Internet Draft
Expires September 19, 1997
File: draft-ietf-rsvp-policy-oops-00.ps

Shai Herzog
Dimitrios Pendarakis
Raju Rajan
Roch Guérin
IBM T.J. Watson Research Center
Apr. 1997

Open Outsourcing Policy Service (OOPS) for RSVP

March 19, 1997

Status of Memo

This document is an Internet-Draft. Internet-Drafts are working documents of the Internet Engineering Task Force (IETF), its areas, and its working groups. Note that other groups may also distribute working documents as Internet-Drafts.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

To learn the current status of any Internet-Draft, please check the "1id-abstracts.txt" listing contained in the Internet-Drafts Shadow Directories on ds.internic.net (US East Coast), nic.nordu.net (Europe), ftp.isi.edu (US West Coast), or munnari.oz.au (Pacific Rim).

Abstract

This document describes a protocol for exchanging policy information and decisions between an RSVP-capable router (client) and a policy server. The OOPS protocol supports a wide range of router configurations and RSVP implementations, and is compatible with the RSVP Extensions for Policy Control [Ext].

Contents

1 Overview

Open Outsourcing Policy Service (OOPS) is a protocol for exchanging policy information and decisions between an RSVP-capable router (client) and a policy server. As the name suggests, OOPS is an outsourcing protocol which allows the partial or complete delegation of the task of policy control from the local router to an external server. Moreover, it is an open protocol in a sense that it does not define or depend on particular policies; instead, it provides a framework for adding, modifying and experimenting with new policies in a modular, plug-n-play fashion.

The OOPS protocol was designed to be compatible with the RSVP Extensions for Policy Control [Ext], both in the format of RSVP objects, as well as the set of supported services.

The basic features of OOPS design are as follows:

Asymmetry between client and server

Adding policy support to RSVP may require substantial modifications to platforms (e.g., routers) which may not have the required implementation flexibility and/or processing power. OOPS assumes that the server is more sophisticated than the client, in terms of processing power and support for diverse policies.

Support for a wide range of client implementation

The OOPS protocol supports a wide range of client implementations. At one end of the spectrum, a "dumb" client may delegate total responsibility to the server for all policy decisions without even maintaining cached states. At the other end, smart clients can perform most policy processing locally and only address the server for a small number of policies and only when they change (otherwise, cache can be used).

Minimal knowledge of RSVP's processing rules.

The server must be aware of the format of several RSVP objects and basic RSVP message types. However, it is not required to understand RSVP's processing rules (e.g., different reservation styles).

Asynchronicity

Both client and server may asynchronously generate queries or requests.

TCP for reliable communications

TCP is used as a reliable communication protocol between client and server.

1.1 Representative OOPS Scenarios

Figure 1 depicts some representative scenarios for policy control along an RSVP path, as envisioned in OOPS. Nodes A, B and C belong to one administrative domain AD-1 (advised by policy server PS-1), while D and E belong to AD-2 and AD-3, respectively.

```
                       AD-1                AD-2       AD-3
         _______________/\___________     __/\__     __/\__
        {                            }   {      }   {      }

        +-------+  +-------+  +-------+  +-------+  +-------+
+----+  |   A   |  |   B   |  |   C   |  |   D   |  |   E   |  +----+
| S1 |--| RSVP  |--| RSVP  |--| RSVP  |--| RSVP  |--| RSVP  |--| R1 |
+----+  +-------+  +-------+  +-------+  +-------+  +-------+  +----+
        | LPM   |             | LPM   |  | LPM   |  | LPM   |
        +-------+             +-------+  +-------+  +-------+
              \                /                        |
               \              /                     +-------+
                \            /                      |Policy |
                 \          /                       |Server |
                  \        /                        | PS-2  |
                   \      /                         +-------+
                   +-------+
                   |Policy |
                   |Server |
                   | PS-1  |
                   +-------+
```

Figure 1: Policy Control along an RSVP path

The scenario includes four typical node types:

