receiving a (*,G) or (*,*,RP) Join/Prune message.

Note that if the set of possible group-range-to-RP mappings changes, each router will need to check whether any existing groups are affected. This may, for example, cause a DR or acting DR to re-join a group, or cause it to re-start register encapsulation to the new RP.

4.7.2. Hash Function

The hash function is used by all routers within a domain, to map a group to one of the RPs from the matching set of group-range-to-RP mappings (this set all have the same longest mask length and same highest priority). The algorithm takes as input the group address, and the addresses of the candidate RPs from the mappings, 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 matching group-range-to-RP mappings, 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) and G must first be derived from the actual RP or group 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 exclusive-or of each 32-bit segment of the address for all other IPv6 addresses. For example, the digest of the IPv6 address 3ffe:b00:c18:1::10 would be computed as 0x3ffe0b00 ^ 0x0c180001 ^ 0x00000000 ^ 0x00000010, where ^ represents the exclusive-or operation.

2 The candidate RP 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 are broken in advantage of the highest address.

4.8. Source-Specific Multicast

The Source-Specific Multicast (SSM) service model [11] can be implemented with a strict subset of the PIM-SM protocol mechanisms. Both regular IP Multicast and SSM semantics can coexist on a single router and both can be implemented using the PIM-SM protocol. A range of multicast addresses, currently 232.0.0.0/8 in IPv4, is reserved for SSM, and the choice of semantics is determined by the multicast group address in both data packets and PIM messages.

4.8.1. Protocol Modifications for SSM destination addresses

The following rules override the normal PIM-SM behavior for a multicast address G in the SSM reserved range:

- A router MUST NOT send a (*,G) Join/Prune message for any reason.
- A router MUST NOT send an (S,G,rpt) Join/Prune message for any reason.
- A router MUST NOT send a Register message for any packet that is destined to an SSM address.
- A router MUST NOT forward packets based on (*,G) or (S,G,rpt) state. The (*,G) and (S,G,rpt) -related state summarization macros are NULL for any SSM address, for the purposes of packet forwarding.
- A router acting as an RP MUST NOT forward any Register-encapsulated packet that has an SSM destination address.

The last two rules are present to deal with "legacy" routers unaware of SSM that may be sending (*,G) and (S,G,rpt) Join/Prunes, or Register messages for SSM destination addresses.

Additionally:

- A router MAY be configured to advertise itself as a Candidate RP for an SSM address. If so, it SHOULD respond with a Register-Stop message to any Register message containing a packet destined for an SSM address.
- A router MAY optimize out the creation and maintenance of (S,G,rpt) and (*,G) state for SSM destination addresses -- this state is not needed for SSM packets.

4.8.2. PIM-SSM-only Routers

An implementor may choose to implement only the subset of PIM Sparse-Mode that provides SSM forwarding semantics.

A PIM-SSM-only router MUST implement the following portions of this specification:

- Upstream (S,G) state machine (Section 4.4.7)
- Downstream (S,G) state machine (Section 4.4.3)
- (S,G) Assert state machine (Section 4.5.1)
- Hello messages, neighbor discovery and DR election (Section 4.6)
- Packet forwarding rules (Section 4.2)

A PIM-SSM-only router does not need to implement the following protocol elements:

- Register state machine (Section 4.3)
- (*,G), (S,G,rpt) and (*,*,RP) Downstream state machines (Sections 4.4.2, 4.4.4, and 4.4.1)
- (*,G), (S,G,rpt), and (*,*,RP) Upstream state machines (Sections 4.4.6, 4.4.8, and 4.4.5)
- (*,G) Assert state machine (Section 4.5.2)
- Bootstrap RP Election (Section 4.7)
- Keepalive Timer
- SptBit (Section 4.2.1)

The `KeepaliveTimer` should be treated as always running and `SptBit` should be treated as being always set for an SSM address. Additionally, the Packet forwarding rules of Section 4.2

can be simplified in a PIM-SSM-only router:

```
if( iif == RPF_interface(S) AND UpstreamJPState(S,G) == Joined ) {
    oiflist = inherited_olist(S,G)
} else if( iif is in inherited_olist(S,G) ) {
    send Assert(S,G) on iif
}

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

This is nothing more than the reduction of the normal PIM-SM forwarding rule, with all (S,G,rpt) and (*,G) clauses replaced with NULL.

4.9. PIM Packet Formats

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

All PIM control messages have IP protocol number 103.

PIM messages are either unicast (e.g. Registers and Register-Stop), or multicast with TTL 1 to the ‘ALL-PIM-ROUTERS’ group (e.g. Join/Prune, Asserts, etc.). The source address used for unicast messages is a domain-wide reachable address; the source address used for multicast messages is the link-local address of the interface on which the message is being sent.

The IPv4 ‘ALL-PIM-ROUTERS’ group is ‘224.0.0.13’. The IPv6 ‘ALL-PIM-ROUTERS’ group is ‘ff02::d’.

