Internet Draft R. Braden, Ed.
Expires May 22, 1996 ISI
File: draft-ietf-rsvp-spec-08.ps L. Zhang
PARC
S. Berson
ISI
S. Herzog
ISI
J. Wroclaswki
MIT
November 1995

Resource ReSerVation Protocol (RSVP) – Version 1 Functional Specification

November 22, 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.

Contents

What's Changed

The most important changes in this document from the rsvp-spec-07 draft are:

- The role and interpretation of the IP Protocol Id is changed. The Protocol Id is now a required part of the session definition, and filter specs and sender templates now assume the Protocol Id from the session rather than stating it explicitly.
- A "soft" reservation confirmation message is added.
- The text states explicitly that an erroneous reservation message is not forwarded. A mechanism to allow a receiver more flexible control over forwarding of its messages after an admission control failure has not been designed and is therefore not included in this version of the protocol.
- A terminology confusion is eliminated. The term "scope" was used both for a set of senders and for a set of sender hosts. A new term "sender selection" is introduced for the first, leaving "scope" for the second.
- The FILTER_SPEC object is dropped from a wildcard sender selection (WF) style reservation, which now selects *all senders* without qualification.
- The StyleID byte is dropped from a STYLE object, as redundant.
- An SE style flow descriptor is simplified to a single flowspec.
- The IP Router Alert option is now required in *Path* *PathTear* and *ResvConfirm* messages.
- The TIME_VALUES object is now required in *Resv* and *Path* messages; there is no default.
- Policing at branch points is now defined in a new section on policing (3.6).
- A 2-second delay is inserted into local repair.
- Merging of SE with WF objects is no longer allowed.
- The Rmax end-to-end bound on the refresh rate R is removed, since its utility was unclear.
- A rule for randomizing refresh timeouts is included.
- The suggestion that TCP could be used for carrying RSVP state through a congested non-RSVP cloud is removed.
- SENDER_TSPECS are now required in *Path*— messages.
- There are new sections on multihomed hosts (3.7) and future compatibility (3.8). The latter section makes clear that a message containing an object with unknown C-Type should be rejected. Any more forgiving treatment seems too complex.
- Appendix C on UDP encapsulation is completely changed.
- Some text was rearranged in Sections 1 and 2.

1 Introduction

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

On behalf of an application data stream, a host uses the RSVP protocol to request a specific quality of service (QoS) from the network. RSVP delivers QoS requests to routers along the path(s) of the data stream and maintains router and host state to provide the requested service. RSVP requests will generally, although not necessarily, result in resources being reserved along the data path.

RSVP requests resources for *simplex* data streams, i.e., it requests resources in only one direction. Therefore, a sender is logically distinct from a receiver, although the same application process may act as both a sender and a receiver at the same time. RSVP operates on top of IP (either IPv4 or IP6), 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. Like the implementations of routing and management protocols, an implementation of RSVP will typically execute in the background, not in the data forwarding path, as shown in Figure 1.

RSVP is not itself a routing protocol; RSVP is designed to operate with current and future unicast and multicast routing protocols. The RSVP daemon consults the local routing protocol(s) to obtain routes. In the multicast case, for example, a host sends IGMP messages to join a multicast group and then sends RSVP messages to reserve resources along the delivery path(s) of that group. Routing protocols determine where packets get forwarded; RSVP only concerns with the QoS of those packets that are forwarded by routing.

Each router that is capable of resource reservation passes incoming data packets through 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. There is a scheduler for each interface, to allocate resources for transmission on the particular link-layer medium used by that 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. This mapping to the link layer QoS may be accomplished in a number of possible ways; the details will be medium-dependent. On a QoS-passive medium such as a leased line, the scheduler itself allocates packet transmission capacity. 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, RSVP makes receivers responsible for requesting resource reservations [RSVP93]. A QoS request, which typically originates from a receiver host application, is passed to the local RSVP implementation, shown as a user daemon in Figure 1. The RSVP protocol then carries the request to all the nodes

```
            HOST                             ROUTER

 _________________________    RSVP  _____________________________
|                         |    .--------------.                  |
|  _______       ______   |   /    | ________  .   ______        |
| |       |     |      |  |  /     ||        |  . |      |       | RSVP
| |Applic-|     | RSVP <----/      ||Routing |   -> RSVP <---------->
| |  App  <----->daemon|  |        ||Protocol|    |daemon|       |
| |       |     |      |  |        || daemon <---->      |       |
| |_______|     |___.__|  |        ||_ ._____|    |__.__.|       |
|   |               |     |        |   |             |   .       |
|===|===============|=====|        |===|=============|====.======|
| data     .........|     |        |   |  ...........|     .____ |
|   |  ____V_   ____V____ |        |  _V__V_    _____V___ | Adm.||
|   | |Class-| |         ||  data  | |Class-|  |         ||Cntrl||
|   |=> ifier|=> Packet  ============> ifier|==> Packet  ||_____|| data
|     |______| |Scheduler||        | |______|  |Scheduler|===========>
|              |_________||        |           |_________|       |
|_________________________|        |_____________________________|
```

Figure 1: RSVP in Hosts and Routers

(routers and hosts) along the reverse data path(s) to the data source(s).

At each node, the RSVP daemon communicates with a local decision module, called *admission control*, to determine if the router can supply the requested QoS. If the admission control check succeeds, the RSVP daemon sets parameters in the packet classifier and scheduler to obtain the desired QoS. If the admission control check fails, the RSVP program immediately returns an error notification to the application process 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 both the membership of a large group and the topology of large multicast trees are likely to change with time, 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. That is, RSVP sends periodic refresh messages to maintain the state along the reserved path(s); in absence of refreshes, the state will automatically time out and be deleted.

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 for more details.

In summary, RSVP has the following attributes:

- RSVP makes resource reservations for both unicast and many-to-many multicast applications, adapting dynamically to changing group membership as well as changing routes.
- RSVP is simplex, i.e., it reserves for data flow in one direction only.
- RSVP is receiver-oriented, i.e., the receiver of a data flow initiates and maintains the resource reservation used for that flow.
- RSVP maintains "soft state" in the routers, providing graceful support for dynamic membership changes and automatic adaptation 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. Section 3 contains the functional specification of RSVP, while Section 4 presents explicit message processing rules. Appendix A defines the variable-length typed data objects used in the RSVP protocol. Appendix B defines error codes and values. Appendix C defines an extension for UDP encapsulation of RSVP messages. Finally, some experimental RSVP features are documented in Appendix D for future reference.

1.1 Data Flows

RSVP defines a *session* as a data flow with a particular destination and transport-layer protocol. The destination for a particular session is generally defined by DestAddress, the IP destination address of the data packets, and perhaps by DstPort, a *generalized destination port*, i.e., some further demultiplexing point in the transport or application protocol layer. RSVP treats each session independently, and this document often assumes the qualification "for the same session".

DestAddress is a group address for multicast delivery or the unicast address of a single receiver. DstPort 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 supports 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 multicast routing. 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 be running in a unique Internet host, or a single host may contain multiple senders and/or receivers, distinguished by generalized ports.

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

Figure 2: Multicast Distribution Session

For unicast transmission, there will be a single destination host but there may be multiple senders; RSVP can set up reservations for multipoint-to-single-point transmission.

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 session definition, specifies 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. Data packets that are addressed to a particular session but do not match any of the filter specs for that session are handled as best-effort traffic.

Note that the action to control QoS occurs at the place where the data enters the medium, i.e., at the upstream end of the link, although an RSVP reservation request originates from receiver(s) downstream. In this document, we define the directional terms "upstream" vs. "downstream", "previous hop" vs. "next hop", and "incoming interface" vs "outgoing interface" with respect to the direction of data flows.

The flowspec in a reservation request will generally include a service class and two sets of numeric parameters: (1) an *Rspec* (R for 'reserve') that defines the desired QoS, and (2) a *Tspec* (T for 'traffic') that describes the data flow. The formats and contents of Tspecs and Rspecs are determined by the integrated service model [ServTempl95a], and are generally opaque to RSVP.

In the most general approach [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, consisting of sender IP address and optionally the UDP/TCP port number SrcPort.

RSVP reservation request messages originate at receivers and are passed upstream towards the sender(s). When a reservation request is received at a node, two general actions are taken.

1. Make 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. Forward the request 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 from downstream, for two reasons. First, it is possible in theory for the traffic control mechanism to modify the flowspec hop-by-hop, although none of the currently defined services does so. Second, reservations for the same sender, or the same set of senders, from different downstream branches of the multicast tree(s) are *merged* as reservations travel upstream; that is, a node forwards upstream only the reservation request with the "maximum" flowspec.

When a receiver originates a reservation request, it can also request a confirmation message to indicate that its request was (probably) installed in the network. A successful reservation request propagates as far as the closest point(s) along the sink tree to the sender(s) where there is an existing reservation level equal or greater than that being requested. At that point, the arriving request will be dropped in favor of the equal or larger reservation in place; the node may then send a reservation confirmation message back to the receiver. Note that the receipt of a confirmation is only a high-probability indication, not a guarantee that the requested service is in place all the way to the sender(s), as explained in Section 2.6.

The basic RSVP reservation model is *one pass*: a receiver sends a reservation request upstream, and each node in the path either accepts or rejects the request. This scheme provides no easy way for a receiver to find out the resulting end-to-end service. Therefore, RSVP supports an enhancement to one-pass service known as *One Pass With Advertising* (OPWA) [Shenker94]. With OPWA, RSVP control packets are sent downstream, following the data paths, to gather information that may be used to predict the end-to-end QoS. The results ("advertisements") are delivered by RSVP to the receiver hosts and perhaps to the receiver applications. The advertisements may then be used by the receiver to construct, or to dynamically adjust, 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 selection of senders: an *explicit* list of all selected senders, or a *wildcard* that implicitly selects all the senders to the session. In an explicit-selection reservation, each filter spec must match exactly one sender, while in a wildcard-selection no filter spec is needed.

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

- Wildcard-Filter (WF) Style

  The WF style implies the options: *shared* reservation and *wildcard* sender selection. 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 from all receivers, independent of the number of senders using it. A WF-style reservation is propagated upstream towards all sender hosts, and automatically extends to new senders as they appear.

  Symbolically, we can represent a WF-style reservation request by:

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

Figure 3: Reservation Attributes and Styles

```
WF( * {Q})
```

where the asterisk represents wildcard sender selection and Q represents the flowspec.

- Fixed-Filter (FF) Style

  The FF style implies the options: *distinct* reservations and *explicit* sender selection. 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.

  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 be merged to share a single reservation.

  Symbolically, we can represent an elementary FF reservation request by:

  ```
  FF( S{Q})
  ```

  where S is the selected sender and Q is the corresponding flowspec; the pair forms a flow descriptor. RSVP allows multiple elementary FF-style reservations to be requested at the same time, using a list of flow descriptors:

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

- Shared Explicit (SE) Style

  The SE style implies the options: *shared* reservation and *explicit* sender selection. Thus, an SE-style reservation creates a single reservation into which flows from all upstream senders are mixed. However, like the FF style, the SE style allows a receiver to explicitly specify the set of senders.

Symbolically, we can represent an SE reservation request by:

```
SE( (S1,S2,...){Q} ),
```

i.e., a flow descriptor composed of a flowspec Q and a list of senders S1, S2, etc.