(1) Policy incapable nodes: Node B. (2) Self-sufficient policy node: Node D is self-sufficient since its local LPM satisfies its entire policy needs. (It has no need for server advice.) (3) "Dumb" policy nodes: Node E is an unsophisticated node that lacks processing power, code support or caching capabilities, and needs to rely on PS-2 for every policy processing operation. In this case, the volume of traffic and delay requirements make it imperative to connect PS-2 to node E by a direct link or a LAN. (4) "Smart" policy nodes: Nodes A and C include sophisticated LPMs, in that these nodes can process some policies, and have the capacity to cache responses from PS-1. In this case, the contact between the clients and server will be limited to occasional updates, and PS-1 could be located somewhere in AD-1.

Consider the case where the receiver R1 sends a Resv message upstream toward sender S1. Assuming that the reservation is successful, the conceptual flow of policy objects is:

```
R1 -- E -- ELPM -- PS-2 -- ELPM -- E -- D -- DLPM -- D -- C -- CLPM
-- PS-1 -- CLPM -- C -- B -- A -- ALPM -- PS-1 -- ALPM -- A -- S1.
```

Of course, other OOPS messages may be exchanged between policy servers and nodes before authorizing the reservation at individual nodes.

2 Query-Response Protocol

OOPS is a transaction protocol, in which most communication is in the form of queries from the client followed by responses from the server. However, a small portion of the communication may also consist of queries originating from the server, or of unidirectional notifications from one entity to another. In this context, it is important that messages be distinguished by a unique sequence number, so that responses may identify the query to which they correspond.

This section discusses two fundamental concepts of the OOPS protocol: (a) flexible division of labor between client and server. (b) consistent management of client, server and RSVP state.

2.1 Division of Labor between Client and Server

The OOPS protocol allows for a flexible division of responsibilities between server and client. Processing of policies (policy elements within POLICY_DATA objects) can be performed by the server, the client, or by both. The decision on which policies are to be handled locally and which are to be sent to the server is always made by the client based on information exchanged during the connection establishment handshake (see Section 3.1).

Before the client forwards incoming POLICY_DATA objects to the server (Incoming-Policy-Query) it removes or marks the policy elements it wishes the server to ignore. (Marking is performed by changing the policy element P-type to zero.) When forwarding incoming policy objects, the client may also set header flags to inform the server that message integrity and/or rsvp hop has been already checked.

OOPS does not impose limitations on the number of servers connected to the client; when appropriate, the client could divide the work along policy lines between several servers, and be responsible for combining their results. In the rest of this document we describe the protocol for a single server-client pair.

When the client receives outgoing POLICY_DATA objects in response to a previous query (Outgoing-Policy-Response) it is responsible for merging the server response with the locally generated outgoing POLICY_DATA object. Merging is performed by concatenating the local and server policy elements and if necessary, computing some of the POLICY_DATA object fields (e.g., length, INTEGRITY, etc.)

When the client receive status results in response to a previous query (Status-Policy-Response) it is responsible for merging the results from the server with the local results. The following rule applies for combining any number of policies, and specifically, local and server policies:

- When responding to a status query (authorization check), individual policy handlers may vote to ACCEPT, SNUB or VETO the request. As their names suggest, a vote of accept authorizes the request; a snub fails it, but remains indifferent on its final outcome (i.e., other policies could provide authorization); a veto vote excludes the possibility of authorizing the request, even if other policy handlers cast accept votes.
- The merge result provides an authorization if there is at least one accept, and no vetoes.[1] (See [LPM]) for more details).
- The client and/or server should complete their policy processing even if a veto was cast by some policy.[2]
- Protocol errors are always considered as snubs, and thus, neutral.

It is recommended (although not required) that all local status processing at the client be completed before querying the server. This allows the server to immediately commit the transaction rather than having to wait until the client is done. (See the Client-Status-Notification operation.)

2.1.1 Error Reporting

Policy error reporting is policy specific; it is performed by sending POLICY_DATA objects with specific error objects toward the originator of the error. The rules governing error reporting are described in [Ext].

