Internet Draft
Expires September 24, 1995
File: draft-ietf-rsvp-spec-05.ps

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

Resource ReSerVation Protocol (RSVP) – Version 1 Functional Specification

Status of Memo

Abstract

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

What's Changed Since Toronto IETF

This version of the document incorporates many of the protocol changes agreed to at the December 1994 IETF meeting in San Jose. The most major changes are:

- The RSVP packet format has been reorganized to carry most data as typed variable-length objects.
- This generality includes provision for 16-byte IP6 addresses.
- Filter specs have been simplified.
- DF style has been moved to an Appendix, as experimental.
- UDP encapsulation has been included.
- OPWA has been included.
- The Drop flag has been eliminated.
- Session groups have been added.
- The routing of *ResvErr* messages has been changed.

1 Introduction

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

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

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

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

```
              HOST                                     ROUTER

  ______________________________     RSVP    ___________________________
 |                              |     .------------------.              |
 |  ________        _______     |    .       | _________  .   _______   |
 | |        |      |       |    |   .        ||         |  . |       || RSVP
 | |Applic- |      | RSVP <------            ||Routing  |  -> RSVP <------->
 | |  App   <------>daemon |    |            ||Protocol |    |daemon ||
 | |        |      |       |    |            || daemon <---->        ||
 | |________|      |___.___|    |            ||_ ._______|   |__.____||
 |===|=================v========|            |===v================v======|
 | data       ..........        |            |   .   ............        |
 |    |   ____v_    ____v_____  |            |  _v__v_      _____v____  |
 |    |  |Class-|  |          | |   data     | |Class-|   |           || data
 |    |=> ifier |=> Packet   =================> ifier |==> Packet     |======>
 |       |______|  |Scheduler | |            | |______|   |Scheduler  ||
 |                 |__________| |            |            |___________||
 |______________________________|            |___________________________|
```

Figure 1: RSVP in Hosts and Routers

Each router that is capable of resource reservation passes incoming data packets to a packet classifier and then queues them as necessary in a packet scheduler. The packet classifier determines the route and the QoS class for each packet. The scheduler allocates a particular outgoing link for packet transmission, and it may also allocate other system resources such as CPU time or buffers.

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

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

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

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

In summary, RSVP has the following attributes:

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

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

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

1.1 Data Flows

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

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

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

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

Figure 2: Multicast Distribution Session

1.2 Reservation Model

An elementary RSVP reservation request consists of a *flowspec* together with a *filter spec*; this pair is called a *flow descriptor*. The flowspec specifies a desired QoS. The filter spec (together with the DestAddress and the generalized destination port defining the session) defines the set of data packets – the "flow" – to receive the QoS defined by the flowspec. The flowspec is used to set parameters to the packet scheduler in the node (assuming that admission control succeeds), while the filter spec is used to set parameters in the packet classifier.

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

The general RSVP reservation model allows filter specs to 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, considerations of both architectural purity and practical requirements have led to the decision that RSVP should use separate sessions for distinct subflows of hierarchically-encoded signals. For multicast sessions, subflows can be distinguished by multicast destination address; for unicast sessions, they must be distinguished by destination port. As a result of these considerations, the present RSVP version includes a quite restricted definition of filter specs, selecting only on sender IP address and UDP/TCP port number, and on protocol id. However, the design of the protocol would easily handle a more general definition in future versions.

Any packets that are addressed to a particular session but do not match any of the filter specs for that session will be sent as best-effort traffic. Under congested conditions, such packets are likely to experience long delays and may be dropped. A receiver may wish to conserve network resources by explicitly asking the network to drop those data packets for which there is no reservation; however, such dropping should be performed by routing, not by RSVP. Determining where packets get delivered should be a routing function; RSVP is concerned only with the QoS of those packets that are delivered by routing.

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

1. 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 it fails this test, the reservation is rejected and RSVP sends back an error message towards the responsible receiver(s). If it passes, the flowspec is used to set up the packet scheduler for the desired QoS and the filter spec is used to set the packet classifier to select the appropriate data packets.

2. Forward reservation upstream.

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

The reservation request that a node forwards upstream may differ from the request that it received, for two reasons. First, it is possible (at least in theory) for the kernel to modify the flowspec hop-by-hop (although currently no realtime services do this). Second, reservations from different downstream branches of the multicast distribution tree(s) must be *merged* as reservations travel upstream. Merging reservations is a necessary consequence of multicast distribution, which creates a single stream of data packets in a particular router from any Si, regardless of the set of receivers downstream. The reservation for Si on a particular outgoing link L should be the “maximum” of the individual flowspecs from the receivers Rj that are downstream via link L. Merging is discussed further in Section 2.3.

For both of these primary actions, there are options controlled by the receiver making the reservation. These options are combined into a control variable called the reservation *style*, which is discussed in section 1.3. One option concerns the treatment of reservations for different senders within the same session: establish a *distinct* reservation for each upstream sender, or else *mix* all senders' packets into a single reservation. Another option controls the scope of the request: *unitary* (i.e., a single specified sender), an explicit sender list, or a *wildcard* that implicitly selects all senders upstream of the given node.

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

1.3 Reservation Styles

Each RSVP reservation request specifies a *reservation style*. The following reservation styles are defined in this version of the protocol.

1. Wildcard-Filter (WF) Style

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

2. Fixed-Filter (FF) Style

   The FF style specifies the options: *distinct* reservation and a *unitary* reservation scope. Thus, an elementary FF-style reservation request creates a distinct reservation for data packets from a particular sender, not mixing 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 necessarily be merged to share a single reservation in a given node.

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

The WF and FF styles are incompatible and cannot be combined within a session. Other reservation styles may be defined in the future (see Appendix C).

2 RSVP Protocol Mechanisms

2.1 RSVP Messages

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

Each receiver host sends RSVP reservation request (*Resv*) messages towards the senders. These reservation messages must follow in reverse the routes the data packets will use, all the way upstream to the senders within the scope. *Resv* messages are delivered to the sender hosts, so that the hosts can set up appropriate traffic control parameters for the first hop. If a reservation request fails at any node, an RSVP error message is returned to the receiver; however, RSVP sends no positive acknowledgment messages to indicate success.

```
      Sender                                            Receiver

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

Figure 3: RSVP Messages

Each sender transmits RSVP *Path* messages forward along the uni-/multicast routes provided by the routing protocol(s); see Figure 3. These *Path* messages store *path state* in each node. Path state is used by RSVP to route the *Resv* messages hop-by-hop in the reverse direction. (In the future, some routing protocols may supply reverse path forwarding information directly, without path state).

Path messages may also carry the following information:

- Sender Template

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

- Tspec

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

- Adspec

  The *Path* message may carry a package of OPWA advertising information, known as an *Adspec*.