Both WF and SE are shared reservations, appropriate for those multicast applications whose application-specific constraints make it unlikely that multiple data sources will transmit simultaneously. Packetized audio is an example of an application suitable for shared reservations; since a limited number of people talk at once, each receiver might issue a WF or SE reservation request for twice the bandwidth required for one sender (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.

The RSVP rules disallow merging of shared reservations with distinct reservations, since these modes are fundamentally incompatible. They also disallow merging explict sender selection with wildcard sender selection, since this might produce an unexpected service for a receiver that specified explicit selection. As a result of these prohibitions, WF, SE, and FF styles are all mutually incompatible.

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

1.4 Examples of Styles

This section presents examples of each of the reservation styles and show the effects of merging.

Figure 4 shows schematically a router with two incoming interfaces through which data streams will arrive, labeled (a) and (b), and two outgoing interfaces through which data will be forwarded, 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 (R2 and R3) are routed via outgoing interface (c) ((d), respectively). We furthermore assume that R2 and R3 arrive via different next hops, e.g., via the two routers D and D' in Figure 9. This illustrates the effect of a non-RSVP cloud or a broadcast LAN on interface (d).

In addition to the connectivity shown in 4, we must also specify the multicast routes within this node. Assume first that data packets from each Si shown in Figure 4 is routed to both outgoing interfaces. Under this assumption, Figures 5, 6, and 7 illustrate Wildcard-Filter, Fixed-Filter, and Shared-Explicit reservations, respectively.

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

Figure 4: Router Configuration

For simplicity, these examples show flowspecs as one-dimensional multiples of some base resource quantity B. The “Receive” column shows the RSVP reservation requests received over outgoing interfaces (c) and (d), and the “Reserve” column shows the resulting reservation state for each interface. The “Send” column shows the reservation requests that are sent upstream to previous hops (a) and (b). In the “Reserve” column, each box represents one reserved “pipe” on the outgoing link, with the corresponding flow descriptor.

Figure 5, showing the WF style, illustrates the two possible merging situations. Each of the two next hops on interface (d) results in a separate RSVP reservation request, as shown. These two requests are merged into the effective flowspec 3B, which is used to make the reservation on interface (d). To forward the reservation requests upstream, the reservations on the interfaces (c) and (d) are merged; as a result, the larger flowspec 4B is forwarded upstream to each previous hop.

Figure 6 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 request forwarded to previous hop (b). On the other hand, the three different flow descriptors for sender S1 are merged into the single request FF(S1{4B}), which is sent to previous hop (a). For each outgoing interface, there is a separate reservation for each source that has been requested, but this reservation is shared among all the receivers that made the request.

Figure 7 shows an example of Shared-Explicit (SE) style reservations. When SE-style reservations are merged, the resulting filter spec is the union of the original filter specs.

```
                       |
         Send          |       Reserve              Receive
                       |
                       |       -------
   WF( *{4B} ) <- (a)  |  (c) | * {4B}|    (c) <- WF( *{4B} )
                       |      |_______|
                       |
-----------------------|----------------------------------------
                       |       -------
   WF( *{4B} ) <- (b)  |  (d) | * {3B}|    (d) <- WF( *{3B} )
                       |      |_______|        <- WF( *{2B} )
```

Figure 5: Wildcard-Filter (WF) Reservation Example

```
                       |
         Send          |       Reserve              Receive
                       |
                       |       --------
  FF( S1{4B} ) <- (a)  |  (c) | S1{4B} |   (c) <- FF( S1{4B}, S2{5B} )
                       |      |________|
                       |      | S2{5B} |
                       |      |________|
  ---------------------|---------------------------------------------
                       |       --------
               <- (b)  |  (d) | S1{3B} |   (d) <- FF( S1{3B}, S3{B} )
  FF( S2{5B}, S3{B} )  |      |________|       <- FF( S1{B} )
                       |      | S3{B}  |
                       |      |________|
```

Figure 6: Fixed-Filter (FF) Reservation Example

```
                     |
       Send          |       Reserve              Receive
                     |
                     |       ________
SE( S1{3B} ) <- (a)  |  (c) |(S1,S2) |   (c) <- SE( (S1,S2){B} )
                     |      |   {B}  |
                     |      |________|
---------------------|-------------------------------------------
                     |       __________
             <- (b)  |  (d) |(S1,S2,S3)|  (d) <- SE( (S1,S3){3B} )
SE( (S2,S3){3B} )    |      |   {3B}   |      <- SE( S2{2B} )
                     |      |__________|
```

Figure 7: Shared-Explicit (SE) Reservation Example

The three examples just shown assume that data packets from S1, S2, and S3 are routed to both outgoing interfaces. The top part of Figure 8 shows another routing assumption: data packets from S2 and S3 are not forwarded to interface (c), e.g., because the network topology provides a shorter path for these senders towards R1, not traversing this node. The bottom part of Figure 8 shows WF style reservations under this assumption. Since there is no route from (b) to (c), the reservation forwarded out interface (b) considers only the reservation on interface (d).

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

                    Router Configuration


                               |
         Send                  |       Reserve              Receive
                               |
                               |       -------
   WF( *{rB} ) <- (a)          |  (c) | * {B} |    (c) <- WF( *{4B} )
                               |      |_______|
                               |
-------------------------------|----------------------------------------
                               |       -------
   WF( *{3B} ) <- (b)          |  (d) | * {3B}|    (d) <- WF( * {3B} )
                               |      |_______|        <- WF( * {2B}
```

Figure 8: WF Reservation Example – Partial Routing

2 RSVP Protocol Mechanisms

2.1 RSVP Messages

```
 Previous        Incoming            Outgoing             Next
 Hops            Interfaces          Interfaces           Hops

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

Figure 9: Router Using RSVP

Figure 9 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 in the same session. Multiple previous hops and/or next hops may be reached through a given physical interface, as a result of the connected network being a shared medium, or the existence of non-RSVP routers in the path to the next RSVP hop (see Section 2.8). An RSVP daemon preserves the next and previous hop addresses in its reservation and path state, respectively.

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

Each receiver host sends RSVP reservation request (*Resv*) messages upstream towards the senders. These reservation messages must follow exactly the reverse of the routes the data packets will use, upstream to all the sender hosts included in the sender selection. *Resv* messages must be delivered to the sender hosts themselves so that the hosts can set up appropriate traffic control parameters for the first hop.

Each RSVP sender host transmits RSVP *Path* messages downstream along the uni-/multicast routes provided by the routing protocol(s), following the paths of the data. These *Path* messages store *path state* in each node along the way. This path state includes at least the unicast IP address of the previous hop node, which is used 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).

A *Path* message may carry the following information in addition to the previous hop address:

- Sender Template

  A *Path* message is required to carry a Sender Template, which 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 the sender IP address and optionally the UDP/TCP sender port. It assumes the protocol Id for the session.

- Sender Tspec

  A *Path* message is required to carry a Sender Tspec, which defines the traffic characteristics of the data stream that the sender will generate. This Tspec is used by traffic control to prevent over-reservation (and perhaps unnecessary Admission Control failure) on all links on which the named sender is the only source sending to the session.

- Adspec

  A *Path* message may optionally 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, which returns an updated Adspec; the updated version is then forwarded downstream.

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

Path messages are sent with the same source and destination addresses as the data, so that they will be routed correctly through non-RSVP clouds (see Section 2.8). On the other hand, *Resv* messages are sent hop-by-hop; each RSVP-speaking node forwards a *Resv* message to the unicast address of a previous RSVP hop.

2.2 Port Usage

At present an RSVP session is defined by the triple: (DestAddress, ProtocolId, DstPort). Here DstPort is a UDP/TCP destination port field (i.e., a 16-bit quantity carried at octet offset +2 in the transport header). DstPort may be omitted (set to zero) if the ProtocolId specifies a protocol that does not have a destination port field in the format used by UDP and TCP.

RSVP allows any value for ProtocolId. However, end-system implementations of RSVP may know about certain values for this field, and in particular must know about the values for UDP and TCP (17 and 6, respectively). An end system should give an error to an application that either:

- specifies a non-zero DstPort for a protocol that does not have UDP/TCP-like ports, or
- specifies a zero DstPort for a protocol that does have UDP/TCP-like ports.

Filter specs and sender templates are defined by the pair: (SrcAddress, SrcPort), where SrcPort is a UDP/TCP source port field (i.e., a 16-bit quantity carried at octet offset +0 in the transport header). SrcPort may be omitted (set to zero) in certain cases. The following rules hold for the use of zero DstPort and/or SrcPort fields in RSVP.

1. Destination ports must be consistent.

   Path state and/or reservation state for the same DestAddress and ProtocolId must have DstPort values that are all zero or all non-zero. Violation of this condition in a node is a "Conflicting Dest Port" error.

2. Destination ports rule.

   If DstPort in a session definition is zero, all SrcPort fields used for that session must also be zero. The assumption here is that the protocol does not have TCP/UDP-like ports. Violation of this condition in a node is a "Conflicting Src Port" error.

3. Source Ports must be consistent.

   A sender host must not send path state both with and without a zero SrcPort. Violation of this condition is an "Ambiguous Path" error.

2.3 Merging Flowspecs

As noted earlier, a single physical interface may receive multiple reservation request from different next hops for the same session and with the same filter spec, but RSVP should install only one reservation on that interface. This reservation should an effective flowspec that is the "maximum" of the flowspecs requested by the different next hops. Similarly, a *Resv* message forwarded to a previous hop should carry a flowspec that is the "maximum" of the flowspecs requested by the different next hops. Both cases represent flowspec merging.

Merging flowspecs requires calculating the "largest" of a set of flowspecs, which are otherwise opaque to RSVP. Since flowspecs are multi-dimensional vectors (they contain both Tspec and Rspec components, each of which may itself be multi-dimensional), generally speaking they cannot be strictly ordered. However, in many cases one can easily determine the "larger" of two flowspecs, such as when both request the same bandwidth but one requests a tighter delay, or when one of the two requests both a higher bandwidth and a tighter delay bound. When the "larger" of the two cannot be determined, RSVP must compute and use a third flowspec that is at least as large as each, i.e., a *least upper bound* (LUB). If the two flowspecs are incomparable, their comparison will treated as an error.

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 9.

1. 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).
2. All Tspecs that were supplied in *Path* messages from different previous hops (e.g., some or all of A, B, and B' in Figure 9) are summed; call this sum Path_Te.
3. The maximum Tspec supplied in *Resv* messages from different next hops (e.g., D and D') is calculated; call this Resv_Te.
4. 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.

Flowspecs, Tspecs, and Adspecs are opaque to RSVP. Therefore, the last of these steps is actually performed by traffic control. The definition and implementation of the rules for comparing flowspecs, calculating LUB's, and summing Tspecs are outside the definition of RSVP [ServTempl95a]. Section 3.9.4 shows generic calls that an RSVP daemon could use for these functions.

2.4 Soft State

RSVP takes a *soft state* approach to managing the reservation state in routers and hosts. 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 by an explicit *teardown* message, described in the next section. At the expiration of each *refresh timeout* period and after a state change, RSVP scans its state to build and forward *Path* and *Resv* refresh messages to succeeding hops.

Path and *Resv* messages are idempotent. 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 existing state at that node.