In this document, we discuss only error reporting between the client and the server, which is intended to help the client determine whether error reporting is required at all.

There are two types of possible errors; policy errors and protocol errors. For the purpose of this protocol, policy errors are considered as legitimate results (e.g., reject) and not as errors. Protocol errors must be reported as such. However, since they do not reveal any policy decisions they should always be considered as snubs (and therefore neutral to the overall policy decision).

When the client (or server) discovers a protocol error (syntax, missing parameters, etc.), it is reported alongside and orthogonal to the status results (accept, reject or veto).

[1] A veto has a stronger semantics than a snub, since it has the power to forcefully reject a flow regardless of any accept decisions made by others.

[2] A wide range of policies may not care about the final status results and should be activated regardless. For instance: a policy that logs all policy queries.

2.2 State Management

In order for policy objects contained in RSVP messages to be processed quickly and correctly, it is often required that the results of past policy decisions be cached and maintained at the LPM or the policy server. Maintenance of policy state must be done in a manner that is consistent with the division of responsibility for policy processing between client and server and with RSVP's state management rules.[3]

The most straightforward method for state maintenance is for the LPM and the policy server to use the same soft-state mechanism as the RSVP capable router. Unfortunately, this soft-state approach has undesirable scaling properties since it requires the client to contact the server on each refresh period (regardless of state changes).

An alternative approach is to allow both client and server to use hard-state mechanisms that could limit the client-server communication to updates only. This alternative implies that the client must be capable of recognizing objects that would result in a change of policy state, as well as being able to translate between the soft-state provided by RSVP and the hard-state exchanged with the server.

Thus, we envision one end of the spectrum where a "dumb" client would use a soft-state approach and simply pass all policy objects to the server relying on it for all policy processing. The rate of queries and lack of caching at the client implies the need for a dedicated, close-by server (PS-2, in our example). As we move towards the other extreme, clients become smarter, more capable of caching, and dividing the work between themselves and the server. Such clients could take advantage of the benefits of hard-state management, and initiate queries only on actual state updates.

OOPS supports soft and hard state mechanisms seamlessly, as described in this section. The client determines its desired type of state management, and communicates it on an object-by-object basis. A single client can use soft-state for some information, and hard state for others. Furthermore, the OOPS protocol allows clients to modify their caching strategies on the fly (without having to renegotiate with the server). While the protocol does not impose strategy limitations, a client implementation could restrict itself to a more modest and simple combination of soft and hard state.

There are two types of state information that is stored at the client: (a) client state information that was forwarded to the server (e.g., policy objects in incoming Path/Resv messages). (b) server state which is cached at the client (e.g., policy results computed by the server). The OOPS protocol addresses each of these types of states:

[3] During normal processing, state split between client and server should remain consistent, and timeout at roughly the same time at RSVP, the client, and the server.

2.2.1 Client State Information Cached at Server

The client indicates that it desires hard (or soft) state management of client state information cached at the server by setting (or resetting) the `OOPS_HardState` flag in objects sent to the server. When the client chooses soft-state management for a particular object, policy state for that object would age and expire at the server according to the timeout specified in the object. The client must, therefore, forward each policy refresh (update or not) to the server, to keep the soft-state at the server from becoming stale and expiring. On the other hand, when the client indicates hard-state management, it assumes responsibility for reliably informing the server on every policy update. In this case, the state cached at the server would not expire unless explicitly modified by the client, or when the communication channel to the client breaks. The client may refrain from forwarding to the server any policy objects that are identical to objects previously sent to the server.

The client may switch between hard and soft states on the fly by modifying the `OOPS_HardState flag` while forwarding input to the server.

2.2.2 Server State Information Cached at Client

The client indicates that it is capable of hard (or soft) state management of server state information by setting (or resetting) the `OOPS_HardState` flag in queries sent to the server. Here, hard state management refers to the caching of response results at the client. Soft state management means that the client, being incapable of caching, would purge them after usage (one-time, or disposable results).