```
 Previous       Incoming           Outgoing             Next
 Hops           Interfaces         Interfaces           Hops

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

Figure 4: Router Using RSVP

Figure 4 illustrates RSVP's model of a router node. Each data stream arrives from a *previous hop* through a corresponding *incoming interface* and departs through one or more *outgoing interface(s)*. The same physical interface may act in both the incoming and outgoing roles (for different data flows but the same session).

As illustrated in Figure 4, there may be multiple previous hops and/or next hops through a given physical interface. This may result from the connected network being a shared medium or from the existence of non-RSVP routers in the path to the next RSVP hop (see Section 2.6). An RSVP daemon must preserve the next and previous hop addresses in its reservation and path state, respectively. A *Resv* message is sent with a unicast destination address, the address of a previous hop. *Path* messages, on the other hand, are sent with the session destination address, unicast or multicast.

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

There are a number of ways for a new reservation request to fail in a given node.

1. There may be no matching path state (i.e., the scope may empty), which would prevent the reservation being propagated upstream.
2. Its style may be incompatible with the style(s) of existing reservations for the same session on the same outgoing interface, so an effective flowspec cannot be computed.
3. Its style may be incompatible with the style(s) of reservations that exist on other outgoing interfaces but will be merged with this reservation when a refresh message to create a refresh message for the previous hop.
4. The effective flowspec may fail admission control.

In any of these cases, an error message is returned to the receiver(s) responsible for the message, but 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.

2.2 Soft State

To maintain reservation state, RSVP keeps *soft state* in router and host nodes. RSVP soft state is created and periodically refreshed by *Path* and *Resv* messages. The state is deleted if no refreshes arrive before the expiration of a *cleanup timeout* interval; it may also be deleted as the result of an explicit *Teardown* message. It is not necessary (although it may be desirable, since the resources being consumed may be "valuable"), to explicitly tear down an old reservation.

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, while the state on the now-unused segment of the route will time out. Thus, whether a message is "new" or a "refresh" is determined separately at each node, depending upon the existence of state at that node. (This document uses the term *refresh message* in this effective sense, to indicate an RSVP message that does not modify the existing state at the node in question.)

In addition to the cleanup timeout, there is a *refresh timeout* period. As messages arrive, the RSVP daemon checks them against the existing state; if it matches, the cleanup timeout timer on the state is reset and the message is dropped. At the expiration of each refresh timeout period, RSVP scans its state to build and forward *Path* and *Resv* refresh messages to succeeding hops.

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

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

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

The RSVP state associated with a session in a particular node is divided into atomic elements that are created, refreshed, and timed out independently. The atomicity is determined by the requirement that any sender or receiver may enter or leave the session at any time, and its state should be created and timed out independently. Management of RSVP state is complex because there may not be a one-to-one correspondence between state carried in RSVP control messages and the resulting state in nodes. Due to merging, a single message contain state referring to multiple stored elements. Conversely, due to reservation sharing, a single stored state element may depend upon (typically, the maximum of) state values received in multiple control messages.

2.3 Merging and Packing

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

Although flowspecs are opaque to RSVP, an RSVP daemon must be able to calculate the "largest" of a set of flowspecs. This is required both to calculate the effective flowspec to install on a given physical interface (see the discussion in connection with Figure 4), and to merge flowspecs when sending a refresh message upstream. Since flowspecs are generally multi-dimensional vectors (they contain both Tspec and Rspec components, each of which may itself be multi-dimensional), they are not strictly ordered. When it cannot take the larger of two flowspecs, RSVP must compute and use a third flowspec that is at least as large as each, i.e., a *least upper bound* (LUB). It is also possible for two flowspecs to be incomparable, which is treated as an error. The definition and implementation of the rules for comparing flowspecs are outside RSVP proper, but they are defined as part of the service templates.

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

2.4 Teardown

RSVP *teardown* messages remove path and reservation state without waiting for the cleanup timeout period, as an optimization to release resources quickly. Although teardown messages (like other RSVP messages) are not delivered reliably, the state will time out even if it is not explicitly deleted.

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. A router may also initiate a teardown message as the result of router or link failures detected by the routing protocol. Once initiated, a teardown request should be forwarded hop-by-hop without delay.

To increase the reliability of teardown, Q copies of any given teardown message can be sent. Note that a node cannot actually delete the state being torn down until it has sent Q Teardown messages; it must place the state in a "moribund" status meanwhile. The appropriate value of Q is an engineering issue. $Q = 1$ would be the simplest and may be adequate, since unrefreshed state will time out anyway; teardown is an optimization. If one or more Teardown message hops are lost, the router that failed to receive a Teardown message will time out its state and initiate a new Teardown message beyond the loss point. Assuming that RSVP message loss probability is small, the longest time to delete state will seldom exceed one refresh timeout period.

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

A teardown message deletes the specified state in the node where it is received. Like any other state change, this will be propagated immediately to the next node, but only if it represents a change. As a result, an *ResvTear* message will prune the reservation state back (only) as far as possible. Note that the *ResvTear* message will cease to be forwarded at the same node where merging suppresses forwarding of the corresponding *Resv* messages. The change will be propagated as a new teardown message if the result has been to remove all state for this session at this node. However, the result may simply be to change the propagated information; thus, the receipt of a *ResvTear* message may result in the immediate forwarding of a modified *Resv* refresh message.

Deletion of path state, whether as the result of a teardown message or because of timeout, may force adjustments in order in related reservation state to maintain consistency in the local node. For example, when a *PathTear* deletes the path state for a sender S, the adjustment in reservation depends upon the style: if the style is WF and S is the only sender to the session, delete the reservation; if the style is FF, delete only reservations for sender S. These reservation changes should not trigger an immediate *Resv* refresh message, since the teardown message will have already made the required changes upstream. However, at the node in which an *ResvTear* message stops, the change of reservation state may trigger a *Resv* refresh starting at that node.

2.5 Security

There are two distinct types of security concerns in RSVP.

1. Protecting RSVP Message Integrity

   It may be necessary to ensure the integrity of RSVP messages against corruption or spoofing, hop by hop. RSVP messages have an optional integrity field that can be created and verified by neighboring RSVP nodes.

2. Authenticating Reservation Requests

   RSVP-mediated resource reservations may reserve network resources, providing special treatment for a particular set of users. Administrative mechanisms will be necessary to control who gets privileged service and to collect billing information. These mechanisms may require secure authentication of senders and/or receivers responsible for the reservation. RSVP messages may contain credential information to verify user identity.

