The Design of the RSVP Protocol

Robert Braden, Deborah Estrin, Steven Berson, Shai Herzog, Daniel Zappala

USC/Information Sciences Institute

FINAL REPORT
Contract DABT63-91-C-0001
27 May 1993 - 30 June 1995

ABSTRACT: RSVP, a setup protocol that creates flow-specific reservation state in routers and hosts, is a component of the QoS extensions to the Internet architecture known as integrated services. RSVP was designed to provide robust, efficient, flexible, and extensible reservation service for multicast and unicast data flows. This report summarizes the key RSVP design decisions and their rationale.

1. Introduction

RSVP (*ReSerVation Protocol*) is a setup protocol for Internet resource reservations; its purpose is to create flow-specific resource reservation state in routers and hosts. RSVP was designed as a component of integrated service, a set of QoS extensions to the Internet architecture [ISIP92, ISarch93]. The fundamental design of RSVP was developed by a research collaboration during the period 1991-1993 [Zhang93]. Beginning in 1993, a further research and development collaboration turned this proto-RSVP into a practical Internet protocol [RSVP95], and a prototype implementation was constructed. Since 1995, the RSVP protocol has been further refined and documented as a potential Internet standard by a Working Group of the IETF; the result is known as Version 1 of RSVP [RSVP97].

This document provides a concise overview of the RSVP protocol and documents the important technical decisions that went into its design. Some of these ideas were expressed in [Zhang93], but others have not previously been documented. The RSVP Functional Specification document describes the final protocol, but it does not describe the technical issues and choices that were made or the alternatives that were considered; this document attempts to fill that gap.

The remainder of this section briefly recapitulates the integrated services context for RSVP and then summarizes RSVP's design objectives and basic architecture. Section 2 contains a concise summary of the design of the RSVP Version 1 protocol. Section 3 discusses the most important particular RSVP design issues and describes the alternatives that were considered. Section 4 concludes the report.

1.1 Integrated Services

The *best-effort* service provided by the original Internet design allows congestion-caused end-to-end delays to grow indefinitely. To better support real-time applications, e.g., packet voice, packet video, and distributed simulation, an extension to the Internet architecture has been developed. This extension is known as *integrated services*, and a network that supports it is called an *integrated services packet network* (ISPN). With integrated services, an end system can request a particular quality of service (QoS), e.g., bounded end-to-end queueing delay, for a particular data flow. Providing the QoS generally requires reservation of network resources in routers hosts along the path(s) of the flow as well as in the end hosts.

In order to provide a requested QoS, the nodes of an ISPN must perform *reservation setup*, *admission control*, *policy control*, *packet scheduling*, and *packet classification* functions. Figure 1 illustrates these functions in an ISPN router.

1. Reservation Setup

A reservation setup protocol is used to pass the QoS request originating in an end-system to each router along the data path, or in the case of multicasting, to each router along the branches of the delivery tree. In particular, RSVP was designed to be the reservation setup protocol for an ISPN.

An RSVP reservation request is basically composed of a *flowspec* and a *filter spec*. The flowspec defines the desired QoS, and the filter spec defines the subset of the data stream, i.e, the flow, that is to receive this QoS.

2. Admission Control

At each node along the path, the RSVP process passes a QoS request (flowspec) to an *admission control* algorithm, to allocate the node and link resources necessary to satisfy the requested QoS. If admission control accepts the request, the necessary state is established for it; otherwise, an error message is sent.

3. Policy Control

Before a reservation can be established, the RSVP process must also consult *policy control* to ensure that the reservation is administratively permissible.

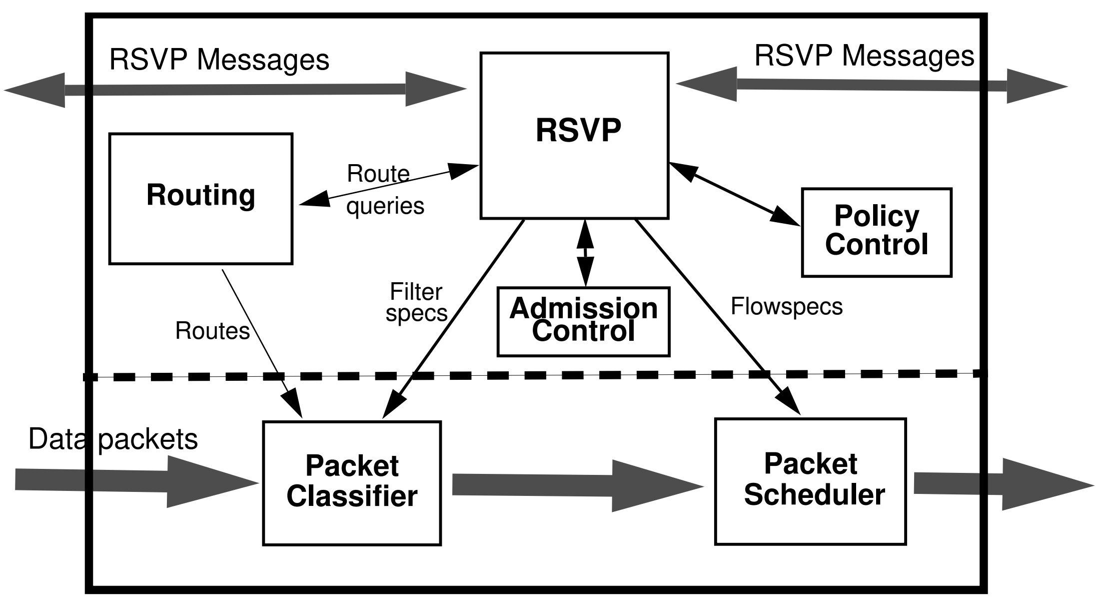

Figure 1: Integrated Services Components in ISPN Router

4. Packet Scheduler

Assuming that admission control and policy control both succeed, the RSVP program installs state in the local *packet scheduler* or other link-layer QoS mechanism, to provide the requested QoS. The packet scheduler multiplexes packets from different reserved flows onto the outgoing links, together with best-effort packets for which there are no reservations.

5. Packet Classifier

The RSVP process also installs state in a *packet classifier,* which sorts the data packets forming the new flow into the appropriate scheduling classes. The state required to select packets for a particular QoS reservation, known as a *filter*, is specified by the filter spec.

In Figure 1, the Packet Scheduler and Packet Classifier components are shown below the horizontal dotted line because they are in the data packet forwarding path; these components are collectively called *traffic control*. The RSVP and routing processes shown above the line typically execute as background processes. In an end system, the dotted line is expected to represent the user/kernel boundary; the forwarding path is in the kernel, while RSVP and routing protocols will execute in user space.