A non-cached response has no strings attached, but the client must issue a query each time that responses are needed. When the server responds to a cached (hard-state) query, it assumes responsibility to reliably inform the client about any changes that may occur later to the original results of this query. The client may rely on cached results as long as the there is no change in RSVP's state (which includes incoming policy objects),[4] and the communication channel with the server is intact.

The client may switch between hard and soft states on the fly by issuing a new query with a modified flag.

[4] A configurable option may allow the client to use cached results even when some RSVP state changes. Clearly, there is a trade-off between fast and accurate policy processing, however, given that the server is up, and that authorization was already granted previously for that RSVP flow, some may find it a reasonable policy approach.

2.2.3 State Change Notification

State change notification is done by resending the same type as the original message but with the modified state instead.

Client notification example (incoming POLICY_DATA objects for Resv-X):

```
                   Seq#  Type          Data
                   ---   ----          ----
Client ==> Server:  50   Notify:input  Resv-X: PD-1

Time passes; the input POLICY_DATA object associated with
Resv-X changed to PD-2.

Client ==> Server:  90   Notify:input  Resv-X: PD-2
```

Server notification example (status query for reservation Resv-X):

```
                    Seq#  Type           Data
                    ---   ----           ----
Client ==> Server: 150   Query:status  Resv-X
Server ==> Client: 151   Resp :status  #150: accept

Time passes; the status of Resv-X changed to "reject".

Server ==> Client: 205   Resp :status  #150: reject
```

3 Client-Server Communications

This section describes the fundamentals of client-server communications: connection establishment, communication channel management, and connection termination.

3.1 Connection Establishment

Connections are always initiated by clients. The client establishes a TCP connection to its preferred policy server, and then initiates the OOPS session through a two way handshake.

- Communication Initiation by the Client

The client sends a Connection-Initiation-Query to the server. This message identifies the client to the server and provides the basic characteristics of the client.

- Response by the Server

  The server responds with a Connection-Accept-Response to connect to the client. It may also respond with a Connection-Reject-Response to refuse and disconnect from the client.

  After connection establishment both the client and server know the set of policies that the client can send to the server, and which one of them should handle default (unrecognized) policies. The Keep-Alive period is determined as the minimum between the two values declared in the handshake messages.

3.1.1 Secure Communications

The integrity of the communication channel between client and server is guaranteed by the use of shared-key message digest. (e.g., keyed MD5). A client, wishing to establish secure communications adds a "Cookie" to the Connection-Initiation-Query. The server may respond with a reply Cookie or with an Error-Description[5]

Shared keys may be obtained from local static configurations or could be distributed dynamically. The exchange of cookies provides the client and server with an opportunity for establishing a temporary shared-key (e.g., from Kerberos) for the connection length.

Once a shared key is available, each message sent by either client or server includes an INTEGRITY object as described in [Bak96]. The format and functionality of the INTEGRITY object are identical to that of RSVP. The sender client or server computes the message digest over the entire OOPS message; if the receiver fails to verify the message, it response with an error message.

The format of "cookies" is left for future versions of this document.

3.2 Reliable Communication

We expect TCP to provide us with reliable, in-order delivery of packets, as well as information on the liveliness of the communication channel. Given that TCP is responsible for all the time critical network operations, reliability errors are assumed to be virtually nonexistent. However, to maintain application-level reliability, OOPS uses a minimalistic reliability mechanism using sequence numbers, selective retransmit and keep-alive messages. This requires no retransmission timeouts, and has low overhead.

[5] The Error-Description provides reasons for rejecting the secure communications request.

3.2.1 Sequence Numbers

Each OOPS message, except a Resend-Notification, is uniquely identified by a sequence number[6] ($Mseq$). These numbers do not imply any order of execution; while the server receives messages in-order, it is free to execute them in any reasonable order.[7] In addition, each message also carries the sequence number of the last received message ($Rseq$). Both client and server begin communication with $Mseq = 0$ (the handshake message), and number consecutive messages in increasing order.