The RSVP packet formats provide for both; see Section 4.

2.6 Automatic RSVP Tunneling

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

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

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

2.7 Session Groups

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.

2.8 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. In order to join an RSVP session, the following events happen at the end systems.

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

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

H3 A receiver listens for *Path* messages.

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

H5 A sender starts sending data packets.

There are several synchronization considerations.

- Suppose that a new sender starts sending data (H5) but no receivers have joined the group (H1). Then there will be no multicast routes beyond the host (or beyond the first RSVP-capable router) along the path; the data will be dropped at the first hop until receivers(s) do appear (assuming a multicast routing protocol that "prunes off" or otherwise avoids unnecessary paths).

- Suppose that a new sender starts sending *Path* messages (H2) and immediately starts sending data (H5), 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.
- If a receiver starts sending *Resv* messages (H4) before any *Path* messages have reached it (H5) (and if path state is being used to route *Resv* messages), RSVP will return error messages to the receiver. The receiver may simply choose to ignore such error messages, or it may avoid them by waiting for *Path* messages before sending *Resv* messages.

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

3 Examples

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

1. Wildcard-Filter

   WF(*{Q})

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

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

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

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

Figure 5 shows schematically a router with two previous hops labeled (a) and (b) and two outgoing interfaces labeled (c) and (d). This topology will be assumed in the examples that follow. There are three upstream senders; packets from sender S1 (S2 and S3) arrive through previous hop (a) ((b), respectively). There are also three downstream receivers; packets bound for R1 and R2 (R3) are routed via outgoing interface (c) ((d) respectively).

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

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

Figure 5: Router Configuration

In Figure 6, the "Receive" column shows the *Resv* messages received over outgoing interfaces (c) and () and the "Reserve" column shows the resulting reservation state for each interface. The "Send" column shows the *Resv* messages forwarded to previous hops (a) and (b). In the "Reserve" column, each box represents one reservation "channel", with the corresponding filter. As a result of merging, only the largest flowspec is forwarded upstream to each previous hop.

Figure 7 shows Fixed-Filter style reservations. The flow descriptors for senders S2 and S3, received from outgoing interfaces (c) and (d), are packed into the message forwarded to previous hop b. On the other hand, the two different flow descriptors for sender S1 are merged into the single message FF(S1{3B}), which is sent to previous hop (a), For each outgoing interface, there is a private reservation for each source that has been requested, but this private reservation is shared among the receivers that made the request.

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

Figure 6: Wildcard-Filter Reservation Example 1

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

Figure 7: Fixed-Filter Reservation Example

The two examples just shown assume full routing, i.e., data packets from S1, S2, and S3 are routed to both outgoing interfaces. Assume the routing shown in Figure 8, in which data packets from S1 are not forwarded to interface (d) (because the mesh topology provides a shorter path for S1 $->$ R3 that does not traverse this node).

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

Figure 8: Router Configuration

Under this assumption, Figure 9 shows Wildcard-Filter reservations. Since there is no route from (a) to (d), the reservation forwarded out interface (a) considers only the reservation on interface (c), so no merging takes place in this case.

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

Figure 9: Wildcard-Filter Reservation Example – Partial Routing

4 RSVP Functional Specification

4.1 RSVP Message Formats

All RSVP messages consist of a common header followed by a variable number of variable-length typed *objects* using a common structure. The subsections that follow define the formats of the common header, the object structures, and each of the RSVP message types. For each RSVP message type, there is a set of rules for the permissible ordering and choice of object types. These rules are specified using Backus-Naur Form (BNF) augmented with square brackets surrounding optional sub-sequences.

4.1.1 Common Header

```
      0             1              2             3
+-------------+-------------+-------------+-------------+
| Vers | Type |    Flags    |       Message Length      |
+-------------+-------------+-------------+-------------+
|       RSVP Checksum       |        Object Count       |
+-------------+-------------+-------------+-------------+
```

The common header fields are as follows:

Vers

Protocol version number. This is version 2.

Type

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

Flags

0x01 = Entry-Police

This flag should be on in a *Path* message sent by an RSVP daemon in a sender host. The first RSVP node that finds the flag on in a *Path* message (i.e., the first-[RSVP-]hop router) should institute policing for the flow(s) described in this message. This flag should never be forwarded in *Path* refresh messages.

0x02 = LUB-Used

This flag is described below in the section on Error Messages.

Message Length

The total length of this RSVP message, including this common header and the objects included in Object Count.

RSVP Checksum

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

Object Count

Count of variable-length objects that follow.

4.1.2 Object Formats

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

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

An object header has the following fields:

Length

Total length in bytes. Must always be a multiple of 4, and at least 4.

Class

Object class. In this document, the names of object classes are capitalized.

0 = **NULL**

A NULL object has a Class of zero; 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.

1 = **SESSION**

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

2 = **SESSION_GROUP**

When present, defines a session group, a set of related sessions whose reservation requests should be passed collectively to Admission Control.

3 = **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.

4 = **STYLE**

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

5 = **FLOWSPEC**

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

6 = **FILTER_SPEC**

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

7 = **SENDER_TEMPLATE**

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

8 = **SENDER_TSPEC**

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

9 = **ADVERT**

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

10 = **TIME_VALUES**

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

11 = **ERROR_SPEC**

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

12 = **CREDENTIAL**

Contains user credential and/or information for policy control and/or accounting.

13 = **INTEGRITY**

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

C-Type

Object type; unique within Class. Values defined in Appendix A.

The Class and C-Type fields may be used together as a 16-bit number to define a unique type for each object.

The formats of specific object types are defined in Appendix A.

4.1.3 Path Message

Path messages carry information from senders to receivers along the same paths, and using the same uni-/multicast routes, as the data packets. The IP destination address of a *Path* message is the DestAddress for the session, and the source address is an address of the node that sent the message (if possible, the address of the particular interface through which it was sent).