1.2 RSVP Architecture

RSVP was designed to provide *robust, scalable, flexible,* and *heterogeneous* resource reservation setup for *multicast* as well as *unicast* data flows in an integrated services packet network. These design requirements led to a number of basic architectural features: (1) a multipoint-to-multipoint communication model, (2) receiver-initiated reservations, (3) cached ("soft") state management in routers, and (4) separation of reservations from routing.

(1) Multipoint-to-Multipoint Communication Model

RSVP was designed from the beginning to support multicast as well as unicast data delivery. In the RSVP model, the basic communication model is a simplex distribution of data from m sources to n receivers to the same destination. RSVP calls such an m-to-n flow a *session*. For a conferencing application, each participating host will play both sender and receiver roles, and typically m = n. At the other extreme are broadcast applications, where m = 1 and n may be >> 1. Figure 2a illustrates the routing of data packets in a multicast session from two sender hosts S1 and S2 to three receiver nodes R1, R2, and R3, through routers A, B, and C.

RSVP must also support unicast sessions, which are not sufficiently defined by the IP destination address; additional demultiplexing information, called a *generalized destination port*, is required at the receiver. Therefore, an RSVP session is logically defined by the pair:

(Destination IP Address, Generalized Destination Port).

It should be possible to make a reservation for a subset of the traffic on a particular multipoint-to-multipoint session. For example, the traffic subset for a reservation might be defined by particular sender host(s), by particular layers of a layer-encoded video signal, or both. We can describe this selection information by the pair:

(Source IP Address, Generalized Source Port).

In principle, selection of the traffic subset to receive a reservation may depend upon any fields in the IP header, in the transport protocol header, or in an application-layer header. The "generalized ports" thus stand for all header fields, other than the IP addresses, that are used for selection. However, for simplicity and efficiency, the current RSVP protocol supports a much more restricted set of header fields for classifying data packets; in particular, it goes no deeper into the packet than the transport layer header.

The multipoint-to-multipoint model allows data flows from different senders for the same session to arrive at a common node. To accommodate a variety of applications, RSVP provides flexible semantics for sharing reservations among such flows. For example, to support reservation channel switching for video, a receiver wants a *distinct* reservation for each sender's traffic. On the other hand, it is more useful to have a single shared reservation channel for audio traffic. Because of silences, such a shared audio channel requires a network bandwidth that is essentially constant, independent of the number of sources m; i.e., all senders can *share* a single reservation. To select one of these cases, each reservation request carries the binary-valued sharing attribute: *shared* vs. *distinct*. The sharing attribute is carried in a vector of attributes called the *style* of the reservation; other style attributes are discussed later.

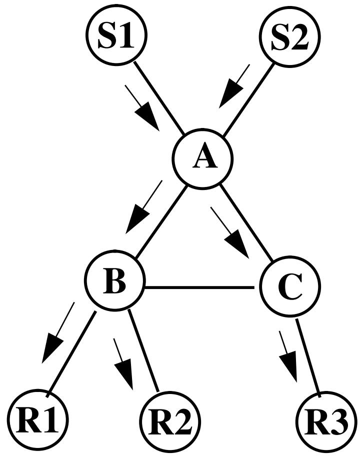

Fig 2a: Session data packets and **Path** messages

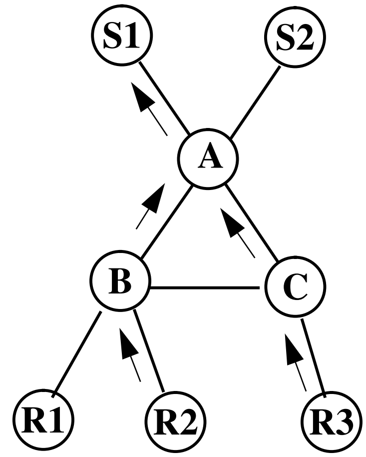

Fig 2b: Reservation requests from receivers R2 and R3 for sender S1.

(2) Receiver-Initiated Reservations

A major concern of RSVP design is scaling well for a large n (number of receivers). The basic RSVP design strategy to achieve large-n scaling is to use *receiver-initiated* (also known as *leaf-initiated*) reservations. The receiver end system initiates an RSVP reservation request at a leaf of the multicast distribution tree or at the unicast destination; this request then travels towards the sender(s), as shown in Figure 2b.

In general, a reservation request message travels only as far as the first multicast router at which another receiver's reservation already exists for the same session. At that point, the new request is *merged* with the previous reservation, and a single request travels upstream. The merged reservation has a flowspec that is the 'largest' of the requests being merged. For example, in Figure 2b the requests from receivers R2 and R3 are merged in router A.

While receiver initiation is critical for multicast sessions, it is not critical for unicast. It was suggested that RSVP could support both sender initiation and receiver initiation, and allow applications to choose. However, economy of mechanism was achieved by supporting only receiver initiation.

(3) Soft State

To achieve robustness and simplicity, RSVP creates “soft” state within the network. Soft state times out if it is not periodically refreshed. The RSVP architecture thus leaves responsibility for state maintenance to the endpoint hosts, which must periodically re-initiate the same RSVP control messages.

Since there is no distinction between initial setup and maintenance of state, RSVP state automatically adapts to routing changes, multicast group membership changes, and reservation modifications. For example, if state is lost or a route changes, the next refresh message will establish the reservation (assuming admission control succeeds) along the new route. Also, an existing reservation can be modified by simply sending an RSVP message containing the new reservation parameters; no separate mechanism is needed for modifying reservations.

(4) Separation of Reservation from Routing

A basic architectural principle is that integrated services is an optional extension to the underlying best-effort data delivery of the Internet; best-effort service must always be available. For example, receiver R1 may join the multicast group in Figure 2a without making a reservation. The data stream received by R1 will then obtain only best-effort service on the last hop from router B. Note that R1’s traffic will then get a “free ride” on the reservations established by receivers R2 and R3 upstream of B. Such “free” reservations are unavoidable without a major change to multicast semantics and mechanism as well as increased cost to the network; they are therefore accepted as part of the service model.

Since multicast forwarding must function whether or not there are reservations, the natural modularity is to separate reservation from routing, making RSVP a pure reservation setup protocol. This separation allows RSVP to operate with the wide variety of present and future routing protocols in the Internet, with minimal changes. Document [RSRR96] describes a generic route lookup interface between RSVP and routing.

The minimum functionality required of routing to support reservations is answering RSVP queries for the next hop for a given Destination Address (unicast) or a given (Destination Address, Sender Address) pair (multicast). This allows RSVP to make its reservations along the path that the data will take, while route computation and installation are left to the routing protocol itself. A small extension to this simple route query function provides asynchronous route-change notification to RSVP, as described below in Section 3.5.