RSVP sends its messages as IP datagrams with no reliability enhancement. Periodic transmission of refresh messages by hosts and routers is expected to handle the occasional loss of RSVP messages. If the effective cleanup timeout is set to K times the refresh timeout period, then RSVP can tolerate K-1 successive RSVP packet losses without falsely erasing a reservation. We recommend that the network traffic control mechanism be statically configured to grant some minimal bandwidth for RSVP messages to protect them from congestion losses.

The state maintained by RSVP is dynamic; to change the set of senders Si 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 in all nodes along the path.

In steady state, refreshing is performed hop-by-hop to allow merging. If the received state differs from the stored state, the stored state is updated. If this update results in modification of state to be forwarded in refresh messages, these refresh messages must be generated and forwarded immediately, so that state changes can be propagated 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.

State that is received through a particular interface I* should never be 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 10, 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) serve as 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. The result is independent reservations in the two directions.

There is an additional rule governing the forwarding of *Resv* messages: state from *Resv* messages received from outgoing interface Io should be forwarded to incoming interface Ii only if *Path* messages from Ii are forwarded to Io.

```
                   ----------------
                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 10: Independent Reservations

2.5 Teardown

Upon arrival, RSVP *teardown* messages remove path and reservation state immediately. Although it is not necessary to explicitly tear down an old reservation, we recommend that all end hosts send a teardown request as soon as an application finishes.

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. An *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 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 must be forwarded hop-by-hop without delay. A teardown message deletes the specified state in the node where it is received. As always, this state change will be propagated immediately to the next node, but only if there will be a net change after merging. As a result, an *ResvTear* message will prune the reservation state back (only) as far as possible.

Like all other RSVP messages, teardown requests are not delivered reliably. The loss of a teardown request message will not cause a protocol failure because the unused state will eventually time out even though it is not explicitly deleted. If a teardown message is lost, the router that failed to receive that 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.

2.6 Errors and Acknowledgments

There are two RSVP error messages, *ResvErr* and *PathErr* and a reservation confirmation message *ResvConfirm*.

There are a number of ways for a syntactically valid reservation request to fail at some node along the path, triggering a *ResvErr* message:

1. The effective flowspec that is computed using the new request may fail admission control.
2. Administrative policy may prevent the requested reservation.
3. There may be no matching path state, so that the request cannot be forwarded towards the sender(s).
4. A reservation style that requires the explicit selection of a unique sender may have a filter spec that is ambiguous, i.e., that matches more than one sender in the path state, due to the use of wildcard fields in the filter spec.
5. The requested style may be incompatible with the style(s) of existing reservations. The incompatibility may occur among reservations for the same session on the same outgoing interface, or among effective reservations on different outgoing interfaces.

In any of these cases, a *ResvErr* message is returned to the receiver(s) responsible for the erroneous request. A node may also decide to preempt an established reservation. A preemption will trigger a *ResvErr* message to all affected receivers. 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 occurred will persist until the responsible receiver(s) explicitly tear down the state or allow it to time out.

In this version of RSVP, detection of an error in a reservation request not only generates a *ResvErr* message, it also prevents the request from being forwarded further. This may not always be the desirable behavior; for example, a receiver may want a reservation request to propagate all the way to the sender despite an admission control failure at a particular link along the path. However, design of the appropriate mechanism has proved difficult, and therefore this version take the simplest approach.

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).

To request a confirmation for its reservation request, a receiver Rj includes in the *Resv* message a confirmation-request object containing its IP address. At each merge point, only the largest flowspec and any accompanying confirmation-request object is forwarded upstream. If the reservation request from Rj is equal to or smaller than the reservation in place on a node, its *Resv* are not forwarded further, and if the *Resv* included an confirmation-request object, a *ResvConfirm* message is sent back to Rj. This mechanism has the following consequences:

- A new reservation request with a flowspec larger than any in place for a session will normally result in either a *ResvErr* or a *ResvConfirm* message back to the receiver from each sender. In this case, the *ResvConfirm* message will be an end-to-end confirmation.
- The receipt of a *ResvConfirm* gives no guarantees. Assume the first two reservation requests from receivers R1 and R2 arrive at the node where they are merged. R2, whose reservation was the second to arrive at that node, may receive a *ResvConfirm* from that node while R1's request has not yet propagated all the way to a matching sender and may still fail. In this case, R2 will receive a *ResvConfirm* although there is no end-to-end reservation in place. Furthermore, if the two flowspecs are equal, R2 may receive a *ResvConfirm* followed by a *ResvErr*. However, if its flowspec is smaller, R2 will receive only the *ResvConfirm*.
- Despite these uncertainties, receipt of a *ResvConfirm* indicates a high probability that the reservation is in place.
- Finally, note that RERR and/or *ResvConfirm* messages may be lost.

2.7 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 is likely to 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 in addition to 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 neighbor 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 be impossible or undesirable to carry all receivers' policy data upstream to the sender(s). The policy data will have to be administratively merged at places near 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.

In summary, different administrative domain in the Internet may have different policies regarding their resource usage and reservation. The role of RSVP is to carry policy data associated with each reservation to the network as needed. 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.

2.8 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. However, if such a cloud has sufficient capacity, it may still provide acceptable realtime service.

RSVP automatically tunnels 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, it carries to the next RSVP-capable node the IP address of the last RSVP-capable router before entering the cloud. This effectively constructs a tunnel through the cloud for *Resv* messages, which can then be forwarded directly to the next RSVP-capable router on the path(s) back towards the source.

Some interconnection topologies of RSVP and non-RSVP routers can cause *Resv* messages to arrive at the wrong RSVP-capable node, or to arrive at the wrong interface at the correct node. An RSVP daemon must be prepared to handle either situation. When a *Resv* message arrives, its IP destination address should normally be the address of one of the local interfaces. If so, the reservation should be made on the addressed interface, even if it is not the one on which the message arrived. If the destination address does not match any local interface and the message is not a *Path* or *PathTear* it should be forwarded without further processing by this node.

2.9 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.

H3 A receiver application receives a *Path* message.

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

H5 A sender application receives a *Resv* message.

H6 A sender starts sending data packets.

There are several synchronization considerations.

- H1 and H2 may happen in either order.
- Suppose that a new sender starts sending data (H6) but there are no multicast routes because no receivers have joined the group (H1). Then the data will be dropped at some router node (which node depends upon the routing protocol) until receivers(s) appear.
- Suppose that a new sender starts sending *Path* messages (H2) and data (H6) simultaneously, 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 receiving any *Path* messages (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. [LZ: should recommend that a receiver wait for at least *Path* messages to arrive 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 3.9.1 discusses the general requirements and presen

3 RSVP Functional Specification

3.1 RSVP Message Formats

An RSVP message consists 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 choice and ordering of object types. These rules are specified using Backus-Naur Form (BNF) augmented with square brackets surrounding optional sub-sequences.

3.1.1 Common Header

```
      0             1              2             3
+-------------+-------------+-------------+-------------+
| Vers | Flags|    Type     |       RSVP Checksum       |
+-------------+-------------+-------------+-------------+
|        RSVP Length        | (Reserved)  |  Send_TTL   |
+-------------+-------------+-------------+-------------+
|                       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*
7 = *ResvConfirm*

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.

Send_TTL: 8 bits

The IP TTL value with which the message was sent.

Message ID: 32 bits

A label shared by all fragments of one message from a given next/previous RSVP hop. An RSVP implementation assigns 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.

3.1.2 Object Formats

Every 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 is always 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), the IP protocol id, and 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

Contains the value for the refresh period R used by the creator of the message; see 3.5. Required in every *Path* and *Resv* message.

STYLE

Defines the reservation style plus style-specific information that is not in FLOWSPEC or FILTER_SPEC objects. Required in every *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 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 list of sender hosts towards which to forward a message. May appear in a *Resv* *ResvErr* or *ResvTear* message.

RESV_CONFIRM

Carries the IP address of a receiver that requested a confirmation. May appear in a *Resv* or *ResvConfirm* 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 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; see Section 3.8.

3.1.3 Path Message

Each sender host periodically sends a *Path* message containing a description of each data stream it originates. The *Path* message travels from a sender to receiver(s) along the same path(s) used by the data packets. The IP source address of a *Path* message is an address of the sender it describes, while the destination address is the DestAddress for the session. These addresses assure that the message will be correctly routed through a non-RSVP cloud.

Each RSVP-capable node along the path(s) captures *Path* messages and processes them to build local path state. The node then forwards the *Path* messages towards the receiver(s), replicating it as dictated by multicast routing, while preserving the original IP source address. *Path* messages eventually reach the applications on all receivers; however, they are not looped back to a receiver running in the same application process as the sender.

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

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

                            [ <INTEGRITY> ]  <TIME_VALUES>

                            <sender descriptor>

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

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

The PHOP (i.e., the RSVP_HOP) object of each *Path* message contains the address of the interface through which the *Path* message was most recently sent. The SENDER_TEMPLATE object defines the format of data packets from this sender, while the SENDER_TSPEC object specifies the traffic characteristics of the flow. Optionally, there may be a POLICY_DATA object specifying user credential and accounting information and/or an ADSPEC object carrying advertising (OPWA) data.

A *Path* message received at a node is processed to create path state for the sender defined by the SENDER_TEMPLATE and SESSION objects. 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 the originating sender of the *Path* message. *Path* messages must satisfy the rules on SrcPort and DstPort in Section 2.2.

Periodically, the RSVP daemon at a node scans the path state to create new *Path* messages to forward downstream. Each message contains a **sender descriptor** defining one sender. The RSVP daemon forwards these messages using routing information it obtains from the appropriate uni-/multicast routing daemon. The route depends upon the session DestAddress, and for some routing protocols also upon the source (sender's IP) address. The routing information generally includes the list of none or more outgoing interfaces to which the *Path* message to be forwarded. Because each outgoing interface has a different IP address, the *Path* messages sent out different interfaces contain different PHOP addresses. In addition, any ADSPEC or POLICY_DATA objects carried in *Path* messages will also generally differ for different outgoing interfaces.

Some IP multicast routing protocols (e.g., DVMRP, PIM, and MOSPF) also keep track of the expected incoming interface for each source host to a multicast group. Whenever this information is available, RSVP should check the incoming interface of each *Path* message and immediately discard those messages that have arrived on the wrong interface.

3.1.4 Resv Messages

Resv messages carry reservation requests hop-by-hop from receivers to senders, along the reverse paths of data flows 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 *Resv* message format is as follows:

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

                        [ <INTEGRITY> ] <TIME_VALUES>

                        [ <S_POLICY_DATA> ]

                        [ <RESV_CONFIRM> ]  [ <SCOPE> ]

                        <STYLE> <flow descriptor list>

<S_POLICY_DATA> ::=  <POLICY_DATA>

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

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

The NHOP (i.e., the RSVP_HOP) object contains the IP address of the (incoming) interface through which the *Resv* message is sent. The appearance of a RESV_CONFIRM object signals a request for a reservation confirmation and carries the IP address of the receiver to which the *ResvConfirm* should be sent. The S_POLICY_DATA object is a POLICY_DATA object that is associated with the entire session. 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 in more detail the composition of a valid **flow descriptor list** for each of the reservation styles.

