

JoinDesired(S,G) becomes True

The macro JoinDesired(S,G) becomes True, e.g., because the downstream state for (S,G) has changed so that at least one interface is in inherited_olist(S,G).

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 macro JoinDesired(S,G) becomes False, e.g., because the downstream state for (S,G) has changed so no interface is in inherited_olist(S,G).

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), and set SPTbit(S,G) to FALSE.

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.

Let t_joinsuppress be the minimum of t_suppressed and the HoldTime from the Join/Prune message triggering this event. If the Join Timer is set to expire in less than t_joinsuppress seconds, reset it so that it expires after t_joinsuppress seconds. If the Join Timer is set to expire in more than t_joinsuppress 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.

Charter Communications, Inc. - Exhibit 1007, part 2
Charter Communications, Inc. v. Iarnach Technologies, Ltd.

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 due to an Assert

The current next hop towards S changes due to 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.

RPF'(S,G) changes not due to an Assert

An event occurred that caused the next hop towards S to change. Note that this transition does not occur if an Assert is active and the upstream interface does not change.

The upstream (S,G) state machine remains in Joined state. Send Join(S,G) to the new upstream neighbor, which is the new value of RPF'(S,G). Send Prune(S,G) to the old upstream neighbor, which is the old value of RPF'(S,G). 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.

4.5.8. (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 a 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 received 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 normally sent not as a periodic message, but only as a triggered message.

4.5.9. 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. The three states are:

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 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.

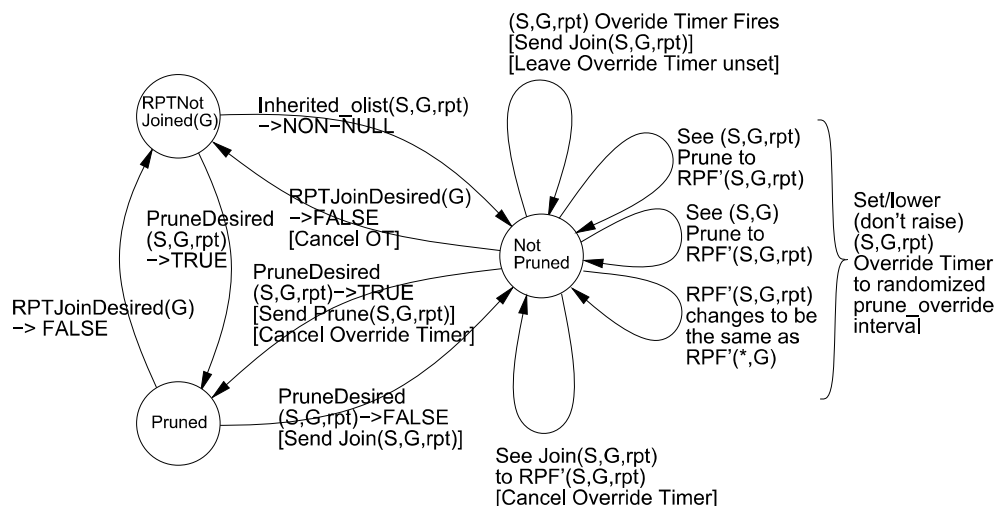


Figure 9: Upstream (S,G,rpt) state machine for triggered messages

Upstream (S,G,rpt) state machine for triggered messages in tabular form:

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); Cancel OT	-	→ NJ state Cancel OT	-

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				
Override 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); Leave OT unset	OT = min(OT, t_override)	Cancel OT	OT = min(OT, t_override)	OT = min(OT, t_override)

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_override) = t_override`).

This state machine uses the following macros:

```
bool RPTJoinDesired(G) {
    return (JoinDesired(*,G) OR 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
             ( inheritedolist(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 either if there are no outgoing interfaces that S would be forwarded on, or if the router has active (S,G) forwarding state but $RPF'(*,G) \neq RPF'(S,G)$.

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 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) Override Timer to the randomized prune-override interval, t_{override} . 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 routers implemented from RFC2362 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 $t_{override}$. 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 noone else has sent a Join, then we will do so.

$PruneDesired(S,G,rpt) \rightarrow TRUE$

See macro above. This event is relevant in the "NotPruned" and "RPTNotJoined(G)" states.

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 $RPF'(S,G,rpt)$, and to cancel the Override Timer. 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 $Join(*,*,RP)$.

$PruneDesired(S,G,rpt) \rightarrow FALSE$

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 " $RPTJoinDesired(G) \rightarrow FALSE$ " transition should happen. Thus, this transition should be interpreted as " $PruneDesired(S,G,rpt) \rightarrow FALSE$ AND $RPTJoinDesired(G) == TRUE$ ".

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

$RPTJoinDesired(G) \rightarrow FALSE$

This event is relevant in the "Pruned" and "NotPruned" states.

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, and the Override Timer is canceled if it was running. Any further 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.

4.5.10. Background: (*,*,RP) and (S,G,rpt) Interaction

In Sections 4.5.8 and 4.5.9, the mechanisms for sending periodic and triggered (S,G,rpt) messages are described. The astute reader will note that periodic Prune(S,G,rpt) messages are only sent in PIM Join/Prune messages containing a Join(*,G), whereas it is possible for a triggered Prune(S,G,rpt) message to be sent when the router has no (*,G) join state. This may seem like a contradiction, but in fact it is intentional and is a side effect of not optimizing (*,*,RP) behavior.

We first note that reception of a Join(*,*,RP) by itself does not cancel (S,G,rpt) prune state on that interface, whereas receiving a Join(*,G) by itself does cancel (S,G,rpt) prune state on that interface. Similarly, reception of a Prune(*,G) on an interface with (*,*,RP) join state does not by itself prevent forwarding of G using the (*,*,RP) state; this is because a Prune(*,G) only serves to cancel (*,G) join state. Conceptually (*,*,RP) state functions "above" the normal (*,G) and (S,G) mechanisms, and so neither Join(*,*,RP) nor Prune(*,*,RP) messages affect any other state.

The upshot of this is that to prevent forwarding (S,G) on (*,*,RP) state, a Prune(S,G,rpt) must be used.

We also note that for historical reasons there is no Assert(*,*,RP) message, so any forwarding contention is resolved using Assert(*,G) messages.

We now need to consider the interaction between (*,*,RP) state and (*,G) state. If there is a need for an assert between two upstream routers on a LAN, we need to ensure that the correct thing happens for all combinations of (*,*,RP) and (*,G) forwarding state. As there is no Assert(*,*,RP) message, no router can tell whether the assert winner has (*,*,RP) state or (*,G) state. Thus, a downstream router has to treat the two the same and send its periodic Prune(S,G,rpt) messages to RPF'(*,G).

To avoid needing to specify all the complex override rules between (*,*,RP), (*,G), and (S,G,rpt), we simply require that to prune (S,G) off the (*,*,RP) tree, a Join(*,G) must also be sent.

If a router is receiving on (*,*,RP) state and has not yet had (*,G) state instantiated, it may still need to send a triggered Join(S,G,rpt) to override a Prune(S,G,rpt) that it sees directed to RPF'(*,G) on its upstream interface. Hence, triggered (S,G,rpt) messages may be sent when JoinDesired(*,G) is false but JoinDesired(*,*,RP) is true.

Finally, we note that there is an unoptimized case when the upstream router on a LAN already has (*,G) join and (S,G,rpt) prune state caused by an existing downstream router. If at this time a new Join(*,*,RP) is sent to the upstream router from a different downstream router, this will not override the (S,G,rpt) prune state at the upstream router. The override will not occur until the next time the original downstream router resends its Prune(S,G,rpt). This case was not considered worth optimizing, as (*,*,RP) state is generally very long lived, and so any minor delays in getting traffic to a new PMBR seem unimportant.

4.6. PIM Assert Messages

Where multiple PIM routers peer over a shared LAN, it is possible for more than one upstream router to have valid forwarding state for a packet, which can lead to packet duplication (see Section 3.6). PIM does not attempt to prevent this from occurring. Instead, it detects when this has happened and elects a single forwarder amongst the upstream routers to prevent further duplication. This election is performed using PIM Assert messages. Assert messages are also received by downstream routers on the LAN, and these cause subsequent Join/Prune messages to be sent to the upstream router that won the Assert.

In general, a PIM Assert message should only be accepted for processing if it comes from a known PIM neighbor. A PIM router hears about PIM neighbors through PIM Hello messages. If a router receives an Assert message from a particular IP source address and it has not seen a PIM Hello message from that source address, then the Assert message SHOULD be discarded without further processing. In addition, if the Hello message from a neighbor was authenticated using the IPsec Authentication Header (AH) (see Section 6.3), then all Assert messages from that neighbor MUST also be authenticated using IPsec AH.

We note that some older PIM implementations incorrectly fail to send Hello messages on point-to-point interfaces, so we also RECOMMEND that a configuration option be provided to allow interoperation with such older routers, but that this configuration option SHOULD NOT be enabled by default.

4.6.1. (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 an Assert Timer (AT) that is used to time out asserts on the assert losers and to resend asserts on the assert winner.

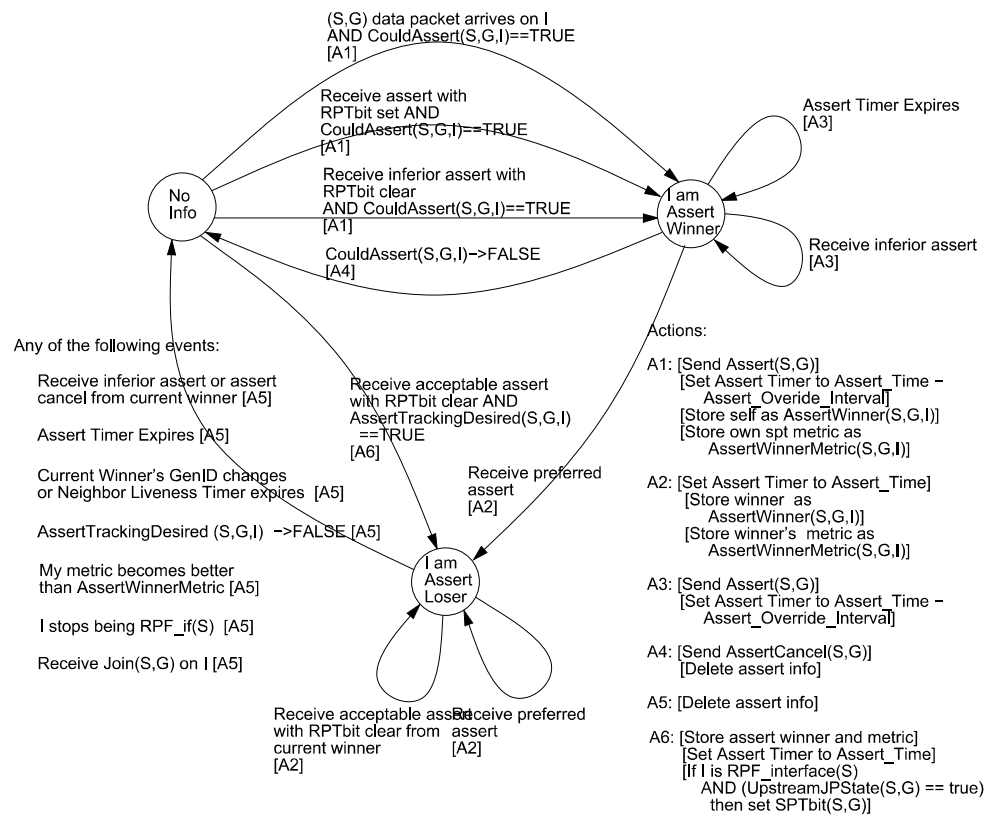


Figure 10: Per-interface (S,G) Assert State machine

Per-interface (S,G) Assert State machine in tabular form:

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 Acceptable 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			
Assert 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 Acceptable Assert with RPTbit clear from Current Winner	Receive Inferior Assert or Assert Cancel from Current Winner	Assert Timer Expires	Current Winner's GenID Changes or NLT Expires
→ L state [Actions A2]	→ L state [Actions A2]	→ NI state [Actions A5]	→ NI state [Actions A5]	→ NI state [Actions A5]

In I Am Assert Loser (L) State			
AssTrDes (S,G,I) → FALSE	my_metric → better than winner's metric	RPF_interface (S) stops being I	Receive Join(S,G) on interface I
→ NI state [Actions A5]	→ NI state [Actions A5]	→ NI state [Actions A5]	→ NI State [Actions A5]

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

Terminology:

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

An "acceptable assert" is one that has a better metric than `my_assert_metric(S,G,I)`. An assert is never considered acceptable if its metric is infinite.

An "inferior assert" is one with a worse metric than `my_assert_metric(S,G,I)`. An assert is never considered inferior if `my_assert_metric(S,G,I)` is infinite.

The state machine uses the following macros:

```
CouldAssert(S,G,I) =
    SPTbit(S,G)==TRUE
    AND (RPF_interface(S) ≠ I)
    AND (I in ( ( joins(*,*,RP(G)) ⊕ 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 that 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(*,*,RP(G)) ⊕ joins(*,G) ⊖
prunes(S,G,rpt) )
        ⊕ ( pim_include(*,G) ⊖ pim_exclude(S,G) )
        ⊖ lost_assert(*,G)
        ⊕ joins(S,G) ) )
    OR (local_receiver_include(S,G,I) == TRUE
        AND (I_am_DR(I) OR (AssertWinner(S,G,I) == me)))
    OR ((RPF_interface(S) == I) AND (JoinDesired(S,G) ==
TRUE))
    OR ((RPF_interface(RP(G)) == 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 and local membership information received on I that might cause the router to be interested in asserts on I.

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

The 5th and 6th lines account for (S,G) local membership information on I. Note that we can't use the pim_include(S,G) macro since it uses lost_assert(S,G,I) and would result in the router forgetting that it lost an assert if the only reason it was interested was local membership. The AssertWinner(S,G,I) check forces an assert winner to keep on being responsible for forwarding as long as local receivers are present. Removing this check would make the assert winner give up forwarding as soon as the information that originally caused it to forward went away, and the task of forwarding for local receivers would revert back to the DR.

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 AND CouldAssert(S,G,I)==TRUE

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 (i.e., CouldAssert(S,G,I)==TRUE)

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 that 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 Acceptable 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 has a better metric than our own, so we do not win the Assert. We transition to "I am Assert Loser" and perform Actions A6 (below).

Transitions from "I am Assert Winner" State

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

Assert Timer Expires

The (S,G) Assert Timer expires. As we're in the Winner state, we must still have (S,G) forwarding state that is actively being kept alive. We resend the (S,G) Assert and restart the Assert Timer (Actions A3 below). Note that the assert winner's Assert 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 resend an (S,G) Assert and restart the Assert Timer (Actions 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) and PruneDesired(S,G,rpt), which could cause transitions in the upstream (S,G) or (S,G,rpt) state machines.

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 "canceling assert" with an infinite metric.

Transitions from "I am Assert 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 Acceptable Assert with RPTbit clear from Current Winner

We receive an assert from the current assert winner that is better than our own metric for this (S,G) (although the metric may be worse than the winner's previous metric). We stay in Loser state and perform Actions A2 below.

Receive Inferior Assert or Assert Cancel 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 or because it is an assert cancel). 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.

Assert Timer Expires

The (S,G) Assert Timer expires. We transition to NoInfo state, deleting the (S,G) assert information (Actions A5 below).

Current Winner's GenID Changes or NLT Expires

The Neighbor Liveness Timer associated with the current winner expires or we receive a Hello message from the current winner reporting a different GenID from the one it previously reported. This indicates that the current winner's interface or router has gone down (and may have come back up), and so we must assume it no longer knows it was the winner. We transition to the NoInfo state, deleting this (S,G) assert information (Actions A5 below).

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 when the underlying routing metric changes, or 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 (Actions A5 below), 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(S) stops being interface I

Interface I used to be the RPF interface for S, and now it is not. We transition to NoInfo state, deleting this (S,G) assert state (Actions A5 below).

Receive Join(S,G) on Interface I

We receive a Join(S,G) that has the Upstream Neighbor Address field set to my primary IP address on interface I. The action is to transition to NoInfo state, delete this (S,G) assert state (Actions A5 below), 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 Assert Timer to (Assert_Time - Assert_Override_Interval).
 Store self as AssertWinner(S,G,I).
 Store spt_assert_metric(S,I) as AssertWinnerMetric(S,G,I).
- A2: Store new assert winner as AssertWinner(S,G,I) and assert winner metric as AssertWinnerMetric(S,G,I).
 Set Assert Timer to Assert_Time.
- A3: Send Assert(S,G).
 Set Assert Timer to (Assert_Time - Assert_Override_Interval).
- A4: Send AssertCancel(S,G).
 Delete assert info (AssertWinner(S,G,I) and AssertWinnerMetric(S,G,I) will then return their default values).
- A5: Delete assert info (AssertWinner(S,G,I) and AssertWinnerMetric(S,G,I) will then return their default values).
- A6: Store new assert winner as AssertWinner(S,G,I) and assert winner metric as AssertWinnerMetric(S,G,I).
 Set Assert Timer to Assert_Time.
 If (I is RPF_interface(S)) AND (UpstreamJPState(S,G) == true) set SPTbit(S,G) to TRUE.

Note that some of these actions may cause the value of JoinDesired(S,G), PruneDesired(S,G,rpt), or RPF'(S,G) to change, which could cause further transitions in other state machines.

4.6.2. (*,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 it 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.

When an Assert message is received with a source address other than zero, a PIM implementation must first match it against the possible events in the (S,G) assert state machine and process any transitions and actions, before considering whether the Assert message matches against the (*,G) assert state machine.

It is important to note that NO TRANSITION CAN OCCUR in the (*,G) state machine as a result of receiving an Assert message unless the (S,G) assert state machine for the relevant S and G is in the "NoInfo" state after the (S,G) state machine

has processed the message. Also, NO TRANSITION CAN OCCUR in the (*,G) state machine as a result of receiving an assert message if that message triggers any change of state in the (S,G) state machine. Obviously, when the source address in the received message is set to zero, an (S,G) state machine for the S and G does not exist and can be assumed to be in the "NoInfo" state.

For example, if both the (S,G) and (*,G) assert state machines are in the NoInfo state when an Assert message arrives, and the message causes the (S,G) state machine to transition to either "W" or "L" state, then the assert will not be processed by the (*,G) assert state machine.

Another example: if the (S,G) assert state machine is in "L" state when an assert message is received, and the assert metric in the message is worse than $\text{my_assert_metric}(S, G, I)$, then the (S,G) assert state machine will transition to NoInfo state. In such a case, if the (*,G) assert state machine were in NoInfo state, it might appear that it would transition to "W" state, but this is not the case because this message already triggered a transition in the (S,G) assert state machine.

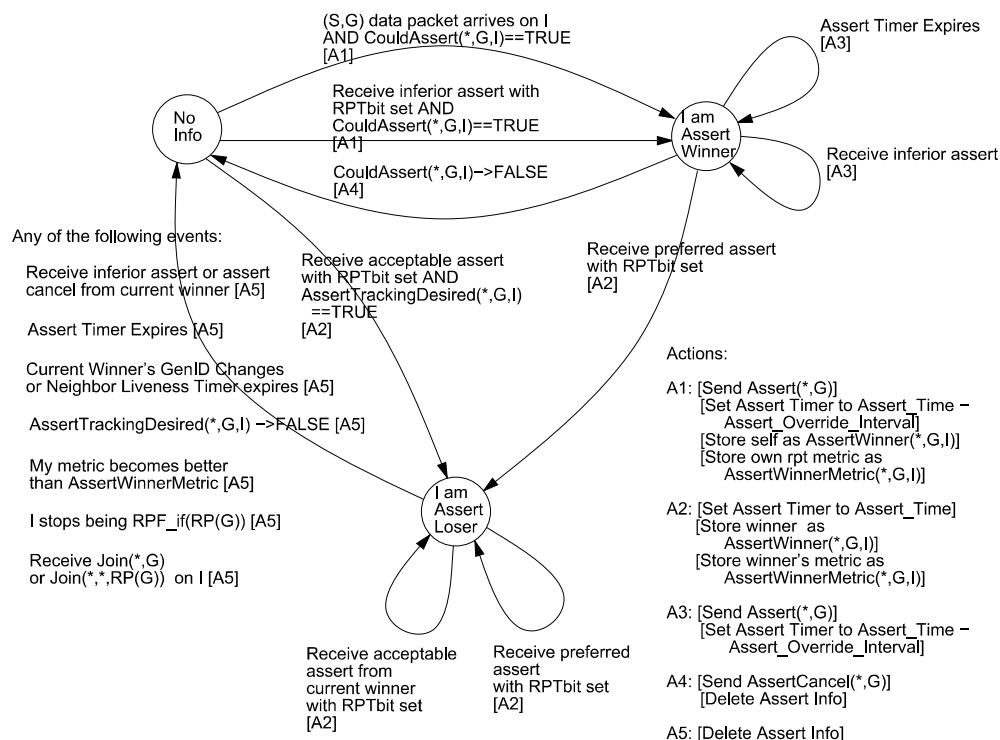


Figure 11: Per-interface (*,G) Assert State machine

Per-interface (*,G) Assert State machine in tabular form:

In NoInfo (NI) State		
Receive Inferior Assert with RPTbit set and CouldAssert(*,G,I)	Data arrives for G on I and CouldAssert(*,G,I)	Receive Acceptable Assert with RPTbit set and AssTrDes(*,G,I)
→ W state [Actions A1]	→ W state [Actions A1]	→ L state [Actions A2]

In I Am Assert Winner (W) State			
Assert 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 with RPTbit set	Receive Acceptable Assert from Current Winner with RPTbit set	Receive Inferior Assert or Assert Cancel from Current Winner	Assert Timer Expires	Current Winner's GenID Changes or NLT Expires
→ L state [Actions A2]	→ L state [Actions A2]	→ NI state [Actions A5]	→ NI state [Actions A5]	→ NI state [Actions A5]

In I Am Assert Loser (L) State			
AssTrDes(*,G,I) → FALSE	my_metric → better than Winner's metric	RPF_interface(RP(G)) 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]	→ NI State [Actions A5]

The state machine uses the following macros:

```
CouldAssert(*,G,I) =
  ( I in ( joins(*,*,RP(G)) ⊕ joins(*,G)
           ⊕ pim_include(*,G)) )
  AND (RPF_interface(RP(G)) ≠ I)
```

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

```
AssertTrackingDesired(*,G,I) =
  CouldAssert(*,G,I)
  OR (local_receiver_include(*,G,I)==TRUE
      AND (I_am_DR(I) OR AssertWinner(*,G,I) == me))
  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 "acceptable assert" is one that has a better metric than `my_assert_metric(*,G,I)`. An assert is never considered acceptable if its metric is infinite.

An "inferior assert" is one with a worse metric than `my_assert_metric(*,G,I)`. An assert is never considered inferior if `my_assert_metric(*,G,I)` is infinite.

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 before and after consideration of the received message:

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 that 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 Acceptable 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 "I am Assert 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 before and after consideration of the received message:

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 resend a (*,G) Assert and restart the Assert Timer (Actions 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:

Assert 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 resend the (*,G) Assert and restart the Assert Timer (Actions 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 "I am Assert 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 before and after consideration of the received message:

Receive Preferred Assert with RPTbit set

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 Acceptable Assert from Current Winner with RPTbit set

We receive a (*,G) assert from the current assert winner that is better than our own metric for this group (although the metric may be worse than the winner's previous metric). We stay in Loser state and perform Actions A2 below.

Receive Inferior Assert or Assert Cancel 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 or is now an assert cancel). We transition to NoInfo state, delete this (*,G) assert state (Actions A5), 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:

Assert Timer Expires

The (*,G) Assert Timer expires. We transition to NoInfo state and delete this (*,G) assert info (Actions A5).

Current Winner's GenID Changes or NLT Expires

The Neighbor Liveness Timer associated with the current winner expires or we receive a Hello message from the current winner reporting a different GenID from the one it previously reported. This indicates that the current winner's interface or router has gone down (and may have come back up), and so we must assume it no longer knows it was the winner. We transition to the NoInfo state, deleting the (*,G) assert information (Actions A5).

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 (Actions A5).

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, delete this `(*,G)` assert state (Actions A5), 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(RP(G)) stops being 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 (Actions A5).

Receive Join(*,G) or Join(*,*,RP(G)) on interface I

We receive a Join(*,G) or a Join(*,*,RP(G)) that has the Upstream Neighbor Address field set to my primary IP address on interface I. The action is to transition to NoInfo state, delete this `(*,G)` assert state (Actions A5), 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, so we may end up being the new forwarder.

(*,G) Assert State machine Actions

- A1: Send Assert(*,G).
Set Assert Timer to `(Assert_Time - Assert_Override_Interval)`.
Store self as AssertWinner(*,G,I).
Store `rpt_assert_metric(G,I)` as AssertWinnerMetric(*,G,I).
- A2: Store new assert winner as AssertWinner(*,G,I) and assert winner metric as AssertWinnerMetric(*,G,I).
Set Assert Timer to Assert_Time.
- A3: Send Assert(*,G)
Set Assert Timer to `(Assert_Time - Assert_Override_Interval)`.
- A4: Send AssertCancel(*,G).
Delete assert info (AssertWinner(*,G,I) and AssertWinnerMetric(*,G,I) will then return their default values).
- A5: Delete assert info (AssertWinner(*,G,I) and AssertWinnerMetric(*,G,I) will then return their default values).

Note that some of these actions may cause the value of JoinDesired(*,G) or RPF'(*,G)) to change, which could cause further transitions in other state machines.

4.6.3. 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 primary 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,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 primary 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,0}
}
```

4.6.4. 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 the source set to zero.

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.

4.6.5. 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(G)) == I OR
        ( RPF_interface(S) == I AND SPTbit(S,G) == TRUE ) ) {
        return FALSE
    } else {
        return ( AssertWinner(S,G,I) != NULL AND
                 AssertWinner(S,G,I) != me )
    }
}

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

Note: the term "AssertWinnerMetric(S,G,I) is better than spt_assert_metric(S,I)" is required to correctly handle the transition phase when a router has (S,G) join state, but has not yet set the SPT bit. In this case, it needs to ignore the assert state if it will win the assert once the SPTbit is set.

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

`AssertWinner(S,G,I)` is the IP source address of the `Assert(S,G)` packet that won an Assert.

`AssertWinner(*,G,I)` is the IP source address of the `Assert(*,G)` packet that won an Assert.

`AssertWinnerMetric(S,G,I)` is the Assert metric of the `Assert(S,G)` packet that won an Assert.

`AssertWinnerMetric(*,G,I)` is the Assert metric of the `Assert(*,G)` packet that won an Assert.

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

Summary of Assert Rules and Rationale

This section summarizes the key rules for sending and reacting to asserts and the rationale for these rules. This section is not intended to be and should not be treated as a definitive specification of protocol behavior. The state machines and pseudocode should be consulted for that purpose. Rather, this section is intended to document important aspects of the Assert protocol behavior and to provide information that may prove helpful to the reader in understanding and implementing this part of the protocol.

1. Behavior: 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. They are not always sent to the RPF neighbor as indicated by the MRIB. Normal suppression and override rules apply.
 Rationale: By sending the periodic and triggered Join messages to the RPF' neighbor instead of to the RPF neighbor, the downstream router avoids re-triggering the Assert process with every Join. A side effect of sending Joins to the Assert winner is that traffic will not switch back to the "normal" RPF neighbor until the Assert times out. This will not happen until data stops flowing, if item 8, below, is implemented.
2. Behavior: The assert winner for `(*,G)` acts as the local DR for `(*,G)` on behalf of IGMP/MLD members.
 Rationale: This is required to allow a single router to merge PIM and IGMP/MLD joins and leaves. Without this, overrides don't work.
3. Behavior: The assert winner for `(S,G)` acts as the local DR for `(S,G)` on behalf of IGMPv3 members.
 Rationale: Same rationale as for item 2.
4. Behavior: `(S,G)` and `(*,G)` prune overrides are sent to the RPF' neighbor and not to the regular RPF neighbor.
 Rationale: Same rationale as for item 1.
5. Behavior: An `(S,G,rpt)` prune override is not sent (at all) if $RPF'(S,G,rpt) \neq RPF'(*,G)$.
 Rationale: This avoids keeping state alive on the `(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 behavior might be confusing although this specification does indicate that such a join should be dropped.
6. Behavior: An assert loser that receives a `Join(S,G)` with an Upstream Neighbor Address that is its primary IP address on that interface cancels the `(S,G)` Assert Timer.
 Rationale: This is necessary in order to have rapid convergence in the event that the downstream router that initially sent a join to the prior Assert winner has undergone a topology change.
7. Behavior: An assert loser that receives a `Join(*,G)` or a `Join(*,*,RP(G))` with an Upstream Neighbor Address that is its primary IP address on that interface 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.

Rationale: Same rationale as for item 6.

8. Behavior: 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 behavior does not apply to (S,G,rpt).

Rationale: This allows switching back to the shared tree after the last SPT router on the LAN leaves. Doing this prevents downstream routers on the shared tree from keeping SPT state alive.

9. Behavior: Resend the assert messages before timing out an assert. (This behavior is optional.)

Rationale: This prevents the periodic duplicates that would otherwise occur each time that an assert times out and is then re-established.

10. Behavior: 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).

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

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.

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 four mechanisms are possible, and all four 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.

Embedded-RP

Embedded-RP defines an address allocation policy in which the address of the Rendezvous Point (RP) is encoded in an IPv6 multicast group address [17].

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 on 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 mechanisms, or through alternative mechanisms not currently specified.

It must be operationally ensured that any RP address configured, learned, or advertised is reachable from all routers in the PIM domain.

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/MLD 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 restart register encapsulation to the new RP.

Implementation note: the bootstrap mechanism described in RFC 2362 omitted step 1 above. However, of the implementations we are aware of, approximately half performed step 1 anyway. Note that implementations of BSR that omit step 1 will not correctly interoperate with implementations of this specification when used with the BSR mechanism described in [11].

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:

$$\text{Value}(G,M,C(i)) = (1103515245 * ((1103515245 * (G \& M) + 12345) \text{ XOR } C(i)) + 12345) \bmod 2^{31}$$

where $C(i)$ is the RP address and M is a hash-mask. If BSR is being used, the hash-mask is given in the Bootstrap messages. If BSR is not being used, the alternative mechanism that supplies the group-range-to-RP mappings may supply the value, or else it defaults to a mask with the most significant 30 bits being one for IPv4 and the most significant 126 bits being one for IPv6. 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 $\wedge$ represents the exclusive-or operation.

2. The candidate RP with the highest resulting hash value is then the RP chosen by this Hash Function. If more than one RP has the same highest hash value, the RP with the highest IP address is chosen.

4.8. Source-Specific Multicast

The Source-Specific Multicast (SSM) service model [6] 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 and `FF3x::/32` for IPv6, 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 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 implementer 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 4.5.7)
- Downstream (S,G) state machine (Section 4.5.3)
- (S,G) Assert state machine (Section 4.6.1)
- Hello messages, neighbor discovery, and DR election (Section 4.3)
- Packet forwarding rules (Section 4.2)

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

- Register state machine (Section 4.4)
- (*,G), (S,G,rpt), and (*,*,RP) Downstream state machines (Sections 4.5.2, 4.5.4, and 4.5.1)
- (*,G), (S,G,rpt), and (*,*,RP) Upstream state machines (Sections 4.5.6, 4.5.8, and 4.5.5)
- (*,G) Assert state machine (Section 4.6.2)
- Bootstrap RP Election (Section 4.7)
- Keepalive Timer
- SPTbit (Section 4.2.2)

The `Keepalive Timer` should be treated as always running, and `SPTbit` should be treated as always being 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  $\ominus$  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'.

