Internet Draft
Expires January 7, 1996
File: draft-ietf-rsvp-spec-07.ps

R. Braden, Ed.
ISI
L. Zhang
PARC
D. Estrin
ISI
S. Herzog
ISI
S. Jamin
USC
July 1995

Resource ReSerVation Protocol (RSVP) – Version 1 Functional Specification

July 7, 1995

Status of Memo

Abstract

This memo describes version 1 of RSVP, a resource reservation setup protocol designed for an integrated services Internet. RSVP provides receiver-initiated setup of resource reservations for multicast or unicast data flows, with good scaling and robustness properties.

What's Changed Since Danvers IETF

The most important changes in this document from the rsvp-spec-05 draft are:

- Added fields to common header for linear fragmentation, and moved all references to semantic fragmentation to Appendix D.
- Added SE (Shared Explicit) style to all parts of the document.
- Further clarified reservation options and added table in Figure 3. Defined option vector in STYLE object.
- Renamed CREDENTIAL object class to POLICY_DATA object class, and rewrote section 2.5 to more fully express its intended usage.
- Clarified the relationship between the wildcard scope reservation option and wildcards in individual FILTER_SPEC objects: wildcard is as wildcard does.
- Added SCOPE object definition and defined the rules for its use to prevent looping of wildcard-scope messages.
- Added some mechanisms for handling backwards compatibility for future protocol extensions: (1) High bit of object class number; (2) unmerged FLOWSPEC C-Type; (3) unmerged POLICY_DATA C-Type.
- Rewrote Section 4.3 on preventing looping. Included rules for SCOPE object.
- Specified rules for local repair upon route change notification (Section 4.4).
- Specified for each error type whether or not the state information in the erroneous packet is to be stored and forwarded.
- Deleted the discussion of retransmitting a Teardown message Q times; assume Q=1 is sufficient.
- Moved Session Groups to Appendix D, *Experimental and Open Issues*. Session Groups should be revisited as part of a larger context of cross-session reservations.
- Changed common header format, removing Object Count (which was redundant) and rearranging the remaining fields. Moved the two common header flags into objects: Entry-Police into SESSION object and LUB-used into ERROR_SPEC object.
- Revised the rules for state timeout (Section 4.5) and redefined the TIME_VALUES object format.
- Changed the error message format: (1) removed required RSVP_HOP object from *PathErr* and *ResvErr* messages; (2) specified more carefully what may appear in **flow descriptor list** of *ResvErr* messages.
- Revised the definitions of error codes and error values, and moved them into a separate Appendix B.
- No longer require CREDENTIAL (i.e., POLICY_DATA) match for teardown.

- Revised routing of *ResvErr* messages to use SCOPE objects to avoid wildcard-induced looping.
- Added LIH (logical interface handle) to RSVP_HOP object, for IP multicast tunnels.
- Specified that addresses should be sorted in SCOPE object.
- Added two new upcall event types in the API: reservation event and policy data event.
- Generalized the generic traffic control calls slightly to allow multiple filter specs per flowspec, for SE style. This introduced a new set of handles, called FHandle. Also added a preemption upcall.
- Added route change notification to the generic interface to routing.
- Updated the message processing rules (Section 5).
- Rewrote Appendix C on UDP encapsulation.
- Removed specification of FLOWSPEC object format (but int-serv working group has since reneged on promise to specify it).

1 Introduction

This document defines RSVP, a resource reservation setup protocol designed for an integrated services Internet [RSVP93,ISInt93].

A host uses the RSVP protocol to request a specific quality of service (QoS) from the network, on behalf of an application data stream. RSVP is also used to deliver QoS requests to routers along the path(s) of the data stream and to maintain router and host state to provide the requested service. This will generally (but not necessarily) require reserving resources along the data path.

RSVP reserves resources for *simplex* data streams, i.e., it reserves resources in only one direction on a link, so that a sender is logically distinct from a receiver. However, the same application may act as both sender and receiver. RSVP operates on top of IP, occupying the place of a transport protocol in the protocol stack. However, like ICMP, IGMP, and routing protocols, RSVP does not transport application data but is rather an Internet control protocol. As shown in Figure 1, an implementation of RSVP, like the implementations of routing and management protocols, will typically execute in the background, not in the data forwarding path.

RSVP is not itself a routing protocol; the RSVP daemon consults the local routing protocol(s) to obtain routes. Thus, a host sends IGMP messages to join a multicast group, and it sends RSVP messages to reserve resources along the delivery path(s) from that group. RSVP is designed to operate with existing and future unicast and multicast routing protocols.

Each router that is capable of resource reservation passes incoming data packets to a packet classifier and then queues them as necessary in a packet scheduler. The packet classifier determines the route and the QoS class for each packet. The scheduler allocates resources for transmission on the particular link-layer medium used by each interface. If the link-layer medium is QoS-active, i.e., if it has its own QoS management capability, then the packet scheduler is responsible for negotiation with the link layer to obtain the QoS requested by RSVP. There are many possible ways this might be accomplished, and the details will be medium-dependent. The scheduler itself allocates packet transmission capacity on a QoS-passive medium such as a leased line. The scheduler may also allocate other system resources such as CPU time or buffers.

In order to efficiently accommodate heterogeneous receivers and dynamic group membership and to be consistent with IP multicast, RSVP makes receivers responsible for requesting resource reservations [RSVP93]. A QoS request, typically originating in a receiver host application, will be passed to the local RSVP implementation, shown as a user daemon in Figure 1. The RSVP protocol is then used to pass the request to all the nodes (routers and hosts) along the reverse data path(s) to the data source(s).

At each node, the RSVP program applies a local decision procedure, called *admission*

```
             HOST                              ROUTER

  _________________________    RSVP    _______________________
 |                         |    .---------------.             |
 |  _______      ______    |   .     | ________  .   ______   |
 | |       |    |      |   |  .      ||        |  . |      || RSVP
 | |Applic-|    | RSVP <-----        ||Routing |  -> RSVP <------>
 | |  App  <----->daemon|  |         ||Protocol|    |daemon||
 | |       |    |      |   |         || daemon <---->      ||
 | |_______|    |___.__|   |         ||_ ._____|    |__.___||
 |===|==============v=====|          |===v============v====|
 | data     ..........     |         |  .  ...........      |
 |   |  ____v_   ____v____ |         |  _v__v_    _____v___ |
 |   | |Class-| |         ||  data   | |Class-|  |         ||  data
 |   |=> ifier|=> Packet  =============> ifier|==> Packet  |======>
 |     |______| |Scheduler||         | |______|  |Scheduler||
 |              |_________||         |           |_________||
 |_________________________|         |______________________|
```

Figure 1: RSVP in Hosts and Routers

control, to determine if it can supply the requested QoS. If admission control succeeds, the RSVP program sets parameters to the packet classifier and scheduler to obtain the desired QoS. If admission control fails at any node, the RSVP program returns an error indication to the application that originated the request. We refer to the packet classifier, packet scheduler, and admission control components as *traffic control*.

RSVP is designed to scale well for very large multicast groups. Since the membership of a large group will be constantly changing, the RSVP design assumes that router state for traffic control will be built and destroyed incrementally. For this purpose, RSVP uses "soft state" in the routers, in addition to receiver-initiation.

RSVP protocol mechanisms provide a general facility for creating and maintaining distributed reservation state across a mesh of multicast or unicast delivery paths. RSVP transfers reservation parameters as opaque data (except for certain well-defined operations on the data), which it simply passes to traffic control for interpretation. Although the RSVP protocol mechanisms are largely independent of the encoding of these parameters, the encodings must be defined in the reservation model that is presented to an application (see Appendix A).

In summary, RSVP has the following attributes:

- RSVP supports multicast or unicast data delivery and adapts to changing group membership as well as changing routes.

- RSVP is simplex.
- RSVP is receiver-oriented, i.e., the receiver of a data flow is responsible for the initiation and maintenance of the resource reservation used for that flow.
- RSVP maintains "soft state" in the routers, enabling it to gracefully support dynamic membership changes and automatically adapt to routing changes.
- RSVP provides several reservation models or "styles" (defined below) to fit a variety of applications.
- RSVP provides transparent operation through routers that do not support it.

Further discussion on the objectives and general justification for RSVP design are presented in [RSVP93,ISInt93].

The remainder of this section describes the RSVP reservation services. Section 2 presents an overview of the RSVP protocol mechanisms, while Section 3 gives examples of the services and mechanism. Section 4 contains the functional specification of RSVP. Section 5 presents explicit message processing rules.

1.1 Data Flows

The set of data flows with the same unicast or multicast destination constitute a *session*. RSVP treats each session independently. All data packets in a particular session are directed to the same IP destination address DestAddress, and perhaps to some further demultiplexing point defined in a higher layer (transport or application). We refer to the latter as a *generalized destination port*.

DestAddress is the group address for multicast delivery, or the unicast address of a single receiver. A generalized destination port could be defined by a UDP/TCP destination port field, by an equivalent field in another transport protocol, or by some application-specific information. Although the RSVP protocol is designed to be easily extendible for greater generality, the present version uses only UDP/TCP ports as generalized ports.

Figure 2 illustrates the flow of data packets in a single RSVP session, assuming multicast data distribution. The arrows indicate data flowing from senders S1 and S2 to receivers R1, R2, and R3, and the cloud represents the distribution mesh created by the multicast routing protocol. Multicast distribution forwards a copy of each data packet from a sender Si to every receiver Rj; a unicast distribution session has a single receiver R. Each sender Si and each receiver Rj may correspond to a unique Internet host, or a single host may contain multiple logical senders and/or receivers, distinguished by generalized ports.

Even if the destination address is unicast, there may be multiple receivers, distinguished by the generalized port. There may also be multiple senders for a unicast destination, i.e., RSVP can set up reservations for multipoint-to-point transmission.

```
       Senders                                    Receivers
                         _____________________
                        (                     ) ===> R1
           S1 ===>      (    Multicast        )
                        (                     ) ===> R2
                        (    distribution     )
           S2 ===>      (                     )
                        (    by Internet      ) ===> R3
                        (_____________________)
```

Figure 2: Multicast Distribution Session

1.2 Reservation Model

An elementary RSVP reservation request consists of a *flowspec* together with a *filter spec*; this pair is called a *flow descriptor*. The flowspec specifies a desired QoS. The filter spec (together with the DestAddress and the generalized destination port defining the session) defines the set of data packets – the "flow" – to receive the QoS defined by the flowspec. The flowspec is used to set parameters to the node's packet scheduler (assuming that admission control succeeds), while the filter spec is used to set parameters in the packet classifier. Note that the action to control the QoS occurs at the place where the data enters the medium, i.e., at the upstream end of the link, although the RSVP reservation request originates from receiver(s) downstream.

The flowspec in a reservation request will generally include a service type and two sets of numeric parameters: (1) an *Rspec* (R for 'reserve'), which defines the desired per-hop reservation, and (2) a *Tspec* (T for 'traffic'), which defines the parameters that may be used to police the data flow, i.e., to ensure it does not exceed its promised traffic level.

The form and contents of Tspecs and Rspecs are determined by the integrated service model [ServTempl95a], and are generally opaque to RSVP. RSVP delivers the Tspec and Rspec, together with an indication whether traffic policing is needed to the admission control and packet scheduling components of traffic control. A service that requires traffic policing might for example apply it at the edge of the network and at data merge points; RSVP knows when these occur and must so indicate to the traffic control mechanism. On the other hand, RSVP cannot interpret the service embodied in the flowspec and therefore does not know whether policing will actually be applied in a particular case.

In the general RSVP reservation model [RSVP93], filter specs may select arbitrary subsets of the packets in a given session. Such subsets might be defined in terms of senders (i.e., sender IP address and generalized source port), in terms of a higher-level protocol, or generally in terms of any fields in any protocol headers in the packet. For example, filter specs might be used to select different subflows in a hierarchically-encoded signal by selecting on fields in an application-layer header. However, in the interest of simplicity (and to minimize layer violation), the present RSVP version uses a much more restricted form of filter spec: select only on sender IP address, on UDP/TCP port number, and perhaps on IP protocol id.

RSVP can distinguish subflows of a hierarchically-encoded signal if they are assigned distinct multicast destination addresses, or, for a unicast destination, distinct destination ports. Data packets that are addressed to a particular session but do not match any of the filter specs for that session are expected to be sent as best-effort traffic, and under congested conditions, such packets are likely to experience long delays, and they may be dropped. When a receiver does not wish to receive a particular (sub-)flow, it can economize on network resources by explicitly asking the network to drop unneeded the data packets; it does so by leaving the multicast group(s) to which these packets are addressed. Thus, determining where packets get delivered should be a routing function; RSVP is concerned only with the QoS of those packets that are delivered by routing.

RSVP reservation request messages originate at receivers and are passed upstream towards the sender(s). (This document defines the directional terms "upstream" vs. "downstream", "previous hop" vs. "next hop", and "incoming interface" vs "outgoing interface" with respect to the data flow direction.) When an elementary reservation request is received at a node, the RSVP daemon takes two primary actions:

1. Daemon makes a reservation

   The flowspec and the filter spec are passed to traffic control. Admission control determines the admissibility of the request (if it's new); if this test fails, the reservation is rejected and RSVP returns an error message to the appropriate receiver(s). If admission control succeeds, the node uses the flowspec to set up the packet scheduler for the desired QoS and the filter spec to set the packet classifier to select the appropriate data packets.

2. Daemon forwards the reservation upstream

   The reservation request is propagated upstream towards the appropriate senders. The set of sender hosts to which a given reservation request is propagated is called the *scope* of that request.

The reservation request that a node forwards upstream may differ from the request that it received, for two reasons. First, it is possible (in theory) for the traffic control mechanism to modify the flowspec hop-by-hop, although currently no realtime services do this. Second, reservations from different downstream branches of the multicast distribution tree(s) must be *merged* as reservations travel upstream. Merging reservations is a necessary consequence of multicast distribution, which creates a single stream of data packets in a particular router from any Si, regardless of the set of receivers downstream. The reservation for Si on a particular outgoing link L should be the "maximum" of the individual flowspecs from the receivers Rj that are downstream via link L. Merging is discussed further in Section 2.2.

The basic RSVP reservation model is *one pass*: a receiver sends a reservation request upstream, and each node in the path can only accept or reject the request. This scheme provides no way to make end-to-end service guarantees, since the QoS request must be applied independently at each hop. RSVP also supports an optional reservation model, known as *One Pass With Advertising* (OPWA) [Shenker94]. In OPWA, RSVP control packets sent downstream, following the data paths, are used to gather information on the end-to-end service that would result from a variety of possible reservation requests. The results ("advertisements") are delivered by RSVP to the receiver host, and perhaps to the receiver application. The information may then be used by the receiver to construct an appropriate reservation request.

1.3 Reservation Styles

A reservation request includes a set of control options, which are collectively called the reservation *style*.

One option concerns the treatment of reservations for different senders within the same session: establish a *distinct* reservation for each upstream sender, or else make a single reservation that is *shared* among all packets of selected senders. Another option controls the scope of the request: an *explicit* sender specification, or a *wildcard* that implicitly selects a group of senders. In an explicit-style reservation, the filter spec must match exactly one sender, while the filter spec in a wildcard reservation must match at least one sender but may match any number.

```
           ||             Reservations:
  Scope    ||     Distinct      |        Shared
  _________||_________________|___________________
           ||                 |                   |
 Explicit  ||  Fixed-Filter   |  Shared-Explicit  |
           ||  (FF) style     |  (SE) Style       |
 __________||_________________|___________________|
           ||                 |                   |
 Wildcard  ||  (None defined) |  Wildcard-Filter  |
           ||                 |  (WF) Style       |
 __________||_________________|___________________|
```

Figure 3: Reservation Attributes and Styles

The styles currently defined are as follows (see Figure 3):

1. Wildcard-Filter (WF) Style

   The WF style implies the options: *shared* reservation and *wildcard* reservation scope. Thus, a WF-style reservation creates a single reservation into which flows from all upstream senders are mixed; this reservation may be thought of as a shared "pipe", whose "size" is the largest of the resource requests for that link from all receivers, independent of the number of senders using it. A WF-style reservation has wildcard scope, i.e., the reservation is propagated upstream towards all sender hosts. A WF-style reservation automatically extends to new senders as they appear.