- WF Style:

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

```
<WF flow descriptor> ::= <FLOWSPEC> [ <F_POLICY_DATA> ]

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

- FF style:

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

              <flow descriptor list> <FF flow descriptor>

<First FF flow descriptor> ::=

           <FLOWSPEC>  [ <F_POLICY_DATA> ] <FILTER_SPEC>

<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 object can be omitted if it is identical to the most recent such object that appeared in the list; the first **FF flow descriptor** must contain a FLOWSPEC.

- SE style:

```
<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, optionally a POLICY_DATA object, and a list of FILTER_SPEC objects.

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

- Explicit sender selection

  Match each FILTER_SPEC object against the path state created from SENDER_TEMPLATE objects to select a particular sender. An ambiguous match, i.e., a FILTER_SPEC matching more than one SENDER_TEMPLATE (e.g. through use of a wildcard port), is an error. Any SCOPE object associated with the reservation should be ignored in this case.

- Wildcard sender selection

  All senders that route to the given outgoing interface match this request. A SCOPE object, if present, contains an explicit list of sender IP addresses. If there is no SCOPE object, the scope is determined by the relevant set of senders in the path state. Whenever a *Resv* message with wildcard sender selection is forwarded to more than one previous hop, a SCOPE object must be included in the message. See Section 3.3 below.

3.1.5 Error and Confirmation Messages

There are three types of RSVP error/confirmation 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.
- *ResvConfirm* messages are sent to (probabilistically) acknowledge reservation requests. A *ResvConfirm* message is sent as the result of the appearance of a RESV_CONFIRM object in a *Resv* message, and contains a copy of that RESV_CONFIRM. The *ResvConfirm* message is sent to the unicast address of a receiver host; the address is obtained from the RESV_CONFIRM object. A *ResvConfirm* message is forwarded to the receiver hop-by-hop by (to accommodate the hop-by-hop integrity check mechanism).

Errors encountered while processing error messages must cause the error message to be discarded without creating further error messages; however, logging of such events may be useful.

None of these messages modify the state of any node through which they pass; instead, they are only reported to the end application.

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

```
                              [ <INTEGRITY> ]  <ERROR_SPEC>

                              <sender descriptor>

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


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

                              [ <INTEGRITY> ]  <ERROR_SPEC>

                              [S_POLICY_DATA]  [ <SCOPE> ]

                             <STYLE> <error flow descriptor>


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

                              [ <INTEGRITY> ]  <ERROR_SPEC>

                              <RESV_CONFIRM>

                              <STYLE> <flow descriptor list>

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

The RESV_CONFIRM object in a *ResvConfirm* message is a copy of the object from the *Resv* message that triggered the confirmation.

The following style-dependent rules define the composition of a valid **error flow descriptor**:

- WF Style:

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

- FF style:

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

- SE style:

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

The ERROR_SPEC object specifies the error and includes the IP address of the node that detected the error (Error Node Address). POLICY_DATA objects are included in error messages in cases where they may provide relevant information (i.e., when an administrative failure is being reported). In a *ResvConfirm* message, the ERROR_SPEC is used only to carry the IP address of the originating node, in the Error Node Address; the error specification is a special value that indicates a confirmation.

When a *Resv* message contains a list of flow descriptors (e.g., FF style), the RSVP implementation should process each flow descritor independently and return a separate *ResvErr* message for each that is in error.

Generally speaking, a *ResvErr* message should be forwarded towards all receivers that may have caused the error being reported. More specifically:

- The node that detects an error in a reservation request sends a *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 3.3.

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 'de-merging' the filter spec).

When a *ResvErr* or *ResvConfirm* 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 semantically equivalent (but not necessarily identical) to the FLOWSPEC originated by this application; otherwise, they may differ.

3.1.6 Teardown Messages

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

- A *PathTear* message deletes path state (which in turn deletes the reservation state for that sender, if there is any) 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 in the same way as 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>

<sender descriptor> ::= (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, unless it is accidentally dropped along the way, a *PathTear* message will reach all the receivers down stream from its origination. On the other hand, a *ResvTear* message will cease to be forwarded at the same node where merging suppresses forwarding of the corresponding *Resv* messages. In each node N along the way, if the *ResvTear* message causes the removal of all state for this session, N will create a new teardown message to be propagated further upstream; otherwise, the *ResvTear* message may result in the immediate forwarding of a modified *Resv* refresh message.

Deletion of path state as the result of a *PathTear* message or a timeout may force adjustments in related reservation state, to maintain state 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 explicit sender selection (FF or SE), delete any reservation with a filter spec matching S; otherwise, the style is wildcard sender selection (WF) and the reservation should be deleted if S is the last sender to the session. These reservation changes should not trigger an immediate *Resv* refresh message, since the *PathTear* message 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.

3.2 Sending RSVP Messages

RSVP messages are sent hop-by-hop between RSVP-capable routers as "raw" IP packets with protocol number 46. Raw IP packets are intended to be used between an end system and the first/last hop router, although it is also possible to encapsulate RSVP messages as UDP datagrams for end-system communication, as described in Appendix C. UDP encapsulation is needed for systems that cannot do raw network I/O.

Path PathTear and *ResvConfirm* messages must be sent with the Router Alert IP option [Katz95] in their IP headers. This option may be used by in the fast forwarding path of a high-speed router to detect datagrams that require special processing.

Upon the arrival of an RSVP message M that changes the state, a node must forward the modified state immediately. However, this must not trigger sending an message out the interface through which M arrived (as could happen if the implementation simply triggered an immediate refresh of all state for the session). 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 can be used as an appropriate reassembly timeout time.

Since RSVP messages are normally generated and sent hop-by-hop, using the RSVP-level fragmentation mechanism should avoid further fragmentation at the IP level. However, IP fragmentation may still occur when RSVP messages travel through a non-RSVP cloud. In case of IP6, which does not support IP fragmentation at routers, an RSVP implementation must use Path MTU Discovery or hand configuration to obtain an appropriate MTU between adjacent RSVP neighbors.

RSVP recovers from occasional packet losses by its periodic refresh mechanism. Under network overload, however, substantial losses of RSVP messages could cause a failure of resource reservations. To control the queueing delay and dropping of RSVP packets, routers should be configured to offer them a preferred class of service. If RSVP packets experience noticeable losses when crossing a congested non-RSVP cloud, a larger value can be used for the timeout factor K (see section 3.5 below).

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 run through 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, called the *logical interface handle* or LIH. The LIH value is carried in the RSVP_HOP object.
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 RSVP_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'.

3.3 Avoiding RSVP Message Loops

Forwarding of RSVP messages must avoid looping. In steady state, *Path* and *Resv* messages are forwarded only once per refresh period on each hop. This avoids looping packets, but there is still the possibility of an *auto-refresh* loop, clocked by the refresh period. Such auto-refresh loops keep state active “forever”, even if the end nodes have ceased refreshing it, until either 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 looping.

Consider each message type.

- *Path* Messages

  Path messages are forwarded in exactly the same way as IP data packets. Therefore there should be no loops of *Path* messages, even in a topology with cycles.

- *PathTear* Messages

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

- *PathErr* Messages

  Since *Path* messages do not 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

  Resv messages directed to particular senders (i.e., with explicit sender selection) cannot loop. However, *Resv* messages with wildcard sender selection (WF style) have a potential for auto-refresh looping.

- *ResvTear* Messages

  Although *ResvTear* messages are routed the same as *Resv* messages, during the second pass around a loop there will be no state so any *ResvTear* message will be dropped. Hence there is no looping problem here.

- *ResvErr* Messages

  ResvErr messages for WF style reservations may loop for essentially the same reasons that *Resv* messages loop.

- *ResvConfirm* Messages

  ResvConfirm messages are forwarded towards a fixed unicast receiver address and cannot loop.

If the topology has no loops, then looping of *wildcard Resv* and *ResvErr* messages, i.e., messages with wildcard sender selection, 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, further effort is needed to prevent auto-refresh loops of wildcard *Resv* messages and fast loops of wildcard *ResvErr* messages. The solution to this problem adopted by this protocol specification is for such messages to carry an explicit sender address list in a SCOPE object.

When a *Resv* message with WF style is to be forwarded to a particular previous hop, a new SCOPE object is computed from the SCOPE objects that were received in matching *Resv* messages. 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* 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 message that arrived on outgoing interface OI, the substitute list is the set of senders that route to OI.
2. Any local senders (i.e., any sender applications on this node) are removed from this set.
3. If the SCOPE object is to be sent to PHOP, remove from the set any senders that did not come from PHOP.

Figure 11 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 sender selection. 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 *ResvErr* messages with WF style:

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 3.1.5. 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.

3.4 Local Repair

When a route changes, the next *Path* or *Resv* refresh message will establish path or reservation state (respectively) along the new route. To provide fast adaptation to routing changes without the

```
                   ________________
               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 11: SCOPE Objects in Wildcard-Scope Reservations

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 a quick 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 destination G, RSVP should wait for a short period W, and then send *Path* refreshes for all sessions G/* (i.e., for any session with destination G, regardless of destination port).

  The short wait period before sending *Path* refreshes is to allow the routing protocol getting settled with the new change(s), and the exact value for W should be chosen accordingly. Currently W = 2 sec is suggested; however, this value should be configurable per interface.
- 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.

3.5 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 generation of successive refreshes for the state by the neighbor node, and the local state's 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 R value is then used to determine the value for L when the state is received and stored. The values for R and L may vary from hop to hop.

In more detail:

1. Floyd and Jacobson [FJ94] have shown that periodic messages generated by independent network nodes can become synchronized. This can lead to disruption in network services as the periodic messages contend with other network traffic for link and forwarding resources. Since RSVP sends periodic refresh messages, it must avoid message synchronization and ensure that any synchronization that may occur is not stable.

   For this reason, the refresh timer should be randomly set to a value in the range [0.5R, 1.5R].
2. To avoid premature loss of state, L must satisfy L >= (K + 0.5)*1.5*R, where K is a small integer. Then in the worst case, K-1 successive messages may be lost without state being deleted. To compute a lifetime L for a collection of state with different R values R0, R1, ..., replace R by max(Ri).

Currently K = 3 is suggested as the default. However, it may be necessary to set a larger K value for hops with high loss rate. K may be set either by manual configuration per interface, or by some adaptive technique that has not yet been specified.

3. Each message that creates state (*Path* or *Resv* message) carries a TIME_VALUES object containing the R used to generate refreshes; the recipient node uses this R to determine L of the stored state.

4. R is chosen locally by each node. If the node does not implement local repair of reservations disrupted by route changes, a smaller R speeds up adaptation to routing changes, while increasing the RSVP 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 control the amount of overhead due to refresh messages.

   The current suggested default for R is 30 seconds. However, the default should be configurable per interface.

5. 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. The values of Rdef, K, and Slew.Max used in an implementation should be easily modifiable per interface, as experience may lead to different values. The possibility of dynamically adapting K and/or Slew.Max in response to measured loss rates is for future study.

3.6 Traffic Policing and TTL

RSVP is required to compute and pass several service-related flags to traffic control: policing flags and a non-RSVP flag.

Some QoS services may require traffic policing at some or all of (1) the edge of the network, (2) a merging point for data from multiple senders, and/or (3) a branch point where traffic flow from upstream may be greater than the downstream reservation. RSVP knows where such points occur and must so indicate to the traffic control mechanism. On the other hand, RSVP does not interpret the service embodied in the flowspec and therefore does not know whether policing will actually be applied in any particular case.