The PIM header common to all PIM messages is:

```

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:

Message Type	Destination
0 = Hello	Multicast to ALL-PIM-ROUTERS
1 = Register	Unicast to RP
2 = Register-Stop	Unicast to source of Register packet
3 = Join/Prune	Multicast to ALL-PIM-ROUTERS
4 = Bootstrap	Multicast to ALL-PIM-ROUTERS
5 = Assert	Multicast to ALL-PIM-ROUTERS
6 = Graft (used in PIM-DM only)	Unicast to RPF'(S)
7 = Graft-Ack (used in PIM-DM only)	Unicast to source of Graft packet
8 = Candidate-RP-Advertisement	Unicast to Domain's BSR

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 |B| Reserved |Z| Mask Len |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|
|                               Group multicast Address
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+...
```

Addr Family

Described above.

Encoding Type

Described above.

[B]idirectional PIM

Indicates the group range should use Bidirectional PIM [13]. For PIM-SM defined in this specification, this bit MUST be zero.

Reserved

Transmitted as zero. Ignored upon receipt.

Admin Scope [Z]one

indicates the group range is an admin scope zone. This is used in the Bootstrap Router Mechanism [11] only. For all other purposes, this bit is set to zero and ignored on receipt.

Mask Len

The Mask length field is 8 bits. The value is the number of contiguous one bits that are left justified and used as a mask; when combined with the group address, it 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, 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

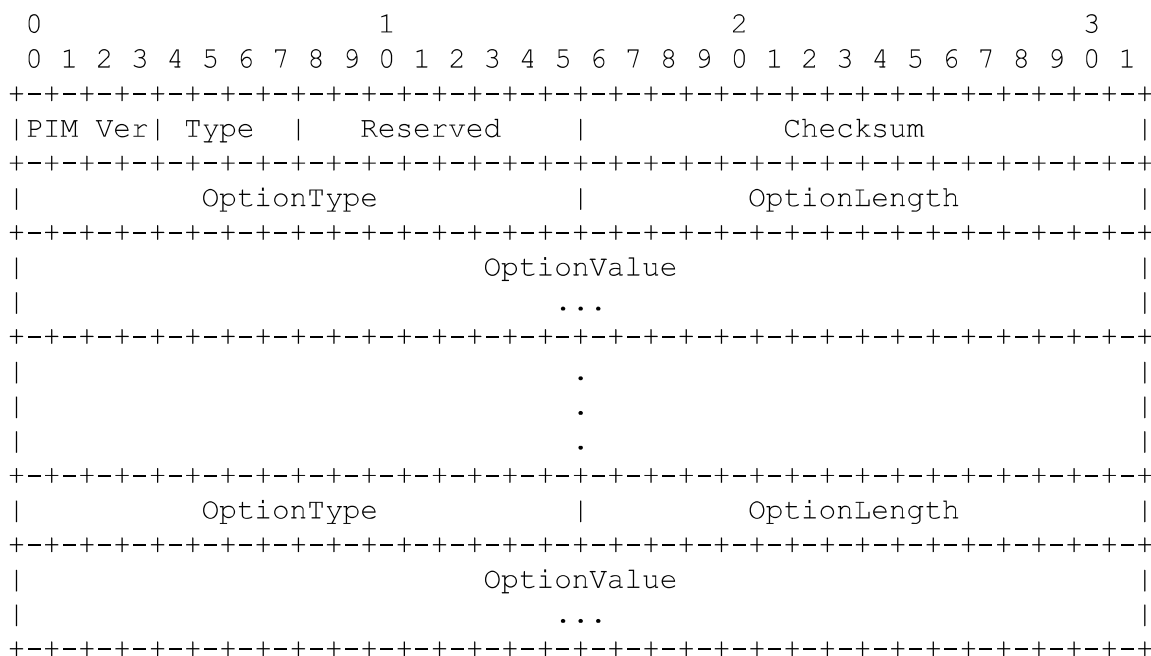
The mask length field is 8 bits. The value is the number of contiguous one bits left justified used as a mask which, combined with the Source Address, describes a source subnet. The mask length MUST be equal to the mask length in bits for the given Address Family and Encoding Type (32 for IPv4 native and 128 for IPv6 native). A router SHOULD ignore any messages received with any other mask length.

Source Address

The source address.

4.9.2. Hello Message Format

It is sent periodically by routers on all interfaces.

**PIM Version, Type, Reserved, Checksum**

Described in Section 4.9.

OptionType

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

OptionLength

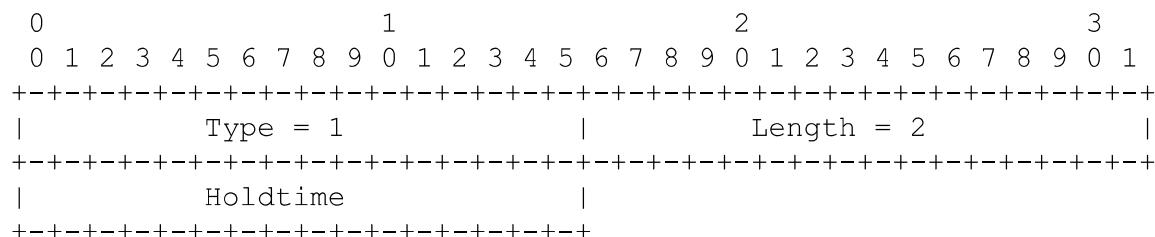
The length of the OptionValue field in bytes.

Option Value

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

The Option fields may contain the following values:

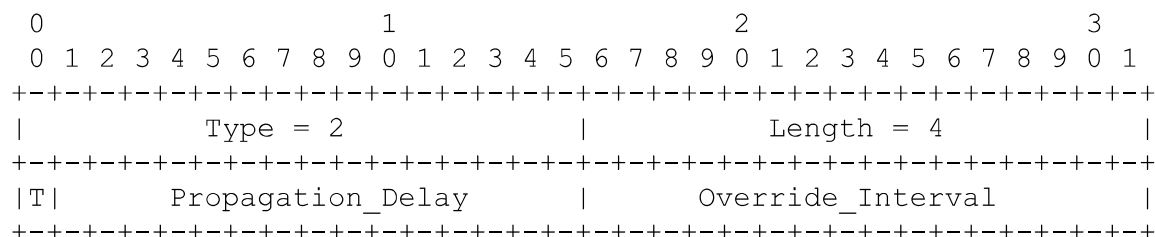
- OptionType 1: Holdtime



Holdtime is the amount of time a receiver must keep the neighbor reachable, in seconds. If the Holdtime is set to '0xffff', the receiver of this message never times out the neighbor. This may be used with dial-on-demand links, to avoid keeping the link up with periodic Hello messages.

Hello messages with a Holdtime value set to '0' are also sent by a router on an interface about to go down or changing IP address (see Section 4.3.1). These are effectively goodbye messages, and the receiving routers should immediately time out the neighbor information for the sender.

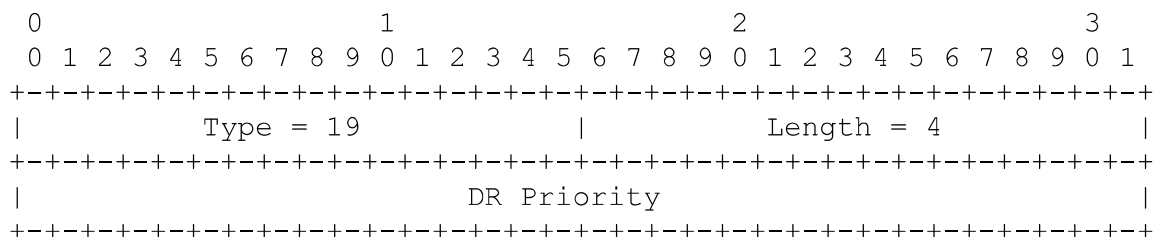
- OptionType 2: LAN Prune Delay



The LAN Prune Delay option is used to tune the prune propagation delay on multi-access LANs. The T bit specifies the ability of the sending router to disable joins suppression. Propagation_Delay and Override_Interval are time intervals in units of milliseconds. A router originating a LAN Prune Delay option on interface I sets the Propagation_Delay field to the configured value of Propagation_Delay(I) and the value of the Override_Interval field to the value of Override_Interval(I). On a receiving router, the values of the fields are used to tune the value of the Effective_Override_Interval(I) and its derived timer values. Section 4.3.3 describes how these values affect the behavior of a router.

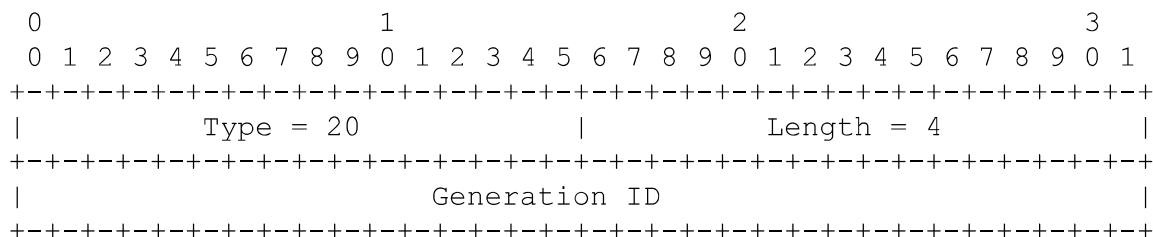
- OptionType 3 to 16: reserved to be defined in future versions of this document.
- OptionType 18: deprecated and should not be used.