2. Fixed-Filter (FF) Style

   The FF style implies the options: *distinct* reservations and *explicit* reservation scope. Thus, an elementary FF-style reservation request creates a distinct reservation for data packets from a particular sender, not sharing them with other senders' packets for the same session. It scope is determined by an explicit list of senders.

   The total reservation on a link for a given session is the total of the FF reservations for all requested senders. On the other hand, FF reservations requested by different receivers Rj but selecting the same sender Si must necessarily be merged to share a single reservation in a given node.

3. Shared Explicit (SE) Style

   The SE style implies the options: *shared* reservation and *explicit* reservation scope. Thus, an SE-style reservation creates a single reservation into which flows from all upstream senders are mixed. However, like a FF reservation the set of senders (and therefore its scope (and therefore the scope) is specified explicitly by the receiver making the reservation.

WF and SE are both shared reservations, appropriate for those multicast applications whose application-specific constraints make it unlikely that multiple data sources will transmit simultaneously. One example is audio conferencing, where a limited number of people talk at once; each receiver might issue a WF or SE reservation request for twice one audio channel (to allow some over-speaking). On the other hand, the FF style, which creates independent reservations for the flows from different senders, is appropriate for video signals.

It is not possible to merge shared reservations with distinct reservations. Therefore, WF and SE styles are incompatible with FF, but are compatible with each other. Merging a WF style reservation with an SE style reservation results in a WF reservation.

Other reservation options and styles may be defined in the future (see Appendix D.4, for example).

2 RSVP Protocol Mechanisms

2.1 RSVP Messages

There are two fundamental RSVP message types: *Resv* and *Path* .

Each receiver host sends RSVP reservation request (*Resv*) messages towards the senders.

These reservation messages must follow in reverse the routes the data packets will use, all the way upstream to the sender hosts included in the scope. *Resv* messages must be delivered to the sender hosts so that the hosts can set up appropriate traffic control parameters for the first hop.

Also note that RSVP sends no positive acknowledgment messages to indicate success (although the delivery of a reservation request to a sender could be used to trigger an acknowledgement at a higher level of protocol.)

```
Sender                                      Receiver

              _____________________
  Path -->  (                     )
Si =======> (    Multicast        ) Path -->
  <-- Resv  (                     ) =========> Rj
            (    distribution     ) <-- Resv
            (_____________________)
```

Figure 4: RSVP Messages

Each sender transmits RSVP *Path* messages forward along the uni-/multicast routes provided by the routing protocol(s); see Figure 4. These *Path* messages store *path state* in each node. Path state is used by RSVP to route the *Resv* messages hop-by-hop in the reverse direction. (In the future, some routing protocols may supply reverse path forwarding information directly, replacing the reverse-routing function of path state).

Path messages may also carry the following information:

- Sender Template

  The Sender Template describes the format of data packets that the sender will originate. This template is in the form of a filter spec that could be used to select this sender's packets from others in the same session on the same link.

  Like a filter spec, the Sender Template is less than fully general at present, specifying only sender IP address, UDP/TCP sender port, and protocol id. The port number and/or protocol id can be wildcarded.

- Tspec

  Path message may optionally carry a Tspec that defines an upper bound on the traffic level that the sender will generate. This Tspec can be used by RSVP to prevent over-reservation (and perhaps unnecessary Admission Control failure) on the non-shared links starting at the sender.

- Adspec

  The *Path* message may carry a package of OPWA advertising information, known as an *Adspec*. An Adspec received in a *Path* message is passed to the local traffic control routines, which return an updated Adspec; the updated version is forwarded downstream.

```
 Previous       Incoming           Outgoing             Next
 Hops           Interfaces         Interfaces           Hops

 _____             _____________________                _____
|     | data -->  |                     |  data -->    |     |
|  A  |-----------| a                 c |--------------|  C  |
|_____|  <-- Resv |                     |   <-- Resv   |_____|
        Path -->  |                     |  Path -->     _____
 _____            |       ROUTER        |           |  |     |
|     |  |        |                     |           |--|  D  |
|  B  |--| data-->|                     |  data --> |  |_____|
|_____|  |--------| b                 d |-----------|
         |<-- Resv|                     |  <-- Resv |   _____
 _____   | Path-->|_____________________|  Path --> |  |     |
|     |  |                                          |--|  D' |
|  B' |--|                                          |  |_____|
|_____|  |                                          |
```

Figure 5: Router Using RSVP

Figure 5 illustrates RSVP's model of a router node. Each data stream arrives from a *previous hop* through a corresponding *incoming interface* and departs through one or more *outgoing interface(s)*. The same physical interface may act in both the incoming and outgoing roles (for different data flows but the same session).

As illustrated in Figure 5, there may be multiple previous hops and/or next hops through a given physical interface. This may result from the connected network being a shared medium or from the existence of non-RSVP routers in the path to the next RSVP hop (see Section 2.6). An RSVP daemon must preserve the next and previous hop addresses in its reservation and path state, respectively. A *Resv* message is sent with a unicast destination address, the address of a previous hop. *Path* messages, on the other hand, are sent with the session destination address, unicast or multicast.

Although multiple next hops may send reservation requests through the same physical interface, the final effect should be to install a reservation on that interface, which is defined by an effective flowspec. This effective flowspec will be the "maximum" of the flowspecs requested by the different next hops. In turn, a *Resv* message forwarded to a particular previous hop carries a flowspec that is the "maximum" over the effective reservations on the corresponding outgoing interfaces. Both cases represent merging, which is discussed further below.

There are a number of ways for a syntactically valid reservation request to fail in a given node:

1. The effective flowspec, computed using the new request, may fail admission control.
2. Administrative policy or control may prevent the requested reservation.
3. There may be no matching path state (i.e., the scope may be empty), which would prevent the reservation being propagated upstream.
4. A reservation style that requires a unique sender may have a filter spec that matches more than one sender in the path state, due to the use of wildcards.
5. The requested style may be incompatible with the style(s) of existing reservations for the same session on the same outgoing interface, so an effective flowspec cannot be computed.
6. The requested style may be incompatible with the style(s) of reservations that exist on other outgoing interfaces but will be merged with this reservation to create a refresh message for the previous hop.

In any of these cases, an error message is returned to the receiver(s) responsible for the erroneous message. An error message does not modify state in the nodes through which it passes. Therefore, any reservations established downstream of the node where the failure was detected will persist until the receiver(s) responsible cease attempting the reservation.

The erroneous message may or may not be propagated forward. In general, if the error is likely to be repeated at every node further along the path, it is best to drop the erroneous message rather than generate a flood of error messages; this is the case for the last four error classes listed above. The first two error classes, admission control and administrative policy, may or may not allow propagation of the message, depending upon the detailed reason and perhaps on local administrative policy and/or the particular service request. More complete rules are given in the error definitions in Appendix B.

An erroneous FILTER_SPEC object in a *Resv* message will normally be detected at the first RSVP hop from the receiver application, i.e., within the receiver host. However, an admission control failure caused by a FLOWSPEC or a POLICY_DATA object may be detected anywhere along the path(s) to the sender(s).

When admission control fails for a reservation request, any existing reservation is left in place. This prevents a new, very large, reservation from disrupting the existing QoS by merging with an existing reservation and then failing admission control (this has been called the “killer reservation” problem).

A node may be allowed to preempt an established reservation, in accordance with administrative policy; this will also trigger an error message to all affected receivers.

2.2 Merging and Packing

A previous section explained that reservation requests in *Resv* messages are necessarily merged, to match the multicast distribution tree. As a result, only the essential (i.e., the "largest") reservation requests are forwarded, once per refresh period. A successful reservation request will propagate as far as the closest point(s) along the sink tree to the sender(s) where a reservation level equal or greater than that being requested has been made. At that point, the merging process will drop it in favor of another, equal or larger, reservation request.

For protocol efficiency, RSVP also allows multiple sets of path (or reservation) information for the same session to be *packed* into a single *Path* (or *Resv*) message, respectively. (For simplicity, the protocol currently prohibits packing different sessions into the same RSVP message). Unlike merging, packing preserves information.

In order to merge reservations, RSVP must be able to merge flowspecs and to merge filter-specs. Merging flowspecs requires calculating the the "largest" of a set of flowspecs, which are otherwise opaque to RSVP. Merging flowspecs is required both to calculate the effective flowspec to install on a given physical interface (see the discussion in connection with Figure 5), and to merge flowspecs when sending a refresh message upstream. Since flowspecs are generally multi-dimensional vectors (they contain both Tspec and Rspec components, each of which may itself be multi-dimensional), they are not strictly ordered. When it cannot take the larger of two flowspecs, RSVP must compute and use a third flowspec that is at least as large as each, i.e., a *least upper bound* (LUB). It is also possible for two flowspecs to be incomparable, which is treated as an error. The definition and implementation of the rules for comparing flowspecs are outside RSVP proper, but they are defined as part of the service templates [ServTempl95a]

We can now give the complete rules for calculating the effective flowspec (Te, Re), to be installed on an interface. Here Te is the effective Tspec and Re is the effective Rspec. As an example, consider interface (d) in Figure 5.

- Re is calculated as the largest (using an LUB if necessary) of the Rspecs in *Resv* messages from different next hops (e.g., D and D') but the same outgoing interface (d).
- The Tspecs supplied in *Path* messages from different previous hops which may send data packets to this reservation (e.g., some or all of A, B, and B' in Figure 5) are summed; call this sum Path_Te.
- The maximum Tspec supplied in *Resv* messages from different next hops (e.g., D and D') is calculated; call this Resv_Te.
- Te is the GLB (greatest lower bound) of Path_Te and Resv_Te. For Tspecs defined by token bucket parameters, this means to take the smaller of the bucket size and the rate parameters.

Two filter specs can be merged only they are identical or if one contains the other through wild-carding. The result is the more general of the two, i.e., the one with more wildcard fields.

2.3 Soft State

To maintain reservation state, RSVP keeps *soft state* in router and host nodes. RSVP soft state is created and periodically refreshed by *Path* and *Resv* messages. The state is deleted if no matching refresh messages arrive before the expiration of a *cleanup timeout* interval. It may also be deleted as the result of an explicit *teardown* message, described in the next section. At the expiration of each *refresh timeout* period, RSVP scans its state to build and forward *Path* and *Resv* refresh messages to succeeding hops.

When a route changes, the next *Path* message will initialize the path state on the new route, and future *Resv* messages will establish reservation state there; the state on the now-unused segment of the route will time out. Thus, whether a message is "new" or a "refresh" is determined separately at each node, depending upon the existence of state at that node.

RSVP sends its messages as IP datagrams without reliability enhancement. Periodic transmission of refresh messages by hosts and routers is expected to replace any lost RSVP messages. To tolerate K-1 successive packet losses, the effective cleanup timeout must be at least K times the refresh timeout. In addition, the traffic control mechanism in the network should be statically configured to grant high-reliability service to RSVP messages, to protect RSVP messages from congestion losses.

The "soft" state maintained by RSVP is dynamic; to change the set of senders Si or receivers Rj or to change any QoS request, a host simply starts sending revised *Path* and/or *Resv* messages. The result should be an appropriate adjustment in the RSVP state and immediate propagation to all nodes along the path.

In steady state, refreshing is performed hop-by-hop, which allows merging and packing as described in the previous section. If the received state differs from the stored state, the stored state is updated. Furthermore, if the result will be to modify the refresh messages to be generated, these refresh messages must be generated and forwarded immediately. This will result in state changes propagating end-to-end without delay. However, propagation of a change stops when and if it reaches a point where merging causes no resulting state change. This minimizes RSVP control traffic due to changes and is essential for scaling to large multicast groups.

The RSVP state associated with a session in a particular node is divided into atomic elements that are created, refreshed, and timed out independently. The atomicity is determined by the requirement that any sender or receiver may enter or leave the session at any time, so its state should be created and timed out independently.

2.4 Teardown

RSVP *teardown* messages remove path and reservation state without waiting for the cleanup timeout period, as an optimization to release resources quickly. It is not necessary to explicitly tear down an old reservation, although it may be desirable in many cases.

A teardown request may be initiated either by an application in an end system (sender or receiver), or by a router as the result of state timeout. Once initiated, a teardown request should be forwarded hop-by-hop without delay.

Teardown messages (like other RSVP messages) are not delivered reliably. However, loss of a teardown message is not considered a problem because the state will time out even if it is not explicitly deleted. If one or more teardown message hops are lost, the router that failed to receive a teardown message will time out its state and initiate a new teardown message beyond the loss point. Assuming that RSVP message loss probability is small, the longest time to delete state will seldom exceed one refresh timeout period.

There are two types of RSVP teardown message, *PathTear* and *ResvTear*. A *PathTear* message travels towards all receivers downstream from its point of initiation and deletes path state along the way. A *ResvTear* message deletes reservation state and travels towards all senders upstream from its point of initiation. A *PathTear* (*ResvTear*) message may be conceptualized as a reversed-sense Path message (Resv message, respectively).

A teardown message deletes the specified state in the node where it is received. Like any other state change, this will be propagated immediately to the next node, but only if it represents a net change after merging. As a result, an *ResvTear* message will prune the reservation state back (only) as far as possible.

2.5 Admission Policy and Security

RSVP-mediated QoS requests will result in particular user(s) getting preferential access to network resources. To prevent abuse, some form of back pressure on users will be required. This back pressure might take the form of administrative rules, or of some form of real or virtual billing for the 'cost' of a reservation. The form and contents of such back pressure is a matter of administrative policy that may be determined independently by each administrative domain in the Internet.

Therefore, admission control at each node is likely to contain a policy component as well as a resource reservation component. As input to the policy-based admission decision, RSVP messages may carry policy data. This data may include credentials identifying users or user classes, account numbers, limits, quotas, etc.

To protect the integrity of the policy-based admission control mechanisms, it may be necessary to ensure the integrity of RSVP messages against corruption or spoofing, hop by hop. For this purpose, RSVP messages may carry integrity objects that can be created and verified by neighboring RSVP-capable nodes. These objects are expected to contain an encrypted part and to assume a shared secret between neighbors.

User policy data in reservation request messages presents a scaling problem. When a multicast group has a large number of receivers, it will not be possible or desirable to carry all the receivers' policy data upstream to the sender(s). The policy data will have to be administratively merged, near enough to the receivers to avoid excessive policy data. Administrative merging implies checking the user credentials and accounting data and then substituting a token indicating the check has succeeded. A chain of trust established using an integrity field will allow upstream nodes to accept these tokens.

Note that the merge points for policy data are likely to be at the boundaries of administrative domains. It may be necessary to carry accumulated and unmerged policy data upstream through multiple nodes before reaching one of these merge points.

2.6 Automatic RSVP Tunneling

It is impossible to deploy RSVP (or any new protocol) at the same moment throughout the entire Internet. Furthermore, RSVP may never be deployed everywhere. RSVP must therefore provide correct protocol operation even when two RSVP-capable routers are joined by an arbitrary "cloud" of non-RSVP routers. Of course, an intermediate cloud that does not support RSVP is unable to perform resource reservation, so service guarantees cannot be made. However, if such a cloud has sufficient excess capacity, it may provide acceptable and useful realtime service.

RSVP will automatically tunnel through such a non-RSVP cloud. Both RSVP and non-RSVP routers forward *Path* messages towards the destination address using their local uni-/multicast routing table. Therefore, the routing of *Path* messages will be unaffected by non-RSVP routers in the path. When a *Path* message traverses a non-RSVP cloud, the copies that emerge will carry as a Previous Hop address the IP address of the last RSVP-capable router before entering the cloud. This will effectively construct a tunnel through the cloud for *Resv* messages, which will be forwarded directly to the next RSVP-capable router on the path(s) back towards the source.

Automatic tunneling is not perfect; in some circumstances it may distribute path information to RSVP-capable routers not included in the data distribution paths, which may create unused reservations at these routers. This is because *Path* messages carry the IP source address of the previous hop, not of the original sender, and multicast routing may depend upon the source as well as the destination address. This can be overcome by manual configuration of the neighboring RSVP programs, when necessary.