```
 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            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

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 on transmission. Ignored upon receipt.

Checksum

The checksum is a 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.

For IPv6, the checksum also includes the IPv6 "pseudo-header", as specified in RFC 2460, section 8.1 [8]. This "pseudo-header" is prepended to the PIM header for the purposes of calculating the checksum. The "Upper-Layer Packet Length" in the pseudo-header is set to the length of the PIM message. The Next Header value used in the pseudo-header is 103. If the packet's length is not an integral number of 16-bit words, the packet is padded with a byte of zero before performing the checksum.

4.9.1. Encoded Source and Group Address Formats

Encoded-Unicast address

An Encoded-Unicast address takes the following format:

```
 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
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Addr Family  | Encoding Type |     Unicast Address
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+...
```

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 [5]. Values 128-250 are reserved to be assigned by the IANA for PIM-specific Address Families. Values 251 though 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 addresses take the following format:

```
 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
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Addr Family  | Encoding Type |   Reserved    |  Mask Len     |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                Group multicast Address
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+...
```

Addr Family

described above.

Encoding Type
described above.

Reserved
Transmitted as zero. Ignored upon receipt.

Mask Len
The Mask length field is 8 bits. The value is the number of contiguous one bits left justified used as a mask which, combined with the group address, describes a range of groups. 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:

```
 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
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| Addr Family   | Encoding Type | Rsrvd   |S|W|R|  Mask Len     |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                        Source Address
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-...
```

Addr Family
described above.

Encoding Type
described above.

Reserved
Transmitted as zero, ignored on receipt.

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

W The WC (or WildCard) bit is a 1 bit value for use with PIM Join/Prune messages (see section 4.9.5.1).

R The RPT (or Rendezvous Point Tree) bit is a 1 bit value for use with PIM Join/Prune messages (see section 4.9.5.1). If the WC bit is 1, the RPT bit MUST be 1.

Mask Len

The mask length field is 8 bits. The value is the number of contiguous one bits left justified used as a mask which, combined with the Source Address, describes a source subnet. 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 source 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.

4.9.2. Hello Message Format

It is sent periodically by routers on all interfaces.

```
 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            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|          OptionType           |         OptionLength          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                          OptionValue                          |
|                              ...                              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                               .                               |
|                               .                               |
|                               .                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|          OptionType           |         OptionLength          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                          OptionValue                          |
|                              ...                              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

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: Hold Time

```
 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
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|          Type = 1             |           Length = 2          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|         Hold Time             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Hold Time 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 dial-on-demand links, 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 18: deprecated and should not be used.
* OptionType 19: DR Priority

```
 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
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|          Type = 19            |           Length = 4          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                         DR Priority                           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

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: Generation ID

```
 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
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|          Type = 20            |           Length = 4          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                       Generation ID                           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

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.

OptionTypes 17 thru 65000 are assigned by the IANA. OptionTypes 65001 through 65535 are reserved for Private Use, as defined in [6].
Unknown options may be ignored. The "Hold Time" option MUST be implemented; the "DR Priority" and "Generation ID" options SHOULD be implemented.

4.9.3. 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. The IP source address is set to the address of the DR, the destination address to the RP's address. The IP TTL of the PIM packet is the system's normal unicast TTL.

```
 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            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|B|N|                         Reserved2                         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
.                     Multicast data packet                     .
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

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 a checksum calculated 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.

Reserved2
Transmitted as zero, ignored on receipt.

Multicast data packet
The original packet sent by the source. This packet must be the of the same address family as the encapsulating PIM packet, e.g. an IPv6 data packet must be encapsulated in an IPv6 PIM packet. Note that the TTL of the original packet is decremented before encapsulation, just like any other packet that is forwarded. In addition, the RP decrements the TTL after decapsulating, before forwarding the packet down the shared tree.

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.

4.9.4. Register-Stop Message Format

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

```
 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            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|             Group Address (Encoded-Group format)              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|            Source Address (Encoded-Unicast format)            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

PIM Version, Type, Reserved, Checksum
Described above.