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

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

                        [ <INTEGRITY> ]  [ <TIME_VALUES> ]

                        <sender descriptor list>

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

                  <sender descriptor list> <sender descriptor>

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

                        [ <SENDER_TSPEC> ]  [ <ADVERT> ]
```

Each **sender descriptor** defines a sender, and the **sender descriptor list** allows multiple **sender descriptor**s to be packed into a *Path* message. For each sender in the list, the SENDER_TEMPLATE object defines the format of data packets, the SENDER_TSPEC object may specify the traffic flow, and the CREDENTIAL object may specify the user credentials. There may also be an ADVERT object carrying advertising (OPWA) data.

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

At each node, a route must be computed independently for each **sender descriptor**s being forwarded. These routes, obtained from the uni/multicast routing table, generally depend upon the (sender host address, DestAddress) pairs, and consist of a list of outgoing interfaces. Then the descriptors being forwarded through the same outgoing interface can be packed into as few *Path* messages as possible. Note that multicast routing of path information is based on the sender address(es) from the **sender descriptor**s, not the IP source address; this is necessary to prevent routing loops; see Section 4.3. PHOP (i.e., the RSVP_HOP object) of each *Path* message should contain the IP source address, the interface address through which the message is sent.

Path messages are processed at each node they reach to create path state, which includes SENDER_TEMPLATE object and possibly CREDENTIAL, SENDER_TSPEC, and ADVERT objects. If an error is encountered while processing a *Path* message, a *PathErr* message is sent to all senders implied by the SENDER_TEMPLATEs in the **sender descriptor list**.

4.1.4 Resv Messages

Resv messages carry reservation requests hop-by-hop from receivers to senders, along the reverse paths of data flow for the session. The IP destination address of a *Resv* message is the unicast address of a previous-hop node, obtained from the path state. The Next Hop address (in the RSVP_HOP object) should be the IP address of the (incoming) interface through which the *Resv* message is sent. The IP source address is an address of the node that sent the message (if possible, the address of the particular interface through which it was sent).

The permissible sequence of objects in a *Resv* message depends upon the reservation style specified in the STYLE object. Currently, object types Style-WF and Style-FF of class STYLE are defined (see Appendix A).

The *Resv* message format is as follows:

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

                        [ <SESSION_GROUP> ]  <RSVP_HOP>

                        [ <INTEGRITY> ] [ <TIME_VALUES> ]

                        [ <CREDENTIAL> ]

                        <style-specific tail>

<style-specific-tail> ::=

            <Style-WF> [ <FILTER_SPEC> ]  <FLOWSPEC> |

            <Style-FF> <flow descriptor list>

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

            <flow descriptor list> <FILTER_SPEC> <FLOWSPEC>
```

The reservation scope, i.e., the set of senders towards which a particular reservation is to be forwarded, is determined by matching FILTER_SPEC objects against the path state created from SENDER_TEMPLATE objects, considering any wildcards that may be present.

4.1.5 Error Messages

There are two types of RSVP error messages:

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

RSVP error messages are triggered only by processing of *Path* and *Resv* messages; errors encountered while processing error or teardown messages must not create error messages.

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

                          [ <INTEGRITY> ]  <ERROR_SPEC>

                          <sender descriptor>

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


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

                          [ <INTEGRITY> ]  <ERROR_SPEC>

                          [ <CREDENTIAL> ] <style-specific tail>

<style-specific tail> ::= (see earlier definition)
```

The ERROR_SPEC specifies the error. It includes the IP address of the node that detected the error, called the Error Node Address.

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

An error message may be duplicated and forwarded unchanged. In general, error messages should be delivered to the applications on all the session nodes that (may have) contributed to this error.

- A *PathErr* message is forwarded to all previous hops for all senders listed in the **Sender Descriptor List**.
- The node that creates a *ResvErr* message as the result of processing a *Resv* message should send the *ResvErr* message out the interface through which the *Resv* arrived.

  In succeeding hops, the routing of a *ResvErr* message depends upon its style and upon routing. In general, a *ResvErr* message is sent out some subset of the outgoing interfaces specified for multicast routing, using Error Node Address as the source address and DestAddress as the destination. (This rule is necessary to prevent packet loops; see Section 4.3 below). Within this set of outgoing interfaces, a *ResvErr* message is sent only to next hop(s) whose *Resv* message(s) created the error; this in turn depends upon the merging of flowspecs. Assume that a reservation whose error is being reported was formed by merging two flowspecs Q1 and Q2 from different next hops.

  - If Q1 = Q2, the error message should be forwarded to both next hops.
  - If Q1 < Q2, the error message should be forwarded only to the next hop for Q2.
  - If Q1 and Q2 are incomparable, the error message should be forwarded to both next hops, and the LUB flag should be turned on.

  The ERROR_SPEC and the LUB-flag should be delivered to the receiver application. In the case of an Admission Control error, the **style-specific tail** will contain the FLOWSPEC object that failed. If the LUB-flag is off, this should be the same as a FLOWSPEC in a *Resv* message sent by this application; otherwise, they may differ.

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

4.1.6 Teardown Messages

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

- *PathTear* messages delete path state (which in turn may delete reservations state) and travel towards all receivers that are downstream from the point of initiation. *PathTear* messages are routed like *Path* messages, and their IP destination address is DestAddress for the session.

- *ResvTear* messages delete reservation state and travel towards all matching senders upstream from the point of teardown initiation. *ResvTear* message are routed like *Resv* messages, and their IP destination address is the unicast address of a previous hop.

```
<PathTear Message> ::= <Common Header> <SESSION> <RSVP HOP>

                            [ <INTEGRITY> ]

                            <sender descriptor list>

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

<ResvTear Message> ::= <Common Header> <SESSION> <RSVP HOP>

                            [ <INTEGRITY> ]  [ <CREDENTIAL> ]