- OptionType 19: DR Priority



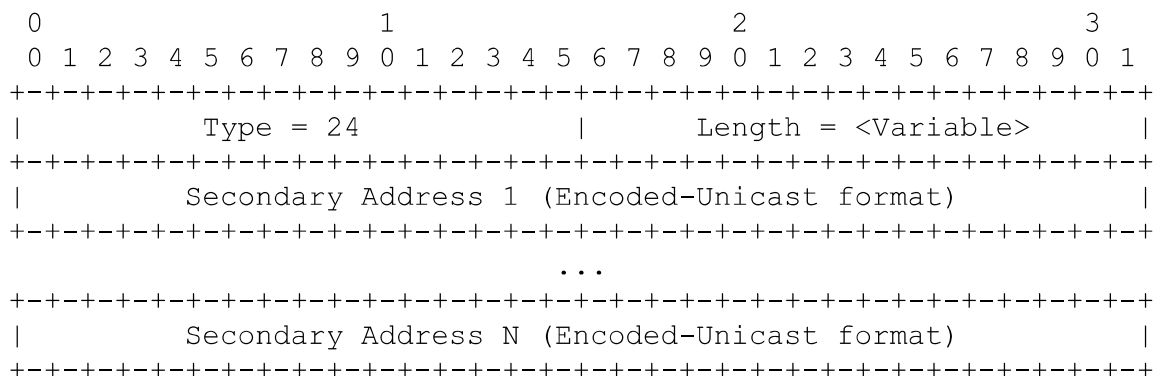
DR Priority is a 32-bit unsigned number and should be considered in the DR election as described in Section 4.3.2.

- OptionType 20: 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.

- OptionType 24: Address List



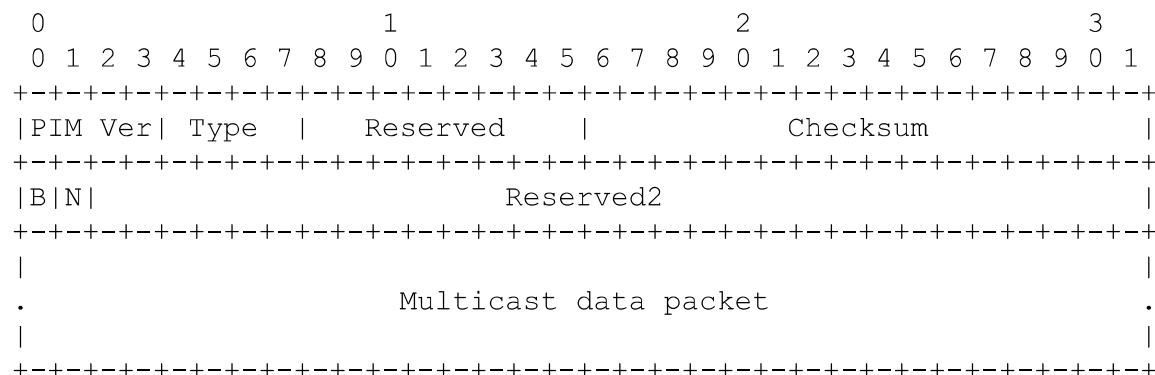
The contents of the Address List Hello option are described in Section 4.3.4. All addresses within a single Address List must belong to the same address family.

OptionTypes 17 through 65000 are assigned by the IANA. OptionTypes 65001 through 65535 are reserved for Private Use, as defined in [9].

Unknown options MUST be ignored and MUST NOT prevent a neighbor relationship from being formed. The "Holdtime" option MUST be implemented; the "DR Priority" and "Generation ID" options SHOULD be implemented. The "Address List" option MUST be implemented for IPv6.

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.



PIM Version, Type, Reserved, Checksum

Described in Section 4.9. Note that in order to reduce encapsulation overhead, the checksum for Registers is done only on the first 8 bytes of the 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 also be accepted. When calculating the checksum, the IPv6 pseudoheader "Upper-Layer Packet Length" is set to 8.

- 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 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 a dummy IP header with S as the source address, G as the destination address. When generating an IPv4 Null-Register message, the fields in the dummy IPv4 header **SHOULD** be filled in according to the following table. Other IPv4 header fields may contain any value that is valid for that field.

Field	Value
IP Version	4
Header Length	5
Checksum	Header checksum
Fragmentation offset	0
More Fragments	0
Total Length	20
IP Protocol	103 (PIM)

On receipt of an (S,G) Null-Register, if the Header Checksum field is non-zero, the recipient **SHOULD** check the checksum and discard null registers that have a bad checksum. The recipient **SHOULD NOT** check the value of any individual fields; a correct IP header checksum is sufficient. If the Header Checksum field is zero, the recipient **MUST NOT** check the checksum.

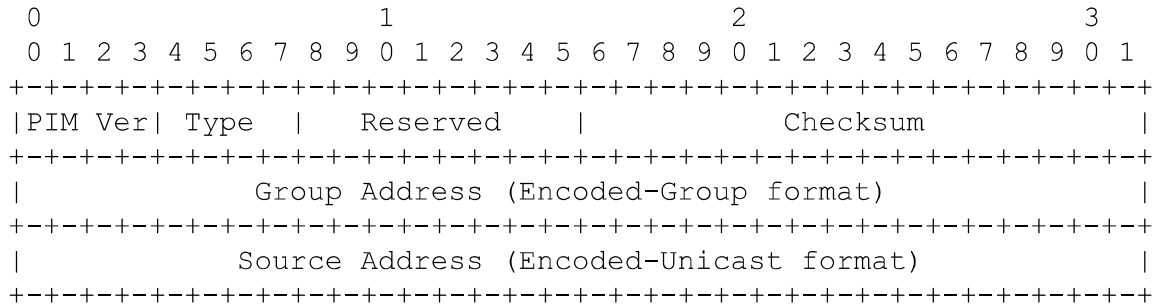
With IPv6, an implementation generates a dummy IP header followed by a dummy PIM header with values according to the following table in addition to the source and group. Other IPv6 header fields may contain any value that is valid for that field.

Header Field	Value
IP Version	6
Next Header	103 (PIM)
Length	4
PIM Version	0
PIM Type	0
PIM Reserved	0
PIM Checksum	PIM checksum including IPv6 "pseudo-header"; see Section 4.9

On receipt of an IPv6 (S,G) Null-Register, if the dummy PIM header is present, the recipient **SHOULD** check the checksum and discard Null-Registers that have a bad checksum.

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.



PIM Version, Type, Reserved, Checksum

Described in Section 4.9.

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

The host address of the source from the multicast data packet in the register. The format for this address is given in the Encoded-Unicast address in Section 4.9.1. A special wild card value consisting of an address field of all zeros can be used to indicate any source.