Group Address
The group address from the multicast data packet in the Register. Format described in section 4.9.1. Note that for Register-Stops the Mask Len field contains the full address length * 8 (e.g. 32 for IPv4 native encoding), if the message is sent for a single group.

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 section 4.9.1. A special wild card value (0's), can be used to indicate any source. XXX note that "(0's)" doesn't really describe what the rest of the field in encoded-unicast-address should be

4.9.5. 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            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|        Upstream Neighbor Address (Encoded-Unicast format)     |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Reserved     | Num groups    |          Holdtime             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|         Multicast Group Address 1 (Encoded-Group format)      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|   Number of Joined Sources    |   Number of Pruned Sources    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|        Joined Source Address 1 (Encoded-Source format)        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                             .                                 |
|                             .                                 |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|        Joined Source Address n (Encoded-Source format)        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|        Pruned Source Address 1 (Encoded-Source format)        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                             .                                 |
|                             .                                 |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|        Pruned Source Address n (Encoded-Source format)        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                           .                                   |
|                           .                                   |
|                           .                                   |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|         Multicast Group Address m (Encoded-Group format)      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|   Number of Joined Sources    |   Number of Pruned Sources    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|        Joined Source Address 1 (Encoded-Source format)        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                             .                                 |
|                             .                                 |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|        Joined Source Address n (Encoded-Source format)        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|        Pruned Source Address 1 (Encoded-Source format)        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                             .                                 |
|                             .                                 |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|        Pruned Source Address n (Encoded-Source format)        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

PIM Version, Type, Reserved, Checksum
Described above.

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 section 4.9.1. This address should be the link-local address of the upstream neighbor, as obtained from the RPF lookup.

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 should hold the state until canceled by the appropriate cancelling Join/Prune message, or timed out according to local policy. This may be used with dial-on-demand links, to avoid keeping the link up with periodic Join/Prune messages.

Note that the HoldTime must be larger than the J/P_Override_Interval.

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

Multicast group address
For format description see Section 4.9.1.

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 Encoded-Source-Address format in section 4.9.1.

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.

4.9.5.1. Group Set Source List Rules

As described above, Join / Prune messages are composed by one or more group sets. Each set contains two source lists, the Join Sources and the Prune Sources. This section describes the different types of group sets and source list entries that can exist in a Join / Prune message.

There are two valid group set types:

Wildcard Group Set

The wildcard group set is represented by the entire multicast range - the beginning of the multicast address range in the group address field and the prefix length of the multicast address range in the mask length field of the Multicast Group Address, e.g. 224.0.0.0/4 for IPv4. Each wildcard group set may contain one or more (*,*,RP) source list entries in either the Join or Prune lists.

A (*,*,RP) source list entry may only exist in a wildcard group set. When added to a Join source list, this type of source entry expresses the routers interest in receiving traffic for all groups mapping to the specified RP. When added to a Prune source list a (*,*,RP) entry expresses the routers interest to stop receiving such traffic.

(*,*,RP) source list entries have the Source-Address set to the address of the RP, the Source-Address Mask-Len set to the full length of the IP address and both the WC and RPT bits of the Source-Address set to 1.

Group Specific Set

For IPv4, a Group Specific Set is represented by a valid IP multicast address in the group address field and the full length of the IP address in the mask length field of the Multicast Group Address. Each group specific set may contain (*,G), (S,G,rpt) and (S,G) source list entries in the Join or Prune lists.

(*,G)

The (*,G) source list entry is used in Join / Prune messages sent towards the RP for the specified group. It expresses interest (or lack of) in receiving traffic sent to the group through the Rendezvous-Point shared tree. There may only be one such entry in both the Join and Prune lists of a group specific set.

(*,G) source list entries have the Source-Address set to the address of the RP for group G, the Source-Address Mask-Len set to the full length of the IP address and have both the WC and RPT bits of the Encoded-Source-Address set.

(S,G,rpt)

The (S,G,rpt) source list entry is used in Join / Prune messages sent towards the RP for the specified group. It expresses interest (or lack of) in receiving traffic through the shared tree sent by the specified source to this group. There may only be one such entry for each source address in both the Join and Prune lists of a group specific set.

(S,G,rpt) source list entries have the Source-Address set to the address of the source S, the Source-Address Mask-Len set to the full length of the IP address and have the WC bit clear and the RPT bit set in the Encoded-Source-Address.