2.7 Host Model

Before a session can be created, the session identification, comprised of DestAddress and perhaps the generalized destination port, must be assigned and communicated to all the senders and receivers by some out-of-band mechanism. When an RSVP session is being set up, the following events happen at the end systems.

H1 A receiver joins the multicast group specified by DestAddress, using IGMP.

H2 A potential sender starts sending RSVP *Path* messages to the DestAddress, using RSVP.

H3 A receiver application receives a *Path* message.

H4 A receiver starts sending appropriate *Resv* messages, specifying the desired flow descriptors, using RSVP.

H5 A sender application receives a *Resv* message.

H6 A sender starts sending data packets.

There are several synchronization considerations.

- Suppose that a new sender starts sending data (H6) but no receivers have joined the group (H1). Then there will be no multicast routes beyond the host (or beyond the first RSVP-capable router) along the path; the data will be dropped at the first hop until receivers(s) do appear (assuming a multicast routing protocol that "prunes off" or otherwise avoids unnecessary paths).
- Suppose that a new sender starts sending *Path* messages (H2) and immediately starts sending data (H6), and there are receivers but no *Resv* messages have reached the sender yet (e.g., because its *Path* messages have not yet propagated to the receiver(s)). Then the initial data may arrive at receivers without the desired QoS. The sender could mitigate this problem by awaiting arrival of the first *Resv* message [H5]; however, receivers that are farther away may not have reservations in place yet.
- If a receiver starts sending *Resv* messages (H4) before any *Path* messages have reached it (H3), RSVP will return error messages to the receiver. The receiver may simply choose to ignore such error messages, or it may avoid them by waiting for *Path* messages before sending *Resv* messages.

A specific application program interface (API) for RSVP is not defined in this protocol spec, as it may be host system dependent. However, Section 4.6.1 discusses the general requirements and presents a generic API.

3 Examples

We use the following notation for a *Resv* message:

1. Wildcard-Filter (WF)

   WF(*{Q})

   Here "*{Q}" represents a Flow Descriptor with a "wildcard" scope (choosing all senders) and a flowspec of quantity Q.

2. Fixed-Filter (FF)

   FF(S1{Q1}, S2{Q2}, ...)

   A list of (sender, flowspec) pairs, i.e., flow descriptors, packed into a single *Resv* message.

3. Shared Explicit (SE)

   SE((S1,S2,...)Q1, (S3,S4,...)Q2, ...)

   A list of shared reservations, each specified by a single flowspec and a list of senders.

For simplicity we assume here that flowspecs are one-dimensional, defining for example the average throughput, and state them as a multiple of some unspecified base resource quantity B.

Figure 6 shows schematically a router with two previous hops labeled (a) and (b) and two outgoing interfaces labeled (c) and (d). This topology will be assumed in the examples that follow. There are three upstream senders; packets from sender S1 (S2 and S3) arrive through previous hop (a) ((b), respectively). There are also three downstream receivers; packets bound for R1 and R2 (R3) are routed via outgoing interface (c) ((d) respectively).

In addition to the connectivity shown in 6, we must also specify the multicast routing within this node. Assume first that data packets (hence, *Path* messages) from each Si shown in Figure 6 is routed to both outgoing interfaces. Under this assumption, Figures 7, 8, and 9 illustrate Wildcard-Filter, Fixed-Filter, and Shared-Explicit reservations, respectively.

```
                     ----------------
                  (a)|                | (c)
( S1 ) ---------->|                |----------> ( R1, R2)
                     |     Router     |
                  (b)|                | (d)
( S2,S3 ) ------->|                |----------> ( R3 )
                     |________________|
```

Figure 6: Router Configuration

```
                          |
        Send              |        Reserve              Receive
                          |
                          |         _______
   WF( *{3B} ) <- (a)     |   (c) | * {B} |     (c) <- WF( *{B} )
                          |       |_______|
                          |
--------------------------|----------------------------------------
                          |         _______
   WF( *{3B} ) <- (b)     |   (d) | * {3B}|     (d) <- WF( *{3B} )
                          |       |_______|
```

Figure 7: Wildcard-Filter (WF) Reservation Example

In Figure 7, the "Receive" column shows the *Resv* messages received over outgoing interfaces (c) and (d) and the "Reserve" column shows the resulting reservation state for each interface. The "Send" column shows the *Resv* messages forwarded to previous hops (a) and (b). In the "Reserve" column, each box represents one reservation "channel", with the corresponding filter. As a result of merging, only the largest flowspec is forwarded upstream to each previous hop.

Figure 8 shows Fixed-Filter (FF) style reservations. The flow descriptors for senders S2 and S3, received from outgoing interfaces (c) and (d), are packed into the message forwarded to previous hop b. On the other hand, the two different flow descriptors for sender S1 are merged into the single message FF(S1{3B}), which is sent to previous hop (a). For each outgoing interface, there is a private reservation for each source that has been requested, but this private reservation is shared among the receivers that made the request.

Figure 9 shows a simple example of Shared-Explicit (SE) style reservations. Here each outgoing interface has a single reservation that is shared by a list of senders.

```
                     |
       Send          |       Reserve             Receive
                     |
                     |       --------
FF( S1{3B} ) <- (a)  |  (c) | S1{B}  |   (c) <- FF( S1{B}, S2{5B} )
                     |      |________|
                     |      | S2{5B} |
                     |      |________|
---------------------|---------------------------------------------
                     |       --------
             <- (b)  |  (d) | S1{3B} |   (d) <- FF( S1{3B}, S3{B} )
FF( S2{5B}, S3{B} )  |      |________|
                     |      | S3{B}  |
                     |      |________|
```

Figure 8: Fixed-Filter (FF) Reservation Example

```
                     |
       Send          |       Reserve             Receive
                     |
                     |       --------
SE( S1{3B} ) <- (a)  |  (c) |(S1,S2) |   (c) <- SE( (S1,S2){B} )
                     |      |   {B}  |
                     |      |________|
---------------------|---------------------------------------------
                     |       --------
             <- (b)  |  (d) |(S1,S3) |   (d) <- SE( (S1,S3){3B} )
SE( (S2,S3){3B} )    |      |   {3B} |
                     |      |________|
```

Figure 9: Shared-Explicit (SE) Reservation Example

The three examples just shown assume full routing, i.e., data packets from S1, S2, and S3 are routed to both outgoing interfaces. The top part of Figure 10 shows another routing assumption: data packets from S1 are not forwarded to interface (d), because the mesh topology provides a shorter path for S1 − > R3 that does not traverse this node. The bottom of Figure 10 shows WF style reservations under this assumption. Since there is no route from (a) to (d), the reservation forwarded out interface (a) considers only the reservation on interface (c); no merging takes place in this case.

```
                  _______________
              (a)|               | (c)
( S1 ) ---------->| --------->--> |----------> ( R1, R2)
                 |        /      |
                 |      /        |
              (b)|    /          | (d)
( S2,S3 ) ------->| ->----------> |----------> ( R3 )
                 |_______________|

                 Router Configuration
```

```
                                |
          Send                  |       Reserve              Receive
                                |
                                |       _______
    WF( *{B} ) <- (a)           |  (c) | * {B} |    (c) <- WF( *{B} )
                                |      |_______|
                                |
--------------------------------|---------------------------------------
                                |       _______
    WF( *{3B} ) <- (b)          |  (d) | * {3B}|    (d) <- WF( * {3B} )
                                |      |_______|
```

Figure 10: WF Reservation Example – Partial Routing

Finally, we note that state that is received through a particular interface I is never forwarded out the same interface. Conversely, state that is forwarded out interface I must be computed using only state that arrived on interfaces different from I. A trivial example of this rule is illustrated in Figure 11, which shows a transit router with one sender and one receiver on each interface (and assumes one next/previous hop per interface). Interfaces (a) and (c) are both outgoing and incoming interfaces for this session. Both receivers are making wildcard-scope reservations, in which the *Resv* messages are forwarded to all previous hops for senders in the group, with the exception of the next hop from which they came. These result in independent reservations in the two directions.

```
                    ----------------
                 a |                | c
( R1, S1 ) <----->|     Router     |<-----> ( R2, S2 )
                   |________________|

       Send                |         Receive
                           |
  WF( *{3B}) <-- (a)       |     (c) <-- WF( *{3B})
                           |
       Receive             |           Send
                           |
  WF( *{4B}) --> (a)       |     (c) --> WF( *{4B})
                           |
       Reserve on (a)      |         Reserve on (c)
        __________         |          __________
       |  * {4B}  |        |         |   * {3B} |
       |__________|        |         |__________|
                           |
```

Figure 11: Independent Reservations

4 RSVP Functional Specification

4.1 RSVP Message Formats

All RSVP messages consist of a common header followed by a variable number of variable-length typed *objects*. The subsections that follow define the formats of the common header, the object structures, and each of the RSVP message types.

For each RSVP message type, there is a set of rules for the permissible ordering and choice of object types. These rules are specified using Backus-Naur Form (BNF) augmented with square brackets surrounding optional sub-sequences.

4.1.1 Common Header

```
       0             1              2             3
+-------------+-------------+-------------+-------------+
| Vers | Flags|    Type     |       RSVP Checksum       |
+-------------+-------------+-------------+-------------+
|         RSVP Length       |        (Reserved)         |
+-------------+-------------+-------------+-------------+
|                       Message ID                      |
+----------+--+-------------+-------------+-------------+
|(Reserved)|MF|            Fragment offset              |
+----------+--+-------------+-------------+-------------+
```

The fields in the common header are as follows:

Vers: 4 bits

Protocol version number. This is version 1.

Flags: 4 bits

(None defined yet)

Type: 8 bits

1 = *Path*
2 = *Resv*
3 = *PathErr*
4 = *ResvErr*
5 = *PathTear*
6 = *ResvTear*

RSVP Checksum: 16 bits

A standard TCP/UDP checksum over the contents of the RSVP message, with the checksum field replaced by zero.

RSVP Length: 16 bits

The total length of this RSVP packet in bytes, including the common header and the variable-length objects that follow. If the MF flag is on or the Fragment Offset field is non-zero, this is the length of the current fragment of a larger message.

Message ID: 32 bits

A label shared by all fragments of one message from a given next/previous RSVP hop. An RSVP implementation assignes a unique Message ID to each message it sends.

MF: More Fragments Flag: 1 bit

This flag is the low-order bit of a byte; the seven high-order bits are reserved. It is on for all but the last fragment of a message.

Fragment Offset: 24 bits

This field gives the byte offset of the fragment in the message.

4.1.2 Object Formats

An object consists of one or more 32-bit words with a one-word header, in the following format:

```
       0             1              2             3
+-------------+-------------+-------------+-------------+
|       Length (bytes)      |  Class-Num  |   C-Type    |
+-------------+-------------+-------------+-------------+
|                                                       |
//                  (Object contents)                   //
|                                                       |
+-------------+-------------+-------------+-------------+
```

An object header has the following fields:

Length

A 16-bit field containing the total object length in bytes. Must always be a multiple of 4, and at least 4.

Class-Num

Identifies the object class; values of this field are defined in Appendix A. Each object class has a name, which will always be capitalized in this document. An RSVP implementation must recognize the following classes:

NULL

A NULL object has a Class-Num of zero, and its C-Type is ignored. Its length must be at least 4, but can be any multiple of 4. A NULL object may appear anywhere in a sequence of objects, and its contents will be ignored by the receiver.

SESSION

Contains the IP destination address (DestAddress) and possibly a generalized destination port, to define a specific session for the other objects that follow. Required in every RSVP message.

RSVP_HOP

Carries the IP address of the RSVP-capable node that sent this message. This document refers to a RSVP_HOP object as a PHOP (*previous hop*) object for downstream messages or as a NHOP (*next hop*) object for upstream messages.

TIME_VALUES

If present, contains values for the refresh period R and the state time-to-live T (see section 4.5), to override the default values of R and T.

STYLE

Defines the reservation style plus style-specific information that is not a FLOWSPEC or FILTER_SPEC object, in a *Resv* message.

FLOWSPEC

Defines a desired QoS, in a *Resv* message.

FILTER_SPEC

Defines a subset of session data packets that should receive the desired QoS (specified by an FLOWSPEC object), in a *Resv* message.

SENDER_TEMPLATE

Contains a sender IP address and perhaps some additional demultiplexing information to identify a sender, in a *Path* message.

SENDER_TSPEC

Defines the traffic characteristics of a sender's data stream, in a *Path* message.

ADSPEC

Carries an Adspec containing OPWA data, in a *Path* message.

ERROR_SPEC

Specifies an error, in a *PathErr* or *ResvErr* message.

POLICY_DATA

Carries information that will allow a local policy module to decide whether an associated reservation is administratively permitted. May appear in a *Path* or *Resv* message.

INTEGRITY

Contains cryptographic data to authenticate the originating node, and perhaps to verify the contents, of this RSVP message.

SCOPE

An explicit specification of the scope for forwarding a *Resv* message.

C-Type

Object type, unique within Class-Num. Values are defined in Appendix A.

The maximum object content length is 65528 bytes. The Class-Num and C-Type fields (together with the 'Optional' flag bit) may be used together as a 16-bit number to define a unique type for each object.

The high-order bit of the Class-Num is used to determine what action a node should take if it does not recognize the Class-Num of an object. If Class-Num < 128, then the node should ignore the object but forward it (unmerged). If Class-Num >= 128, the message should be rejected and an "Unknown Object Class" error returned. Note that merging cannot be performed on unknown object types; as a result, unmerged objects may be forwarded to the first node that does know how to merge them. The scaling limitations that this imposes must be considered when defining and deploying new object types.

4.1.3 Path Message

Path messages carry information from senders to receivers along the paths used by the data packets. The IP destination address of a *Path* message is the DestAddress for the session; the source address is an address of the node that sent the message (preferably the address of the interface through which it was sent). The PHOP (i.e., the RSVP_HOP) object of each *Path* message must contain the address of the interface through which the *Path* message was sent.

The format of a *Path* message is as follows:

```
<Path Message> ::= <Common Header> <SESSION> <RSVP_HOP>

                         [ <INTEGRITY> ]  [ <TIME_VALUES> ]

                         <sender descriptor list>

<sender descriptor list> ::= <empty > |

                  <sender descriptor list> <sender descriptor>

<sender descriptor> ::= <SENDER_TEMPLATE>  [ <SENDER_TSPEC> ]

                        [ <POLICY_DATA> ]   [ <ADSPEC> ]
```

Each **sender descriptor** defines a sender, and the **sender descriptor list** allows multiple **sender descriptor**s to be packed into a *Path* message. For each sender in the list, the SENDER_TEMPLATE object defines the format of data packets; in addition, a

SENDER_TSPEC object may specify the traffic flow, a POLICY_DATA object may specify user credential and accounting information, and an ADSPEC object may carry advertising (OPWA) data.

Each sender host must periodically send *Path* message(s) containing a **sender descriptor** for each its own data stream(s). Each **sender descriptor** is forwarded and replicated as necessary to follow the delivery path(s) for a data packet from the same sender, finally reaching the applications on all receivers (except that it is not looped back to a receiver included in the same application process as the sender).

It is an error to send ambiguous path state, i.e., two or more Sender Templates that are different but overlap, due to wildcards. For example, if we represent a Sender Template as (IP address, sender port, protocol id and use '*' to represent a wildcard, then each of the following pairs of Sender Templates would be an error:

```
(10.1.2.3, 34567, *) and (10.1.2.3, *, *)

(10.1.2.3, 34567, *) and (10.1.2.3, 34567, 17)
```

A *Path* message received at a node is processed to create path state for all senders defined by SENDER_TEMPLATE objects in the **sender descriptor list**. If present, any POLICY_DATA, SENDER_TSPEC, and ADSPEC objects are also saved in the path state. If an error is encountered while processing a *Path* message, a *PathErr* message is sent to all senders implied by the SENDER_TEMPLATEs.

Periodically, the path state is scanned to create new *Path* messages to be forwarded downstream. A node must independently compute the route for each **sender descriptor** being forwarded. These routes, obtained from uni-/multicast routing, generally depend upon the (sender host address, DestAddress) pairs and consist of a list of outgoing interfaces. The descriptors being forwarded through the same outgoing interface may be packed into as few *Path* messages as possible. Note that multicast routing of path information is based on the sender address(es) from the **sender descriptor**s, not the IP source address; this is necessary to prevent routing loops; see Section 4.3.