The RSVP daemon passes to traffic control a separate policing flag for each of these three situations.

- E_Police_Flag – Entry Policing

  This flag is set in the first-hop RSVP node that implements traffic control (and is therefore capable of policing).

  For example, sender hosts must implement RSVP but currently many of them do not implement traffic control. In this case, the E_Police_Flag should be off in the sender host, and it should only be set on when the first hop capable of traffic control is reached. This is controlled by the E_Police flag in SESSION objects.

- M_Police_Flag – Merge Policing

  This flag should be set on for a reservation using a shared style (WF or SE) when flows from more than one sender are being merged.

- B_Police_Flag – Branch Policing

  This flag should be set on when the flowspec being installed is smaller than, or incomparable to, a FLOWSPEC in place on any other interface, for the same FILTER_SPEC and SESSION.

RSVP must also detect and report to receivers the presence of non-RSVP hops in the path. For this purpose, an RSVP daemon must place into each *Path* message that it sends the value of the IP TTL with which the message was sent. The RSVP-capable node that receives this message compares this field to the TTL with which the message was actually received, and if they differ it turns on the Non_RSVP flag. This flag is carried forward to receivers in the ADSPEC [??].

3.7 Multihomed Hosts

Accommodating multihomed hosts requires some special rules in RSVP. We use the term 'multihomed host' to cover both hosts (end systems) with more than one network interface [could ref. section 3.3.4 of RFC-1122], and routers that are supporting local application programs.

An application executing on a multihomed host may explicitly specify which interface any given flow will use for sending and/or for receiving data packets, to override the system-specified default interface. The RSVP daemon must be aware of the default, and if an application sets a specific interface, it must also pass that information to RSVP.

- Sending Data

  A sender application uses an API call (SENDER in Section 3.9.1) to declare to RSVP the characteristics of the data flow it will originate. This call may optionally include the local IP address of the sender. If it is set by the application, this parameter must be the interface address for sending the data packets; otherwise, the system default interface is implied.

The RSVP daemon on the host then sends *Path* messages for this application out the specified interface (only).

- Making Reservations

  A receiver application uses an API call (called RESERVE in Section 3.9.1) to request a reservation from RSVP. This call may optionally include the local IP address of the receiver, i.e., the interface address for receiving data packets. In the case of multicast sessions, this is the interface on which the group has been joined. If the parameter is omitted, the system default interface is used.

  In general, the RSVP daemon should send *Resv* messages for application out the specified interface. However, when the application is executing on a router and the session is multicast, a more complex situation arises. Suppose in this case that a receiver application joins the group on an interface Iapp that differs from Isp, the shortest-path interface to the sender. Then there are two possible ways for multicast routing to deliver data packets to the application. The RSVP daemon must determine which case holds by examining the path state, to decide which incoming interface to use for sending *Resv* messages.

  1. The multicast routing protocol may create a separate branch of the multicast distribution 'tree' to deliver to Iapp. In this case, there will be path state for both Isp and Iapp. The path state on Iapp should only match a reservation from the local application; it must be marked "Local_only" by the RSVP daemon. If "Local_only" path state for Iapp exists, the *Resv* message should be sent out Iapp.

     Note that it is possible for the path state blocks for Isp and Iapp to have the same next hop, if there is an intervening non-RSVP cloud.

  2. The multicast routing protocol may forward data within the router from Isp to Iapp. In this case, Iapp will appear in the list of outgoing interfaces of the path state for Isp, and the *Resv* message should be sent out Isp.

3.8 Future Compatibility

We may expect that in the future new object C-Types will be defined for existing object classes, and perhaps new object classes will be defined. It will be desirable to employ such new objects within the Internet using older implementations that do not recognize them. Unfortunately, this is only possible to a limited degree with reasonable complexity. The rules are as follows.

1. Unknown Class

   There are two possible ways that an RSVP implementation can treat an object with unknown class. This choice is determined by the high-order bit of the Class-Num octet, as follows.

- Class-Num >= 128

  In this case, the entire message should be rejected and an "Unknown Object Class" error returned.

- Class-Num < 128

  In this case, the node should ignore the object but forward it, unexamined and unmodified, in all messages resulting from the state contained in this message.

  For example, suppose that a *Resv* message that is received contains an object of unknown class. Such an object should be saved in the reservation state without further examination; however, only the latest object with a given (unknown class, C-Type) pair should be saved. When a *Resv* message is forwarded, it should include copies of such saved unknown-class objects from all reservations that are merged to form the new *Resv* message.

  Note that objects with unknown class cannot be merged; however, unmerged objects may be forwarded until they reach a node that knows how to merge them. Forwarding objects with unknown class enables incremental deployment of new objects; however, the scaling limitations of doing so must be carefully examined before a new object class is deployed with Class-Num < 128.

These rules should be considered when any new Class-Num is defined.

2. Unknown C-Type for Known Class

   One might expect the known Class-Num to provide information that could allow intelligent handling of such an object. However, in practice such class-dependent handling is complex, and in many cases it is not useful.

   Generally, the appearance of an object with unknown C-Type should result in rejection of the entire message and generation of an error message (*ResvErr* or *PathErr* as appropriate). The error message will include the Class-Num and C-Type that failed (see Appendix B); the end system that originated the failed message may be able to use this information to retry the request using a different C-Type object, repeating this process until it runs out of alternatives or succeeds.

   Objects of certain classes (FLOWSPEC, ADSPEC, and POLICY_DATA) are opaque to RSVP, which simply hands them to traffic control or policy modules. Depending upon its internal rules, either of the latter modules may reject a C-Type and inform the RSVP daemon; RSVP should then reject the message and send an error, as described in the previous paragraph.

3.9 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).

3.9.1 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 Session

```
Call: SESSION( DestAddress , ProtocolId, DstPort ,

                [ , SESSION_object ]

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

  This call initiates RSVP processing for a session, defined by DestAddress together with ProtocolId and possibly a port number DstPort. If successful, the SESSION call returns immediately with a local session identifier Session-id, which may be used in subsequent calls.

  The Upcall_Proc_addr parameter defines the address of an upcall procedure to receive asynchronous error or event notification; see below. 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.

- Define Sender

```
Call: SENDER( Session-id,

                [ , Source_Address ]  [ , Source_Port ]

                [ , Sender_Template ]

                [ , Sender_Tspec ]   [ , Data_TTL ]
```

```
[ , Sender_Policy_Data ] )
```

A sender uses this call to define, or to modify the definition of, the attributes of the data stream. The first SENDER call for the session registered as 'Session-id' will cause RSVP to begin sending *Path* messages for this session; later calls will modify the path information.

The SENDER parameters 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.
- 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.
- Sender_Tspec

  This optional parameter describes 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.

• Reserve

```
Call: RESERVE( session-id, [ receiver_address , ]

              [ ACK_flag, ] style, style-dependent-parms )
```

A receiver uses this call to make or to modify a resource reservation for the session registered as 'session-id'. 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 existing reservations may result in admission control failure).

The optional 'receiver_address' parameter may be used by a receiver on a multihomed host (or router); it is the IP address of one of the node's interfaces. The ACK_flag should be set on if a reservation ACK is desired, off otherwise. 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 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 removes RSVP state for the session specified by session-id. The node then sends appropriate teardown messages and ceases sending refreshes for this session-id.

- Error/Event Upcalls

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

                [ Error_code , Error_value ,

                     Error_Node , 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 SESSION call.

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

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

   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 is triggered by the receipt of the first reservation message or by modification of a 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 a SENDER call.

   The Error_code parameter will define the error, and Error_value may supply some additional (perhaps system-specific) data about the error. The Error_Node parameter will specify the IP address of the node that detected the error.

   'List_count' will be 1, and Filter_spec_list will contain the Sender_Template supplied in the SENDER call; Flow_Spec_list and Advert_list will each contain one NULL object. The Policy_data parameter will contain any POLICY_DATA objects in the *PathErr* message.

4. Info_type = Resv Error/Confirmation

   An Resv Error/Confirmation event indicates an error in a reservation message to which this application contributed, or the receipt of a *ResvConfirm* message. The Error_code parameter will define the error or confirmation. For an error, Error_value may supply some additional (perhaps system-specific) data. The Error_Node parameter will specify the IP address of the node that detected the event being reported.

   Filter_spec_list and Flowspec_list will contain the FILTER_SPEC and FLOWSPEC objects from the **error flow descriptor** (see Section 3.1.5). List_count will specify the number of FILTER_SPECS in Filter_spec_list, while there will be one FLOWSPEC in Flowspec_list. For an error, the Policy_data parameter will contain any POLICY_DATA objects in the *ResvErr* message.

Although RSVP messages indicating path or resv events 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. All error and confirmation events should be reported to the application.

3.9.2 RSVP/Traffic Control Interface

In an RSVP-capable node, 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.

- Make a Reservation

```
Call: Rhandle =  TC_AddFlowspec( Interface, TC\_Flowspec,

                       TC_Tspec, E_Police_Flag,

                       M_Police_Flag, B_Police_Flag )
```

The TC_Flowspec parameter defines the desired effective QoS to admission control; its value is computed as the maximum over the flowspecs of different next hops (see the Compare_Flowspecs call below). It contains the effective reservation Tspec Resv_Te (although the RSVP daemon itself has no means to extract the Tspec). The TC_Tspec parameter defines the effective sender Tspec Path_Te (see Section 2.3). We assume that traffic control takes the min of Resv_Te and Path_Te (see step (4) in Section 2.3).

E_Police_Flag, M_Police_Flag, and B_Police_Flag are Boolean parameters whose values should be set as described in Section 3.6.

The TC_AddFlowspec 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.

- Modify Reservation

```
Call: TC_ModFlowspec( Rhandle, new_Flowspec,

                      Sender_Tspec,  E_Police_flag,

                       M_Police_Flag, B_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. The other parameters are defined as in TC_AddFlowspec.

- Delete Flowspec

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

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

- 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.

- Delete Filter Spec

  `Call: TC_DelFilter( FHandle )`

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

- OPWA Update

  ```
  Call: TC_Advertise( interface, Adspec,

                  [ , Non_RSVP_flag ] ) -> New_Adspec
  ```

  This call is used for OPWA to compute the outgoing advertisement New_Adspec for a specified interface.

- 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.

3.9.3 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 packet received for IP protocol 46 must be diverted to the RSVP program for processing, without being forwarded. On a router, the identity of the interface, real or virtual, on which it is received must also be available to the RSVP daemon.

- Route Query

  To forward *Path* and *PathTear* messages, an RSVP daemon must be able to query the routing daemon(s) for routes.

```
Ucast_Route_Query( [ SrcAddress, ] DestAddress, Notify_flag )

                      -> OutInterface

Mcast_Route_Query( [ SrcAddress, ] DestAddress, Notify_flag )

                      -> [ IncInterface, ] OutInterface_list
```

Depending upon the routing protocol, the query may or may not depend upon SrcAddress, i.e., upon the sender host IP address, which is also the IP source address of the message. Here IncInterface is the interface through which the packet is expected to arrive; some multicast routing protocols may not provide it.

If the Notify_flag is True, routing will save state necessary to issue unsolicited route change notification callbacks (see below) whenever the specified route changes. Such callbacks will be enabled until routing receives a route query call with the Notify_Flag set False.