(S,G)

The (S,G) source list entry is used in Join / Prune messages sent towards the specified source. It expresses interest (or lack of) in receiving traffic through the shortest path tree sent by the source to the specified group. There may only be one such entry for each source address in both the Join and Prune lists of a group specific set.

(S,G) source list entries have the Source-Address set to the address of the source S, the Source-Address Mask-Len set to the full length of the IP address and have both the WC and RPT bits of the Encoded-Source-Address clear.

The rules described above are sufficient to prevent invalid combinations of source list entries in group-specific sets. There are however a number of combinations that have a valid interpretation but which are not generated by the protocol as described in this specification:

- Combining a (*,G) Join and a (S,G,rpt) Join entry in the same message is redundant as the (*,G) entry covers the information provided by the (S,G,rpt) entry.
- The same applies for a (*,G) Prunes and (S,G,rpt) Prunes.
- The combination of a (*,G) Prune and a (S,G,rpt) Join is also not generated. (S,G,rpt) Joins are only sent when the router is receiving all traffic for a group on the shared tree and it wishes to indicate a change for the particular source. As a (*,G) prune indicates that the router no longer wishes to receive shared tree traffic, the (S,G,rpt) Join is meaningless.
- As Join / Prune messages are targeted to a single PIM neighbour, including both a (S,G) Join and a (S,G,rpt) prune in the same message is redundant. The (S,G) Join informs the neighbour that the sender wishes to receive the particular source on the shortest path tree. It is therefore unnecessary for the router to say that it no longer wishes to receive it on the shared tree.
- The combination of a (S,G) Prune and a (S,G,rpt) Join could possibly be used by a router to switch from receiving a particular source on the shortest-path tree back to receiving it on the shared tree (provided that the RPF neighbor for the shortest-path and shared trees is common). However Sparse-Mode PIM does not provide a mechanism for switching back to the shared tree.

The rules are summarised in the table below.

	(*,G)J	**(*,G)P**	**(S,G,rpt)J**	**(S,G,rpt)P**	**(S,G)J**	**(S,G)P**
(*,G)J	-	no	?	yes	yes	yes
(*,G)P		-	?	?	yes	yes
(S,G,rpt)J			-	no	yes	yes
(S,G,rpt)P				-	?	?
(S,G)J					-	no
(S,G)P						-

yes Allowed and expected.

no Combination is not allowed by the protocol and MUST not be generated by a router.

? Combination not expected by the protocol, but well-defined. A router MAY accept it but SHOULD not generate it.

The order of source list entries in a group set source list is not important. As a result the table above is symmetric and only entries on the upper right half have been specified as entries on the lower left are just a mirror.

4.9.5.2. Group Set Fragmentation

When building a Join / Prune for a prticular neighbour, a router should try and include in the message as much of the information it needs to convey to the neighbour as possible. This implies adding one group set for each multicast group that has information pending transmission and within each set including all relevant source list entries.

On a router with a large amount of multicast state the number of entries that must be included may result in packets that are larger in the maximum IP packet size. In most such cases the information may be split into multiple messages.

There is an exception with group sets that contain a (*,G) Join source list entry. The group set expresses the routers interest in receiving all traffic for the specified group on the shared tree and it MUST include an (S,G,rpt) Prune source list entry for every source that the router does not wish to receive. This list of (S,G,rpt) Prune source-list entries MUST not be split in multiple messages.

If only N (S,G,rpt) Prune entries fit into a maximum-sized Join / Prune message, but the router has more than N (S,G,rpt) Prunes to add, then the router MUST choose to include the first N (numerically smallest) IP addresses.