Multicast routing may also report the expected incoming interface (i.e., the shortest path back to the sender). If so, any *Path* message that arrives on a different interface should be discarded immediately.

It is possible that routing will report no routes for a (sender, DestAddress) pair; path state for this sender should be stored locally but not forwarded.

4.1.4 Resv Messages

Resv messages carry reservation requests hop-by-hop from receivers to senders, along the reverse paths of data flow for the session. The IP destination address of a *Resv* message is the unicast address of a previous-hop node, obtained from the path state. The IP source address is an address of the node that sent the message. The NHOP (i.e., the RSVP_HOP) object must contain the IP address of the (incoming) interface through which the *Resv* message is sent.

The *Resv* message format is as follows:

```
<Resv Message> ::= <Common Header> <SESSION>  <RSVP_HOP>

                         [ <INTEGRITY> ] [ <TIME_VALUES> ]

                         [ <S_POLICY_DATA> ] [ <SCOPE> ]

                         <STYLE> <flow descriptor list>

<S_POLICY_DATA> ::=  <POLICY DATA>

<flow descriptor list> ::=  <flow descriptor> |

                  <flow descriptor list>  <flow descriptor>
```

Here the S_POLICY_DATA object is a POLICY_DATA object that is associated with the session, i.e., with all the flows that may be listed. There may also be flow-specific POLICY_DATA objects, as described below.

The BNF above defines a **flow descriptor list** as simply a list of **flow descriptor**s. The following style-dependent rules specify more exactly the composition of a valid **flow descriptor list**.

- WF Style:

```
<flow descriptor list> ::=  <WF flow descriptor>

<WF flow descriptor> ::=

              <FLOWSPEC> [ <F_POLICY_DATA> ] <FILTER_SPEC>

<F_POLICY_DATA> ::=  <POLICY_DATA>
```

- FF style:

```
<flow descriptor list> ::=   <FF flow descriptor>  |

              <flow descriptor list> <FF flow descriptor>
```

```
<FF flow descriptor> ::=

          [ <FLOWSPEC> ] [ <F_POLICY_DATA> ] <FILTER_SPEC>
```

Each elementary FF style request is defined by a single (FLOWSPEC, FILTER_SPEC) pair, and multiple such requests may be packed into the **flow descriptor list** of a single *Resv* message. A FLOWSPEC or POLICY_DATA object can be omitted if it is identical to the most recent such object that appeared in the list.

- SE style:

```
<flow descriptor list> ::= <SE descriptor>

             | <flow descriptor list> <SE flow descriptor>

<SE flow descriptor> ::=

         <FLOWSPEC> [ <F_POLICY_DATA> ] <filter spec list>

<filter spec list> ::=  <FILTER_SPEC>

                   |  <filter spec list> <FILTER_SPEC>
```

Each elementary SE style request is defined by a single **SE descriptor**, which includes a FLOWSPEC defining the shared reservation, possibly a POLICY_DATA object, and a list of FILTER_SPEC objects. Multiple elementary requests, each representing an independent shared reservation, may be packed into the **flow descriptor list** of a single *Resv* message. A POLICY_DATA object may be omitted if it is identical to the most recent such object that appeared in the list.

The reservation scope, i.e., the set of sender hosts towards which a particular reservation is to be forwarded, is determined as follows:

- For a style with explicit scope, match each FILTER_SPEC object against the path state created from SENDER_TEMPLATE objects to select a particular sender. It is an error if a FILTER_SPEC matches more than one SENDER_TEMPLATE, due to wildcarding. A SCOPE object, if present, should be ignored.
- For a style with wildcard scope, a SCOPE object, if present, defines the scope with an explicit list of sender IP addresses (see Section 4.3 below). If there is no SCOPE object, the scope is determined by the relevant set of senders in the path state. A SCOPE object must be sent in any wildcard scope *Resv* message that is forwarded to more than one previous hop. See Section 4.3 below.

4.1.5 Error Messages

There are two types of RSVP error messages.

- *PathErr* messages result from *Path* messages and travel towards senders. *PathErr* messages are routed hop-by-hop using the path state; at each hop, the IP destination address is the unicast address of a previous hop.
- *ResvErr* messages result from *Resv* messages and travel towards the appropriate receivers. They are routed hop-by-hop using the reservation state; at each hop, the IP destination address is the unicast address of a next-hop node.

Errors encountered while processing error messages must not create further error messages.

```
<PathErr message> ::= <Common Header> <SESSION>

                            [ <INTEGRITY> ]  <ERROR_SPEC>

                            <sender descriptor>

<sender descriptor> ::= (see earlier definition)


<ResvErr Message> ::= <Common Header> <SESSION>

                            [ <INTEGRITY> ]  [S_POLICY_DATA]

                            <ERROR_SPEC>

                           <STYLE> <error flow descriptor>
```

The following style-dependent rules define the composition of a valid **error flow descriptor** in terms of sequences defined earlier:

- WF Style:

  ```
  <error flow descriptor> ::= <WF flow descriptor>
  ```

- FF style:

  ```
  <error flow descriptor> ::= <FF flow descriptor>
  ```

- SE style:

```
<error flow descriptor> ::= <SE flow descriptor>
```

POLICY_DATA objects need be included in error messages only for information when they are relevant (i.e., when an administrative failure is being reported).

The ERROR_SPEC object specifies the error and includes the IP address of the node that detected the error (Error Node Address).

When a *Path* or *Resv* message has been *packed* with multiple sets of elementary parameters, the RSVP implementation should process each set independently and return a separate error message for each that is in error.

In general, error messages should be delivered to the applications on all the session nodes that (may have) contributed to this error. A *PathErr* message is forwarded to all previous hops for all senders listed in the **Sender Descriptor List**. A *ResvErr* message is generally forwarded towards all receivers that may have caused the error being reported. More specifically:

- The node that detects an error in a reservation request creates and sends an *ResvErr* message to the next hop from which the erroneous reservation came.

  The message must contain the information required to define the error and to route the error message. Routing requires at least a STYLE object and one or more FILTER_SPEC object(s) from the erroneous *Resv* message. For an admission control failure, for example, the erroneous FLOWSPEC must be included.
- Succeeding nodes forward the *ResvErr* message using their local reservation state, to the next hops of reservations that match the FILTER_SPEC(s) in the message. For reservations with wildcard scope, there is an additional limitation on forwarding *ResvErr* messages, to avoid loops; see Section 4.3.

When the error is an admission control failure, a node is allowed (but not required) to match the FLOWSPEC as well as the FILTER_SPEC object(s), to limit the distribution of a *ResvErr* message to those receivers that 'caused' the error. Suppose that a *ResvErr* message contains a FLOWSPEC Qerr that is being matched against the FLOWSPEC Qlocal in the local reservation state in node N. Qerr, which originated in a node upstream from N, resulted from merging of flowspecs that included Qlocal. Generally, a *ResvErr* message can be forwarded to the receiver(s) that specified the 'biggest' flowspec. The comparison of Qerr against a particular Qlocal to determine whether Qlocal qualifies as (one of) the 'biggest', may be called 'de-merging'. As with merging, the details of de-merging depend upon the service and the FLOWSPEC format, and are outside RSVP itself.

A *ResvErr* message that is forwarded should carry the FILTER_SPEC from the corresponding reservation state (thus 'un-merging' the filter spec).

When a *ResvErr* message reaches a receiver, the STYLE object, **flow descriptor list**, and ERROR_SPEC object (which contains the LUB-Used flag) should be delivered to the receiver application. In the case of an Admission Control error, the **flow descriptor list** will contain the FLOWSPEC object that failed. If the LUB-Used flag is off, this should be 'equal' to (but not necessarily identical to) the FLOWSPEC originated by this application; otherwise, they may differ.

4.1.6 Teardown Messages

There are two types of RSVP Teardown message, *PathTear* and *ResvTear*.

- A *PathTear* message deletes path state (which may, in turn, delete reservation state) and travels towards all receivers that are downstream from the point of initiation. A *PathTear* message is routed like a *Path* message, and its IP destination address is DestAddress for the session.
- A *ResvTear* message deletes reservation state and travels towards all matching senders upstream from the point of teardown initiation. A *ResvTear* message is routed like a corresponding *Resv* message (using the same scope rules). Its IP destination address is the unicast address of a previous hop.

```
<PathTear Message> ::= <Common Header> <SESSION> <RSVP_HOP>

                            [ <INTEGRITY> ]

                            <sender descriptor list>

<sender descriptor list> ::= (see earlier definition)

<ResvTear Message> ::= <Common Header> <SESSION> <RSVP_HOP>

                            [ <INTEGRITY> ] [ <SCOPE> ]

                            <STYLE> <flow descriptor list>

<flow descriptor list> ::= (see earlier definition)
```

FLOWSPEC or POLICY_DATA objects in the **flow descriptor list** of a *ResvTear* message will be ignored and may be omitted.

Note that the *ResvTear* message will cease to be forwarded at the same node where merging suppresses forwarding of the corresponding *Resv* messages. The change will be propagated as a new teardown message if the result has been to remove all state for this session at this node; otherwise, it may result in the immediate forwarding of a modified *Resv* refresh message.

Deletion of path state, whether as the result of a teardown message or because of timeout, may force adjustments in related reservation state to maintain consistency in the local node. The adjustment in reservation state depends upon the style. For example, suppose a *PathTear* deletes the path state for a sender S. If the style specifies distinct reservations (FF), only reservations for sender S should be deleted; if the style specifies shared reservations (WF or SE), delete the reservation if this was the last filter spec. These reservation changes should not trigger an immediate *Resv* refresh message, since the teardown message will have already made the required changes upstream. However, at the node in which a *ResvTear* message stops, the change of reservation state may trigger a *Resv* refresh starting at that node.

4.2 Sending RSVP Messages

RSVP messages are sent hop-by-hop between RSVP-capable routers as "raw" IP datagrams with protocol number 46. Raw IP datagrams are similarly intended to be used between an end system and the first/last hop router; however, it is also possible to encapsulate RSVP messages as UDP datagrams for end-system communication, as described in Appendix C. UDP encapsulation may simplify installation of RSVP on current end systems, particularly when firewalls are in use.

Upon the arrival of an RSVP message M that changes the state, a node must forward the modified state immediatly. If this is implemented as an immediate refresh of all the state for the session, then no refresh messages should be sent out the interface through which M arrived. This rule is necessary to prevent packet storms on broadcast LANs.

An RSVP message must be fragmented when necessary to fit into the MTU of the interface through which it will be sent. All fragments of the message should carry the same unique value of the Message ID field, as well as appropriate Fragment Offset and MF bits, in their common headers. When an RSVP message arrives, it must be reassembled before it can be processed. The refresh period R is appropriate as a ressembly timeout time.

Since RSVP messages are normally expected to be generated and sent hop-by-hop, using the RSVP-level fragmentation mechanism should result in no IP fragmentation. However, IP fragmentation may occur through a non-RSVP cloud. For IP6, which does not support router fragmentation, this case will require that the RSVP implementation use Path MTU Discovery or hand configuration to obtain an appropriate MTU.

Under overload conditions, lost RSVP control messages could cause a failure of resource reservations. Routers should be configured to give a preferred class of service to RSVP packets. RSVP should not use significant bandwidth, but queueing delay and dropping of

RSVP messages needs to be controlled. Loss of RSVP packets through a congested non-RSVP cloud may still be a problem. The simplest solution is to adopt a larger value for the timeout factor K (see section 4.5 below). If this does not suffice, neighboring RSVP routers could use a TCP connection to pass RSVP messages through a non-RSVP cloud. The current protocol contains no automatic mechanism to setting up such connections; hand configuration is assumed.

Some multicast routing protocols provide for *multicast tunnels*, which encapsulate multicast packets for transmission through routers that do not have multicast capability. A multicast tunnel looks like a logical outgoing interface that is mapped into some physical interface. A multicast routing protocol that supports tunnels will describe a route using a list of logical rather than physical interfaces. RSVP can support multicast tunnels in the following manner:

1. When a node N forwards a *Path* message out a logical outgoing interface L, it includes in the message some encoding of the identity of L. This information is carried (in the HOP object) as a value called the *logical interface handle* or LIH.
2. The next hop node N' stores the LIH value in its path state.
3. When N' sends a *Resv* message to N, it includes the LIH value from the path state (again, in the HOP object).
4. When the *Resv* message arrives at N, its LIH value provides the information necessary to attach the reservation to the appropriate logical interface. Note that N creates and interprets the LIH; it is an opaque value to N'.

4.3 Avoiding RSVP Message Loops

We must ensure that the rules for forwarding RSVP control messages avoid looping. In steady state, *Path* and *Resv* messages are forwarded only once per refresh period on each hop. This avoids directly looping packets, but there is still the possibility of an *auto-refresh* loop, clocked by the refresh period. The effect of such a loop is to keep state active "forever", even if the end nodes have ceased refreshing it (but the state will be deleted when the receivers leave the multicast group and/or the senders stop sending *Path* messages). On the other hand, error and teardown messages are forwarded immediately and are therefore subject to direct looping.

- *Path* Messages

  Path messages are forwarded using routes determined by the appropriate routing protocol. For routing that is source-dependent (e.g., some multicast routing algorithms), the RSVP daemon must route each **sender descriptor** separately using the source addresses found in the SENDER_TEMPLATE objects. This should ensure that there will be no auto-refresh loops of *Path* messages, even in a topology with cycles.

Consider each message type.

- *PathTear* Messages

  PathTear messages use the same routing as *Path* messages and therefore cannot loop.

- *PathErr* Messages

  Since *Path* messages don't loop, they create path state defining a loop-free reverse path to each sender. *PathErr* messages are always directed to particular senders and therefore cannot loop.

- *Resv* Messages

  Like *PathErr* message, *Resv* messages directed to particular senders (i.e., with explicit scope) cannot loop. However, there is a potential for auto-refresh of *Resv* messages with wildcard scope; the solution is presented below.

- *ResvTear* Messages

  ResvTear messages are routed the same as *Resv* messages and have an analogous looping problem for wildcard scope.

- *ResvErr* Messages

  ResvErr messages for wildcard scope reservations have the same potential for looping as the reservations themselves, and the solution presented below is required.

If the topology has no loops, then looping of wildcard-scoped messages can be avoided by simply enforcing the rule given earlier: state that is received through a particular interface must never be forwarded out the same interface. However, when the topology does have cycles then further effort is needed to prevent auto-refresh loops in wildcard-scope *Resv*, *ResvTear*, and *ResvErr* messages. The solution is for such messages to carry an explicit sender address list in a SCOPE object.

When a *Resv* or *ResvTear* message with wildcard scope is to be forwarded to a particular previous hop, a new SCOPE object is computed from the SCOPE objects that were received (in messages of the same type). If the computed SCOPE object is empty, the message is not forwarded to the previous hop; otherwise, the message is sent containing the new SCOPE object. The rules for computing a new SCOPE object for a *Resv* or *ResvTear* message are as follows:

1. The union is formed of the sets of sender IP addresses listed in all SCOPE objects in the reservation state for the given session.

   If reservation state from some NHOP does not contain a SCOPE object, a substitute sender list must be created and included in the union. For a wildcard scope (WF) message that arrived on outgoing interface OI, the substitute list is the set of senders that route to OI. For an explicit scope (SE) message, it is the set of senders explicitly listed in the message.

2. Any local senders (i.e., any sender applications on this node) are removed from this set.

```
                  ----------------
              a  |                | c
    R4, S4<----->|     Router     |<-----> R2, S2, S3
                 |                |
              b  |                |
    R1, S1<----->|                |
                 |________________|

   Send on (a):            |    Receive on (c):
                           |
      <-- WF( [S4] )       |       <-- WF( [S4, S1])
                           |
   Send on (b):            |
                           |
      <-- WF( [S1] )       |
                           |
   Receive on (a):         |    Send on (c):
                           |
      WF( [S1,S2,S3]) -->  |       WF( [S2, S3]) -->
                           |
   Receive on (b):         |
                           |
      WF( [S2,S3,S4]) -->  |
                           |
```

Figure 12: SCOPE Objects in Wildcard-Scope Reservations

3. If the SCOPE object is to be sent to PHOP, remove from the set any senders that did not come from PHOP.

Figure 12 shows an example of wildcard-scoped (WF style) *Resv* messages. The address lists within SCOPE objects are shown in square brackets. Note that there may be additional connections among the nodes, creating looping topology that is not shown.