However, this minimum functionality has its limitations: it forces RSVP to use whatever multicast routes are available for a particular multicast group. If any of these routes do not have the resources to support a reservation request, the requesting application must either lower its request or give up. If the multicast routes change, RSVP must adapt to the new routes and re-establish its reservation by contacting admission control at each new node. This may cause at least a temporary service disruption. Moreover, there is no guarantee that the new routes will be able to support the same QoS as the old routes.

In order to more effectively support real-time applications, research is being pursued on how the network routing protocol and RSVP can cooperate to provide stable routes that deliver the applications’ desired QoS [Zappala97]. Since global knowledge cannot be assumed, this more advanced routing/reservation cooperation must be based upon local or partial knowledge. It is expected to result in an iterative process, as follows. (1) The routing protocol is responsible for identifying a route that has the possibility of meeting the receiver's requirements. (2) The reservation protocol verifies the availability of resources along the route and attempts to reserve resources for the user. (3) If the reservation request is unsuccessful, this process repeats, with the routing protocol finding a new route for the reservation protocol to try. Once a reserved route is obtained, the routing protocol maintains (“pins”) the route for the receiver.

2. OVERVIEW OF THE RSVP PROTOCOL

This section summarizes the RSVP protocol with minimal explanation. The reasoning behind many of the protocol features is presented later in Section 3.

2.1 Data Flow Definitions

In principle, a filter spec defines the flow (i.e., the subset of the data stream) to receive a particular QoS. In practice, flow definition is divided into two parts, the session definition and the filter spec. Thus, the session specifies the destination and the filter spec specifies the “rest”. In particular, the basic filter spec format defines a sender to the session.

Although generalized ports could in principle be defined by application-layer header fields, RSVP Version 1 restricts the generalized ports to be transport-layer demultiplexing fields. The generalized destination port is specified by a port number and the IP Protocol Id, while the generalized source port is specified by a port number (assuming the same protocol Id as the session). Thus, an RSVP Version 1 *session* is defined by the triple:

(Destination Address, IP Protocol Id, Destination Port),

while the basic form of RSVP filter spec selects one specific sender defined by the pair:

(Source Address, Source Port)

Source Port and Destination Port values are the corresponding UDP/TCP port fields if the Protocol Id is UDP or TCP; otherwise, the value zero is assumed for both port values. Two other filter spec forms have been defined, for IPv6 flow ids [RSVP97] and for data streams using IPSEC [IPSEC97]; others may be introduced in the future.

2.2 RSVP Messages

RSVP creates and maintains state by periodically sending control messages in both directions along the m-to-n data paths for a session. RSVP messages are sent as IP datagrams and are captured and processed in each node -- router or host -- along the path(s), to establish, modify, or refresh state.

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

o **Resv** (Reservation request) messages

Resv messages are periodically initiated by receivers to request reservations. **Resv** messages travel upstream (i.e., against the direction of data flow) to create *reservation state* in each node. See Figure 2b. Each **Resv** message is sent to the unicast address of the *previous* RSVP hop.

A **Resv** message forwarded by a node is obtained by merging all related incoming reservation requests, as described later. A new or modified **Resv** message is forwarded only as far as the node at which merging causes it to have no net effect on upstream reservations.

o **Path** messages

A Path message is initiated by a sender and travels downstream, addressed to the multicast or unicast destination address of the session, to create *path state* in each node. RSVP in each node queries routing in order to forward a **Path** message along the exact path(s) that the corresponding data packets will traverse, as shown in Figure 2a. Each **Path** message contains information about the sender that initiated it, and the path state basically consists of a description of every sender to the given session. **Path** messages carry the same IP TTL as the data packets, so they have the same range.

Path messages carry three distinct kinds of information.

(1) Specification of previous RSVP hop.

Each **Path** message carries the IP address and a logical interface number (Logical Interface Handle or LIH) of the node that last forwarded this message. This information, recorded in the path state, is used to route **Resv** messages upstream to that previous hop. The purpose of the LIH is described in Section 3.4.

(2) Tspec describing sender traffic

This Tspec is used to prevent over-reservation on the links nearest to the sender, as described in Section 3.2.

(3) Adspec to measure path properties

The Adspec is updated at each hop to measure properties of the path for the use of the receiver for making reservations.

Although path state and reservation state time out if not refreshed, the following two message types can be used to promptly and explicitly 'tear down' state.

o **ResvTear** (Reservation Teardown) messages

A **ResvTear** message deletes specific reservation state, traveling upstream following the same path(s) as the corresponding **Resv** message. It travels only as far as the node where merging will cause it to have no effect on upstream reservations.

o **PathTear** (Path Teardown) messages

A **PathTear** message deletes path state, traveling downstream following the same path(s) as the corresponding **Path** message (and the data).

There are also error messages corresponding to **Path** and **Resv** messages, as follows.

o **ResvErr** (Reservation Error) messages

A **ResvErr** message may report an error found while processing a **Resv** message or a spontaneous error in reservation state. It travels downstream to all receivers that may have contributed to the error (see below).

o **PathErr** (Path Error) messages

A **PathErr** message reports an error in processing a **Path** message. It travels upstream to the corresponding sender.

A ResvErr message includes the flowspec that failed and an information bit called InPlace. This bit is set on if the error being reported was an admission control failure in which an existing reservation was being modified or increased. In this case, the RSVP specification requires the existing reservation remain in effect (see Section 3.3), and this is reflected by setting the InPlace bit on in the resulting ResvErr message.

Normally, a node forwards a ResvErr message to all next hops for which there is reservation state in the node, for the matching sender and session. That is, a ResvErr message is normally flooded to the entire subtree of receivers with reservations, downstream of the error. However, if a node receives a ResvErr message with the InPlace bit on, it suppresses forwarding this message to any next hop whose reservation request carried a flowspec that is strictly smaller than the flowspec that failed.

2.3 RSVP State

2.3.1 Reservation State

An elementary reservation request contains the following information that is saved in the reservation state:

* D= (Destination Address, IP Protocol ID, Destination Port) triple defining the session.

* NHOP = Next Hop address, i.e., the IP address of the RSVP-capable node from which the **Resv** message arrived.

* LIH = Logical Interface Handle defining the outgoing interface to which the reservation applies.

* Q= flowspec defining the requested QoS.

* F_1, ... F_k, a list of filter specs defining the senders to receive the QoS.

* Style

This state is stored in a reservation state element (RSE), which is distinguished by:

(D, NHOP) for a shared style

(D, NHOP, F) for a distinct style

A **Resv** message may carry a single elementary reservation request, or for one style (FF; see below), multiple elementary reservation requests may be packed into a single **Resv** message.