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)          |
+-+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|          :          |
|          :          |
+-+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|          Pruned Source Address n (Encoded-Source format)          |
+-+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+

```

PIM Version, Type, Reserved, Checksum

Described in Section 4.9.

Unicast Upstream Neighbor Address

The address of the upstream neighbor that is the target of the message. The format for this address is given in the Encoded-Unicast address in Section 4.9.1. For IPv6 the source address used for multicast messages is the link-local address of the interface on which the message is being sent. For IPv4, the source address is the primary address associated with that interface.

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 canceling 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(I).

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 joined source addresses listed for a given group.

Joined Source Address 1 .. n

This list contains the sources for a given group that the sending router will forward multicast datagrams from if received on the interface on which the Join/Prune message is sent.

See Encoded-Source-Address format in Section 4.9.1.

Number of Pruned Sources

Number of pruned source addresses listed for a group.

Pruned Source Address 1 .. n

This list contains the sources for a given group that the sending router does not want to forward multicast datagrams from when received on the interface on which the Join/Prune message is sent.

Within one PIM Join/Prune message, all the Multicast Group Addresses, Joined Source addresses, and Pruned Source addresses MUST be of the same address family. It is NOT PERMITTED to mix IPv4 and IPv6 addresses within the same message. In addition, the address family of the fields in the message SHOULD be the same as the IP source and destination addresses of the packet. This permits maximum implementation flexibility for dual-stack IPv4/IPv6 routers. If a router receives a message with mixed family addresses, it SHOULD only process the addresses that are of the same family as the unicast upstream neighbor address.

4.9.5.1. Group Set Source List Rules

As described above, Join/Prune messages are composed of one or more group sets. Each set contains two source lists, the Joined Sources and the Pruned 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 (i.e., '224.0.0.0/4' for IPv4 or 'ff00::/8' for IPv6). Each Join/Prune message SHOULD contain at most one wildcard group set. Each wildcard group set may contain one or more (*,*,RP) source list entries in either the Joined or Pruned lists.

A (*,*,RP) source list entry may only exist in a wildcard group set. When added to a Joined source list, this type of source entry expresses the router's interest in receiving traffic for all groups mapping to the specified RP. When added to a Pruned source list a (*,*,RP) entry expresses the router's interest to stop receiving such traffic. Note that as indicated by the Join/Prune state machines, such a Join or Prune will NOT override Join/Prune state created using a Group-Specific Set (see below).

(*,*,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

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 Join/Prune message SHOULD NOT contain more than one group-specific set for the same IP multicast address. Each group-specific set may contain (*,G), (S,G,rpt), and (S,G) source list entries in the Joined or Pruned 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 thereof) in receiving traffic sent to the group through the Rendezvous-Point shared tree. There may only be one such entry in both the Joined and Pruned 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 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 thereof) in receiving traffic through the shared tree sent by the specified source to this group. For each source address, the entry may exist in only one of the Joined and Pruned source lists of a group-specific set, but not both.

(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 the WC bit cleared 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 thereof) in receiving traffic through the shortest path tree sent by the source to the specified group. For each source address, the entry may exist in only one of the Joined and Pruned source lists of a group-specific set, but not both.

(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 both the WC and RPT bits of the Encoded-Source-Address cleared.

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 that 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 would be meaningless.
- As Join/Prune messages are targeted to a single PIM neighbor, including both a (S,G) Join and a (S,G,rpt) Prune in the same message is usually redundant. The (S,G) Join informs the neighbor 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. However, there is a valid interpretation for this combination of entries. A downstream router may have to instruct its upstream only to start forwarding a specific source once it has started receiving the source on the shortest-path 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 explicitly switching back to the shared tree.

The rules are summarized in the tables below.

	Join (* ,G)	Prune (* ,G)	Join (S,G,rpt)	Prune (S,G,rpt)	Join (S,G)	Prune (S,G)
Join (* ,G)	-	no	?	yes	yes	yes
Prune (* ,G)	no	-	?	?	yes	yes
Join (S,G,rpt)	?	?	-	no	yes	?
Prune (S,G,rpt)	yes	?	no	-	yes	?
Join (S,G)	yes	yes	yes	yes	-	no
Prune (S,G)	yes	yes	?	?	no	-

	Join (* ,*,RP)	Prune (* ,*,RP)	all others
Join (* ,*,RP)	-	no	yes
Prune (* ,*,RP)	no	-	yes
all others	yes	yes	see above

yes Allowed and expected.

no Combination is not allowed by the protocol and MUST NOT be generated by a router. A router MAY accept these messages, but the result is undefined. An error message MAY be logged to the administrator in a rate-limited manner.

? 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, except where limited by the packet format itself.

4.9.5.2. Group Set Fragmentation

When building a Join/Prune for a particular neighbor, a router should try to include in the message as much of the information it needs to convey to the neighbor 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.

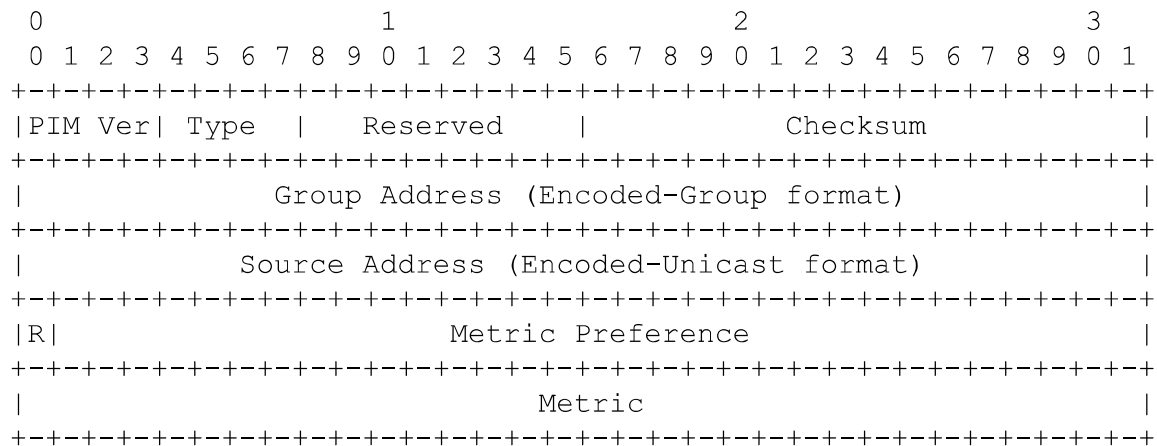
On a router with a large amount of multicast state, the number of entries that must be included may result in packets that are larger than the maximum IP packet size. In most such cases, the information may be split into multiple messages.

There is an exception with group sets that contain a (*,G) Joined source list entry. The group set expresses the router's interest in receiving all traffic for the specified group on the shared tree, and it MUST include an (S,G,rpt) Pruned source list entry for every source that the router does not wish to receive. This list of (S,G,rpt) Pruned 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 in network byte order) IP addresses.

4.9.6. Assert Message Format

The Assert message is used to resolve forwarder conflicts between routers on a link. It is sent when a router receives a multicast data packet on an interface on which the router would normally have forwarded that packet. Assert messages may also be sent in response to an Assert message from another router.



PIM Version, Type, Reserved, Checksum

Described in Section 4.9.

Group Address

The group address for which the router wishes to resolve the forwarding conflict.

This is an Encoded-Group address, as specified in Section 4.9.1.

Source Address

Source address for which the router wishes to resolve the forwarding conflict. The source address MAY be set to zero for (*,G) asserts (see below). The format for this address is given in Encoded-Unicast-Address in Section 4.9.1.

R RPT-bit is a 1-bit value. The RPT-bit is set to 1 for Assert(*,G) messages and 0 for Assert(S,G) messages.

Metric Preference

Preference value assigned to the unicast routing protocol that provided the route to the multicast source or Rendezvous-Point.

Metric

The unicast routing table metric associated with the route used to reach the multicast source or Rendezvous-Point. The metric is in units applicable to the unicast routing protocol used.

Assert messages can be sent to resolve a forwarding conflict for all traffic to a given group or for a specific source and group.

Assert(S,G)

Source-specific asserts are sent by routers forwarding a specific source on the shortest-path tree (SPTbit is TRUE). (S,G) Asserts have the Group-Address field set to the group G and the Source-Address field set to the source S. The RPT-bit is set to 0, the Metric-Preference is set to MRIB.pref(S) and the Metric is set to MRIB.metric(S).

Assert(*,G)

Group-specific asserts are sent by routers forwarding data for the group and source(s) under contention on the shared tree. (*,G) asserts have the Group-Address field set to the group G. For data-triggered Asserts, the Source-Address field MAY be set to the IP source address of the data packet that triggered the Assert and is set to zero otherwise. The RPT-bit is set to 1, the Metric-Preference is set to MRIB.pref(RP(G)), and the Metric is set to MRIB.metric(RP(G)).

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

Per interface (I):

Hello Timer: HT(I)

Per neighbor (N):

Neighbor Liveness Timer: NLT(N,I)

Per active RP (RP):

(* ,*,RP) Join Expiry Timer: ET(* ,*,RP,I)**(* ,*,RP) Prune-Pending Timer: PPT(* ,*,RP,I)**

Per Group (G):

(*,G) Join Expiry Timer: ET(*,G,I)**(*,G) Prune-Pending Timer: PPT(*,G,I)****(*,G) Assert Timer: AT(*,G,I)**

Per Source (S):

(S,G) Join Expiry Timer: ET(S,G,I)**(S,G) Prune-Pending 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) Prune-Pending 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.

Note that protocol events or configuration may change the default value of a timer on a specific interface. When timers are initialized in this document, the value specific to the interface in context must be used.

Some of the timers listed below (Prune-Pending, Upstream Join, Upstream Override) can be set to values that depend on the settings of the Propagation_Delay and Override_Interval of the corresponding interface. The default values for these are given below.

Variable Name: Propagation_Delay(I)

Value Name	Value	Explanation
Propagation_delay_default	0.5 secs	Expected propagation delay over the local link.

The default value of the Propagation_delay_default is chosen to be relatively large to provide compatibility with older PIM implementations.

Variable Name: Override_Interval(I)

Value Name	Value	Explanation
t_override_default	2.5 secs	Default delay interval over which to randomize when scheduling a delayed Join message.

Timer Name: Hello Timer (HT(I))

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

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
Default_Hello_Holdtime	$3.5 * \text{Hello_Period}$	Default holdtime to keep neighbor state alive
Hello_Holdtime	from message	Holdtime from Hello Message Holdtime option.

The Holdtime in a Hello Message should be set to $(3.5 * \text{Hello_Period})$, 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	Holdtime from Join/Prune Message

See details of JT(*,G) for the Holdtime 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(I)	Default: $\text{Effective_Propagation_Delay(I)} + \text{EffectiveOverride_Interval(I)}$	Short period after a join or prune to allow other routers on the LAN to override the join or prune

Note that both the `Effective_Propagation_Delay(I)` and the `Effective_Override_Interval(I)` are interface-specific values that may change when Hello messages are received (see Section 4.3.3).

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($1.1 * t_periodic$, $1.4 * t_periodic$) when Suppression_Enabled(I) is true, 0 otherwise	Suppression period when someone else sends a J/P message so we don't need to do so.
t_override	rand(0, Effective_Override_Interval(I))	Randomized delay to prevent response implosion when sending a join message to override someone else's Prune message.

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

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

t_override depends on the Effective_Override_Interval of the upstream interface, which may change when Hello messages are received.

t_suppressed depends on the Suppression State of the upstream interface (Section 4.3.3) and becomes zero when suppression is disabled.