SCOPE objects are not necessary if the multicast routing uses shared trees or if the reservation style has explicit scope. Furthermore, attaching a SCOPE object to a reservation may be deferred to a node which has more than one previous hop upstream.

The following rules are used for SCOPE objects in wildcard-scoped *ResvErr* messages:

1. The node that detected the error initiates an *ResvErr* message containing a copy of the SCOPE object associated with the reservation state or message in error.

2. Suppose a wildcard-scoped *ResvErr* message arrives at a node with a SCOPE object containing the sender host address list L. The node forwards the *ResvErr* message using the rules of Section 4.1.5. However, the *ResvErr* message forwarded out OI must contain a SCOPE object derived from L by including only those senders that route to OI. If this SCOPE object is empty, the *ResvErr* message should not be sent out OI.

4.4 Local Repair

When a route changes, the next *Path* or *Resv* refresh will establish path or reservation state (respectively) along the new route. To provide fast adaptation to routing changes without the overhead of short refresh periods, the local routing protocol module can notify the RSVP daemon of route changes for particular destinations. The RSVP daemon should use this information to trigger an immediate refresh of state for these destinations, using the new route.

More specifically, the rules are as follows:

- When routing detects a change of the set of outgoing interfaces for sending *Path* messages for destination G, RSVP sends immediate *Path* refreshes for all sessions G/* (i.e., for any session with destination G, regardless of destination port). Such refresh messages are to be sent to at least the new outgoing interfaces for these sessions.
- When a *Path* message arrives with a Previous Hop address that differs from the one stored in the path state, RSVP should send immediate *Resv* refreshes for that session.

4.5 Time Parameters

There are two time parameters relevant to each element of RSVP path or reservation state in a node: the refresh period R between receiving successive refreshes for the state, and its lifetime L. Each RSVP *Resv* or *Path* message may contain a TIME_VALUES object specifying the R value that was used to generate this refresh message; this is used to determine the L when the state is received and stored.

In more detail:

1. To avoid premature loss of state, we require that L >= (K + 0.5)* R, where K is a small integer. Then K-1 successive messages may be lost without state being deleted. Currently K = 3 is suggested.
2. Each message will generally carry a TIME_VALUES object containing the R used to generate refreshes; the recipient node uses this R to determine L of the stored state.

However, if a default R = Rdef is used, the TIME_VALUES object may be omitted from a message. Rdef is currently defined to be 30 seconds.

3. This document does not specify the interval R to be used for generating refresh messages. If the node does not implement local repair of reservations disrupted by route changes, a smaller R improves the speed of adapting to routing changes (but increases overhead). With local repair, a router can be more relaxed about R since the periodic refresh becomes only a backstop robustness mechanism. A node may therefore adjust the effective R dynamically to limit the overhead due to refresh messages.
4. The TIME_VALUES object could contain, in addition to the hop-by-hop R value, an end-to-end upper bound on R, called Rmax. When Rmax is specified, a node cannot set R > Rmax. However, a node is allowed to refuse an RSVP message (i.e., drop it and return an error) when it specifies an Rmax value that is so small that it would create unacceptable overhead. This refusal would look like a kind of admission control failure.
5. However, when R is changed dynamically, there is a limit to how fast it may increase. Specifically, the ratio of two successive values R2/R1 must not exceed 1 + Slew.Max.

   Currently, Slew.Max is 0.30. With K = 3, one packet may be lost without state timeout while R is increasing 30 percent per refresh cycle.
6. To improve robustness, a node may temporarily send refreshes more often than R after a state change (including initial state establishment).
7. A node should randomize its refresh timeouts to avoid synchronization and burstiness of refreshes.
8. The values of Rdef, K, and Slew.Max used in an implementation should be easily modifiable, as experience may lead to different values. The possibility of dynamically changing K and/or Slew.Max in response to measured loss rates is for future study.

4.6 RSVP Interfaces

RSVP on a router has interfaces to routing and to traffic control. RSVP on a host has an interface to applications (i.e, an API) and also an interface to traffic control (if it exists on the host).

4.6.1 Application/RSVP Interface

This section describes a generic interface between an application and an RSVP control process. The details of a real interface may be operating-system dependent; the following can only suggest the basic functions to be performed. Some of these calls cause information to be returned asynchronously.

- Register

```
Call: REGISTER( DestAddress , DestPort

                [ , SESSION_object ]  , SND_flag , RCV_flag

                [ , Source_Address ]  [ , Source_Port ]

                [ , Source_ProtID ]  [ , Sender_Template ]

                [ , Sender_Tspec ]   [ , Data_TTL ]

                [ , Sender_Policy_Data ]

                [ , Upcall_Proc_addr ] )  -> Session-id
```

  This call initiates RSVP processing for a session, defined by DestAddress together with the TCP/UDP port number DestPort. If successful, the REGISTER call returns immediately with a local session identifier Session-id, which may be used in subsequent calls.

  The SESSION_object parameter is included as an escape mechanism to support some more general definition of the session ("generalized destination port"), should that be necessary in the future. Normally SESSION_object will be omitted; if it is supplied, it should be an appropriately-formatted representation of a SESSION object.

  SND_flag should be set true if the host will send data, and RCV_flag should be set true if the host will receive data. Setting neither true is an error. The optional parameters Source_Address, Source_Port, Sender_Template, Sender_Tspec, Data_TTL, and Sender_Policy_Data are all concerned with a data source, and they will be ignored unless SND_flag is true.

If SND_FLAG is true, a successful REGISTER call will cause RSVP to begin sending *Path* messages for this session using these parameters, which are interpreted as follows:

- Source_Address

  This is the address of the interface from which the data will be sent. If it is omitted, a default interface will be used. This parameter is needed on a multihomed sender host.
- Source_Port

  This is the UDP/TCP port from which the data will be sent. If it is omitted or zero, the port is "wild" and can match any port in a FILTER_SPEC.
- Source_ProtID

  This is the IP protocol ID for the sender data. If it is omitted or zero, the protocol id is "wild" and can match any protocol id in a FILTER_SPEC.
- Sender_Template

  This parameter is included as an escape mechanism to support a more general definition of the sender (*generalized source port*). Normally this parameter may be omitted; if it is supplied, it should be an appropriately formatted representation of a SENDER_TEMPLATE object.
- Sender_Tspec

  This parameter is a Tspec describing the traffic flow to be sent. It may be included to prevent over-reservation on the initial hops.
- Data_TTL

  This is the (non-default) IP Time-To-Live parameter that is being supplied on the data packets. It is needed to ensure that Path messages do not have a scope larger than multicast data packets.
- Sender_Policy_Data

  This optional parameter passes policy data for the sender. This data may be supplied by a system service, with the application treating it as opaque.

Finally, Upcall_Proc_addr is the address of an upcall procedure to receive asynchronous error or event notification; see below.

- Reserve

```
Call: RESERVE( session-id,

                         style, style-dependent-parms )
```

A receiver uses this call to make a resource reservation for the session registered as 'session-id'. The style parameter indicates the reservation style. The rest of the parameters depend upon the style, but generally these will include appropriate flowspecs, filter specs, and possibly receiver policy data objects.

The first RESERVE call will initiate the periodic transmission of *Resv* messages. A later RESERVE call may be given to modify the parameters of the earlier call (but note that changing the reservations may result in admission control failure, depending upon the style).

The RESERVE call returns immediately. Following a RESERVE call, an asynchronous ERROR/EVENT upcall may occur at any time.

- Release

```
Call: RELEASE( session-id )
```

This call will terminate RSVP state for the session specified by session-id. It may send appropriate teardown messages and will cease sending refreshes for this session-id.

- Error/Event Upcalls

```
Upcall: <Upcall_Proc>( ) -> session-id, Info_type,

                [ Error_code , Error_value , LUB-Used, ]

                List_count, [ Flowspec_list,]

                [ Filter_spec_list, ] [ Advert_list, ]

                [ Policy_data ]
```

Here *Upcall_Proc* represents the upcall procedure whose address was supplied in the REGISTER call.

This upcall may occur asynchronously at any time after a REGISTER call and before a RELEASE call, to indicate an error or an event. Currently there are three upcall types, distinguished by the Info_type parameter:

1. Info_type = Path Event

   A Path Event upcall indicates to a receiver application that there is at least one active sender. It results from receipt of the first *Path* message for this session.

   This upcall provides synchronizing information to the receiver application, and it may also provide parallel lists of senders (in Filter_spec_list), traffic descriptions (in Flowspec_list), and service advertisements (in Advert_list). 'List_count'will be the number in each list; where these objects are missing, corresponding null objects must appear. The Error_code, Error_value, LUB-Used flag, and Policy_data parameters will be undefined in this upcall.

2. Info_type = Resv Event

   A Resv Event upcall indicates to a sender application that a reservation for this session in place along the entire path to at least one receiver. It is triggered by the receipt of the first reservation message or by modification of previous reservation state, for this session.

   'List_count' will be 1, and Flowspec_list will contain one FLOWSPEC, the effective QoS that would be applicable to the application itself. Filter_spec_list and Advert_list will contain one NULL object. The Error_code, Error_value, LUB-Used flag, and Policy_data parameters will be undefined in this upcall.

3. Info_type = Path Error

   An Path Error event indicates an error in sender information that was specified in the REGISTER call.

   The Error_code parameter will define the error, and Error_value may supply some additional (perhaps system-specific) data about the error. 'List_count' will be 1, and Filter_spec_list and Flowspec_list will contain the Sender_Template supplied in the REGISTER call; Sender_Tspec and Advert_list will each contain one NULL object. The Policy_data parameter will be undefined in this upcall.

4. Info_type = Resv Error

   An Resv Error event indicates an error in processing a reservation message to which this application contributed. The Error_code parameter will define the error, and Error_value may supply some additional (perhaps system-specific) data on the error.

   Filter_spec_list and Flowspec_list will contain the FILTER_SPEC and FLOWSPEC objects from the **error flow descriptor** (see Section 4.1.5). List_count will specify the number of FILTER_SPECS in Filter_spec_list, while there will be one FLOWSPEC in Flowspec_list. The Policy_data parameter will be undefined in this upcall.

5. Info_type = Policy Data

   A Policy Information upcall passes a Policy_data parameter containing policy information (accounting, current costs, prices, quota, etc.) that arrived at the receiver.

   List_count will be zero, and the Error_code, Error_value, and LUB-Used flag parameters will be undefined in this upcall.

Although RSVP messages indicating path events or errors may be received periodically, the API should make the corresponding asynchronous upcall to the application only on the first occurrence, or when the information to be reported changes.

4.6.2 RSVP/Traffic Control Interface

In each router and host, enhanced QoS is achieved by a group of inter-related traffic control functions: a packet classifier, an admission control module, and a packet scheduler. This section describes a generic RSVP interface to traffic control.

1. Make a Reservation

```
Call: Rhandle =  TC_AddFlowspec( Interface, Flowspec

                            [ , Sender_Tspec]
```

```
, E_Police_Flag , M_Police_Flag )
```

This call passes a Flowspec defining a desired QoS to admission control. It may also pass Sender_Tspec, the maximum traffic characteristics computed over the SENDER_TSPECs of senders that will contribute data packets to this reservation.

E_Police_Flag and M_Police_Flag are Boolean parameters. E_Police_Flag is on if this is an entry node, while M_Police is on if this node is an interior data merge point for a shared reservation style. These flags are used to enable traffic policing or shaping when appropriate, in accordance with the service.

This call returns an error code if Flowspec is malformed or if the requested resources are unavailable. Otherwise, it establishes a new reservation channel corresponding to Rhandle. It returns the opaque number Rhandle for subsequent references to this reservation.

2. Modify Reservation

   ```
   Call: TC_ModFlowspec( Rhandle, new_Flowspec

                       [ , Sender_Tspec] , Police_flag )
   ```

   This call can modify an existing reservation. If new_Flowspec is included, it is passed to Admission Control; if it is rejected, the current flowspec is left in force. The corresponding filter specs, if any, are not affected.

3. Delete Flowspec

   ```
   Call: TC_DelFlowspec( Rhandle )
   ```

   This call will delete an existing reservation, including the flowspec and all associated filter specs.

4. Add Filter Spec

   ```
   Call: FHandle = TC_AddFilter( Rhandle, Session , FilterSpec )
   ```

   This call is used to associate an additional filter spec with the reservation specified by the given Rhandle, following a successful TC_AddFlowspec call. This call returns a filter handle FHandle.

5. Delete Filter Spec

   ```
   Call: TC_DelFilter( FHandle )
   ```

   This call is used to remove a specific filter, specified by FHandle.

6. OPWA Update

```
Call: TC_Advertise( interface, Adspec

                [ ,Sender_TSpec ] ) -> New_Adspec
```

This call is used for OPWA to compute the outgoing advertisement New_Adspec for a specified interface. Sender_TSpec is also passed if it is available.

7. Preemption Upcall

```
Upcall: TC_Preempt() -> RHandle, Reason_code
```

In order to grant a new reservation request, the admission control and/or policy modules may be allowed to preempt an existing reservation. This might be reflected in an upcall to RSVP, passing the RHandle of the preempted reservation, and some indication of the reason.

4.6.3 RSVP/Routing Interface

An RSVP implementation needs the following support from the packet forwarding and routing mechanisms of the node.

- Promiscuous receive mode for RSVP messages

  Any datagram received for IP protocol 46 must be diverted to the RSVP program for processing, without being forwarded. The identity of the interface on which it is received should also be available to the RSVP daemon.

- Route Query

  RSVP must be able to query the routing daemon for the route(s) for forwarding a specific datagram.

```
Ucast_Route_Query( DestAddress, Notify_flag ) -> OutInterface

Mcast_Route_Query( SrcAddress, DestAddress, Notify_flag )

                              -> OutInterface_list
```

  If the Notify_flag is True, routing will save state necessary to issue unsolicited route change notification callbacks whenever the specified route changes. This will continue until routing receives a route query call with the Notify_Flag set False.

- Route Change Notification

  If requested by a route query with the Notify_flag True, the routing daemon may provide an asynchronous callback to RSVP that a specified route has changed.

```
Ucast_Route_Change( ) ->   DestAddress, OutInterface

Mcast_Route_Change( )

             -> SrcAddress, DestAddress, OutInterface_list
```

- Outgoing Link Specification

  RSVP must be able to force a (multicast) datagram to be sent on a specific outgoing virtual link, bypassing the normal routing mechanism. A virtual link may be a real outgoing link or a multicast tunnel. Outgoing link specification is necessary because RSVP may send different versions of outgoing *Path* messages for the same source and destination addresses on different interfaces. It is also necessary in some cases to avoid routing loops.

- Discover Interface List

  RSVP must be able to learn what real and virtual interfaces are active, with their IP addresses.

5 Message Processing Rules

This generic description of RSVP operation assumes the following data structures. An actual implementation may use additional or different structures to optimize processing.

- PSB – Path State Block

  Each PSB holds path state for a particular (session, sender) pair, which are defined by SESSION and SENDER_TEMPLATE objects, respectively. PSB contents include a PHOP object and possibly SENDER_TSPEC, POLICY_DATA, and/or ADSPEC objects from *Path* messages.

- RSB – Reservation State Block

  Each RSB holds reservation state for a particular 4-tuple: (session, next hop, style, filterspec), which are defined in SESSION, NHOP, STYLE, and FILTER_SPEC objects, respectively. RSB contents also include a FLOWSPEC object and may include a POLICY_DATA object. We assume that RSB contents include the outgoing interface OI that is implied by NHOP.

MESSAGE ARRIVES

Verify version number, checksum, and length fields of common header, and discard message if any mismatch is found.

Further processing depends upon message type.