4.9.6. 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.

```
 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            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|              Group Address (Encoded-Group format)             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|            Source Address (Encoded-Unicast format)            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|R|                     Metric Preference                       |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                             Metric                            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

PIM Version, Type, Reserved, Checksum
Described above.

Group Address
The group address to which the data packet was addressed, and which triggered the Assert. This is an Encoded-Group address, as specified in 4.9.1.

Source Address
Source address from multicast datagram that triggered the Assert packet to be sent. The format for this address is given in Encoded-Unicast-Address in section 4.9.1.

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.

4.10. 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

- **Hello Timer: HT**

Per interface (I):

- Per neighbor (N):
  - **Neighbor liveness Timer: NLT(N,I)**
- Per active RP (RP):
  - **(*,*,RP) Join Expiry Timer: ET(*,*,RP,I)**
  - **(*,*,RP) PrunePending Timer: PPT(*,*,RP,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 active RP (RP):

- **(*,*,RP) Upstream Join Timer: JT(*,*,RP)**

Per Group (G):

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

Per Source (S):

(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 DRs or relevant Assert Winners only:

Per Source,Group pair (S,G):

Register Stop Timer: RST(S,G)

4.11. Timer Values

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

Timer Name: Hello Timer (HT)

Value Name	Value	Explanation
Hello_Period	30 sec	Periodic interval for hello messages.
Triggered_Hello_Delay	5 sec	Randomized interval for initial Hello message on bootup or triggered Hello message to a rebooting neighbor.

Hello messages are sent on every active interface once every Hello_Period seconds. At system power-up, the timer is initialized to rand(0,Triggered_Hello_Delay) to prevent synchronization. When a new or rebooting neighbor is detected, a responding Hello is sent within rand(0,Triggered_Hello_Delay).

Timer Name: Neighbor Liveness Timer (NLT(N,I))

Value Name	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: Expiry Timer (ET(*,*,RP,I), ET(*,G,I), ET(S,G,I), ET(S,G,rpt,I))

Value Name	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 Names: Prune Pending Timer (PPT(*,*,RP,I), PPT(*,G,I), PPT(S,G,I), PPT(S,G,rpt,I))

Value Name	Value	Explanation
J/P Override Interval	Default: 3 secs	Short period after a join or prune to allow other routers on the LAN to override the join or prune

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

Value Name	Value	Explanation
Assert Override Interval	Default: 3 secs	Short interval before an assert times out where the assert winner resends an assert message
Assert Time	Default: 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: Upstream Join Timer (JT(*,*,RP), JT(*,G), JT(S,G))

Value Name	Value	Explanation
t_periodic	Default: 60 secs	Period between Join/Prune Messages
t_suppressed	rand(1.1 * t_periodic, 1.4 * t_periodic)	Suppression period when someone else sends a J/P message so we don't need to do so.
t_override	rand(0, 0.9 * J/P Override Interval)	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.

0.9 * J/P Override Interval is really an attempt to estimate the true desired max value of t_override, which is the J/P Override Interval minus the local network's propagation delay. If the network's propagation delay is actually known, the value (J/P Override Interval - propagation delay) may be used instead.

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

Timer Name: KeepAlive Timer (KAT(S,G))

Value Name	Value	Explanation
Keepalive_Period	Default: 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 messages.
RP_Keepalive_Period	(3 * 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 Name: Upstream Override Timer (OT(S,G,rpt))

Value Name	Value	Explanation
t_po	rand(0, 0.9 * Join/Prune Override Interval)	Randomized delay to prevent response implosion when sending a join message to override someone else's prune message.

As with t_override, the network's propagation delay may be used if known. XXX t_po and t_override seem to be the same thing???

Timer Name: Register Stop Timer (RST(S,G))

Value Name	Value	Explanation
Register Suppression Time	Default: 60 seconds	Period during which a DR stops sending Register-encapsulated data to the RP after receiving a RegisterStop
Register Probe Time	Default: 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 [5].

Values 128 through 250 are designated to be assigned by the IANA based upon IESG Approval, as defined in [6]. XXX note: is the IESG OK with this?

Values 251 through 255 are designated for Private Use, as defined in [6].

5.2. PIM Hello Options

Values 17 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 [6]. Such assignments are valid for one year, and may be renewed. Permanent assignments require a specification (see "Specification Required" in [6].)

6. Security Considerations

All PIM control messages MAY use IPsec [7] to address security concerns. The authentication methods are addressed in a companion document [9]. Keys may be distributed as described in [10].

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
170 W. Tasman Drive
San Jose, CA 95134
holbrook@cisco.com

Isidor Kouvelas
Cisco Systems
170 W. Tasman Drive
San Jose, CA 95134
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, Girish Chandranmenon, Brian Haberman, Hal Sandick, and Garry Kump.

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

This specification has been blessed by St. Isidor.

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] W. Fenner, M. Handley, H. Holbrook, I. Kouvelas, "Bootstrap Router (BSR) Mechanism for PIM Sparse Mode", draft-ietf-pim-sm-bsr-00.txt, work in progress.

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

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

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

[8] S. Deering, R. Hinden, "Internet Protocol, Version 6 (IPv6) Specification", RFC 2460.

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

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

[11] H. Holbrook, B. Cain, "Source-Specific Multicast for IP", draft-holbrook-ssm-00.txt, work in progress.

10. Index