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

Value Name	Value	Explanation
t_override	see Upstream Join Timer	see Upstream Join Timer

The upstream Override Timer is only ever set to t_override; this value is defined in the section on Upstream Join Timers.

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	($3 * \text{Register_Suppression_Time}$) + Register_Probe_Time	As Keepalive_Period, but at the RP when a Register-Stop 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

$\max(\text{Keepalive_Period}, \text{RP_Keepalive_Period})$ when a Register-Stop is sent.

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

Value Name	Value	Explanation
Register_Suppression_Time	Default: 60 secs	Period during which a DR stops sending Register-encapsulated data to the RP after receiving a Register-Stop message.
Register_Probe_Time	Default: 5 secs	Time before RST expires when a DR may send a Null-Register to the RP to cause it to resend a Register-Stop message.

If the Register_Suppression_Time or the Register_Probe_Time are configured to values other than the defaults, it MUST be ensured that the value of the Register_Probe_Time is less than half the value of the Register_Suppression_Time to prevent a possible negative value in the setting of the Register-Stop Timer.

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. Because 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 [7].

Values 128 through 250 are designated to be assigned for PIM by the IANA based upon IESG Approval, as defined in [9].

Values 251 through 255 are designated for Private Use, as defined in [9].

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 [9]. Such assignments are valid for one year and may be renewed. Permanent assignments require a specification (see "Specification Required" in [9].)

6. Security Considerations

This section describes various possible security concerns related to the PIM-SM protocol, including a description of how to use IPsec to secure the protocol. The reader is referred to [15] and [16] for further discussion of PIM-SM and multicast security. The IPsec authentication header [8] MAY be used to provide data integrity protection and groupwise

data origin authentication of PIM protocol messages. Authentication of PIM messages can protect against unwanted behaviors caused by unauthorized or altered PIM messages.

6.1. Attacks Based on Forged Messages

The extent of possible damage depends on the type of counterfeit messages accepted. We next consider the impact of possible forgeries, including forged link-local (Join/Prune, Hello, and Assert) and forged unicast (Register and Register-Stop) messages.

6.1.1. Forged Link-Local Messages

Join/Prune, Hello, and Assert messages are all sent to the link-local ALL_PIM_ROUTERS multicast addresses and thus are not forwarded by a compliant router. A forged message of this type can only reach a LAN if it was sent by a local host or if it was allowed onto the LAN by a compromised or non-compliant router.

1. A forged Join/Prune message can cause multicast traffic to be delivered to links where there are no legitimate requesters, potentially wasting bandwidth on that link. A forged leave message on a multi-access LAN is generally not a significant attack in PIM, because any legitimately joined router on the LAN would override the leave with a join before the upstream router stops forwarding data to the LAN.
2. By forging a Hello message, an unauthorized router can cause itself to be elected as the designated router on a LAN. The designated router on a LAN is (in the absence of asserts) responsible for forwarding traffic to that LAN on behalf of any local members. The designated router is also responsible for register-encapsulating to the RP any packets that are originated by hosts on the LAN. Thus, the ability of local hosts to send and receive multicast traffic may be compromised by a forged Hello message.
3. By forging an Assert message on a multi-access LAN, an attacker could cause the legitimate designated forwarder to stop forwarding traffic to the LAN. Such a forgery would prevent any hosts downstream of that LAN from receiving traffic.

6.1.2. Forged Unicast Messages

Register messages and Register-Stop messages are forwarded by intermediate routers to their destination using normal IP forwarding. Without data origin authentication, an attacker who is located anywhere in the network may be able to forge a Register or Register-Stop message. We consider the effect of a forgery of each of these messages next.

1. By forging a Register message, an attacker can cause the RP to inject forged traffic onto the shared multicast tree.
2. By forging a Register-stop message, an attacker can prevent a legitimate DR from Registering packets to the RP. This can prevent local hosts on that LAN from sending multicast packets.

The above two PIM messages are not changed by intermediate routers and need only be examined by the intended receiver. Thus, these messages can be authenticated end-to-end, using AH. Attacks on Register and Register-Stop messages do not apply to a PIM-SSM-only implementation, as these messages are not required for PIM-SSM.

6.2. Non-Cryptographic Authentication Mechanisms

A PIM router SHOULD provide an option to limit the set of neighbors from which it will accept Join/Prune, Assert, and Hello messages. Either static configuration of IP addresses or an IPsec security association may be used. Furthermore, a PIM router SHOULD NOT accept protocol messages from a router from which it has not yet received a valid Hello message.

A Designated Router **MUST NOT** register-encapsulate a packet and send it to the RP unless the source address of the packet is a legal address for the subnet on which the packet was received. Similarly, a Designated Router **SHOULD NOT** accept a Register-Stop packet whose IP source address is not a valid RP address for the local domain.

An implementation **SHOULD** provide a mechanism to allow an RP to restrict the range of source addresses from which it accepts Register-encapsulated packets.

All options that restrict the range of addresses from which packets are accepted **MUST** default to allowing all packets.

6.3. Authentication Using IPsec

The IPsec [8] transport mode using the Authentication Header (AH) is the recommended method to prevent the above attacks against PIM. The specific AH authentication algorithm and parameters, including the choice of authentication algorithm and the choice of key, are configured by the network administrator. When IPsec authentication is used, a PIM router should reject (drop without processing) any unauthorized PIM protocol messages.

To use IPsec, the administrator of a PIM network configures each PIM router with one or more security associations (SAs) and associated Security Parameter Indexes (SPIs) that are used by senders to authenticate PIM protocol messages and are used by receivers to authenticate received PIM protocol messages. This document does not describe protocols for establishing SAs. It assumes that manual configuration of SAs is performed, but it does not preclude the use of a negotiation protocol such as the Internet Key Exchange [14] to establish SAs.

IPsec [8] provides protection against replayed unicast and multicast messages. The anti-replay option for IPsec **SHOULD** be enabled on all SAs.

The following sections describe the SAs required to protect PIM protocol messages.

6.3.1. Protecting Link-Local Multicast Messages

The network administrator defines an SA and SPI that are to be used to authenticate all link-local PIM protocol messages (Hello, Join/Prune, and Assert) on each link in a PIM domain.

IPsec [8] allows (but does not require) different Security Policy Databases (SPD) for each router interface. If available, it may be desirable to configure the Security Policy Database at a PIM router such that all incoming and outgoing Join/Prune, Assert, and Hello packets use a different SA for each incoming or outgoing interface.

6.3.2. Protecting Unicast Messages

IPsec can also be used to provide data origin authentication and data integrity protection for the Register and Register-Stop unicast messages.

6.3.2.1. Register Messages

The Security Policy Database at every PIM router is configured to select an SA to use when sending PIM Register packets to each rendezvous point.

In the most general mode of operation, the Security Policy Database at each DR is configured to select a unique SA and SPI for traffic sent to each RP. This allows each DR to have a different authentication algorithm and key to talk to the RP. However, this creates a daunting key management and distribution problem for the network administrator.

Therefore, it may be preferable in PIM domains where all Designated Routers are under a single administrative control that the same authentication algorithm parameters (including

the key) be used for all Registered packets in a domain, regardless of who are the RP and the DR.

In this "single shared key" mode of operation, the network administrator must choose an SPI for each DR that will be used to send it PIM protocol packets. The Security Policy Database at every DR is configured to select an SA (including the authentication algorithm, authentication parameters, and this SPI) when sending Register messages to this RP.

By using a single authentication algorithm and associated parameters, the key distribution problem is simplified. Note, however, that this method has the property that, in order to change the authentication method or authentication key used, all routers in the domain must be updated.

6.3.2.2. Register-Stop Messages

Similarly, the Security Policy Database at each Rendezvous Point should be configured to choose an SA to use when sending Register-Stop messages. Because Register-Stop messages are unicast to the destination DR, a different SA and a potentially unique SPI are required for each DR.

In order to simplify the management problem, it may be acceptable to use the same authentication algorithm and authentication parameters, regardless of the sending RP and regardless of the destination DR. Although a unique SA is needed for each DR, the same authentication algorithm and authentication algorithm parameters (secret key) can be shared by all DRs and by all RPs.

6.4. Denial-of-Service Attacks

There are a number of possible denial-of-service attacks against PIM that can be caused by generating false PIM protocol messages or even by generating data false traffic. Authenticating PIM protocol traffic prevents some, but not all, of these attacks. Three of the possible attacks include:

- Sending packets to many different group addresses quickly can be a denial-of-service attack in and of itself. This will cause many register-encapsulated packets, loading the DR, the RP, and the routers between the DR and the RP.
- Forging Join messages can cause a multicast tree to get set up. A large number of forged joins can consume router resources and result in denial of service.
- Forging a (*,*,RP) join presents a possibility for a denial-of-service attack by causing all traffic in the domain to flow to the PMBR issuing the join. (*,*,RP) behavior is included here primarily for backwards compatibility with prior revisions of the spec. However, the implementation of (*,*,RP) and PMBR is optional. When using (*,*,RP), the security concerns should be carefully considered.

7. Acknowledgements

PIM-SM was designed over many years by a large group of people, including ideas, comments, and corrections 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, Mike Mroz, Garry Kump, Pavlin Radoslavov, Mike Davison, James Huang, Christopher Thomas Brown, and James Lingard.

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

8. Normative References

- [1] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.
- [2] Cain, B., Deering, S., Kouvelas, I., Fenner, B., and A. Thyagarajan, "Internet Group Management Protocol, Version 3", RFC 3376, October 2002.
- [3] Deering, S., "Host extensions for IP multicasting", STD 5, RFC 1112, August 1989.
- [4] Deering, S., Fenner, W., and B. Haberman, "Multicast Listener Discovery (MLD) for IPv6", RFC 2710, October 1999.
- [5] Deering, S. and R. Hinden, "Internet Protocol, Version 6 (IPv6) Specification", RFC 2460, December 1998.
- [6] Holbrook, H. and B. Cain, "Source-Specific Multicast for IP", RFC 4507, August 2006.
- [7] IANA, "Address Family Numbers", <<http://www.iana.org/assignments/address-family-numbers>>
- [8] Kent, S. and K. Seo, "Security Architecture for the Internet Protocol", RFC 4301, December 2005.