A multicast route query may return an empty OutInterface_list if there are no receivers downstream of a particular router. A route query may also return a 'No such route' error, probably as a result of a transient inconsistency in the routing (since a *Path* or *PathTear* message for the requested route did arrive at this node). In either case, the local state should be updated as requested by the message, although it cannot be forwarded further. Updating local state will make path state available immediately for a new local receiver, or it will tear down path state immediately.

- Route Change Notification

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

```
Ucast_Route_Change( ) -> DestAddress, OutInterface

Mcast_Route_Change( ) -> [ SrcAddress, ] DestAddress,

              [ IncInterface, ] 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 to send different versions of an outgoing *Path* message on different interfaces. It is also necessary in some cases to avoid routing loops.

- Source Address Specification

  RSVP must be able to specify the IP source address to be used when sending *Path* messages.

- Interface List Discovery

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

3.9.4 Service-Dependent Manipulations

Flowspecs, Tspecs, and Adspecs are opaque objects to RSVP; their contents are defined in service specification documents. In order to manipulate these objects, RSVP daemon must have available to it the following service-dependent routines.

- Compare Flowspecs

  ```
  Compare_Flowspecs( Flowspec_1, Flowspec_2 ) -> result_code
  ```

  The possible result_codes indicate: flowspecs are equal, Flowspec_1 is greater, Flowspec_2 is greater, flowspecs are incomparable but LUB can be computed, or flowspecs are incompatible.

  Note that comparing two flowspecs implicitly compares the Tspecs that are contained. Although the RSVP daemon cannot itself parse a flowspec to extract the Tspec, it can use the Compare_Flowspecs call to implicitly calculate Resv_Te (see Section 2.3).

- Compute LUB of Flowspecs

  ```
  LUB_of_Flowspecs( Flowspec_1, Flowspec_2 ) ->
  Flowspec_LUB
  ```

- Compare Tspecs

  ```
  Compare_Tspecs( Tspec_1, Tspec_2 ) -> result_code
  ```

  The possible result_codes indicate: Tspecs are equal, or Tspecs are unequal.

- Sum Tspecs

  ```
  Sum_Tspecs( Tspec_1, Tspec_2 ) -> Tspec_sum
  ```

This call is used to compute Path_Te (see Section 2.3).

4 Message Processing Rules

This section provides a generic description of the rules for RSVP operation. It is intended to outline a set of algorithms that will accomplish the needed function. An actual implementation may use different but equivalent algorithms. This section assumes the generic interface calls defined in Section 3.9 and the following data structures. An actual implementation may use additional or different data structures and interfaces.

[NOTE: This section is always the last to be updated when changes are made, and it is neither correct nor complete at the present time. Therefore, when this section disagrees with the rest of the text, you should believe the rest of the text!]

- PSB – Path State Block

  Each PSB holds path state for a particular (session, sender) pair, defined by SESSION and SENDER_TEMPLATE objects, respectively, received in a *Path* message.

  PSB contents include the following values from a *Path* message:

  – The previous hop IP address from a PHOP object (required)
  – LIH, the Logical Interface Handle from the previous hop, from a PHOP object (required).
  – The remaining IP TTL (required)
  – SENDER_TSPEC (required)
  – POLICY_DATA and/or ADSPEC objects (optional)
  – Non_RSVP flag (required); see Section 3.6.

  In addition, the PSB contains the following information provided by routing: OutInterface_list, the list of outgoing interfaces for this (sender, destination), and IncInterface, the expected incoming interface. For a unicast destination, OutInterface_list contains one entry and IncInterface is undefined.

- RSB – Reservation State Block

  Each RSB holds a reservation request that arrived in a particular *Resv* message, corresponding to the triple: (session, next hop, filter_spec_list). Here "filter_spec_list" may be a list of FILTER_SPECs (for SE style), a single FILTER_SPEC (FF style), or empty (WF style). We use the symbol "FILTER_SPEC*" to indicate such a FILTER_SPEC list.

  RSB contents include:

  – The outgoing (logical) interface OI on which the reservation is to be made or has been made (required).
  – FLOWSPEC*, list of FLOWSPEC objects (required)

- The style (required)
- A POLICY_DATA object (optional)
- A SCOPE object (optional, depending on style)
- A RESV_CONFIRM object (optional)

- TCSB – Traffic Control State Block

  TCSB's hold the reservation specifications that have been handed to traffic control for specific outgoing interfaces. In general, information in TCSB's is derived from RSB's for the same outgoing interface. Each TCSB defines a single reservation for a particular triple: (session, OI, filter_spec_list). TCSB contents include:

  - TC_Flowspec, the effective flowspec, i.e., the maximum over the corresponding FLOWSPEC values from matching RSB's. TC_Flowspec is passed to traffic control to make the actual reservation. The Tspec part of TC_Flowspec is the effective reservation Tspec Resv_Te (Section 2.3).
  - TC_Tspec, equal to the effective sender Tspec Path_Te.
  - Police Flags

    The flags E_Police_Flag, M_Police_Flag,and B_Police_Flag are defined in Section 3.6.
  - Rhandle, F_Handle_list

    Handles returned by the traffic control interface, corresponding to the reservation (flowspec) and to the list of filter specs.

Boolean flags Path_Refresh_Needed, Resv_Refresh_Needed, and Tear_Needed will also be used in this section.

[LZ: It might be very helpful to have a short section to summarize the management of all the timers.]

MESSAGE ARRIVES

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

Reassemble a fragmented message.

Parse the sequence of objects in the message to verify the length field of the common header; discard message if there is a mismatch.

If the message type is not *Path* or *PathTear* and if the IP destination address does not match any of the addresses of the local interfaces, then forward the message to IP destination address and return.

Verify the INTEGRITY object, if any. If the check fails, discard the message and return.

Further processing depends upon message type.

PATH MESSAGE ARRIVES

Process the **sender descriptor** object sequence in the message as follows. The flags Path_Refresh_Needed and Resv_Refresh_Needed flags are initially off.

- 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.
- If the DstPort in the SESSION object is zero but the SrcPort in the SENDER_TEMPLATE object is non-zero, build a send a "Conflicting Src Port" *PathErr* message, drop the *Path* message, and return.
- Search for a path state block (PSB) whose (SESSION, SENDER_TEMPLATE) pair matches the corresponding objects in the message, considering any wildcard ports.
- If, during the PSB search, a PSB is found whose session matches the DestAddress and Protocol Id fields of the received SESSION object, but the DstPorts differ and one is zero, then build and send a "Conflicting Dst Port" *PathErr* message, drop the *Path* message, and return.
- If, during the PSB search, a PSB is found with a matching sender host (in SENDER_TEMPLATE) but the SrcPorts differ and one is zero, then build and send a "Ambiguous Path" *PathErr* message, drop the *Path* message, and return.
- If there was no matching PSB, then:
  1. Create a new PSB.
  2. Call the appropriate Route_Query routine, using DestAddress from SESSION and (for multicast routing) SrcAddress from SENDER_TEMPLATE. Store the values of OutInterface_list and IncInterface into the PSB. However, if the sender is from the local API, then instead of invoking routing, set OutInterface_List to the single interface whose address matches the sender address; IncInterface is undefined in this case.
  3. If IncInterface is defined and if a multicast message arrived on an interface different from IncInterface, drop the message and return.
  4. Set a cleanup timer for the PSB. If this is the first PSB for the session, set a refresh timer for the session.
  5. Copy contents of the SESSION, SENDER_TEMPLATE, SENDER_TSPEC, and PHOP (IP address and LIH) objects into the PSB. Store the received TTL into the PSB. Copy into the PSB either of the following objects that are present: POLICY_DATA and ADSPEC.
  6. Turn on the Path_Refresh_Needed flag.

- Otherwise (there is a matching PSB and there is no dest port conflict):
  1. If there is no route change notification in place, call the appropriate Route_Query routine using DestAddress from SESSION and (for multicast routing) SrcAddress from SENDER_TEMPLATE.
     - If the OutInterface_list that is returned differs from that in the PSB, execute the PATH LOCAL REPAIR event sequence below.
     - If a multicast message arrived on an interface different from IncInterface, drop the message and return.
  2. If the PHOP IP address, the LIH, or SENDER_TSPEC differs between the message and the PSB, copy the new value into the PSB, execute the RESV REFRESH event sequence for the sender defined by the PSB, and turn on the Path_Refresh_Needed flag.
     [LZ: [When] should ADSPEC change trigger a refresh?]
     However, if the *Path* message being processed came from a local application and if there is reservation state for this session, then make a Resv Event upcall to that application instead of executing the RESV REFRESH sequence.
     ```
     Call: <Upcall_Proc>( session-id, Resv Event, 1,
                 {Flowspec}, NULL, NULL, NULL )
     ```
  3. Restart the cleanup timer.
- If the message arrived with a TTL different from Send_TTL in the RSVP common header, set the Non_RSVP flag on in the PSB.
- If the Path_Refresh_Needed flag is now set then:
  1. If this *Path* message came from a network interface and not from a local application, make a Path Event upcall for each local application for this session:
     ```
     Call: <Upcall_Proc>( session-id, Path Event, 1,
                 {SENDER_TSPEC}, {SENDER_TEMPLATE},
                 {ADSPEC}, {POLICY_DATA} )
     ```
  2. Execute the PATH REFRESH event sequence (below) for the sender defined by the PSB.

PATH TEAR MESSAGE ARRIVES

- Search for a PSB whose (SESSION, SENDER_TEMPLATE) pair matches the corresponding objects in the message. If no matching PSB is found, drop the *PathTear* message and return.
- Forward a copy of the *PathTear* message to each outgoing interface listed in OutInterface_list of the PSB.

- Find each RSB that matches this PSB, i.e., whose FILTER_SPEC object matches the SENDER_TEMPLATE in the PSB and whose OI is included in OutInterface_list.

  If this RSB matches no other PSB, then tear down the RSB, as described below under RESV TEAR MESSAGE ARRIVES.
- Delete the PSB.
- Drop the *PathTear* message and return.

PATH ERROR MESSAGE ARRIVES

- Search for a PSB whose (SESSION, SENDER_TEMPLATE) pair matches the corresponding objects in the message. If no matching PSB is found, drop the *PathErr* message and return.
- If the previous hop address in the PSB is the local API, make an error upcall to the application:

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

  Any POLICY_DATA, SENDER_TSPEC, or ADSPEC object in the message is ignored. [LZ: Why we don't send these objects up to application? They might of some help to understand the errors.] Drop the *PathErr* message and return.
- Otherwise, send a copy of the *PathErr* message to the PHOP IP address, drop the *PathErr* message, and return.

RESV MESSAGE ARRIVES

Initially, the Resv_Refresh_PHOP* list is empty and the Resv_Refresh_Needed flag is off. These variables are used to control immediate reservation refreshes.

- Process the NHOP object

  The logical outgoing interface OI is taken from the LIH in the NHOP object. (If the physical interface is not implied by the LIH, it can be learned from the interface matching the IP destination address).
- 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 the receiver host itself.

  [LZ: Explain this?]

- Check the S_POLICY_DATA object.

  If there is an S_POLICY_DATA object in the message, check permission to create a reservation for the session. If the check fails, build and send an "Administrative rejection" *ResvErr* message, drop the *Resv* message, and return. Otherwise, copy the S_POLICY_DATA object into the RSB.

Now process the STYLE object and the **flow descriptor list** to make reservations, as follows.

For FF style, execute the following steps independently for each flow descriptor, i.e., for each (FLOWSPEC, FILTER_SPEC) pair. For FF style, FILTER_SPEC* consists of the single FILTER_SPEC from the flow descriptor.

For SE style, execute the following steps once, with FILTER_SPEC* consisting of the list of FILTER_SPEC objects from the flow descriptor.

For WF style, execute the following steps once, with FILTER_SPEC* consisting of a single internal placeholder "WILD_FILTER".

- If the DstPort in the SESSION object is zero but the SrcPort in the FILTER_SPEC object is non-zero, build a send a "Conflicting Src Port" *ResvErr* message, drop the *Resv* message, and return.
- Find or create a reservation state block (RSB) for the triple: (SESSION, NHOP, FILTER_SPEC*). Call this the *active RSB*.
- If the RSB is not new and if its style is incompatible with the STYLE object in the message, build and send a *ResvErr* message specifying "Conflicting Style", drop the *Resv* message, and return.
- Start or restart the cleanup timer on the the active RSB.
- If the active RSB is not new, check whether FLOWSPEC or SCOPE objects have changed. If not, continue with the next **flow descriptor** in the *Resv* message, if any.
- If the active RSB is new, set its OI and style, and copy any FLOWSPEC, POLICY_DATA, and/or SCOPE objects into it.
- If there is a RESV_CONFIRM in the message, turn on Resv_Refresh_Needed and save the object in the RSB.
- The active RSB must be new or changed. Compute the traffic control parameters, using the following steps.

  1. Locate the set of PSBs (senders) whose SENDER_TEMPLATEs match FILTER_SPEC* in the active RSB and whose OutInterface_list includes OI.
     If this set is empty, build and send an error message specifying "No sender information", and continue with the next flow descriptor in the *Resv* message.
  2. If this set contains more than one PSB and if the style has explicit sender selection (e.g., FF or SE), build and send an error message specifying "Ambiguous filter spec" and continue with the next flow descriptor.

3. Add the PHOP from the PSB to the Resv_Refresh_PHOP* list, if the PHOP is not already on the list.
4. Set TC_E_Police_flag on if any of these PSBs have their E_Police flag on. Set TC_M_Police_flag on if it is a shared style and there is more than one PSB in the set.
5. Compute Path_Te as the sum of the SENDER_TSPEC objects in this set of PSBs.
6. Scan all RSB's matching the SESSION and Filter_Spec_list from the message.
   - If any of these RSB's has a style that is incompatible with the specifying "Conflicting Style", drop the *Resv* message, delete the RSB if it has just been created, and return.
   - Set TC_B_Police_flag on if TC_Flowspec is smaller than, or incomparable to, any FLOWSPEC in those RSB's.
7. Consider the set of RSB's for the same (SESSION, OI, Filter_Spec_list) triple from the message.
   - Compute the effective kernel flowspec, TC_Flowspec, as the maximum of the FLOWSPEC values in these RSB's.
   - Compute the effective kernel filter spec (list), TC_Filter*. by merging the FILTER_SPEC* object (lists) from these RSB's.

- Search for a TCSB matching the triple (SESSION, OI, FILTER_SPEC*), taken from the RSB.

  1. If none is found but style is SE, search for a TCSB matching (SESSION, OI). If find one and if TCSB's TC_Flowspec, Path_Te, and police flags match the computed values, then
     - Make an appropriate set of TC_DelFilter and TC_AddFilter calls to transform the Filter_Spec_list in the TCSB into the Filter_Spec_list from the message.
     - Set Resv_Refresh_Needed on, drop the *Resv* message, and return.
  2. Otherwise, if none is found:
     - Create a new TCSB.
     - Store TC_Flowspec, Filter_Spec_list, Path_Te, and the police flags into TCSB. [SCOPE?]
     - Set Resv_Refresh_Needed on.
     - Make the traffic control call:

```
Rhandle = TC_AddFlowspec( OI, TC_flowspec, Path\_Te,
                          TC_E_Police_flag, TC_M_Police_flag,
                          TC_B_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 Rhandle in the TCSB.

- For each filter_spec F in Filter_Spec_list, call:

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

  and record the returned Fhandle in the TCSB.
- Continue with the next flow descriptor.

3. Otherwise (found existing TCSB), check whether TC_Flowspec, Path_Te, and/or any of the police flags has changed, and if so:
   - Store TC_Flowspec, Filter_Spec_list, Path_Te, and the police flags into it. [SCOPE?]
   - Set Resv_Refresh_Needed on.
   - Make the traffic control call:

```
TC_ModFlowspec( Rhandle, K_Flowspec, Path\_Te,
                    TC_E_Police_flag, TC_M_Police_flag,
                    TC_B_Police_flag )
```

4. Continue with the next flow descriptor.

- If the Resv_Refresh_Needed flag is now on, execute the RESV REFRESH sequence (below) for each PHOP in the Resv_Refresh_PHOP* set.

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 with an IP destination address matching outgoing interface OI. Flags Tear_Needed and Resv_Refresh_Needed are initially off and Resv_Refresh_PHOP* list is empty.

- Process the STYLE object and the **flow descriptor list** in the *ResvTear* message to tear down local reservation state, as follows.

  For FF style, execute the following steps for each flow descriptor, i.e., for each (FLOWSPEC, FILTER_SPEC) pair independently, with Filter_Spec_list consisting of a single FILTER_SPEC object.

  For SE style, execute the following steps once, with Filter_Spec_list consisting of a list of FILTER_SPEC objects.

  For WF style, execute the following steps once, with Filter_Spec_list consisting of a single internal placeholder “WILD_FILTER”.

  1. Find matching RSB for the 4-tuple: (SESSION, NHOP, style, Filter_Spec_list); call this the active RSB. If no active RSB is found, continue with next flow descriptor.

2. Delete the active RSB.
3. Find TCSB for the triple: (SESSION, OI, Filter_Spec_list).
4. Consider the set of RSB's matching this TCSB. If there are none:
   - Call the traffic control interface routine:

     ```
     TC_DelFlowspec( Rhandle )
     ```

   - Delete the TCSB and set Tear_Needed flag on.
   - Continue with the next flow descriptor.
5. Otherwise (there are other RSB's for the same TCSB), recompute TC_Flowspec and Path_Te (see RESV MESSAGE ARRIVES). (This also adds the appropriate PHOP addresses to the Resv_Refresh_PHOP* list¿) If either changed, update the TCSB, set flag Resv_Refresh_Needed on, and call the traffic control interface module:

   ```
   TC_ModFlowspec( Rhandle, TC_Flowspec, Path\_Te)
                        TC_E_Police_flag, TC_M_Police_flag,
                        TC_B_Police_flag )
   ```

   This kernel call should not fail, since the reservation can only be reduced.

[LZ: Suppose receiver R has the credential to make the reservation and others took a ride on top of R's credential. Now R tears down its request, what should happen? Shouldn't TEAR take policy data as input?]

- If Tear_Needed and Resv_Refresh_Needed flags are both off, then drop the *ResvTear* message and return.
- If Tear_Needed is off but Resv_Refresh_Needed is on, then execute the RESV REFRESH sequence for each PHOP in the Resv_Refresh_PHOP* set, drop the *ResvTear* message, and return.
- Otherwise (Tear_Needed is on), need to forward *ResvTear* and/or *Resv* refresh messages. Do the following for each PSB whose OutInterface_list includes the outgoing interface OI:
  1. Pick each flow descriptor Fj in the *ResvTear* message whose FILTER_SPEC matches the PSB, and do the following.
     - If there is no RSB whose FILTER_SPEC matches the PSB, then add Fj to the new *ResvTear* message being built.
     - Otherwise (there is a matching RSB), note the PSB as needing a *Resv* refresh message and set the Resv_Refresh_Needed flag True.
  2. If the new *ResvTear* message contains any flow descriptors, send it to PHOP in the PSB.
- If the Resv_Refresh_Needed flag is now on, execute the RESV REFRESH sequence (below) for each PHOP in the Resv_Refresh_PHOP* set.

If the Refresh_Needed flag is true, then execute the RESV REFRESH sequence for the PSB's that have been noted.

- Drop the *ResvTear* message and return.

RESV ERROR MESSAGE ARRIVES

A *ResvErr* message arrives through the (real) incoming interface In_If.

- If there is no path state for SESSION, drop the *ResvErr* message and return.
- Do the following with each RSB for this SESSION whose OI does not match In_If and whose FILTER_SPEC matches that in the *ResvErr* message.
  1. Copy the **error flow descriptor** from the incoming *ResvErr* message.
  2. 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.
  3. If NHOP in RSB is the local API, deliver an error upcall to application:

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

     and continue with the next RSB. Here k, Filter_Spec_List, and Flowspec_List are constructed from the **error flow descriptor**.
  4. If the *Resv* message has wildcard sender selection, 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.
  5. Create a new *ResvErr* message containing the **error flow descriptor** and send to the NHOP address specified by the RSB. Include SC.Out if the sender selection is wildcard.
  6. Continue with the next RSB.
- Drop the *ResvErr* message and return.

RESV CONFIRMATION ARRIVES

If the (unicast) IP address found in its RESV_CONFIRM object matches an interface of the node, a confirmation upcall is made to the matching application:

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

Otherwise, the *ResvConfirm* message is forwarded immediately to the address in the IP address in its RESV_CONFIRM object.

PATH REFRESH

This sequence sends a path refresh for a particular sender, i.e., a PSB. This sequence may be entered by either the expiration of the path refresh timer or directly as the result of the Path_Refresh_Needed flag being turned on during the processing of a received *Path* message.

- 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.
- Insert TIME_VALUES and PHOP objects into the *Path* message being built.
- Create a **sender descriptor** containing the SENDER_TEMPLATE, SENDER_TSPEC, and POLICY_DATA objects, if present in the PSB, and pack it into the *Path* message being built.
- Pass any ADSPEC and SENDER_TSPEC objects present in the PSB to the traffic control call TC_Advertise. Insert the modified ADSPEC object that is returned 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.
- Send a copy of the *Path* message to each interface in OutInterfact_list. Before sending each copy, insert into its PHOP object the interface address and the LIH for the interface.

RESV REFRESH

This sequence sends a reservation refresh towards a particular previous hop with IP address PH. This sequence may be entered by either the expiration of a reservation refresh timer or directly as the result of the Resv_Refresh_Needed flag being turned on as the result of processing a *Resv* or *ResvTear* message.

In general, this sequence considers each of the PSB's with PHOP address PH. For a given PSB, it scans the RSBs for matching reservations and merges the styles, FLOWSPECs and FILTER_SPEC*'s appropriately. It then builds a *Resv* message and sends it to PH. The details depend upon the attributes of the style(s) included in the reservations.

- If there are PSB's from more than one PHOP and if the multicast routing protocol does not use shared trees, set the Need_Scope flag on, otherwise set it off.