                            <style-specific tail>

<style-specific tail> ::= (see earlier definition)
```

Flowspec objects in the **style-specific tail** of a *ResvTear* message will be ignored and may be omitted.

If the state being deleted was created with user credentials from a CREDENTIAL field, then the matching *PathTear* or *ResvTear* message must include matching CREDENTIAL field(s).

[There is a problem here: tearing down path state may implicitly delete reservation state. But a *PathTear* message does not have credentials for the reservation state, only for the path state. Some argue that a CREDENTIAL may not be needed in teardown messages, on the assumption that false teardown messages can be injected only with the collusion of routers along the data path, and in that case, the colluding router can just as well stop delivering the *Resv* messages, which will have the same effect.]

4.2 Sending RSVP Messages

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

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

An RSVP *Path* or *Resv* message consists of a small root segment followed by a variable-length list of objects, which may overflow the capacity of one datagram. IP fragmentation is inadvisable, since it has bad error characteristics; RSVP-level fragmentation should be used. That is, a message with a long list of descriptors will be divided into segments that will fit into individual datagrams, each carrying the same root fields. Each of these messages will be processed at the receiving node, with a cumulative effect on the local state. No explicit reassembly is needed.

Since RSVP messages are normally expected to be generated and sent hop-by-hop, their MTU should be determined by the MTU of each interface.

[There may be rare instances in which this does not work very well, and in which manual configuration would not help. The problem case is an interface connected to a non-RSVP cloud in which some particular link far away has a smaller MTU. This would affect only those sessions that happened to use that link. Proper solution to this case would require MTU discovery separately for each interface and each session, which is a very large amount of machinery and some overhead for a rare (?) case. Best approach seems to be to rely on IP fragmentation and reassembly for this case.]

4.3 Avoiding RSVP Message Loops

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

In addition, error and teardown messages are forwarded immediately and are therefore subject to direct looping.

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

Since *Path* messages don't loop, they create path state defining a loop-free reverse path to each sender. As a result, *Resv* and *ResvTear* messages directed to particular senders cannot

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

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

Figure 10: Avoiding Auto-Refresh in Non-Looping Topology

loop. *PathErr* messages are always directed to particular senders and therefore cannot loop. However, there is a potential auto-refresh problem for *Resv*, *ResvTear*, and *ResvErr* messages with wildcard scope, as we now discuss.

If the topology has no loops, then auto-refresh can be avoided, even for wildcard scope, with the following rule:

> A reservation request received from next hop N must not be forwarded to N.

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 are both outgoing and incoming interfaces for this session. Both receivers are making wildcard-scope reservations, in which the *Resv* messages are forwarded to all previous hops for senders in the group, with the exception of the next hop from which they came. These result in independent reservation requests in the two directions, without an auto-refresh loop.

However, further effort is needed to prevent auto-refresh loops from wildcard-scope reservations in the presence of cycles in the topology. [TBD!!].

We treat routing of *ResvErr* messages as a special case. They are sent with unicast addresses of next hops, but the multicast routing is used to prevent loops. As explained above, *ResvErr* messages are forwarded to a subset of the multicast tree to DestAddress, rooted at the node on which the error was discovered. Since multicast routing cannot create loops, this will prevent loops for *ResvErr* messages.

[Open question about Figure 10: should it be possible to have incompatible reservation styles on the two interfaces? For example, 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.]

4.4 Local Repair

Each RSVP daemon periodically sends refreshes to its next/previous hops. An important optimization would allow the local routing protocol module to notify the RSVP daemon of route changes for particular destinations. The RSVP daemon should use this information to trigger an immediate refresh of state for these destinations, using the new route. This allows fast adaptation to routing changes without the overhead of a short refresh period.

4.5 Time Parameters

For each element of state, there are two time parameters: the refresh period R and the time-to-live value T. R specifies the period between sending successive refreshes of this data. T controls how long state will be retained after refreshes stop appearing, and depends upon period between receiving successive refreshes. Specifically, R <= T, and the *cleanout time* is K * T. Here K is a small integer; K-1 successive messages may be lost before state is deleted. Currently K = 3 is suggested.

Clearly, a smaller T means increased RSVP overhead. If the router does not implement local repair, a smaller T improves the speed of adapting to routing changes. With local repair, a router can be more relaxed about T, since the periodic refresh becomes only a backstop robustness mechanism.

There are three possible ways for a router to determine R and T.

- Default values are configured in the router. Current defaults are 30 seconds for T and R.
- A router may adjust the value of T dynamically to keep a constant total overhead due to refresh traffic; as more sessions appear, the period would be lengthened. In this case, R = T could be used.
- R and T can be specified by the end systems. For this purpose, *Path* and *Resv* messages may contain the optional TIM_VALUES object. When messages are merged and forwarded to the next hop, R should be the minimum R that has been received, and T should be the maximum T that has been received. Thus, the largest T determines how long state is retained, and the smallest R determines the responsiveness of RSVP

to route changes. In the first hop, they are expected to be equal. The RSVP API might allow an application to override the default value for a particular session.

4.6 RSVP Interfaces

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

4.6.1 Application/RSVP Interface

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

- Register

```
Call: REGISTER( DestAddress , DestPort

                [ , SESSION_object ]  , SND_flag , RCV_flag

                [ , Source_Address ]  [ , Source_Port ]

                [ , Sender_Template ]  [ , Sender_Tspec ]

                [ , Data_TTL ]  [ , UserCredential ]

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

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

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

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

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

- Source_Address

  This is the address of the interface from which the data will be sent. If it is omitted, a default interface will be used.

- 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 FILTERSPEC.

- Sender_Template

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

- Sender_Tspec

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

- Data_TTL

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

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

- Reserve

```
Call: RESERVE( session-id, style, style-dependent-parms )
```

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

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

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

- Release

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

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

- Error/Event Upcalls

```
Call: <Upcall_Proc> (session-id, Info_type, List_count

               [ ,Error_code ,Error_value ,LUB-flag ]

               [ ,Filter_spec_list ]  [ ,Flowspec_list ]

               [ ,Advert_list ] )
```

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

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

1. Info_type = Path Event

   A Path Event upcall indicates the receipt of a *Path* message, indicating to the 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' is the number in each list; where these objects are missing, corresponding null objects must appear.

   Error_code and Error_value, and LUB-flag should be ignored in a Path Event upcall.

2. Info_type = Path Error

   An Path Error event indicates an error in processing a **sender descriptor** originated by this sender. The Error_code parameter will define the error, and Error_value may supply some additional (perhaps system-specific) data about the error. 'List_count' will be 1, and Filter_spec_list and Flowspec_list will contain the Sender_Template and the Sender_Tspec supplied in the REGISTER call; Advert_list will contain one NULL object.

3. Info_type = Resv Error

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

   'List_count' will be 1, and Filter_spec_list and Flowspec_list will contain one FILTER_SPEC and one FLOWSPEC object. These objects are taken from the *Resv* message that caused the error (unless the LUB-flag is on, in which case FLOWSPEC may differ).