A transmitted message with $Mseq = m$ is considered to be acknowledged if $m \leq Rseq$ ($Rseq$ from the latest received message).[8] The sender must be prepared to retransmit (as requested) any message that has not been acknowledged yet. Missing, or out-of-order messages are identified by a gap in sequence numbers of received messages.

3.2.2 Receiver initiated retransmit

When the receiver (client or server) detects missing messages it immediately sends an explicit Resend-Notification listing these messages. The Resend-Notification has a sequence number 0.[9] Upon receiving the Resend-Notification, the sender must retransmit all the requested messages before sending new ones.

3.2.3 Keep-Alive Messages

Many platforms provide system support for detecting broken TCP connections. OOPS can utilize, but does not depend on such mechanisms. Instead, it relies on Keep-Alive messages to provide application-level communication-channel verification, as a server may be in a dysfunctional state while its TCP connection is still open and viable.

The client sends a Keep-Alive message to the server only after the receiving channel has been idle for longer than the Keep-Alive period. The server responds promptly with a Keep-Alive ack.

---

[6] Not counting wraparounds

[7] Execution order is implementation and policy specific; any order that does not violate the policy specific requirements is assumed to be reasonable.

[8] $Mseq \leq Rseq$ should take into account possible wrap-around of sequence numbers.

[9] Thus, Resend-Notification cannot participate in sequence number reliability verification. A lost Resend-Notification cannot not be detected, however, a new one is bound to be triggered sometime again.

3.2.4 Overhead

These reliability mechanisms were designed to be simple and impose minimal overhead in a busy working environment. When the client supports a large number of RSVP sessions and has frequent message exchange with the server, it would not be sending Keep-Alive messages. Similarly, since TCP is used for reliable communications, there is a virtually zero probability that Resend-Notification messages would be required. The only timer required is for the Keep-Alive period; the timer is reset on each message arrival and a Keep-Alive message is initiated only when it expires.

3.3 Connection Termination

This section describes how communication breakdown is handled.

3.3.1 Explicit Termination

The client (or server) may terminate the connection by sending a Bye-Notification, and wait until either it receives an echoed Bye-Notification or a Keep-Alive period had passed. In between, it should ignore incoming messages (and not reset the Keep-Alive timer).

At the opposite side, when a client (or server) receive a Bye-Notification message, they should echo it, and close the connection.

After an explicit termination, both client and server may cleans up and purges the state related to the closed connection.

3.3.2 Implicit Termination

The communication channel may be unexpectedly disconnected because of a misbehaving client or server, network split, or other reasons. Both client and server must be able to detect such channel failures and act accordingly.

Consider the case where OOPS is used for quota enforcement. The server may approve a reservation while debiting $\$X/min$ from a local account. If the OOPS communication channel breaks, it is critical for the server to detect it and stop debiting this account.

A communication channel is assumed to be disconnected when the channel was idle (no message was received on it) for over two Keep-Alive periods.

3.3.3 Post Termination

Soft-state has an inherent cleanup mechanism; when the channel disconnects, the soft-state would age and eventually expire based on the same mechanism and refresh-period used by RSVP.

When hard-state is used, cached state is assumed to be valid unless explicitly modified. However, when the channel disconnects such an explicit notification is not possible. Purging all state immediately upon disconnection is not an acceptable approach since it may cause a disruption of service before an alternate server is contacted. OOPS uses the following simple rule:

When the communication channel disconnects, the hard state associated with it is assumed to be soft-state that was just refreshed.

Naturally, when any RSVP state changes (e.g., routing changes, policy input changes, etc.), cached results at the client should not be used and must be purged.

3.3.4 Switching to An Alternative Server

We assume that the client is provided a list of policy servers and site specific selection criteria.

A switch to an alternate server may be triggered by a voluntary disconnection (i.e., Bye-Notification) or an unexpected break in the communication channel.

During normal operations, the client may wish to switch to an alternate server (for any reason). The client is advised to first connect to the new server before sending a Bye-Notification to the original one. If the communication channel unexpectedly disconnects, the client should quickly attempt to connect to an alternate server.