PATH MESSAGE ARRIVES

Each **sender descriptor** object sequence in the message defines a sender. Process each sender as follows, starting the Path_Refresh_Needed and Resv_Refresh_Needed flags off.

1. If there is a POLICY_DATA object, verify it; if it is unacceptable, build and send a "Administrative Rejection" *PathErr* message, drop the *Path* message, and return.
2. Call the appropriate Route_Query routine, using DestAddress from SESSION and (for multicast routing) SrcAddress from SENDER_TEMPLATE. This provides a routing bit mask ROUTE_MASK and (for a multicast destination) an EXPECTED_INTERFACE.
3. If the message arrived on an interface different from EXPECTED_INTERFACE, drop it and return.
4. Search for a path state block (PSB) whose (SESSION, SENDER_TEMPLATE) pair matches the corresponding objects in the message.

If there is a match considering wildcards in the SENDER_TEMPLATE objects, but the two SENDER_TEMPLATEs differ, build and send a "Ambiguous Path" *PathErr* message, drop the *Path* message, and return.

5. If there is no matching PSB for the (SESSION, SENDER_TEMPLATE) pair then:
   - Create a new PSB.
   - Set a cleanup timer for the PSB. If this is the first PSB for the session, set a refresh timer for the session.
   - Copy the SESSION, TIME_VALUES, and PHOP objects into the PSB. Copy into the PSB any of the following objects that are present: POLICY_DATA, SENDER_TSPEC, and ADSPEC.
   - Store ROUTE_MASK and EXPECTED_INTERFACE in the PSB.
   - Turn on the Path_Refresh_Needed flag.
6. Otherwise (there is a matching PSB):
   - Restart cleanup timer.
   - If the SENDER_TSPEC and/or ADSPEC values differ between the message and the PSB, copy the new values into the PSB and turn on the Path_Refresh_Needed flag. Note that if SEND_TSPEC has changed, reservations matching S may also change; this may be deferred until a *Resv* refresh arrives.
   - If the new ROUTE_MASK differs from that stored in the PSB, turn on the Path_Refresh_Needed flag, and store the new ROUTE_MASK into the PSB.
   - If the new EXPECTED_INTERFACE differs from that stored in the PSB, turn on the Resv_Refres_Needed flag and store the new EXPECTED_INTERFACE value into the PSB.
7. Save the IP TTL with which the message arrived in the PSB .
8. If the Path_Refresh_Needed flag is now set, execute the PATH REFRESH event sequence (below); however, send no *Path* refresh messages out the interface through which the *Path* message arrived.
9. If the Resv_Needed flag is now set, execute the RESV REFRESH event sequence (below).

PATH TEAR MESSAGE ARRIVES

- If there is no path state for this destination, drop the message and return.
- Forward a copy of the *PathTear* message using the same rules as for a *Path* message (see PATH REFRESH).
- Each **sender descriptor** in the *PathTear* message contains a SENDER_TEMPLATE object defining a sender S; process it as follows.
  1. Locate the PSB for the pair: (session, S). If none exists, continue with next **sender descriptor**.
  2. Examine the RSB's for this session and delete reservation state that is associated with sender S and no other sender.

3. Delete the PSB.

- Drop the *PathTear* message and return.

PATH ERROR MESSAGE ARRIVES

- If there are no existing PSB's for SESSION then drop the *PathErr* message and return.
- Look up the PSB for (session, sender); sender is defined by SENDER_TEMPLATE. If no PSB is found, drop *PathErr* message and return.
- If PHOP in PSB is local API, deliver error to application via an upcall:

```
Call: <Upcall_Proc>( session-id, Path Error,
               Error_code, Error_value, 0,
               1, SENDER_TEMPLATE, NULL, NULL, NULL)
```

  Any POLICY_DATA, SENDER_TSPEC, or ADSPEC object in the message is ignored.
- Otherwise (PHOP is not local API), forward a copy of the *PathErr* message to the PHOP node.

RESV MESSAGE ARRIVES

A *Resv* message arrives through outgoing interface OI.

- Check the SESSION object.

  If there are no existing PSB's for SESSION then build and send a *ResvErr* message (as described later) specifying "No path information", drop the *Resv* message, and return. However, do not send the *ResvErr* message if the style has wildcard reservation scope and this is not the receiver host itself.
- Check the STYLE object.

  If the style in the message conflicts with the style of any reservation for this session in place on any interface, reject the *Resv* message by building and sending a *ResvErr* message specifying "Conflicting Style", drop the *Resv* message, and return.
- Check the POLICY_DATA object.

  Verify the POLICY_DATA field (if any) to check permission to create a reservation. If it is unacceptable, build and send an "Administrative rejection" *ResvErr* message, drop the *Resv* message, and return.
- Make reservations

  Process the STYLE object and the **flow descriptor list**.

  For FF style, execute the following steps for each flow descriptor, i.e., for each (FLOWSPEC, FILTER_SPEC) pair. For SE style, execute the following steps for each FILTER_SPEC in the list, using the given FLOWSPEC. For WF style, execute the following once, using an internal placeholder "WILD_FILTER" for FILTERSPEC if it is omitted.

1. Find or create a reservation state block (RSB) for the 4-tuple: (SESSION, NHOP, style, FILTER_SPEC).
2. Start or restart the cleanout timer on the RSB. Start a refresh timer for this session if none was started.
3. If the RSB existed and contains state matching this flow descriptor, continue with the next flow descriptor. Otherwise (the state is new or modified), continue processing the current flow descriptor with the following steps.
4. Scan the set of PSBs (senders) whose SENDER_TSPECs match FILTER_SPEC.
   – If this set is empty, build and send an error message specifying "No sender information", and continue with the next flow descriptor.
   – If this set contains more than one PSB and if the style has the explicit option (e.g., FF or SE), build and send an error message specifying "Ambiguous filter spec" and continue with the next flow descriptor.
   – Set K_E_Police_flag on if any of these PSBs have the E_Police flag on, otherwise set K_E_Police_flag off. Set K_M_Police_flag on if the style has wildcard scope and there is more than one PSB in the scope, otherwise, set K_M_Police_flag off.
   – Compute K_Tspec as the sum of the SENDER_TSPEC objects, if any, in this set of PSBs.
5. Compute the parameters for the effective reservation, by considering all RSB's for the same (SESSION, OI, FILTERSPEC) triple.
   – Compute the effective kernel flowspec, K_Flowspec, as the maximum of the FLOWSPEC values in these RSB's
   – Compute the effective kernel filter spec K_Filter by merging the FILTER_SPEC objects in these RSB's.
6. If this reservation has wildcard scope and this is not the first flow descriptor in the message, one of the filter specs must have changed; delete the old one and install the new:

   ```
   TC_DelFilter( old_Fhandle );

   Fhandle = TC_AddFilter( Rhandle, SESSION, K_filter)
   ```

   Then continue with the next flow descriptor.
7. Otherwise, if there was no previous kernel reservation in place for (SESSION, OI, FILTERSPEC), call the kernel interface module:

   ```
   Rhandle = TC_AddFlowspec( OI, K_flowspec, K_Tspec,
                             K_E_Police_flag, K_M_Police_flag )
   ```

   If this call fails, build and send a *ResvErr* message specifying "Admission control failed", and continue with the next flow descriptor. Otherwise, record the kernel handle Rhandle returned by the call in the RSB(s). Then call:

   ```
   TC_AddFilter( Rhandle, SESSION, K_Filter)
   ```

to set the filter, and continue with the next flow descriptor.
However, if there was a previous kernel reservation with handle Rhandle, and the flowspec has changed, call:

```
TC_ModFlowspec( Rhandle, K_Flowspec, K_Tspec,
                        K_E_Police_flag, K_M_Police_flag )
```

If this call fails, build and send a *ResvErr* message specifying "Admission control failed". In any case, drop the *Resv* message and return.
If the flowspec is unchanged but the filter spec has changed, install the new:

```
TC_DelFilter( old_Fhandle )
        Fhandle = TC_AddFilter( Rhandle, SESSION, K_filter)
```

Then continue with the next flow descriptor.

If processing a *Resv* message finds an error, a *ResvErr* message is created containing flow descriptor and an ERRORS object. The Error Node field of the ERRORS object (see Appendix A) is set to the IP address of OI, and the message is sent unicast to NHOP.

RESV TEAR MESSAGE ARRIVES

A *ResvTear* message arrives on outgoing interface OI.

- Initialize flag Tear_Needed to False.
- Execute the following steps for each flow descriptor, i.e., each (FLOWSPEC, FILTERSPEC) pair, in the **flow descriptor list**:
  1. Find matching RSB for the 4-tuple: (SESSION, NHOP, style, FILTER_SPEC). If no RSB is found, continue with next flow descriptor.
  2. Delete the RSB.
  3. If there are no more RSBs for the same (SESSION, OI, FILTER_SPEC) triple, call the kernel interface to delete the reservation:

     ```
     TC_DelFlowspec( K_handle )
     ```

     and set Tear_Needed to True.
  4. Otherwise (there are other RSB's for the same reservation), recompute K_Flowspec and call the kernel interface module:

     ```
     TC_ModFlowspec( K_handle, K_Flowspec, Sender_Tspec)
     ```

     to update the reservation. If this kernel call fails, return; the prior reservation will remain in place.

- If Tear_Needed is False (the resulting merged state may have changed but is still in place), then execute the RESV REFRESH sequence below, drop *ResvTear* message, and return.
- Otherwise, need to create new *ResvTear* message for each PHOP, and perhaps some *Resv* refresh messages.

  Set Refresh_Needed flag to False. Do the following for each sender Si (in the path stat) whose ROUTE_MASK includes the outgoing interface OI and for each PHOP:

  1. Pick each flow descriptor Fj in the *ResvTear* message whose FILTER_SPEC matches Si, and do the following.
     - If there is no RSB whose FILTER_SPEC matches Si, then add Fj to the new *ResvTear* message being built.
     - Otherwise (there is a matching RSB), note the incoming interface of Si as an interface needing a *Resv* refresh message and set the Refresh_Needed flag True.
  2. If the new *ResvTear* message contains any flow descriptors, forward it to PHOP.

     If the scope is wildcard, include only a single flow descriptor in the message.
- If the Refresh_Needed flag is true, then execute the RESV_REFRESH sequence below, for the incoming interfaces that have been noted.

RESV ERROR MESSAGE ARRIVES

- If there is no state for SESSION, then drop the *ResvErr* mesasge and return.
- For each RSB, do the following. Note that an RSB implies an outgoing interface OI and a next hop NHOP.
  1. If OI differs from the incoming interface through which the *ResvErr* message arrived, continue with the next RSB.
  2. Compare the FILTER_SPEC(s) in the **error flow descriptor** with the FILTER_SPEC(s) in the RSB. If no match, continue with the next RSB.

     Otherwise, form a new **error flow descriptor** with the subset of FILTER_SPECs that matched.
  3. Compare the FLOWSPEC in the *ResvErr* message with the FLOWSPEC in the RSB. If they don't match along any coordinate (i.e., if the RSB FLOWSPEC is strictly 'smaller'), continue with the next RSB.

     If they agree on some but not all coordinates, turn on the LUB-used flag.
  4. If NHOP in PSB is local API, deliver error to application via an upcall:

```
Call: <Upcall_Proc>( session-id, Resv Error, k,
             Error_code, Error_value, LUB-Used,
               Filter\_Spec\_List, Flowspec\_List, NULL,
  NULL)
```

and continue with the next RSB. Here k, Filter_Spec_List, and Flowspec_List are constructed from the new **error flow descriptor**.

5. If the *Resv* message has wildcard scope, use its SCOPE object SC.In to construct a SCOPE object SC.Out to be forwarded. SC.Out should contain those sender addresses that appeared in SC.In and that route to OI [LIH?], as determined by scanning the PSB's. If SC.Out is empty, continue with the next RSB.
6. Create a new *ResvErr* message containing the new **error flow descriptor** and send to the NHOP address specified by the RSB. Include SC.Out if the scope is wildcard.
7. Continue with the next RSB.

- Drop the *ResvErr* message and return.

PATH REFRESH

This sequence may be entered by either the expiration of the path refresh timer for a particular session, or immediately as the result of processing a *Path* message turning on the Path_Refresh_Needed flag.

For each outgoing interface OI, build a *Path* message and send it to OI. To build the message, consider each PSB whose ROUTE_MASK includes OI, and do the following:

- Pass the ADSPEC and SENDER_TSPEC objects present in the PSB to the kernel call TC_Advertise, and get back a modified ADSPEC object. Pack this modified object into the *Path* message being built.
- Create a **sender descriptor** sequence containing the SENDER_TEMPLATE, SENDER_TSPEC, and POLICY_DATA objects, if present in the PSB. Pack the **sender descriptor** into the *Path* message being built.
- If the PSB has the E_Police flag on and if interface OI is not capable of policing, turn the E_Police flag on in the *Path* message being built.
- Compute the IP TTL for the *Path* message as one less than the maximum of the TTL values from the senders included in the message. However, if the result is zero, return without sending the *Path* message.
- If the maximum size of the *Path* message is reached, send the packet out interface OI and start packing a new one.

RESV REFRESH

This sequence may be entered by either the expiration of the reservation refresh timer for a particular session, or immediately as the result of processing a *Resv* or *ResvTear* message.

For each PHOP defined by the path state, scan the RSBs, merge the style, FLOWSPECs and FILTER_SPECs appropriately, build a new *Resv* message, and send it to PHOP. Each message carries a NHOP object containing the local address of the interface through which it is sent.

The details of building the *Resv* messages depend upon the shared/distinct option of the style. For each PHOP, do the following:

- Distinct style

  Select each sender Si (PSB) for PHOP, and do the following:

  1. Select all RSB's whose FILTER_SPECs match the SENDER_TEMPLATE object for Si and whose OI matches a bit in the ROUTE_MASK of the PSB for Si.
  2. Compute the maximum over the FLOWSPEC objects of this set of RSB's, and merge their FILTER_SPEC, STYLE, and POLICY_DATA objects.
  3. Append the (FLOWSPEC, FILTER_SPEC pair) to the *Resv* message being built for destination PHOP. When the packet fills, or upon completion of all PSB's with the same PHOP, send it.

- Shared style

  1. Select each sender Si (PSB) for PHOP, and select all RSB's that: (a) have an OI matching a bit in the ROUTE_MASK for Si, and (b) contain at least one FILTER_SPEC that matches the SENDER_TEMPLATE object for Si.
  2. For all selected RSB's for all Si corresponding to a given PHOP:
     - Compute the maximum over the FLOWSPEC objects of this set of RSB's.
     - Merge the metching FILTER_SPEC objects; this will in general result in a list of non-overlapping FILTER_SPECs, but where there are overlaps due to wildcards, use the 'wildest'.
     - Merge the STYLE and POLICY_DATA objects.
     - Place the resulting merged objects into a *Resv* message and send it to PHOP.
  3. If the scope is wildcard, a forwarded *Resv* must contain a SCOPE object. The set of IP addresses in the SCOPE object sent to a given PHOP is formed as follows.
     - Take the union of the senders listed in SCOPE objects in all RSB's.
     - Intersect that set with the set of sender hosts listed in path state for PHOP.
     - If the resulting set is empty, no *Resv* should be forwarded to this PHOP.

A Object Definitions

C-Types are defined for the two Internet address families IPv4 and IP6. To accomodate other address families, additional C-Types could easily be defined. These definitions are contained as an Appendix, to ease updating.

All unused fields should be sent as zero and ignored on receipt.

A.1 SESSION Class

SESSION Class = 1.

- IPv4/UDP SESSION object: Class = 1, C-Type = 1

```
+-------------+-------------+-------------+-------------+
|              IPv4 DestAddress (4 bytes)               |
+-------------+-------------+-------------+-------------+
|   //////    |    Flags    |          DestPort         |
+-------------+-------------+-------------+-------------+
```

- IP/UDP SESSION object: Class = 1, C-Type = 2

```
+-------------+-------------+-------------+-------------+
|                                                       |
+                                                       +
|                                                       |
+                IP6 DestAddress (16 bytes)             +
|                                                       |
+                                                       +
|                                                       |
+-------------+-------------+-------------+-------------+
|  ///////    |     Flags   |          DestPort         |
+-------------+-------------+-------------+-------------+
```

DestAddress

The IP unicast or multicast destination address of the session.