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

4.6.2 RSVP/Traffic Control Interface

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

1. Make a Reservation

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

                              [ , Sender_Tspec]

                              [ , SD_rank , SD_end_flag ] )
   ```

   This call passes a Flowspec defining a desired QoS to admission control. It may also pass Sender_Tspec, the maximum traffic characteristics computed over the SENDER_TSPECs of senders that will contribute data packets to this reservation. Police_Flag is a Boolean parameter that indicates whether traffic policing should be applied at this point.

   The SD_rank and SD_end_flag fields are used for a member of a session group. SD_rank is the rank value from the SESSION_GROUP object. The call is made with each of the sessions in the group, and SD_end_flag is set true for the last one.

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

2. Add Filter

   ```
   Call: TC_AddFilter( Rhandle, Session, Filterspec )
   ```

   This call is used to define a filter corresponding to the given handle, following a successful TC_AddFlowspec call.

3. Modify or Delete Filter

   ```
   Call: TC_ModFilter( Rhandle, Session,

                                        [ new_Filterspec] )
   ```

   This call can modify an existing filter or replace an existing filter with no filter (i.e., delete the filter).

4. Modify or Delete Flowspec

```
Call: TC_ModFlowspec( Rhandle

                [, new_Flowspec [ ,Sender_Tspec]] )
```

This call can modify an existing reservation or delete the reservation. If new_Flowspec is included, it is passed to Admission Control; if it is rejected, the current flowspec is left in force. If new_Flowspec is omitted, the reservation is deleted and Rhandle is invalidated.

5. OPWA Update

```
Call: TC_Advertise( interface, Adspec

                [ ,Sender_TSpec ] ) -> New_Adspec
```

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

6. Initialize Traffic Control

```
Call: TC_Initialize(interface )
```

This call is used when RSVP initializes its state, to clear out all existing classifier and/or packet scheduler state for a specified interface.

4.6.3 RSVP/Routing Interface

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

- Promiscuous receive mode for RSVP messages

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

- Route discovery

  RSVP must be able to discover the route(s) that the routing algorithm would have used for forwarding a specific datagram.

```
GetUcastRoute( DestAddress ) -> OutInterface

GetMcastRoute( SrcAddress, DestAddress )

                            -> OutInterface_list
```

- Route Change Notification

  Routing may provide an asynchronous notification to RSVP that a specified route has changed.

  ```
  New_Ucast_Route( DestAddress ) -> new_OutInterface

  New_Mcast_Route( SrcAddress, DestAddress )

                                 -> new_OutInterface_list
  ```

- Outgoing Link Specification

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

- Discover Interface List

  RSVP must be able to learn what real and virtual interfaces exist.

5 Message Processing Rules

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

- PSB – Path State Block

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

- RSB – Reservation State Block

  RSB's are used to hold reservation state. Each RSB holds reservation state for the 4-tuple: (session, next hop, style, filterspec), defined in SESSION, NHOP (i.e., RSVP_HOP), STYLE, and FILTER_SPEC objects, respectively. We assume that RSB contents include the outgoing interface OI that is implied by NHOP. RSB contents also include a FLOWSPEC object and may include a CERTIFICATE object.

MESSAGE ARRIVES

Verify version number, checksum, and length fields of common header, and discard message if it fails.

Further processing depends upon message type.

PATH MESSAGE ARRIVES

Start with the Refresh_Needed flag off.

Each **sender descriptor** object sequence in the message defines a sender. Process each sender as follows.

1. If there is a CREDENTIAL object, verify it; if it is unacceptable, build and send a *PathErr* message, drop the *Path* message, and return.
2. If there is no path state block (PSB) for the (session, sender) pair then:
   - Create a new PSB.
   - Set a cleanup timer for the PSB. If this is the first PSB for the session, set a refresh timer for the session.
   - Copy PHOP into the PSB. Copy into the PSB any of the following objects that are present in the message: CREDENTIAL, SENDER_TSPEC, and/or ADVERT. Copy the EntryPolice flag from the common header into the PSB.
   - Call the appropriate route discovery routine, using DestAddress from SESSION and (for multicast routing) SrcAddress from SENDER_TEMPLATE. Store the resulting routing bit mask ROUTE_MASK in the PSB.

3. Otherwise (there is a matching PSB):
   - If CREDENTIAL differs between message and PSB, verify new CREDENTIAL. If it is acceptable, copy it into PSB. Otherwise, build and send a *PathErr* message for "Bad Credential", drop the *Path* message, and return.
   - Restart cleanup timer.
   - Update the PSB with values from the message, as follows. Copy the ADVERT object, if any, into the PSB. Copy the EntryPolice flag into the PSB. If the values of PHOP or SEND_TSPEC differ between the message and the PSB, copy the new values into the PSB and turn on the Refresh_Needed flag. If SEND_TSPEC has changed, reservations matching S may also change; this may be deferred until a *Resv* refresh arrives.
   - Call the appropriate route discovery routine and compare the route mask with the ROUTE_MASK value already in the PSB; if a new bit (interface) has been added, turn on the Refresh_Needed flag. Store new ROUTE_MASK in the PSB.
4. If the Refresh_Needed flag is now set, execute the PATH REFRESH event sequence (below).

PATH TEAR MESSAGE ARRIVES

- If there is no path state for this destination, drop the message and return.
- Forward a copy of the *PathTear* message using the same rules as for a *Path* message (see PATH REFRESH).
- Each **sender descriptor** in the *PathTear* message contains a SENDER_TEMPLATE object defines a sender S; process it as follows.
  1. Locate the PSB for the pair: (session, S). If none exists, continue with next **sender descriptor**.
  2. Examine the RSB's for this session and delete any reservation state associated with sender S, depending upon the reservation style. For example:
     Delete a WF reservation for which S is the only sender.
     Delete an FF reservation for S.
  3. Delete the PSB.

PATH ERROR MESSAGE ARRIVES

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

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

Note that CREDENTIAL, SENDER_TSPEC, and ADVERT objects in the message is ignored.

Otherwise (PHOP is not local API), forward a copy of the *PathErr* message to the PHOP node.

RESV MESSAGE ARRIVES

A *Resv* message arrives through outgoing interface OI.

- Check the SESSION object.

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

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

  Verify the CREDENTIAL field (if any) to check permission to create a reservation. [This check may also involve the CREDENTIAL fields from the PSB's in the scope of this reservation; in that case, it would better fit below in processing the individual flow descriptors.]
- Check for path state

  If there are no PSB's matching the scope of this reservation, build and send a *ResvErr* message specifying "No Sender Information", drop the *Resv* message, and return.
- Make reservations

  Process the **style-specific tail** sequence.

  For FF style, execute the following steps for each flow descriptor, i.e., each (FLOWSPEC, FILTERSPEC) pair. For WF style execute the following once, using some internal placeholder "WILD_FILTER" for FILTERSPEC to indicate wildcard scope.

  1. Find or create a reservation state block (RSB) for the 4-tuple: (SESSION, NHOP, style, FILTERSPEC).
  2. Start or restart the cleanout timer on the RSB.
  3. Start a refresh timer for this session if none was started.
  4. If the RSB existed and if FLOWSPEC and the SENDER_TSPEC objects are unchanged, drop the *Resv* message and return.
  5. Compute Sender_Tspec as the maximum over the SENDER_TSPEC objects of the PSB's within the scope of the reservation.
  6. Set Police_flag on if any PSB's in the scope have the EntryPolice flag on, or if the style is WF and there is more than one PSB in the scope, otherwise off.

7. Computer K_Flowspec, the effective kernel flowspec, as the maximum of the FLOWSPEC values in all RSB's for the same (SESSION, OI, FILTERSPEC) triple. Similarly, the kernel filter spec K_filter is either the FILTER_SPEC object under consideration (unitary scope), or it is WILD_FILTER (wildcard scope).

   If there was no previous kernel reservation in place for (SESSION, OI, FILTERSPEC), call the kernel interface module:

   `TC_AddFlowspec( Sender_Tspec, K_flowspec, Police_Flag )`

   If this call fails, build and send a *ResvErr* message specifying "Admission control failed", drop the *Resv* message, and return. Otherwise, record the kernel handle K_handle returned by the call in the RSB(s). Then call:

   `TC_AddFilter( K_handle, K_Filter)`

   to set the filter, drop the *Resv* message and return.

   /item However, if there was a previous kernel reservation with handle K_handle, call the kernel interface module:

   `TC_ModFlowspec( K_handle, K_Flowspec, Sender_Tspec)`

   If this call fails, build and send a *ResvErr* message specifying "Admission control failed". In any case, drop the *Resv* message and return.

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.

created

RESV TEAR MESSAGE ARRIVES

A *ResvTear* message arrives on outgoing interface OI.

- If there are no existing PSB's for SESSION then drop the *ResvTear* message and return.
- Process the **style-specific tail** sequence to tear down reservations.

  For FF style, execute the following steps for each flow descriptor, i.e., each (FLOWSPEC, FILTERSPEC) pair. For WF style execute the following once, using some internal placeholder "WILD_FILTER" for FILTERSPEC to indicate wildcard scope.

  1. Find matching RSB(s) for the 4-tuple: (SESSION, NHOP, style, FILTERSPEC). If no RSB is found, continue with next flow descriptor, if any.
  2. Delete the RSB(s).
  3. If there are no more RSBs for the same (SESSION, OI, FILTERSPEC/) triple, call the kernel interface module:

     `TC_ModFlowspec( K_handle )`

     to delete the reservation. Then build and forward a new *ResvErr* message.
     - WF style: send a copy to each PHOP among all matching senders.
     - FF style: Send to PHOP of matching PSB.

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

   `TC_ModFlowspec( K_handle, K_Flowspec, Sender_Tspec)`

   to update the reservation, and then execute the RESV REFRESH sequence (below). If this kernel call fails, return; the prior reservation will remain in place.

RESV ERROR MESSAGE ARRIVES

- Call the appropriate route discovery routine, using DestAddress from SESSION and (for multicast routing) SrcAddress from the Error Node field in the ERRORS object. Let the resulting routing bit mask be M.
- Determine the set of RSBs matching the triple: (SESSION, style, FILTER-SPEC). If no RSB is found, drop *ResvErr* message and return.

  Recompute the maximum over the FLOWSPEC objects of this set of RSB's. If the LUB was used in this computation, turn on the LUB-flag in the received *Resv* message.
- Delete from the set of RSVs any whose OI does not appear in the bit mask M and whose NHOP is not the local API. If none remain, drop *ResvErr* message and return.

  For each PSB in the resulting set, do the following step.
- If NHOP in PSB is local API, deliver error to application via an upcall:

  ```
  Call: <Upcall_Proc>( session-id, Resv Error, 1,
                       Error_code, Error_value, LUB-flag,
                       FILTER_SPEC, FLOWSPEC, NULL)
  ```

  Here LUB-flag is taken from the received packet, as possibly modified above.

  Otherwise (NHOP is not local API), forward a copy of the *ResvErr* message to the PHOP node.

PATH REFRESH

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

For each virtual outgoing interface ("vif") V, build a *Path* message and send it to V. To build the message, consider each PSB whose ROUTE_MASK includes V, and do the following:

- Pass the ADVERT and SENDER_TSPEC objects present in the PSB to the kernel call TC_Advertise, and get back a modified ADVERT object. Pack this modified object into the *Path* message being built.

- Create a **sender descriptor** sequence containing the SENDER_TEMPLATE, CREDENTIAL, and SENDER_TSPEC objects, if present in the PSB. Pack the **sender descriptor** into the *Path* message being built.
- If the PSB has the EntryPolice flag on and if interface V is not capable of policing, turn the EntryPolice flag on in the *Path* message being built.
- If the maximum size of the *Path* message is reached, send the packet out interface V and start packing a new one.

RESV REFRESH

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

Each PSB for this session is considered in turn, to compute a **style-dependent tail** sequence. These sequences for a given PHOP are then packed into the same message(s) and sent to that PHOP. The logic is somewhat different depending upon whether the scope of the reservations is wildcard or not (they may not be mixed).

For each PSB that does not correspond to the API, do the following.

- Compute (FLOWSPEC, FILTER_SPEC) Pair

  Select each RSB in whose reservation scope the PSB falls, and compute the maximum over the FLOWSPEC objects of this set of RSB's. Also, select an appropriate FILTER_SPEC. The scope depends upon the style and the filter spec of the RSB:

  1. WF: Select every RSB whose OI matches a bit in the ROUTE_MASK of the PSB.

     In this case, FILTER_SPEC is the standard WILD_FILTER.
  2. FF: Select every RSB whose FILTER_SPEC matches SENDER_TEMPLATE in the RSB. This matching process should consider wildcards.

     In this case, FILTER_SPEC is taken from any of the matching RSB's. [?? Need to either 'merge' filter specs, which probably means to remove gratuitous wildcards??]

  This computation also yields a style (since style must be consistent across RSB's for given session). [??Again, need merging rules]]

- Build *Resv* packets

  Append this (FLOWSPEC, FILTER_SPEC pair) to the *Resv* message being built for destination PHOP (from the PSB). When the packet fills, or upon completion of all PSB's with the same PHOP, set the NHOP address in the message to the interface address and send the packet out that interface to the PHOP address.

A Object Type Definitions

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

A.1 SESSION Class

Currently, SESSION objects contain the pair: (DestAddress, DestPort), where DestAddress is the data destination address and DestPort is the UDP/TCP destination port. Other SESSION C-Types could be defined in the future to support other demultiplexing conventions in the transport-layer or application layer.

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

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

- IP6/UDP SESSION object: Class = 1, C-Type = 129

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

A.2 SESSION_GROUP Class

- 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 = 129:

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

A.3 RSVP_HOP Class

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

```
+-------------+-------------+-------------+-------------+
|             IPv4 Next/Previous Hop Address            |
+-------------+-------------+-------------+-------------+
```

- IP6 RSVP_HOP object: Class = 3, C-Type = 129