2.3.2 Path State

A **Path** message contains the following information:

* D= (Destination Address, IP Protocol ID, Destination Port) triple defining the session.

* PHOP = Previous Hop address, i.e., the IP address of the RSVP-capable node from which the **Path** message arrived.

* LIH = Logical Interface Handle (discussed below) for the outgoing interface of the previous hop.

* T= Tspec defining the QOS parameters of the data traffic stream that will be sent.

* S= Sender Template defining the sender IP address and source port from which the data will be sent.

* Adspec, used by integrated services to gather end-to-end characteristics of the path.

The state received in a **Path** message is stored as a path state element (PSE), which is distinguished by:

(D, S) for unicast sessions, or

(D, S, Incoming interface) for multicast sessions.

2.4 Styles

Much of RSVP's flexibility resides in the vector of reservation attributes called the *style*. The style currently specifies values for two attributes:

o Sharing attribute: values are *Shared* or *Distinct*.

o Sender Selection attribute: values are *Explicit*, *Wildcard*, or *Assured*.

The sharing attribute was described earlier. The *sender selection* attribute controls how senders are selected; it also controls the *scope* of a request, i.e., the set of senders towards which it is forwarded. The following two values are defined for sender selection:

-- **Explicit**: The **Resv** contains filter specs that explicitly select those senders whose packets will receive the reservation; this set of senders is also the scope of the reservation.

-- **Wildcard**: The **Resv** contains no filter spec; the reservation applies to all upstream senders, and the scope is also all upstream senders.

Table 1 shows the three attribute combinations that are included in the protocol and the abbreviated names given to them. These styles are defined as follows, using the notation <style name>(<parameter> ...) for a reservation request. Here Fi representsa filter spec, while {Q} represents a flowspec.

o WF style: No filter spec ("*") meaning all senders, sharing the flowspec Q.

Reservation request: WF(*{Q})

o FF style: A list of k > 0 (filter spec, flowspec) pairs, defining k independent and distinct reservations.

Reservation request: FF(F1{Q1}, ... , Fk{Qk})

o SE style: A list of k > 0 filter specs defining senders sharing the same reservation, which is defined by the flowspec Q.

Reservation request: SE((F1, ... , Fk){Q})

2.5 Merging Reservations

RSVP's receiver-initiated reservations accommodate heterogeneous QoS requests from different receivers. At each node in which multicast delivery replicates data packets, RSVP merges the corresponding reservations into a single reservation message to be sent upstream.

When RSVP merges two reservations, their flowspecs are combined to define a reservation 'large' enough to satisfy both requests. If the flowspecs can be strictly ordered, the 'larger' of the two is used; otherwise, a third flowspec that is 'larger' than both is constructed. This

	Sharing	
Sender Selection	Distinct	Shared
Explicit	FF style	SE style
Wildcard	(not defined)	WF style

Table 1: Styles

combination, which is called the Least-Upper-Bound (LUB), is associative and commutative. In Figure 2b, for example, if R2 and R3 request QoS defined by flowspecs Q2 and Q3, then router A computes merges these into LUB(Q2, Q3) which is forwarded towards S1. The parameters passed to traffic control in router A cause traffic policing of the outgoing data stream with the 'smaller' reservation, or both if they are not orderable.

Merging of different styles is not allowed. The following rules are used for merging reservations of the same style. The issue of which reservations to merge will be discussed below.

o Merging WF style: No change (except to SCOPE object; see Section 3.7).

o Merging FF style: Combine the lists of Fi{Qi} pairs into a single list. Wherever the same filter spec occurs twice, replace by a single pair using the LUB of the Q's:

F{Qa}, F{Qb} ==> F { LUB(Qa,Qb) }

o Merging SE style: Take the union of the filter spec lists and the LUB of the flow specs:

(Fa1, ...){Qa}, (Fb1, ...){Qb} ==> ((Fa1, ...) U (Fb1, ...)) { LUB(Qa,Qb) }

An elementary RSVP reservation request is forwarded towards all senders whose data packets will be sorted into that reservation class. This establishes a fundamental predicate relation between path state (PSEs) elements and reservation state elements (RSEs). Consider an RSE on an outgoing interface OI and with filter specs F_1, ... F_k, and a PSE for sender S. We say that the PSE 'maps onto' the RSE if data packets from S to the session are routed to OI and if the RSE includes an F_j that specifies the sender S. Let X be a PSE for sender S; in general, the flowspec that is forwarded towards S is formed by merging the flowspecs in all RSEs that X maps onto.

RSVP forwards each **Resv** message to a particular previous hop (PHOP) node for which it has path state. Assume a particular PHOP node P, and let G_S = { S_1, ... S_n } be the set of senders (or alternatively, the corresponding PSEs) whose **Path** messages came from P. A **Resv** message sent to P may contain only filter specs that match a sender in G_S. The detailed rules for forming the Resv message for P depend upon the *style*.

o WF Style:

A reduced set G_S' of eligible senders is formed from G_S using the received SCOPE objects (see Section 3.7 below). Then the outgoing flowspec Q is formed by merging the flowspecs from all RSEs that any of the PSEs remaining in G'$_S$ map onto.

o SE Style

A set G_S' is formed, containing those senders in G_S that map onto some RSE. Then the outgoing flowspec is formed by merging the flowspecs in the RSEs that any of the PSEs in G_S' map onto. The outgoing filter spec is the simply the filter specs that match the PSEs for G_S'.

o FF Style

The outgoing Resv message will contain a list of (flowspec, filter spec) pairs, one for each sender in G_S. Each filter spec matches a PSE in G_S, and the corresponding flowspec is formed by merging the flowspecs in the RSEs that this PSE maps onto.

Finally, we must specify *when* to forward a merged reservation message. For explanatory purposes, suppose that RSVP in each node maintained explicit state for outgoing **Resv** refresh messages; we call this the merged reservation state (MRS). RSVP would maintain its MRS consistent with the path and reservation state in the node. A node would send a **Resv** refresh message to a particular PHOP node whenever:

(1) a new MRS element was added for that PHOP,

(2) the MRS element for that PHOP changed (to reflect a change in path and/or reservation state),

(3) the refresh timer went off, or

(4) local repair was needed (see Section 3.5).

If an element of MRS was deleted, a ResvTear message would be sent to that PHOP. Actual implementations may not keep MRS state, but instead dynamically compute the merged state as needed; however, the effect must be the same as the rules above.

2.6 Distributed State Management

RSVP is designed to create "soft" state, which times out if it is not refreshed soon enough. The only permanent state is in the end nodes. The algorithms for refreshing and timing out state are a fundamental aspect of RSVP.