Flags

0x01 = E_Police flag

The E_Police flag is used in *Path* messages to determine the effective "edge" of the network, to control traffic policing. If the sender host is not itself capable of traffic policing, it will set this bit on in *Path* messages it sends. The first node whose RSVP is capable of traffic policing will do so (if appropriate to the service) and turn the flag off.

[It might make more sense to include this flag in ADSPEC object.]

DestPort

The UDP/TCP destination port for the session. Zero may be used to indicate a 'wildcard', i.e., any port.

Other SESSION C-Types could be defined in the future to support other demultiplexing conventions in the transport-layer or application layer.

A.2 RSVP_HOP Class

RSVP_HOP class = 3.

- IPv4 RSVP_HOP object: Class = 3, C-Type = 1

```
+-------------+-------------+-------------+-------------+
|             IPv4 Next/Previous Hop Address            |
+-------------+-------------+-------------+-------------+
|                 Logical Interface Handle              |
+-------------+-------------+-------------+-------------+
```

- IP6 RSVP_HOP object: Class = 3, C-Type = 2

```
+-------------+-------------+-------------+-------------+
|                                                       |
+                                                       +
|                                                       |
+             IP6 Next/Previous Hop Address             +
|                                                       |
+                                                       +
|                                                       |
+-------------+-------------+-------------+-------------+
|                 Logical Interface Handle              |
+-------------+-------------+-------------+-------------+
```

This object provides the IP address of the interface through which the last RSVP-knowledgeable hop forwarded this message. The Logical Interface Handle is a 32-bit number which may be used to distinguish logical outgoing interfaces as described in Section 4.2; it should be identically zero if there is no logical interface handle.

A.3 INTEGRITY Class

INTEGRITY class = 4.

See draft-ietf-rsvp-md5-00.txt.

A.4 TIME_VALUES Class

TIME_VALUES class = 5.

- TIME_VALUES Object: Class = 5, C-Type = 1

```
+-------------+-------------+-------------+-------------+
|                   Refresh Period                      |
+-------------+-------------+-------------+-------------+
|                 Max Refresh Period                    |
+-------------+-------------+-------------+-------------+
```

Refresh Period

The refresh timeout period R used to generate this message; in milliseconds.

Max Refresh Period

The largest R value that a node is allowed to apply to the downstream state for this session. A node may refuse to accept this requirement, by ignoring the message containing this TIME_VALUES object and sending a "R too small" error message.

If this value is zero, no limit is set.

A.5 ERROR_SPEC Class

ERROR_SPEC class = 6.

- IPv4 ERROR_SPEC object: Class = 6, C-Type = 1

```
+-------------+-------------+-------------+-------------+
|            IP4 Error Node Address (4 bytes)           |
+-------------+-------------+-------------+-------------+
|    Flags    | Error Code  |        Error Value        |
+-------------+-------------+-------------+-------------+
```

- IP6 ERROR_SPEC object: Class = 6, C-Type = 2

```
+-------------+-------------+-------------+-------------+
|                                                       |
+                                                       +
|                                                       |
+           IP6 Error Node Address (16 bytes)           +
|                                                       |
+                                                       +
|                                                       |
+-------------+-------------+-------------+-------------+
|    Flags    | Error Code  |        Error Value        |
+-------------+-------------+-------------+-------------+
```

Error Node Address

The IP address of the node in which the error was detected.

Flags

0x01 = LUB-Used

The use of this flag is described in section 4.1.5.

Error Code

A one-octet error description.

Error Value

A two-octet field containing additional information about the error. Its contents depend upon the Error Type.

The values for Error Code and Error Value are defined in Appendix B.

A.6 SCOPE Class

SCOPE class = 7.

This object contains a list of IP addresses, used for routing messages with wildcard scope without loops. The addresses must be listed in ascending numerical order.

- IPv4 SCOPE List object: Class = 7, C-Type = 1

```
+-------------+-------------+-------------+-------------+
|                IP4 Src Address (4 bytes)              |
+-------------+-------------+-------------+-------------+
//                                                     //
+-------------+-------------+-------------+-------------+
|                IP4 Src Address (4 bytes)              |
+-------------+-------------+-------------+-------------+
```

- IP6 SCOPE list object: Class = 7, C-Type = 2

```
+-------------+-------------+-------------+-------------+
|                                                       |
+                                                       +
|                                                       |
+                IP6 Src Address (16 bytes)             +
|                                                       |
+                                                       +
|                                                       |
+-------------+-------------+-------------+-------------+
//                                                     //
+-------------+-------------+-------------+-------------+
|                                                       |
+                                                       +
|                                                       |
+                IP6 Src Address (16 bytes)             +
|                                                       |
+                                                       +
|                                                       |
+-------------+-------------+-------------+-------------+
```

A.7 STYLE Class

STYLE class = 8.

- STYLE object: Class = 8, C-Type = 1

```
+-------------+-------------+-------------+-------------+
|   Style ID  |              Option Vector              |
+-------------+-------------+-------------+-------------+
```

Style ID

An integer identifying the style, as follows:

0 = No ID assigned; use option vector.
1 = WF
2 = FF
3 = SE

Option Vector

A set of bit fields giving values for the reservation options. If new options are added in the futre, corresponding fields in the option vector will be assigned from the least-significant end. If a node does not recognize a style ID, it may interpret as much of the option vector as it can, ignoring new fields that may have been defined.

The option vector bits are assigned (from the left) as follows:

19 bits: Reserved

2 bits: Sharing control

00b: Reserved
01b: Distinct reservations
10b: Shared reservations
11b: Reserved

3 bits: Scope control

000b: Reserved
001b: Wildcard scope
010b: Explicit scope
011b - 111b: Reserved

The low order bits of the option vector are determined by the style id, as follows:

```
WF 10001b
FF 01010b
SE 10010b
```

A.8 FLOWSPEC Class

FLOWSPEC class = 9.

- Class = 9, C-Type = 1: int-serv flowspec

  The contents of this object will be specified in documents prepared by the int-serv working group.

- Class = 9, C-Type = 254: Unmerged Flowspec List

```
+-------------+-------------+-------------+-------------+
|                                                       |
//                FLOWSPEC object  1                   //
|                                                       |
+-------------+-------------+-------------+-------------+
|                                                       |
//                FLOWSPEC object  2                   //
|                                                       |
+-------------+-------------+-------------+-------------+
//                                                     //
//                                                     //
+-------------+-------------+-------------+-------------+
|                                                       |
//                FLOWSPEC object  k                   //
|                                                       |
+-------------+-------------+-------------+-------------+
```

  This is a container C-Type, used to enclose a set of FLOWSPEC objects that could not be merged at the next hop downstream because they include unrecognized C-Types. The node that receives this object may merge those it recognizes and forward the rest in another Unmerged Flowspec List object.

A.9 FILTER_SPEC Class

FILTER_SPEC class = 10.

- IPv4 FILTER_SPEC object: Class = 10, C-Type = 1

```
+-------------+-------------+-------------+-------------+
|               IPv4 SrcAddress (4 bytes)               |
+-------------+-------------+-------------+-------------+
| Protocol Id |    //////   |          SrcPort          |
+-------------+-------------+-------------+-------------+
```

- IP6 FILTER_SPEC object: Class = 10, C-Type = 2

```
+-------------+-------------+-------------+-------------+
|                                                       |
+                                                       +
|                                                       |
+               IP6 SrcAddress (16 bytes)               +
|                                                       |
+                                                       +
|                                                       |
+-------------+-------------+-------------+-------------+
| Protocol Id |    //////   |          SrcPort          |
+-------------+-------------+-------------+-------------+
```

- IP6 Flow-label FILTER_SPEC object: Class = 10, C-Type = 3

```
+-------------+-------------+-------------+-------------+
|                                                       |
+                                                       +
|                                                       |
+               IP6 SrcAddress (16 bytes)               +
|                                                       |
+                                                       +
|                                                       |
+-------------+-------------+-------------+-------------+
|   ///////   |         Flow Label (24 bits)            |
+-------------+-------------+-------------+-------------+
```

SrcAddress

The IP source address for a sender host, or zero to indicate a 'wildcard'.

Protocol Id

The IP protocol Identifier, or zero to indicate a 'wildcard'.

SrcPort

The UDP/TCP source port for a sender, or zero to indicate a 'wildcard' (i.e., any port).

Flow Label

A 24-bit Flow Label, defined in IP6. This value may be used by the packet classifier to efficiently identify the packets belonging to a particular (sender− >destination) data flow.

A.10 SENDER_TEMPLATE Class

SENDER_TEMPLATE class = 11.

- IPv4/UDP SENDER_TEMPLATE object: Class = 11, C-Type = 1
  Definition same as IPv4/UDP FILTER_SPEC object.
- IP6/UDP SENDER_TEMPLATE object: Class = 11, C-Type = 2
  Definition same as IP6/UDP FILTER_SPEC object.

A.11 SENDER_TSPEC Class

SENDER_TSPEC class = 12.

The only current form of Tspec is a token bucket.

- Token Bucket SENDER_TSPEC object: Class = 12, C-Type = 1

```
+-----------+-----------+-----------+-----------+
|        b: Token Bucket Depth (bits)           |
+-----------+-----------+-----------+-----------+
|        r: Average data rate (bits/sec)        |
+-----------+-----------+-----------+-----------+
```

A.12 ADSPEC Class

ADSPEC class = 13.

[TBD]

A.13 POLICY_DATA Class

POLICY_DATA class = 14.

- Type 1 POLICY_DATA object: Class = 14, C-Type = 1

  [TBD]

- Unmerged POLICY_DATA object: Class = 14, C-Type = 254

  This object is a container for a list of POLICY_DATA objects (none of which may have C-Type = 254). The contained objects have not yet been merged.

```
+-------------+-------------+-------------+-------------+
|                                                       |
//                POLICY\_DATA object  1               //
|                                                       |
+-------------+-------------+-------------+-------------+
|                                                       |
//                POLICY\_DATA object  2               //
|                                                       |
+-------------+-------------+-------------+-------------+
//                                                     //
//                                                     //
+-------------+-------------+-------------+-------------+
|                                                       |
//                POLICY\_DATA object  k               //
|                                                       |
+-------------+-------------+-------------+-------------+
```

B Error Codes and Values

The following Error Codes are defined.

- Error Code = 01: Admission failure

  Reservation rejected by admission control.

  For this Error Code, the 16 bits of the Error Value field are:

      ussr cccc cccc cccc

  where the bits are:

  u = 0: RSVP should reject the message without updating local state.

  u = 1: RSVP may use message to update local state and forward it.

  ss = 00: Low order 12 bits contain a globally-defined sub-code (values listed below).

  ss = 10: Low order 12 bits contain a sub-code that is specific to local organization. RSVP is not expected to be able to interpret this except as a numeric value.

  ss = 11: Low order 12 bits contain a sub-code that is specific to the service. RSVP is not expected to be able to interpret this except as a numeric value. Since the traffic control mechanism might substitute a different service, this encoding may include some representation of the service in use.

  r: Reserved bit, should be zero.

  cccc cccc cccc: 12 bit code.

  The following globally-defined sub-codes may appear in the low-order 12 bits when uu = 00:

  - Sub-code = 1: Delay bound cannot be met
  - Sub-code = 2: Requested bandwidth unavailable
  - Sub-code = 11: Service conflict
  - Sub-code = 12: Service unsupported

    Traffic control can provide neither the requested service nor an acceptable substitute.
  - Sub-code = 13: Bad Flowspec or Tspec value

    Unreasonable request. High order 4 bits should be 000r, so that RSVP will reject the message.
  - Sub-code = 14: Rmax value too small.

    Rmax would result in excessive refresh overhead.

- Error Code = 02: Administrative rejection

  Reservation has been rejected for administrative reasons.

  For this Error Code, the high order 4 bits of the Error Value field are assigned as for Code = 01 (above). For this case, the following global sub-codes may be used:

  - Sub-code = 1: Required credential(s) not presented.
  - Sub-code = 2: Request too large

    Reservation request exceeds allowed value for this user class.
  - Sub-code = 3: Insufficient quota or balance.
  - Sub-code = 4: Administrative preemption

- Error Code = 03: No path information for this Resv

  RSVP should reject the message.

- Error Code = 04: No sender information for this Resv

  There is path information, but it does not include the sender specified in any of the Filterspecs listed in the Resv message. RSVP should reject the message.

- Error Code = 05: Ambiguous path

  Sender specification is ambiguous with existing path state. RSVP should reject the message.

- Error Code = 06: Ambiguous filter spec

  Filter spec matches more than one sender, in style that requires a unique match. RSVP should reject the message.

- Error Code = 07: Conflicting or unknown style

  Reservation style conflicts with style(s) of existing reservation state, or it is unknown. If the high-order bit of Error Value is zero, RSVP should reject the message.

- Error Code = 11: Missing required object

  RSVP was unable to find or construct required object data from message. Error Value will be Class-Num that is missing. RSVP should reject the message.

- Error Code = 12: Unknown object class

  Error Value will contain 16-bit value composed of (Class-Num, C-Type) of unknown object. This error should be sent only if RSVP is going to reject the message.

- Error Code = 13: Unknown object C-Type

  Error Value will contain 16-bit value composed of (Class-Num, C-Type) of object. This error should be sent only if RSVP is going to should reject the message.

- Error Code = 14: Object error

  A non-specific error indicating bad format or contents of an object. The Error Value will contain 16-bits value (Class-Num, C-Type) from header of bad object. RSVP should reject the message.

- Error Code = 21: Traffic Control error

  Some system error was detected and reported by the traffic control modules. The Error Value will contain a system-specific value giving more information about the error.

- Error Code = 22: RSVP System error

  The Error Value field will provide implementation- dependent information on the error.

C UDP Encapsulation

As described earlier, RSVP control messages are intended to be carried directly within IP datagrams as "raw packets". Implementing RSVP in a node will require an intercept in the packet forwarding path for protocol 46, and the necessary kernel change is incorporated in the recent releases of IP multicasting

There are particular circumstances where it may be desirable to encapsulate RSVP messages in UDP packets, as a short-term measure.

1. UDP encapsulation can be used between hosts and the last- (or first-) hop router(s). This may ease installing RSVP on some host systems, by avoiding a kernel change for the RSVP intercept.
2. UDP encapsulation may be useful for legal penetration of firewalls.
3. UDP encapsulation might be used on each interface of an intermediate RSVP router whose kernel supported multicast but which did not have the RSVP intercept.

In the following discussion, we concentrate on (1) and (2).

Figure 13 shows a typical situation for a host running RSVP. Here two RSVP-capable hosts Hu and Hr within a corporation are connected to the Internet through some arbitrarily complex set of networks and routers that is labelled the "Corporate cloud". The border router R is assumed to be RSVP-capable, but the corporate cloud is not.

```
                 - - - -
 ------        (         )      RSVP-capable
|      |      (           )       router
|  Hu  |-----(  Corporate  )      ------
|______|      (           )     a|      |b
             (    cloud    )-----|  R   |---->Internet
 ------       (           )      |______|
|      |     (             )
|  Hr  |------(           )
|______|       (_ _ _ _ _)
```

Figure 13: End Host Situation

We assume that Hu is a "UDP-only" host that requires UDP encapsulation, while Hr is a "raw-capable" host that can use raw RSVP packets. The UDP encapsulation scheme should allow RSVP interoperation among an arbitrary topology of Hr hosts and Hu hosts as well as routers R.

Resv messages are always sent unicast; once path state has been established, the unicast destination address of each *Resv* message is known. If the path state also indicates whether the next host node needs UDP encapsulation, a *Resv* message can simply be sent to the next-hop node, either in raw mode or with UDP encapsuation.

UDP encapsulation of PATH messages poses a more difficult problem. To solve it, we define two new well-known multicast addresses G1 and G2, and a well-known UDP port Pu. Then the table in Figure 14 shows the rules. Under the 'Send' column, the notation is $< mode >$(destaddr, destport, TTL), where TTL is the IP-layer hop count. The 'Receive' column shows the group that is joined and, where relevant, the UDP Listen port. T1 and T2 are configured IP TTL values used for encapsulation, while Tr is the local TTL value of the specific *Path* message. Finally, D is the DestAddress for the particular session.