- Create an output message containing SESSION, RSVP_HOP, INTEGRITY, and TIME_VALUES objects.
- Select each sender PSB whose PHOP has address PH.
  1. Select all RSB's whose FILTER_SPEC*'s match the SENDER_TEMPLATE object in the PSB and whose OI appears in the OutInterface_list of the PSB.
  2. Get a STYLE object from the first RSB and move it into the output message. (Note that the present set of styles are never themselves merged; if future styles can be merged, these rules will become more complex).
  3. Compute the maximum/LUB over the FLOWSPEC objects of this set of RSB's.
  4. While computing the maximum/LUB, check for a RESV_CONFIRM object in each RSB. If a RESV_CONFIRM object is found and if the FLOWSPEC in that RSB is larger than all other flowspecs being compared, then save this RESV_CONFIRM object. If a RESV_CONFIRM object is found but the corresponding FLOWSPEC is equal or smaller than the largest, or if the result of merging was a LUB, then create and send a *ResvConfirm* message to the address in the RESV_CONFIRM object.
     - Include the RESV_CONFIRM object in the *ResvConfirm* message.
     - Build a confirmation ERROR_SPEC object and include it in the *ResvConfirm* message. The Error_Node parameter in this object should be the IP address of OI from the RSB.

     Then delete the RESV_CONFIRM object from the RSB.
  5. Merge the matching FILTER_SPEC objects from this set of RSB's. The merging rule depend upon the style:

     Explicit sender selection (FF, SE) styles:

     Use the SENDER_TEMPLATE as the merged FILTER_SPEC.

     Wildcard sender selection (WF) style:

     There is no filter spec to merge.
  6. If the Need_Scope flag is on, compute a new SCOPE object as the union of the SCOPE objects found in the RSB's.
  7. Merge the F_POLICY_DATA objects from the RSB's.
  8. (All matching RSB's have been processed). The next step depends upon the style attributes.

     Distinct reservation (FF) style

     Pack the merged (FLOWSPEC, FILTER_SPEC, F_POLICY_DATA) triplet into the message as a **flow descriptor**.

     Shared reservation (SE, WF) styles

     Merge (take the maximum) across all PSB's the merged FLOWSPECS from the RSB's.

     If the sender selection is not wildcard (i.e., if it is SE), form the union of the FILTER_SPECs obtained from the RSB's. For Wildcard sender selection (WF) style, there is not filter spec to merge.

9. If the Need_Scope flag is on, remove from the merged SCOPE object all sender addresses that do not match the set of PSB's for PH, and all senders addresses that are local. If the resulting set is empty, no *Resv* should be forwarded to this PHOP; return; otherwise (set is not empty), move the new SCOPE object into the message.

- (All PSB's have been processed). If a shared reservation style is being built, move the final merged FLOWSPEC, F_POLICY_DATA, and FILTER_SPEC (if SE) objects into the message.
- If a RESV_CONFIRM object was saved earlier, copy it into the new *Resv* message and delete it from the RSB in which it was found.
- Set the RSVP_HOP object in the message to contain the IncInterface address through which it will be sent and the LIH from (one of) the PSB's.
- Send the message to the address PH.

A Object Definitions

C-Types are defined for the two Internet address families IPv4 and IP6. To accommodate 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)                |
+-------------+-------------+-------------+-------------+
| Protocol Id |    Flags    |          DstPort          |
+-------------+-------------+-------------+-------------+
```

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

```
+-------------+-------------+-------------+-------------+
|                                                       |
+                                                       +
|                                                       |
+               IP6 DestAddress (16 bytes)              +
|                                                       |
+                                                       +
|                                                       |
+-------------+-------------+-------------+-------------+
| Protocol Id |    Flags    |          DstPort          |
+-------------+-------------+-------------+-------------+
```

DestAddress

The IP unicast or multicast destination address of the session. This parameter must be supplied.

Protocol Id

The IP Protocol Identifier for the data flow. This parameter must be supplied.

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.]

DstPort

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 3.2; 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                     |
+-------------+-------------+-------------+-------------+
```

Refresh Period

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

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 3.1.5.

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

```
+-------------+-------------+-------------+-------------+
|                       Option Vector                   |
+-------------+-------------+-------------+-------------+
```

Option Vector

A set of bit fields giving values for the reservation options. If new options are added in the future, 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:

27 bits: Reserved

2 bits: Sharing control

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

3 bits: Sender selection control

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

The low order bits of the option vector are determined by the style, 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)               |
+-------------+-------------+-------------+-------------+
|    //////   |    //////   |          SrcPort          |
+-------------+-------------+-------------+-------------+
```

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

```
+-------------+-------------+-------------+-------------+
|                                                       |
+                                                       +
|                                                       |
+               IP6 SrcAddress (16 bytes)               +
|                                                       |
+                                                       +
|                                                       |
+-------------+-------------+-------------+-------------+
|    //////   |    //////   |          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'.

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.

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

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

A.12 ADSPEC Class

ADSPEC class = 13.

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

A.13 POLICY_DATA Class

POLICY_DATA class = 14.

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

  The contents of this object are for further study.

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

A.14 RESV_CONFIRM Class

RESV_CONFIRM class = 15.

- IPv4 RESV_CONFIRM object: Class = 15, C-Type = 1

```
+-------------+-------------+-------------+-------------+
|            IPv4 Receiver Address (4 bytes)            |
+-------------+-------------+-------------+-------------+
```

- IP6 RESV_CONFIRM object: Class = 15, C-Type = 2

```
+-------------+-------------+-------------+-------------+
|                                                       |
+                                                       +
|                                                       |
+            IP6 Receiver Address (16 bytes)            +
|                                                       |
+                                                       +
|                                                       |
+-------------+-------------+-------------+-------------+
```

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 rejects the message without updating local state.

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

  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 ss = 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 replacement.
  - Sub-code = 13: Bad Flowspec or Tspec value

    Unreasonable request. High order bit u = 0, i.e., 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.

  The high order 4 bits of the Error Value field are assigned as for Error Code = 01 (above). For Error Code = 02, the following global sub-codes are defined:

- 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 one of the Filterspecs listed in the Resv message. RSVP should reject the message.

• Error Code = 05: Ambiguous path

  Sender port appears both zero and non-zero in same session. RSVP should reject the message.

• Error Code = 06: Ambiguous filter spec

  Filter spec matches more than one sender, in a 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 = 08: Conflicting dest port

  Sessions for same destination address and protocol have appeared with both zero and non-zero dest port fields.

• Error Code = 09: Conflicting source port

  The source port is non-zero in a filter spec or sender template for a session with destination port zero.

• 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 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. RSVP is not expected to be able to interpret this value.

- Error Code = 22: RSVP System error

  The Error Value field will provide implementation-dependent information on the error. RSVP is not expected to be able to interpret this value.

C UDP Encapsulation

An RSVP implementation will generally require the ability to perform "raw" network I/O, i.e., to send and receive IP datagrams using protocol 46. However, some important classes of host systems may not support raw network I/O. To use RSVP, such hosts must encapsulate RSVP messages in UDP.

The basic UDP encapsulation scheme makes two assumptions:

1. All hosts are capable of sending and receiving multicast packets.
2. The first/last-hop routers are RSVP-capable.

A method of relaxing the second assumption is given later.

Let Hu be a "UDP-only" host that requires UDP encapsulation, and Hr a host that can do raw network I/O. The UDP encapsulation scheme must allow RSVP interoperation among an arbitrary topology of Hr hosts, Hu hosts, and routers.

Resv ResvErr ResvTear and *PathErr* messages are sent to unicast addresses learned from the path or reservation state in the node. If the node keeps track of which previous hops and which interfaces need UDP encapsulation, these messages message can be sent using UDP encapsulation when necessary.

On the other hand, *Path* and *PathTear* messages are send to the unicast or multicast destination address for the session. The table in Figure 12 shows the basic rules for UDP encapsulation of such messages. 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. The following symbols are also used:

- D is the DestAddress for the particular session.
- G* is a well-known group address of the form 224.0.0.x, i.e., a group that is limited to the local connected network. [TO BE DEFINED]
- Pu is the well-known UDP port for UDP encapsulation of RSVP: 3455.
- Ra is the IP address of the router interface 'a'.
- Tr is the TTL value of the specific *Path* message.
- Router interface 'a' is on the local network connected to Hu and Hr, while interface 'b' is connected only to another router.

```
                        RSVP              RSVP
Node  Node Type         Send              Receive
---   ----------     -------------     ---------------
Hu   UDP-only host    UDP(G*,Pu,1)     UDP(G*,Pu)
                     or UDP(Ra,Pu,1)   and UDP(D,Pu)
                     [Note 1]          [Note 3]

Hr   Raw-mode host    UDP(G*,Pu,1)     UDP(G*,Pu)
                     and Raw(D,,Tr)    and Raw()

R    Router
      Interface a:    UDP(D,Pu,Tr)     UDP(G*,Pu) [Note 2]
                     and Raw(D,,Tr)    and UDP(Ra,Pu)
                                       and Raw()

      Interface b:    Raw(D,,Tr)           Raw()
```

Figure 12: UDP Encapsulation Rules for Path Messages

[Note 1] Hu sends a *Path* message to Ra only if session destination address D is unicast.

[Note 2] R ignores *Path* messages addressed to G* if D is unicast. (This is necessary to prevent routing and reservation anomalies).

[Note 3] The DestAddress D is the IP address of Hu in this case.

R and Hr 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 destination may take steps to ignore the duplicates, although this redundancy has no ill effect other than overhead for processing the extra messages.

A router may determine if its interface X needs UDP encapsulation by listening for UDP-encapsulated *Path* messages that were sent to either G* (multicast D) or to the address of interface X (unicast D). There is one failure mode for this scheme: if no host on the connected network acts as an RSVP sender, there will be no *Path* messages to trigger UDP encapsulation. In this (unlikely) case, it will be necessary to explicitly configure UDP encapsulation on the local network interface of the router.

A UDP-only host Hu supporting unicast RSVP sessions must somehow know the address Ra, presumably by configuration.

When a UDP-encapsulated packet is received, the IP TTL is not available to the application on most systems. The RSVP daemon that receives a UDP-encapsulated *Path* or *PathTear* message should therefore use the Send_TTL field of the RSVP common header as the effective receive TTL.

We have assumed that the first-hop RSVP-capable router R is on the directly-connected network. There are several possible approaches if this is not the case.

1. Hu can send both unicast and multicast sessions to UDP(Ra,Pu,Ta).

   Here Ta must be the TTL to exactly reach R. If Ta is too small, the *Path* message will not reach R. If Ta 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. The host Hu must be explicitly configured with Ra and Ta.

2. A particular host on the LAN connected to Hu could be designated as an *RSVP relay host*. A relay host would listen on (G*,Pu) and forward any *Path* messages directly to R, although it would not be in the data path. The relay host would have to be configured with Ra and Ta.

D Experimental and Open Issues

D.1 Reservation Compatibility

How strong is the requirement for compatibility of reservations in different directions? For example, see Figure 10; 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, ...)

```
                        |
         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 13: Dynamic-Filter Reservation Example

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 13 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 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 13). 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**.

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.

[FJ94] Floyd, S. and V. Jacobson, *Synchronization of Periodic Routing Messages*, IEEE/ACM Transactions on Networking, Vol. 2, No. 2, April, 1994.

[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.

[Katz95] Katz, D., *IP Router Alert Option*, Internet Draft draft-katz-router-alert-01.txt, Cisco Systems, November 16, 1995.

[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 2.7.

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