2.6.1 Hop-by-Hop Refreshes

It is not useful to think of a **Resv** refresh message as travelling end-to-end; instead, each RSVP node sends refreshes for its local reservation state independently of refreshes it receives. This is necessary to accomodate merging: when a **Resv** refresh message is sent, its contents are generally computed by merging multiple reservation state elements, each of which is itself refreshed independently. However, whenever the contents of a potential refresh message changes, that refresh message must be sent immediately. As a result,, a new or modified **Resv** message is forwarded immediately by each hop, until it reaches a hop at which merging will not modify the state at the next hop.

The case of **Path** messages is simpler, because there is no merging of path state. However, for both Resv and Path messages, independent refreshing of each hop improves resiliency against packet loss, so **Path** refresh messages are also sent independently by each node along the path.

Independent refreshing of each node brings another benefit: it allows each hop to choose an appropriate refresh rate in order to adapt to the level of RSVP control messages and perhaps congestion in the link; this is discussed below.

Once local state is established, an RSVP node sends refreshes forward for that state until it is removed or times out. Any other rule (e.g., ceasing to send refreshes when one or more refreshes are not received) would make losses accumulate along the path; another rule would also be incompatible with merging. A consequence of this rule is that successive nodes time out sequentially; the Nth node will time out after a time proportional to N. RSVP avoids this O(N) time out period by requiring that timeout of local state generate a teardown message. If the teardown message is generated in the first node and travels to the Nth node, all state will be deleted at once. If the Teardown message is lost at some node, an additional timeout period will be needed to delete the state beyond the loss, but the total time will be O(1) rather than O(N).

2.6.2 Refresh Period and State Timeout

Suppose that a node sets a refresh timeout period of R, so that it generates a **Resv** refresh every R seconds. Actually, a node must randomize the inter-transmission interval, in the range [0.5R,1.5R]. The value of R is carried in each refresh message. The node that receives the message then refreshes the corresponding state element by resetting its timeout timer with a lifetime T_L that is a multiple of the received R. In particular, the RSVP spec calls for:

$$T_L = (K + 0.5) * 1.5 * R \qquad [1]$$

If the number K is an integer and R is a constant, this formula should ensure that state will not time out even if K-1 or fewer successive refresh messages are lost. The factor 1.5 handles the worst-case due to randomization of the actual inter-refresh times. The 0.5 term is to avoid a race condition due to variations of clocks in different nodes.

As a result of this mechanism, the values of R and K do not have to be standardized across all routers; they can be configured according to experience and circumstances, and they can be adjusted adaptively. Furthermore, R and K can be determined independently by the sending and receiving ends of the refresh hop.

If the sender increases R rapidly but one or more successive packets are lost, the receiving node's value of R may be out of date, allowing a false timeout. Version 1 RSVP solves this problem by limiting the rate of increase in R: the ratio of R in two successive timeout intervals cannot exceed a fixed constant λ, the *maximum slew ratio*. For a given K, we can calculate a λ that will protect against false timeout, assuming that L or fewer successive messages are lost. As illustrated in Figure 3a, suppose that packets sent at time t_0 and at time t_{L+1} are received, but packets sent at times t_1, t2, ... t_L are lost. Then:

$$t_i \leq t_{i-1} + \lambda^i * R * 1.5 \qquad i = 1, \ldots, L+1 \qquad [2]$$

If state is not to time out before the packet sent at time t_{L+1} is received, then $t_{L+1} - t_0$ must be less than or equal to 1.5*R*K (by [1] and omitting the 0.5 term). Using equation [2], we get:

$$1.5*R*(\lambda + \lambda^2 + \ldots + \lambda^{L+1}) = 1.5*R*K \qquad [3]$$

Results of solving this equation for λ, for small K and L, are shown in Figure 3b. This calculation makes very conservative assumptions: the 0.5 term in equation [1] is ignored, and the R sent in a message is that for the interval just ending. On the basis of this table, the RSVP spec places a limit λ = 1.30 on the slew rate, assuming K = 3. Experience may suggest other values in the future.

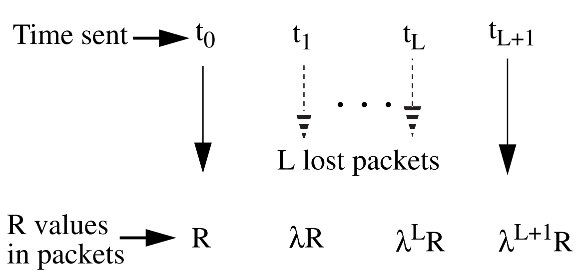

Fig. 3a: L successive lost packets and slew rate λ

	K= 3	4	5
L= 1	1.30	1.56	1.79
2	1	1.15	1.28
3	x	1	1.09
4	x	x	1

Fig 3b: Max λ as function of K and L

2.6.3 State Timeout Granularity

A PSE (sender) is the unit for timing out and refreshing path state. The timeout unit for reservation state is a single filter spec. That is, individual state timers are associated with: an entire WF reservation, each (flowspec, filter spec) pair of an FF reservation, or each filter spec in an SE reservation.

3. RSVP Design Issues

The RSVP protocol described in Section 2 represents the resolution of a number of protocol design issues. This section discusses some of the important issues and the reasoning behind the decisions.

3.1 Data Selection

RSVP allows each receiver to selectively make reservations for subsets of a session's data packets. In particular, FF and SE styles allow sender selection. For example, in Figures 2a and 2b, sender S2 is sending data to the session but no receiver is making a reservation for it.

If the application is really not interested in receiving data from S2, it is desirable in the interest of network efficiency to suppress data packets from S2 until a receiver "switches channels" to it. One approach would be a reservation style in which data packets are dropped if there is no reservation for them. However, this would directly contradict the basic architectural principle that best-effort service must be the default; for example, it would prevent R1 from receiving S2's data if it joins the multicast group as a best-effort receiver.

Therefore, the fundamental design decision was made that:

* Routing determines *which* packets are forwarded, but

* RSVP determines *how* they are forwarded,

and best-effort is always the default QoS. To select among senders, multicast routing must provide a *sender-join* capability, a feature that would be equally useful for best-effort service. To allow selection among layer-encoded subsets of a video stream, each layer can be sent as a different multicast group. A receiver will simply join the group(s) of the data subflows it wanted to receive.

3.2 Heterogeneity

RSVP supports what we may call "downstream heterogeneity", that is, heterogeneous QoS requests from different receivers. This requires merging flowspecs when **Resv** messages are forwarded, as explained earlier. RSVP also handles "upstream heterogeneity", that is senders with heterogeneous traffic flows (Tspecs); this might result from different link access speeds, for example. RSVP will not make a reservation larger than the traffic flow that can come down a particular link. For example, consider a shared reservation for senders S1 and S2 in Figure 2a. S2 might have a lower bandwidth access link that cannot support the LUB of the requests from R2 and R3. This problem is handled by installing a reservation using an effective Tspec that is the minimum of the Tspec from the path state and the Tspec included within the flowspec. For a shared reservation, where there may be multiple senders, the sum of the sender Tspecs is used.