- [9] Narten, T. and H. Alvestrand, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 2434, October 1998.

9. Informative References

- [10] Bates, T., Rekhter, Y., Chandra, R., and D. Katz, "Multiprotocol Extensions for BGP-4", RFC 2858, June 2000.
- [11] Bhaskar, N., Gall, A., Lingard, J., and S. Venaas, "Bootstrap Router (BSR) Mechanism for PIM Sparse Mode", Work in Progress, May 2006.
- [12] Black, D., "Differentiated Services and Tunnels", RFC 2983, October 2000.
- [13] Handley, M., Kouvelas, I., Speakman, T., and L. Vicisano, "Bi-directional Protocol Independent Multicast", Work in Progress, October 2005.
- [14] Kaufman, C., "Internet Key Exchange (IKEv2) Protocol", RFC 4306, December 2005.
- [15] Savola, P., Lehtonen, R., and D. Meyer, "Protocol Independent Multicast - Sparse Mode (PIM-SM) Multicast Routing Security Issues and Enhancements", RFC 4609, August 2006.
- [16] Savola, P. and J. Lingard, "Last-hop Threats to Protocol Independent Multicast (PIM)", Work in Progress, January 2005.
- [17] Savola, P. and B. Haberman, "Embedding the Rendezvous Point (RP) Address in an IPv6 Multicast Address", RFC 3956, November 2004.
- [18] Thaler, D., "Interoperability Rules for Multicast Routing Protocols", RFC 2715, October 1999.

10. Appendix A: PIM Multicast Border Router Behavior

In some cases, PIM-SM domains will interconnect with non-PIM multicast domains. In these cases, the border routers of the PIM domain speak PIM-SM on some interfaces and speak other multicast routing protocols on other interfaces. Such routers are termed PIM Multicast Border Routers (PMBRs). In general, RFC 2715 [18] provides rules for interoperability between different multicast routing protocols. In this appendix, we specify how PMBRs differ from regular PIM-SM routers.

From the point of view of PIM-SM, a PMBR has two tasks:

- To ensure that traffic from sources outside the PIM-SM domain reaches receivers inside the domain.
- To ensure that traffic from sources inside the PIM-SM domain reaches receivers outside the domain.

We note that multiple PIM-SM domains are sometimes connected together using protocols such as Multicast Source Discovery Protocol (MSDP), which provides information about active external sources, but does not follow RFC 2715. In such cases, the domains are not connected via PMBRs because Join(S,G) messages traverse the border between domains. A PMBR is required when no PIM messages can traverse the border.

10.1. Sources External to the PIM-SM Domain

A PMBR needs to ensure that traffic from multicast sources external to the PIM-SM domain reaches receivers inside the domain. The PMBR will follow the rules in RFC 2715, such that traffic from external sources reaches the PMBR itself.

According to RFC 2715, the PIM-SM component of the PMBR will receive an (S,G) Creation event when data from an (S,G) data packet from an external source first reaches the PMBR. If RPF_interface(S) is an interface in the PIM-SM domain, the packet cannot be originated into the PIM domain at this router, and the PIM-SM component of the PMBR will not process the packet. Otherwise, the PMBR will then act exactly as if it were the DR for this source (see Section 4.4.1), with the following modifications:

- The Border-bit is set in all PIM Register messages sent for these sources.
- DirectlyConnected(S) is treated as being TRUE for these sources.
- The PIM-SM forwarding rule "iif == RPF_interface(S)" is relaxed to be TRUE if iif is any interface that is not part of the PIM-SM component of the PMBR (see Section 4.2).

10.2. Sources Internal to the PIM-SM Domain

A PMBR needs to ensure that traffic from sources inside the PIM-SM domain reaches receivers outside the domain. Using terminology from RFC 2715, there are two possible scenarios for this:

- Another component of the PMBR is a wildcard receiver. In this case, the PIM-SM component of the PMBR must ensure that traffic from all internal sources reaches the PMBR until it is informed otherwise.

Note that certain profiles of PIM-SM (e.g., PIM-SSM, PIM-SM with Embedded RP) cannot interoperate with a neighboring wildcard receiver domain.

- No other component of the PMBR is a wildcard receiver. In this case the PMBR will receive explicit information as to which groups or (source,group) pairs the external domains wish to receive.

In the former case, the PMBR will need to send a Join(*,*,RP) to all the active RPs in the PIM-SM domain. It may also send a Join(*,*,RP) to all the candidate RPs in the PIM-SM domain. This will cause all traffic in the domain to reach the PMBR. The PMBR may then act as if it were a DR with directly connected receivers and trigger the transition to a shortest path tree (see Section 4.2.1).

In the latter case, the PMBR will not need to send Join(*,*,RP) messages. However, the PMBR will still need to act as a DR with directly connected receivers on behalf of the external receivers in terms of being able to switch to the shortest-path tree for internally-reached sources.

According to RFC 2715, the PIM-SM component of the PMBR may receive a number of alerts generated by events in the external routing components. To implement the above behavior, one reasonable way to map these alerts into PIM-SM state is as follows:

- When a PIM-SM component receives an (S,G) Prune alert, it sets local_receiver_include(S,G,I) to FALSE for the discard interface.
- When a PIM-SM component receives a (*,G) Prune alert, it sets local_receiver_include(*,G,I) to FALSE for the discard interface.
- When a PIM-SM component receives an (S,G) Join alert, it sets local_receiver_include(S,G,I) to TRUE for the discard interface.
- When a PIM-SM component receives a (*,G) Join alert, it sets local_receiver_include(*,G,I) to TRUE for the discard interface.
- When a PIM-SM component receives a (*,*) Join alert, it sets DownstreamJPState(*,*,RP,I) to Join state on the discard interface for all RPs in the PIM-SM domain.
- When a PIM-SM component receives a (*,*) Prune alert, it sets DownstreamJPState(*,*,RP,I) to NoInfo state on the discard interface for all RPs in the PIM-SM domain.

We refer above to the discard interface because the macros and state machines are interface specific, but we need to have PIM state that is not associated with any actual PIM-SM interface. Implementers are free to implement this in any reasonable manner.

Note that these state changes will then cause additional PIM-SM state machine transitions in the normal way.

These rules are, however, not sufficient to allow pruning off the (*,*,RP) tree. Some additional rules provide guidance as to one way this may be done:

- If the PMBR has joined on the (*,*,RP) tree, then it should set DownstreamJPState(*,G,I) to JOIN on the discard interface for all active groups.
- If the router receives a (S,G) prune alert, it will need to set DownstreamJPState(S,G,rpt,I) to PRUNE on the discard interface.
- If the router receives a (*,G) prune alert, it will need to set DownstreamJPState(S,G,rpt,I) to PRUNE on the discard interface for all active sources sending to G.

The rationale for this is that there is no way in PIM-SM to prune traffic off the (*,*,RP) tree, except by Joining the (*,G) tree and then pruning each source individually.

11. Index

Address_List	22
Assert(*,G)	19,97
Assert(S,G)	19,96
AssertCancel(*,G)	74,75
AssertCancel(S,G)	61,69,75
AssertTimer(*,G,I)	11,17,69,99
AssertTimer(S,G,I)	12,17,63,99
AssertTrackingDesired(*,G,I)	71,72,73
AssertTrackingDesired(S,G,I)	65,66,67,68
AssertWinner(*,G,I)	11,15,17,71,74,76
AssertWinner(S,G,I)	12,15,17,66,69,76,76
AssertWinnerMetric(*,G,I)	11,74,77
AssertWinnerMetric(S,G,I)	12,69,76
assert_metric	74
Assert_Override_Interval	69,74,99
Assert_Time	69,74,99
AT(*,G,I)	11,17,69,97,99
AT(S,G,I)	12,17,63,97,99
CheckSwitchToSpt(S,G)	19,20
CouldAssert(*,G,I)	71,71,72,73,75
CouldAssert(S,G,I)	65,66,67,67,68,75
CouldRegister(S,G)	29,30
Default_Hello_Holdtime	23
DirectlyConnected(S)	19,19,20,30,106
DownstreamJPState(*,*,RP,I)	16,107
DownstreamJPState(*,G,I)	16
DownstreamJPState(S,G,I)	16,29
DownstreamJPState(S,G,rpt,I)	16
DR(I)	23
dr_is_better(a,b,I)	23,24
DR_Priority	22,23,24
Effective_Override_Interval(I)	26,86,99
Effective_Propagation_Delay(I)	25,99
ET(*,*,RP,I)	10,34,97,99
ET(*,G,I)	11,37,97,99
ET(S,G,I)	12,40,97,99
ET(S,G,rpt,I)	14,43,45,97,99
GenID	11,12,13,22,48,51,53,55,65,71
Hash_Function	9,80
Hello_Holdtime	23,99
Hello_Period	22,98
HT(I)	22,98
IGMP	5,6,11,15,77,79
immediate_olist(*,*,RP)	15,48
immediate_olist(*,G)	15,52
immediate_olist(S,G)	15,29,55
infinite_assert_metric()	75
inherited_olist(S,G)	15,19,29,32,55,66,82
inherited_olist(S,G,rpt)	15,19,20,58,60,61
I_Am_Assert_Loser(*,G,I)	17
I_Am_Assert_Loser(S,G,I)	17
I_am_DR(I)	15,24,30,66,71
I_am_RP(G)	32,32

J/P_Holdtime	35,38,41,45,49,52,56,92, 99 ,100
J/P_Override_Interval(I)	36,39,41,45,92, 99
JoinDesired(*,*,RP)	48 ,60
JoinDesired(*,G)	11, 52 ,60,66,74
JoinDesired(S,G)	13,20, 55 ,66,67,69
joins(*,*,RP(G))	15
joins(*,*,RP)	15, 16 ,66,71
joins(*,G)	15, 16 ,66,71
joins(S,G)	15, 16 ,66
JT(*,*,RP)	10,47,97, 100
JT(*,G)	11,50,97, 100
JT(S,G)	12,54,97, 100
KAT(S,G)	12,18,19,20,30,32,55,81,97, 100
KeepaliveTimer(S,G)	12,18,19,19,20,30,32,55,81,97, 100
Keepalive_Period	19, 100
lan_delay_enabled(I)	25 ,26
LAN_Prune_Delay	22
local_receiver_exclude(S,G,I)	15
local_receiver_include(*,G,I)	15 ,71,107
local_receiver_include(S,G,I)	15 ,66
local_receiver_include(S,G,I).	107
lost_assert(*,G)	15, 16 ,66
lost_assert(*,G,I)	15, 16 , 76
lost_assert(S,G)	15, 16
lost_assert(S,G,I)	15, 16 , 76
lost_assert(S,G,rpt)	16
lost_assert(S,G,rpt,I)	16, 76
MBGP	4,5
MFIB	5,9
MLD	5,6,11,15,77,79
MRIB	4,5,8,11,13,17,47,50,50,57,75,78,96
MRIB.next_hop(host)	17,17,47,48
my_assert_metric(*,G,I)	72
my_assert_metric(S,G,I)	65 ,68,70, 75
NBR(Interface,IP_address)	17 ,26,47,49,50
NLT(N,I)	10,23,97,99
OT(S,G,rpt)	14,58,97, 100
Override_Interval(I)	10,22,24,26,86, 98 ,99
packet_arrives_on_rp_tunnel(pkt)	32
pim_exclude(S,G)	15, 15 ,20,66
pim_include(*,G)	11,15, 15 ,20,66,71
pim_include(S,G)	13,15, 15 ,20,66
PPT(*,*,RP,I)	10,34,97, 99
PPT(*,G,I)	11,37,97, 99
PPT(S,G,I)	12,40,97, 99
PPT(S,G,rpt,I)	14,43,45,97, 99
Propagation_Delay(I)	22,25, 98 ,99
Propagation_delay_default	25 , 98
PruneDesired(S,G,rpt)	60 ,61,67,69
prunes(S,G,rpt)	15, 16 ,66
Register-Stop(*,G)	31
Register-Stop(S,G)	32
Register-StopTimer(S,G)	28,29,98, 101
Register_Probe_Time	29,33, 101

Register_Suppression_Time	.29,33, 101
RP(G)	4,15,17,29,32,37,52,58,66,71,75,77,97
RPF	4
RPF'(*,G)	.17,20,50,51,53,58,60,74,77
RPF'(S,G)	17,20,54,58,60,69,77
RPF'(S,G,rpt)	17,58,60,77
RPF_interface	71
RPF_interface(host)	.17,19,20,30,52,52,56,66,71,76,82,106
RPTJoinDesired(G)	60,61,71
rpt_assert_metric(G,I)	74,74, 75
RST(S,G)	28,29,98, 101
SPTbit(S,G)	13 ,19,20,32,39,56,58,60,66,66,68,69,76,81
spt_assert_metric(S,I)	69, 75 ,76
SSM	8,80
Suppression_Enabled(I)	26 ,100
SwitchToSptDesired(S,G)	20, 20 ,32
TIB	5,9,18
Triggered_Hello_Delay	22,22, 98
t_joinsuppress	48,49,51,52,56
t_override	.48,51,55,59,100, 100
t_override_default	26,98
t_periodic	48,51,55,100
t_suppressed	26,49,52,55,56,100
Update_SPTbit(S,G,iif)	19, 20
UpstreamJPState(S,G)	19 ,82

Authors' Addresses

Bill Fenner
 AT&T Labs - Research
 1 River Oaks Place
 San Jose, CA 95134

EEmail: fenner@research.att.com

Mark Handley
 Department of Computer Science
 University College London
 Gower Street
 London WC1E 6BT
 United Kingdom

EEmail: M.Handley@cs.ucl.ac.uk

Hugh Holbrook
 Arastra, Inc.
 P.O. Box 10905
 Palo Alto, CA 94303

EEmail: holbrook@arastra.com

Isidor Kouvelas
Cisco Systems
170 W. Tasman Drive
San Jose, CA 95134

EMail: kouvelas@cisco.com

Full Copyright Statement

Copyright (C) The Internet Society (2006).

This document is subject to the rights, licenses and restrictions contained in BCP 78, and except as set forth therein, the authors retain all their rights.

This document and the information contained herein are provided on an "AS IS" basis and THE CONTRIBUTOR, THE ORGANIZATION HE/SHE REPRESENTS OR IS SPONSORED BY (IF ANY), THE INTERNET SOCIETY AND THE INTERNET ENGINEERING TASK FORCE DISCLAIM ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY WARRANTY THAT THE USE OF THE INFORMATION HEREIN WILL NOT INFRINGE ANY RIGHTS OR ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

Intellectual Property

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

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

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

Acknowledgement

Funding for the RFC Editor function is provided by the IETF Administrative Support Activity (IASA).