Node	Node Type	Send	Receive
Hu	UDP-only host	UDP(G1,Pu,T1)	UDP(G1,Pu) and UDP(G2,Pu)
Hr	Raw-mode host	UDP(G1,Pu,T1) and Raw(D,,Tr)	UDP(G1,Pu) and Raw()
R	Router		
	Interface a:	UDP(G2,Pu,T2) and Raw(D,,Tr)	UDP(G1,Pu) and Raw()
	Interface b:	Raw(D,,Tr)	Raw()

Figure 14: UDP Encapsulation Rules for Path Messages

Note that R and Hr must send their *Path* messages twice, once with UDP encapsulation and once in raw mode. In two cases (Hr $- >$ R and Hr $- >$ Hr), each *Path* message will be delivered twice. The router may take steps to ignore the duplicates, but this redundancy actually has no ill effect other than overhead for processing the extra messages.

A router must keep track of which of its interfaces are using UDP encapsulation and which are not. A node can always listen for UDP(G1,Pu) on each interface, and if it receives a UDP-encapsulated *Path* message, mark the corresponding path state as UDP-needed. Then matching *Resv* messages will be correctly encapsulated.

Two provisions are necessary for this automatic determination of encapsulation to work.

C1 A router must use different groups G1 and G2 for sending and receiving, as already shown.

C2 The TTL value T1 used by a host must be exactly enough to reach the router R.

If T1 is too small to pass through the corporate cloud, of course *Path* messages will not be forwarded. If T1 is too large, multicast routing in R will forward the UDP packet into the Internet until its hop count expires. This will turn on UDP encapsulation between routers within the Internet, causing bogus UDP traffic. (Note that UDP packets addressed to G2 by a router will not be received by a neighboring router).

However, there are possible situations where it will be impossible to find a value of T1 that meets condition C2. Within the corporate cloud there might be a multicast tunnel with an outgoing threshold larger than the hop count through the cloud. Another possibility is that there might be more than one border router R, with different TTL's. There are several possible ways that C2 might be satisfied in such cases.

1. It might be possible to configure the hosts' RSVP daemons with the IP address for R; the daemons could then *unicast Path* messages to this address. This solution will be feasible as long as the number of Hr and Hu hosts is small.
2. A particular host on the LAN including Hu could be designated as an *RSVP relay host*. This system would listen on (G1,Pu) and be configured with the IP address of R. It could then forward any (*Path*) messages it received directly to R, and T1 could be set only large enough to reach local hosts and the relay.

Finally, manual configuration of T1 could be replaced by an expanding ring search conducted by host RSVP daemons. This possibility is for future study.

D Experimental and Open Issues

D.1 Reservation Compatability

How strong is the requirement for compatability of reservations in different directions? For example, see Figure 11; should it be possible to have incompatible reservation styles on the two interfaces? If R1 requests a WF reservation and R2 requests a FF reservation, it is logically possible to make the corresponding reservations on the two different interfaces. The current implementation does NOT allow this; instead, it prevents mixing of incompatible styles in the same session on a node, even if they are on different interfaces.

D.2 Session Groups (Experimental)

Section 1.2 explained that a distinct destination address, and therefore a distinct session, will be used for each of the subflows in a hierarchically encoded flow. However, these separate sessions are logically related. For example it may be necessary to pass reservations for all subflows to Admission Control at the same time (since it would be nonsense to admit high frequency components but reject the baseband component of the session data). Such a logical grouping is indicated in RSVP by defining a *session group*, an ordered set of sessions.

To declare that a set of sessions form a session group, a receiver includes a data structure we call a SESSION_GROUP object in the *Resv* message for each of the sessions. A SESSION_GROUP object contains four fields: a reference address, a session group ID, a count, and a rank.

- The reference address is an agreed-upon choice from among the DestAddress values of the sessions in the group, for example the smallest numerically.
- The session group ID is used to distinguish different groups with the same reference address.
- The count is the number of members in the group.
- The rank, an integer between 1 and count, is different in each session of the session group.

The SESSION_GROUP objects for all sessions in the group will contain the same values of the reference address, the session group ID, and the count value. The rank values establishes the desired order among them.

If RSVP at a given node receives a *Resv* message containing a SESSION_GROUP object, it should wait until *Resv* messages for all 'count' sessions have appeared (or until the end of the refresh cycle) and then pass the *Resv* requests to Admission Control as a group. It is normally expected that all sessions in the group will be routed through the same nodes. However, if not, only a subset of the session group reservations may appear at a given node; in this case, the RSVP should wait until the end of the refresh cycle and then perform Admission Control on the subset of the session group that it has received. The rank values will identify which are missing.

Note that routing different sessions of the session group differently will generally result in delays in establishing or rejecting the desired QoS. A *bundling* facility could be added to multicast routing, to force all sessions in a session group to be routed along the same path.

D.2.1 Resv Messages

Add:

```
[ <SESSION_GROUP> ]
```

after the SESSION object.

D.2.2 SESSION_GROUP Class

SESSION_GROUP class = 2.

- IPv4 SESSION_GROUP Object: Class = 2, C-Type = 1:

```
+-------------+-------------+-------------+-------------+
|               IPv4 Reference DestAddress              |
+-------------+-------------+-------------+-------------+
|      Session_Group ID     |    Count    |     Rank    |
+-------------+-------------+-------------+-------------+
```

- IP6 SESSION_GROUP Object: Class = 2, C-Type = 2:

```
+-------------+-------------+-------------+-------------+
|                                                       |
+                                                       +
|                                                       |
+               IP6 Reference DestAddress               +
|                                                       |
+                                                       +
|                                                       |
+-------------+-------------+-------------+-------------+
|      Session-Group ID     |    Count    |     Rank    |
+-------------+-------------+-------------+-------------+
```

The variables are defined in above.

D.3 DF Style (Experimental)

In addition to the WF and FF styles defined in this specification, a Dynamic Filter (DF) style has also been proposed. The following describes this style and gives examples of its usage. At this time, DF style is experimental.

D.3.1 Reservation Styles

A Dynamic-Filter (DF) style reservation makes *distinct* reservations with *wildcard* scope, but it decouples reservations from filters.

- Each DF reservation request specifies a number D of distinct reservations using the same specified flowspec. These reservations are distributed with wildcard scope, i.e., to all senders.

  The number of reservations that are actually made in a particular node is $D' = min(D, Ns)$, where Ns is the total number of senders upstream of the node.

- In addition to D and the flowspec, a DF style reservation may also specify a list of K filterspecs, for some K in the range: $0 <= K <= D'$. These filterspecs define particular senders to use the D' reservations, and this list establishes the scope for the filter specs.

  Once a DF reservation has been established, the receiver may change the set of filterspecs to specify a different selection of senders, without a new admission control check (assuming D' and the common flowspec remain unchanged). This is known as *channel switching*, in analogy with a television set.

In order to provide assured channel switching, each node along the path must reserve enough bandwidth for *all* D' channels, even though some of this bandwidth may be unused at any one time. If D' changes (because the receiver changed D or because the number Ns of upstream sources changed), or if the common flowspec changes, the refresh message is treated as a new reservation that is subject to admission control and may fail.

The DF style allows a receiver to switch channels without danger of an admission denial due to limited resources (unless a topology change reroutes traffic along a lower-capacity path or new senders appear), once the initial reservations have been made. This in turn implies that the DF style creates reservations that may not be in use at any given time.

The DF style is compatible with the FF style but not the WF or SE style.

D.3.2 Examples

To give an example of the DF style, we use the following notation:

- DF Style

  DF(n, {r} ;) or DF(n, {r} ; S1, S2, ...)

This message carries the count n of channels to be reserved, each using common flowspec r. It also carries a list, perhaps empty, of filterspecs defining senders.

Figure 15 shows an example of Dynamic-Filter reservations. The receivers downstream from interface (d) have requested two reserved channels, but selected only one sender, S1. The node reserves $min(2, 3) = 2$ channels of size B on interface (d), and it then applies any

```
                        |
         Send           |      Reserve             Receive
                        |
                        |       --------
 DF( 1,{B}; S1) <- (a)  |  (c) |  S1{B} |  (c) <- DF( 2,{B}; S1, S2)
                        |      |________|
                        |      |  S2{B} |
                        |      |________|
                        |
------------------------|-------------------------------------------
                        |       --------
 DF( 2,{B}; S2) <- (b)  |  (d) |  S1{B} |   (d) <- DF( 2,{B}; S1)
                        |      |________|
                        |      |   ?{B} |
                        |      |________|
```

Figure 15: Dynamic-Filter Reservation Example

specified filters to these channels. Since only one sender was specified, one channel has no corresponding filter, as shown by '?'.

Similarly, the receivers downstream of interface (c) have requested two channels and selected senders S1 and S2. The two channels might have been one channel each from R1 and R2, or two channels requested by one of them, for example.

A router should not reserve more Dynamic-Filter channels than the number of upstream sources (three, in the router of Figure 15). Since there is only one source upstream from previous hop (a), the first parameter of the DF message (the count of channels to be reserved) was decreased to 1 in the forwarded reservations. However, this is unnecessary, because the routers upstream will reserve only one channel, regardless.

When a DF reservation is received, it is labeled with the IP address of the next hop (RSVP-capable) router, downstream from the current node. Since the outgoing interface may be directly connected to a shared medium network or to a non-RSVP-capable router, there may be more than one next-hop node downstream; if so, each sends independent DF *Resv* messages for a given session. The number N' of DF channels reserved on an outgoing interface is given by the formula:

$N' = min(D1 + D2 + ...Dn, Ns)$,

where Di is the D value (channel reservation count) in a *Resv* from the ith next-hop node.

For a DF reservation request with a Dynamic Reservation Count = C, RSVP should call TC_AddFlowspec C times.

D.3.3 Resv Messages

Add the following sequence:

```
<flow descriptor list> ::=

            <FLOWSPEC> <filter spec list>
```

D.3.4 STYLE Class

- STYLE-DF object: Class = 8, C-Type = 2

```
+-------------+------------+------------+-------------+
| Style ID=4  |   Attribute Vector  0...0101001b      |
+-------------+------------+------------+-------------+
|    //////       ///////   |    Dynamic Resv Count   |
+-------------+------------+--------------------------+
```

Style ID

4 = Dynamic-Filter (DF)

Attribute Vector

18 bits: Reserved

1 bit: Decoupled if 1.

2 bits: Sharing control (as before)

3 bits: Scope control (as before)

Dynamic Resv Count

The number of channels to be reserved for a Dynamic Filter style reservation. This integer value must not less than the number of FILTER_SPEC objects in **filter spec list**.

D.4 Semantic Fragmentation

Long RSVP messages are fragmented into MTU-sized packets when they are sent and reassembled upon receipt. This is normally expected to be done at the RSVP layer, but may also occur at the IP layer (when fragmentation occurs within a non-RSVP cloud). It is well known that such *linear fragmentation* amplifies the effect of packet loss. There is some concern that this could result in lost RSVP state across congested paths through non-RSVP clouds.

One way to avoid this problem would be to use *semantic* fragmentation, exploiting the structure of an RSVP message. With semantic fragmentation, the state information that would have been packed into one large message is sent in multiple packets, each of which is constructed to be logically complete. Upon receipt, each packet can be processed independently of the other packets, with no explicit reassembly required.

Semantic fragmentation causes some redundancy of information; for example, each packet of a *Resv* message must include SESSION, NHOP/PHOP, TIME_VALUES, and STYLE objects. More importantly, the rules for semantic fragmentation are complex, since a single *Resv* message may contain two unbounded lists, and different styles require different rules. Finally, the largest atomic message must still fit into an MTU-sized packet, leading to a complex set of limits on the sizes of individual objects. At present, most objects are known to be small, but POLICY_DATA objects are variable and may perhaps grow large.

The text of this section describes (some of) the rules for semantic fragmentation. It has been removed from the main body of the document, but is kept here for futur consideration.

D.4.1 Semantic Fragmentation of *Resv* Messages

An outgoing *Resv* message that is too large for the MTU of the interface can be sent as multiple messages, as follows:

- For FF style, the **flow descriptor list** can be split as required to fit; the rest of the message should be replicated into each packet.
- For WF style, a SCOPE object containing an explicit list of sender IP addresses can be split as required to fit; the rest of the message should be replicated into each packet.
- For SE style, the **flow descriptor list** can be split as required to fit; the rest of the message should be replicated into each packet.

  If a single **SE descriptor** is too large to fit, its **filter spec list** can similarly be split as required. However, the subsets of a particular **filter spec list** must each be enclosed in TAG objects carrying the same tag value, so the receiver will be able to match each FILTER_SPEC object to the appropriate shared reservation.

D.4.2 TAG class

TAG class = 15.

- TAG object: Class = 15, C-Type = 1

```
+-------------+-------------+-------------+-------------+
|                        Tag Value                      |
+-------------+-------------+-------------+-------------+
|                                                       |
```

```
//                    Tagged Sublist                     //
|                                                         |
+-------------+-------------+-------------+-------------+
```

Tag Value

The value of the tag being attached to the objects in the Tagged Sublist. The tag value is unique for each session and next/previous hop.

Tagged Sublist

A list of objects with the same class-num (but not necessarily the same C-Type).

A TAG object encloses a list of one or more objects and attaches a logical name or *tag* value to them. The tag value is unique to the next/previous hop and the session (specified by HOP and SESSION objects, respectively). The enclosed object list is the *tagged sublist*, and the objects in it said to be *tagged* with the tag value. Objects in a particular tagged sublist must all have the same class-num.

Tagged objects with the same tag value are declared to be logically related, i.e., to be members of some larger logical set of objects. Note that the tagged sublist implies no ordering; it defines only a set of objects.

The meaning of the logical relationship depends upon the class-num of the tagged objects.

References

[CSZ92] Clark, D., Shenker, S., and L. Zhang, *Supporting Real-Time Applications in an Integrated Services Packet Network: Architecture and Mechanisms*, Proc. SIGCOMM '92, Baltimore, MD, August 1992.

[ISInt93] Braden, R., Clark, D., and S. Shenker, *Integrated Services in the Internet Architecture: an Overview*, RFC 1633, ISI, MIT, and PARC, June 1994.

[IServ93] Shenker, S., Clark, D., and L. Zhang, *A Service Model for an Integrated Services Internet*, Work in Progress, October 1993.

[Partridge92] Partridge, C., *A Proposed Flow Specification*, RFC 1363, BBN, September 1992.

[RSVP93] Zhang, L., Deering, S., Estrin, D., Shenker, S., and D. Zappala, *RSVP: A New Resource ReSerVation Protocol*, IEEE Network, September 1993.

[ServTempl95a] Shenker, S., *Network Element Service Specification Template*, Internet Draft draft-ietf-intserv-svc-template-00.txt, Integrated Services Working Group, March 1995.

[Shenker94] Shenker, S., "Two-Pass or Not Two-Pass", Current Meeting Report, RSVP Working Group, Proceedings of the Thirtieth Internet Engineering Task Force, Toronto, Canada, July 1994.

Security Considerations

See Section 2.5.

Authors' Addresses

Lixia Zhang
Xerox Palo Alto Research Center
3333 Coyote Hill Road
Palo Alto, CA 94304

Phone: (415) 812-4415
EMail: Lixia@PARC.XEROX.COM

Bob Braden
USC Information Sciences Institute
4676 Admiralty Way
Marina del Rey, CA 90292

Phone: (310) 822-1511
EMail: Braden@ISI.EDU

Deborah Estrin
Computer Science Department
University of Southern California
Los Angeles, CA 90089-0871

Phone: (213) 740-4524
EMail: estrin@USC.EDU

Shai Herzog
USC Information Sciences Institute
4676 Admiralty Way
Marina del Rey, CA 90292
Palo Alto, CA 94304

Phone: (310) 822 1511
EMail: Herzog@ISI.EDU

Sugih Jamin
Computer Science Department
University of Southern California
Los Angeles, CA 90089-0871

Phone: (213) 740-6578
EMail: jamin@catarina.usc.edu