One of the unexpected results of the RSVP research effort has been the realization that the heterogeneity that can be achieved without severe complexity is quite limited. For example, RSVP does not allow the style to be heterogeneous; merging of reservations with different styles was considered and rejected. Shared and distinct styles would be fundamentally incompatible, while merging explicit and wildcard styles (which in practice means SE + WF => WF) could cause unexpected flows in an SE reservation. Another example of a major consequence of heterogeneity and is the "killer reservation" problem, discussed in the next section.

Furthermore, the value of heterogeneous reservations for real applications is unclear. Enforcing heterogeneity at multicast branch points requires policing the flows with the smaller reservations. The application has no control over which particular data packets will receive the requested QoS and which will receive only best-effort service and perhaps be dropped as a result. Layered encoding using multiple multicast groups provides a much more promising solution to heterogeneous receiver capability.

However, if heterogeneity were once excluded from the spec, it might be very difficult to add it at a later time. The decision was therefore made to include the provisions for heterogeneity, especially merging of reservations.

3.3 Killer Reservations

The *"killer reservation"* (KR) problem is a denial of quality-of-service that can result from merging two different flowspecs; thus, it is a result of heterogeneity. If the path towards the source has sufficient resources for the smaller of the reservations but not for the merged reservation request, then the effect of merging can be to deny reservations to both.

We distinguish two versions of the KR problem, known as KR-I and KR-II.

KR-I Problem

The first killer reservation problem occurs when a reservation with flowspec Q1 is already successfully in place, and another reservation with flowspec Q2 larger than Q1 arrives such that the merged flowspec LUB(Q1,Q2) fails admission control in the current node or in some node upstream. If RSVP simply replaces Q1 with the larger reservation and then gets an Admission Control failure, the Q1 reservation will be lost.

The solution to KR-I is fortunately simple: RSVP keeps an existing reservation in place when making an admission control decision for a replacement reservation. If the new reservation fails, a **ResvErr** message is sent back but the original reservation is left in place. Furthermore, the original (smaller) reservation Q1 continues to be refreshed upstream (however, see later discussion of "Error Behavior").

A further feature was added to shield "innocent" receivers from ResvErr messages created by a new receiver making a larger reservation; this mechanism involves the InPlace bit and was described in Section 2.2.

KR-II Problem

In the second killer reservation problem, receiver 1 is persistently trying to make a reservation Q2 that is being rejected somewhere upstream, and then receiver 2 attempts to make a 'smaller' reservation Q1 that would succeed if it were not merged with Q2. It is considered reasonable behavior for receiver 1 to (slowly but) persistently retry its failing reservation, as a user application may reasonably poll for availability of a desired reservation. However, such behavior should not prevent a smaller reservation from succeeding.

Unfortunately, there is no simple solution to this second problem, without adding at least some additional state and processing complexity. The best solution to the KR-II problem is still an open issue. A partial solution has been included in the Version 1 RSVP spec, but since it is not believed to be the best solution, we do not comment on it further here.

3.4 Non-RSVP Regions

To be useful, RSVP must be deployable in the real Internet. Deployment will be gradual, and at any time there will be paths and regions of the Internet that do not support RSVP. Therefore, RSVP was designed to work transparently through arbitrary non-RSVP-capable (*non-R*) routers. This had a surprisingly large impact on major details of the protocol.

(a) An **Resv** message must be routed through a non-R region to the appropriate previous RSVP-capable node. RSVP handles this routing problem by using **Path** messages. Each **Path** message carries the IP address of the previous RSVP-capable hop, and since it has the IP destination address of the session, the **Path** message is automatically and correctly forwarded along the data path(s) through non-R clouds.

We note that if there were no non-R nodes, the routing function of **Path** messages would not be needed, since multicast routing protocols directly provide the previous hop address.

(b) For multicast delivery, the route depends upon the IP source address as well as the destination address. To be reliably routed through a non-R cloud, a multicast **Path** message must therefore carry the original sender address as its IP source address as it is forwarded hop by hop. Sending a datagram with a specific non-local source address requires a kernel modification in some implementations.

(c) Early RSVP designs allowed **Path** messages to be *packed*, i.e., to carry multiple sender descriptions. However, packed **Path** messages cannot satisfy point (b), so each **Path** message can describe only a single sender. This is unfortunate for **Path** message overhead.

(d) A non-R region can result in a **Resv** message arriving on the wrong interface. As illustrated in Figure 4a, the data flow and the **Path** messages go A->B->D, but a **Resv** message might take a different return path. The **Resv** message is unicast to the previous RSVP hop, which is router A, and it arrives at A on the wrong interface. Routers B, C, and D are non-R.

To solve this problem, a **Path** message carries a specification of the outgoing interface through which it was sent, e.g., the A->B interface of router A. This specification, called the Logical Interface Handle, is saved in the path state at the next hop and returned in a subsequent **Resv** message to the previous hop. Thus, router A makes the reservation on the A->B interface, as a result of the LIH in the **Resv**, regardless of which interface the **Resv** arrives on.

(e) A non-R region can cause a message to arrive at the wrong RSVP-capable router, as illustrated in Figure 4b. The situation is essentially the same as in Figure 4a, but here the **Resv** message arrives at router C, which is RSVP-capable. If it tried to process the **Resv** message, C would find no matching path state and send an error message. To solve this problem, an RSVP message (other than **Path** or **PathTear**) that is addressed to an RSVP-capable node must be forwarded towards the destination without processing. Router C will forward the **Resv** message to A, where the LIH will cause the correct reservation to be made.