```
+-------------+-------------+-------------+-------------+
|                                                       |
+                                                       +
|                                                       |
+             IP6 Next/Previous Hop Address             +
|                                                       |
+                                                       +
|                                                       |
+-------------+-------------+-------------+-------------+
```

This object provides the IP address of the interface through which the last RSVP-knowledgeable hop forwarded this message.

A.4 STYLE Class

- STYLE-WF object: Class = 4, C-Type = 1

```
+-------------+-------------+-------------+-------------+
|   Style=1   |   ////////  |   ////////  | /////////   |
+-------------+-------------+-------------+-------------+
```

- STYLE-FF object: Class = 4, C-Type = 2

```
+-------------+-------------+-------------+-------------+
|   Style=2   |   ////////  |   ////////  | FD Count    |
+-------------+-------------+-------------+-------------+
```

FD Count

The count of elements in the variable-length object list that follows. See the *Resv* message format definition earlier.

A.5 Flowspec Class

- CSZ FLOWSPEC object: Class = 5, C-Type = 1

```
+-----------+----------+----------+-----------+
|               QoS Service Code              |
+-----------+----------+----------+-----------+
|        b: Token Bucket Depth (bits)         |
+-----------+----------+----------+-----------+
|        r: Average data rate (bits/sec)      |
+-----------+----------+----------+-----------+
|        d: Max end-to-end delay (ms)         |
+-----------+----------+----------+-----------+
|             (For Future Use)                |
+-----------+----------+----------+-----------+
```

QoS Service Code

Integer value defining what service is being requested. The values currently defined for this code are:

1 = Guaranteed Service
The Tspec is (b, r), while the Rspec is (r). (d) is ignored.

2 = Bounded-Delay Predictive Service
The Tspec is (b, r), while the Rspec is (d).

A.6 FILTER_SPEC Class

- IPv4/UDP FILTER_SPEC object: Class = 6, C-Type = 1

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

- IP6/UDP FILTER_SPEC object: Class = 6, C-Type = 129

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

SrcAddress is an IP address for a host, and SrcPort is a UDP/TCP source port, defining a sender.

A.7 SENDER_TEMPLATE Class

- IPv4/UDP SENDER_TEMPLATE object: Class = 7, C-Type = 1

  Definition same as IPv4/UDP FILTER_SPEC object.
- IP6/UDP SENDER_TEMPLATE object: Class = 7, C-Type = 129

  Definition same as IP6/UDP FILTER_SPEC object.

A.8 SENDER_TSPEC Class

The most common form of Tspec is a token bucket.

- Token Bucket SENDER_TSPEC object: Class = 8, C-Type = 1

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

A.9 ADVERT Class

[TBD]

A.10 TIME_VALUES Class

- TIME_VALUES Object: Class = 10, C-Type = 1

```
+-------------+-------------+-------------+-------------+
|                    Refresh Period                     |
+-------------+-------------+-------------+-------------+
|                    State TTL Time                     |
+-------------+-------------+-------------+-------------+
```

A.11 ERROR_SPEC Class

- IPv4 ERROR_SPEC object: Class = 11, C-Type = 1

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

- IP6 ERROR_SPEC object: Class = 11, C-Type = 129

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

Errnor Node

The IP address

Error Code

A one-octet error description.

01 = Insufficient memory
02 = Count Wrong
The FD Count field does not match length of message.
03 = No path information for this Resv
04 = No Sender information for this Resv
There is path information, but it does not include the sender specified in any of the Filterspecs listed in the Resv messager.
05 = Incorrect Dynamic Reservation Count
Dynamic Reservation Count is zero or less than FD Count.
06 = Filterspec error
07 = Flowspec syntax error
08 = Flowspec value error

Internal inconsistency of values.
[What should be done with Flowspec Feature Not Supported?]

09 = Resources unavailable [Sub-reasons? Depend upon traffic control and admission control algorithms?]

10 = Illegal style

Error Value

Specific cause of the error described by the Error Code.

A.12 CREDENTIAL Class

[TBD]

A.13 INTEGRITY Class

[TBD]

B UDP Encapsulation

As described earlier, RSVP control messages are intended to be carried as "raw packets", directly within IP datagrams. Implementing RSVP in a node will typically require an intercept in the packet forwarding path for protocol 46, which means a kernel change. However, for ease of installing RSVP on host systems in the short term, it may be desirable to avoid host kernel changes by supporting UDP encapsulation of RSVP messages. This encapsulation would be used between hosts and the last- (or first-) hop router(s). This scheme will also support the case of an intermediate RSVP router whose kernel supports multicast but does not have the RSVP intercept, by allowing UDP encapsulation on each interface.

The UDP encapsulation approach must support a domain that contains a mix of "UDP-only" hosts, which require UDP encapsulation, and "raw-capable" host, which can use raw RSVP packets. Raw-capable hosts and first-hop router(s) must send each RSVP message twice in the local domain, once as a raw packet and once with UDP encapsulation; these nodes will also receive some local RSVP packets in both formats. We assume that the only negative impact of this duplication will be (negligible) additional packet processing overhead in the raw-capable hosts and first-hop routers.

[REST TBD]

C 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.

C.1 Reservation Styles

A Dynamic-Filter (DF) style reservation decouples reservations from filters.

Each DF reservation request specifies a number D of distinct reservations to be made using the same specified flowspec, and these reservations have a wildcard reservation scope, so they go everywhere. 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. Like a FF style request, a DF style request causes distinct reservations for different senders.

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 essential difference between the FF and DF styles is that 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 style.

C.2 Examples

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

- DF Style

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

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

Figure 11 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 11). Since there is only one source upstream from previous hop (a), the first parameter of the DF message (the count of channels to be

```
                         |
         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 11: Dynamic-Filter Reservation Example

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.

C.3 Resv Messages

Add the following sequence:

```
    <style-specific-tail> ::=

                <Style-DF> <FLOWSPEC> <filter spec list>
```

```
<filter spec list> ::=  <empty>  |  <filter spec list> <FILTER_SPEC>
```

C.4 STYLE Class

- STYLE-DF object: Class = 4, C-Type = 3

```
+------------+------------+------------+------------+
|  Style=3   |  ////////  | Dyn Resv Cnt|  FD Count  |
+------------+------------+------------+------------+
```

Style

3 = Dynamic-Filter

Dyn Resv Count

The number of channels to be reserved for a Dynamic Filter style reservation. This integer value must not less than FD Count.

References

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

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

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

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

[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.

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

Security Considerations

See Section 2.5.

Authors' Addresses

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

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

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

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

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

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

Shai Herzog

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

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

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

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