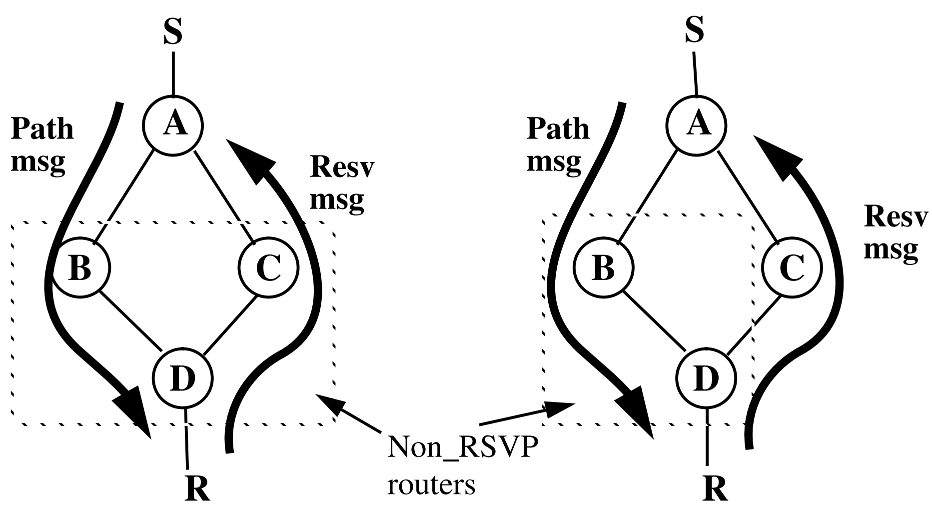

Fig 4a: Non-RSVP Region: Resv arrives on wrong interface.

Fig 4b: Non-RSVP Router: Resv arrives at wrong RSVP router.

(f) Since a non-RSVP region may perturb the QoS being provided to the user, an RSVP-capable node must be able to detect the region and inform the receiver application. There is no perfect solution to detection of a non-RSVP region. Each RSVP message carries the IP TTL (hop count) field with which the datagram was originated. If the datagram arrives with a different TTL, the node assumes that it must have traversed an non-R region. This technique does not work in all cases, because of TTL adjustments due to tunnels; it may need to be supplemented with explicit configuration.

3.5 Liveliness and Local Repair

Refresh messages make the RSVP state self-healing. If a route changes, for example, the next refresh will (try to) re-establish a reservation along the new path. However, this basic robustness mechanism can create excessive RSVP message overhead if the refresh rate R is set very small to achieve good user responsiveness. It is therefore important to supplement the basic refresh mechanism with “liveliness” mechanisms to provide more immediate response to changes.

One liveliness mechanism depends upon a new service from routing: an immediate notification to RSVP when a route that RSVP is using changes. Upon such route change notification, RSVP institutes a procedure called “*local repair*”: it sends immediate **Path** and/or **Resv** refreshes as appropriate down the new path.

Here are some examples of liveliness considerations for RSVP.

(1) A host H1 is the first to join a particular multicast group down a new branch of the multicast distribution tree.

In the node R at which the new branch is spliced onto the tree, routing should signal a route change (new outgoing interface) to RSVP. Local repair in R will then forward a **Path** message down the new branch to H1, which does not have to await a refresh timeout to send its **Resv** message.

(2) A host H2 on a LAN joins a group to which another host on the LAN already belongs.

Each node on the LAN will be receiving all RSVP **Path** messages, whether or not they are destined for the host. Note that a host, like a router, intercepts RSVP packets without regard to whether they are addressed to the node. Therefore, host H2 should already have any path state for the new group, so H2 should not have to await a refresh timeout to send its **Resv** message.

(3) An application in host H3 joins an RSVP session for which path state is already available, as in (2).

RSVP must generate an immediate Path Event upcall to the application.

There may be cases where a specific routing protocol has properties that create race conditions with RSVP, so that the liveliness approaches in (1) and (2) may sometimes fail. In these cases, the RSVP refresh timeout is guaranteed to heal the situation in time. However, it is possible that experience will show that there are important failures of liveliness that will require additional RSVP protocol mechanisms. For example, a host might send a new type of RSVP message that triggers an immediate path refresh when needed.

3.6 Extensibility

To turn the general architecture of RSVP into a practical Internet protocol, many design choices had to be made. Every one of the many possible services and features of RSVP has a cost in complexity of design and implementation. It was therefore necessary to choose a limited subset of the possible features for version 1 of the RSVP protocol. Since we have no experience with an integrated services packet network, these choices must be based only on prognostication of the likely importance of each feature, and it is very likely that future experience will require revision and expansion of the feature set. It was therefore important to choose an easily-extensible packet format for RSVP messages.

The RSVP message therefore carries a minimal fixed-format header followed by a variable list of (type, length, value) data structures that are called *objects*. The type of an object is the pair: (class, C-type); class defines the function of the object and C-Type specifies a particular format for a given class.

From the message syntax viewpoint, many RSVP extensions require only the definition of one or more new C-Type formats for existing object classes. This extension mechanism has already been used to add support for IPv6 as well as IPv4. It has also been used to extend RSVP to support data flows that use the IP Security (IPSEC) mechanism [IPSEC97]. Several other object extensions have also been proposed.

3.7 Wildcard Reservation Looping

Two unexpected problems were found for reservations with wildcard sender selection [Scope96]. The first is that reservations may be made along links that are not used by data. The second is that in a topology with loops, the reservation state can become self-refreshing, so that the reservation state will persist indefinitely long even after the receivers stop refreshing it. The solution to both of these problems requires that the **Resv** message carry an explicit list of the IP addresses of sender hosts; this list is contained in a SCOPE object.

See [Scope96] for a careful explanation of these issues and why a SCOPE list solves them. Also note that the SCOPE object can be greatly reduced in size for a multicast session when routing uses a shared tree [RSRR96].

The requirement for a SCOPE object was a disappointment, because the primary argument in favor of the WF style was that it would scale well for a large number of senders m. However, the SCOPE object means that the size of a WF-style **Resv** message will increase linearly with m. It might appear that WF style could similarly be omitted, since it can be simulated using SE style. However, the WF style has a potentially important advantage over SE when there is a large number of senders: WF can significantly reduce the amount of state in the Classifier. WF requires only one classifier entry, while SE requires an entry per sender. WF also is able to respond with a reservation more quickly when a new sender appears. Therefore, WF was retained in the spec.

3.8 Extended Styles

Table 1 showed the RSVP styles that are included in Version 1 of RSVP. The undefined combination (Wildcard, Distinct) was not used because it adds no new capabilities. For example, an application (or a last-hop router) could create the same effect by sending an FF reservation listing all senders, using the complete sender list available in the path state.

The early RSVP design work defined another style, called dynamic filter (DF), whose attributes are shown in Table 2. DF style is defined using the following additional value of the sender selection attribute:

-- Assured: The **Resv** message contains filter specs to explicitly select those senders whose packets will receive the reservation. However, the scope is wildcard, i.e., reservations will be put in place towards all senders, regardless of whether they are included in the explicit selection.

The Assured attribute must be accompanied with an integer parameter N, the maximum number of distinct reservations to be made; however, in each node no more distinct reservations will be made than the actual number of upstream senders.

The Assured attribute is designed to allow channel switching without fear of admission control failure, once a reservation is in place.

DF style: A list of k ≥ 0 filter specs, and a max number of reservation channels N.

Sender Selection	Sharing: Distinct	Sharing: Shared
Explicit	FF style	SE style
Wildcard	(not defined)	WF style
Assured	DF style	(not defined)

Table 2: Extended Styles

DF(N, (F1, ... , Fk){Q})

Merging DF style:

Take the union of the filter spec lists, and the LUB of the flowspecs, and the sum of the channel numbers but limited to the number of upstream senders.

DF(Nab, ((Fa1, ...) U (Fb1, ...)) { LUB(Qa,Qb) })

where:

Nab = min(Na + Nb, Number of senders upstream)

DF style was dropped from the Version 1 specification because of its complexity and because its usefulness was in doubt.

There are many other possible styles, representing a trade-off among functionality, complexity, and overhead. Furthermore, design choices were required even for some of the styles that were included. Consider for example the rule for merging SE style reservations in the case when the merged filter spec lists overlap.

* The chosen interpretation is to simply take the union of the filter spec lists and the LUB of the flowspecs. However, this approach may cause unexpected interference between merged reservations. Suppose SE((F1,F2){Q}) and SE((F2,F3){Q}) are merged, and that the common flowspec Q can provide lossless service for (only) two senders. The merged **Resv** request: SE((F1,F2,F3){Q}) shares the reservation among three senders, which may cause packet losses.

* An alternative approach might split a single SE reservation into multiple parallel SE reservations when filter spec lists overlap. In the example above, the result of the merge would be three separate reservations: SE(F1{Q}), SE(F2{Q}), and SE(F3{Q}). This approach would protect users from interference by other receivers but use more resources.

An application might want either interpretation of the SE style; perhaps two different styles are needed.

3.9 Other Simplifications

As RSVP was turned into a practical protocol by the IETF, a number of other simplifications were made.

3.9.1 Sessions

RSVP defines the flow to a given destination address and port as a session. This is a natural basis for reservations, assuming that all data packets with the same address will be take the same route (in steady state) through the Internet, because reservation state can be installed along this unique route. RSVP also treats sessions as independent for the purpose of reservations; it does not support multi-session (i.e., multiple destination address) reservations. This restriction is expected to be satisfactory for the majority of the expected applications of integrated services, but some examples are already known that require a more general reservation model. Such a generalization would add syntactic as well as semantic complexity to the protocol.

3.9.2 Filter Specs

If RSVP were designed in full generality, a filter spec would select on arbitrary combinations of data packet header fields. An early version of the RSVP spec approached this generality by using filter specs that were variable-length mask-and-match bit strings. An even more general approach was suggested: pseudo-code defining a program for a filter pseudo-machine. However, all of this generality was dropped in favor of simple UDP/TCP port numbers.

Another generalization that was considered was to allow wildcard port numbers. However, this led to somewhat complex matching rules, and wildcard ports were eliminated from Version 1.

3.9.3 Error Behavior

There are a number of issues in the error model that RSVP presents to a receiver and the control over error behavior available to an application. The current RSVP design has made the very simplest choices for error behavior.

As a simple example of error behavior, suppose that Admission Control fails in a particular node. The current rule is that the node returns a **ResvError** message and neither installs the reservation state nor forwards it upstream. However, some applications may wish to take a more aggressive approach, obtaining reservations across as many of the nodes of the path as possible, hoping that it will get adequate service despite failure in one or more nodes. The risk here is that every upstream router might reject the request, resulting in a flood of **ResvErr** messages. Although the current RSVP spec takes the conservative approach, we anticipate that a future revision of RSVP will provide applications with some control over their aggressiveness in the face of localized failures.

3.9.4 Liveliness Control by Applications

The refresh rate is a trade-off between RSVP message overhead and the liveliness of RSVP's adaptation to changes. An early version of the protocol allowed an end user to specify the maximum refresh rate, i.e., to set a minimum liveliness. Such a provision may only be viable if there is some form of 'cost' attached to a user's liveliness specification. The importance of this feature was unclear, so it was eliminated from the protocol.

4. Conclusions and Acknowledgments

This document has presented an overview of the elaboration of the RSVP protocol architecture into a practical Internet protocol. There have been some surprises -- in particular, the problem of wildcard routing, the many consequences of non-RSVP regions, and the limitations of heterogeneity -- but the general architecture of RSVP has been vindicated, and the general objectives of the protocol have been met. RSVP is simple, robust, scalable (with the size of a multicast group), flexible, deployable, and extensible.

The basic conceptual design of RSVP was performed during 1991-1993 by a research collaboration of Lixia Zhang (Xerox PARC), Deborah Estrin (USC/ISI), Scott Shenker (Xerox PARC), Sugih

Jamin (USC/Xerox PARC), and Daniel Zappala (USC). Zhang did the design and simulation of the major RSVP features, and Jamin successfully demonstrated an early research prototype of RSVP in May 1993. Further definition and refinement of the RSVP protocol was then carried out by a collaboration of ISI, PARC, MIT, and BBN researchers: in particular, Scott Shenker and Lee Breslau (PARC), John Wroclawski and Dave Clark (MIT), Craig Partridge (BBN), and the authors of this report (ISI).

5. References

[IPSEC97] Berger, L. and T. O'Malley, *RSVP Extensions for IPSEC Data Flows*. IETF Work in Progress, March 1997.

[ISIP92] 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.

[ISarch93] Braden, R., Clark, D., and S. Shenker, *Integrated Services in the Internet Architecture, an Overview*. RFC1633, October 1993.

[Zhang93] Zhang, L., Deering, S., Estrin, D., Shenker, S., and D. Zappala, *RSVP: A New Resource ReSerVation Protocol.*, IEEE Network, September 1993.

[RSVP95] Zhang, L., Braden, R., Estrin, D., Herzog, S., and S. Jamin, *Resource ReSerVation Protocol -- Version 1 Functional Specification*. IETF Work in Progress, July 1995.

[RSVP97] Braden, R. (Ed.), Zhang, L., Berson, S., Herzog, S., and S. Jamin, *Resource ReSerVation Protocol -- Version 1 Functional Specification*. IETF Work in Progress, May 1997.

[Scope96] Zappala, D., *RSVP LOOP Prevention for Wildcard Reservations*. ISI internal document, February 1996. Available from ftp://ftp.isi.edu/rsvp/docs/WF.scope.ps.

[RSRR96] Zappala, D., *RSRR: A Routing Interface for RSVP*. IETF Work in Progress, November 1996. Available from: http://www.isi.edu/rsvp/pubs.html.

[Zappala96] Zappala, D., Braden, B., Estrin, D., and S. Shenker, *Interdomain Multicast Routing Support for Integrated Services Networks*, White Paper, December 1996. Available as IETF Work in Progress, March 1997.