In both cases, after the connection to a new server[10] is established, the aging cached state from the old server would be gradually replaced by responses from the new server.[11] As general guidelines, state replacement from a new server should not cause a disruption of service that would not otherwise occur (if a new server was not found).[12]

[10] The term "new server" may be the same as the "previous server"; it may happen that the connection encounters a problem and the client chooses to disconnected and re-established the connection.

[11] The client could speed-up replacement of cached state by sending copies of cached input to the server and issuing repeated queries, on connection establishment (instead of waiting until objects arrive from RSVP).

[12] Practically, this means that as long as there is no change in RSVP messages, the client is advised to choose between cached and new results in favor of authorizing the request.

4 OOPS Message Format

OOPS messages serve as a wrapper that may include one or more protocol operations; this wrapper allows common operation (e.g., MD5 integrity, RSVP_HOPs, protocol version, etc.) to be verified and performed in one-shot.

```
+---------------+---------------+---------------+---------------+
|    Vers       |   Flags       |    op-objs#   | Reserved (0)  |
+---------------+---------------+---------------+---------------+
|                        Message Length                         |
+---------------+---------------+---------------+---------------+
|                   Message Sequence Number                     |
+---------------+---------------+---------------+---------------+
|                   Ack-ed Sequence Number                      |
+---------------+---------------+---------------+---------------+
|                      INTEGRITY Object... (optional)           |
+---------------+---------------+---------------+---------------+
|                      List of operations                       |
+---------------+---------------+---------------+---------------+
```

Any OOPS message is composed of the following fields:

Version: 8 bits

Protocol version number. The current version is 1.

Flags: 8 bits

```
0x01 H_Integrity_Checked      Integrity already checked by client
0x01 H_Hops_Checked           RSVP\_HOPs already checked by client
```

op-objs#: 8 bits

Number of objects included in this message.

Message Length: 32 bits

The total length of this OOPS message in bytes.

Message Sequence Number: 32 bits

The sequence number of the message being sent.

Ack-ed Sequence Number: 32 bits

The sequence number of the last message received in-order from the peer entity (client or server).

RSVP INTEGRITY Object (optional): variable length

This object is defined in [Bak96]. It provides a message digest based on a shared key between the client and sender. The message digest is calculated over the entire OOPS message.

List of OOPS operations: variable length

Described in the following section.

4.1 OOPS Operations

Each OOPS message may contain multiple OOPS operations each encapsulating a different query, response or notification. For example, multiple Incoming-Policy-Queries might be followed by a Status-Query operation in the same message. Operations within an OOPS message are sequentially numbered.

Individual OOPS operations have the following header:

```
+---------------+---------------+---------------+---------------+
| Operation Type| Op. Subtype   | Op. Seq#      |     Flags     |
+---------------+---------------+---------------+---------------+
|                     Length (bytes)                            |
+---------------+---------------+---------------+---------------+
|               |  RSVP's Refresh Period                        |
+---------------+---------------+---------------+---------------+
```

The operation header has the following fields:

operation Type: 8 bits

The type of OOPS operation.

Operation Subtype: 8 bits

This field can be used to indicate an attribute of the operation type, such as its version; currently it is always set to 1.

Operation Sequence Number: 8 bits

The operation sequence number within the message.

Flags: 8 bits

```
0x01 OOPS_HardState:   Hard State (soft-state if not set (0) )
0x02 OOPS_Shared   :   Resv shared among sources as filter specs
0x02 OOPS_FullList :   Last in the set of status queries.
```

Length: 32 bits

Contains the total operation length in bytes.

RSVP's Refresh Period

The refresh-period RSVP associates with this object.

This remainder of this section describes the set of operations that may appear in OOPS messages. Many data fields of these operations are RSVP objects; they are typed in uppercase letters and their format is defined in [RSVPSP]. The format of other operations is listed in the following section.

4.1.1 Null-Notification (a.k.a Keep-Alive)

```
Operation Type = 0, sub-type = 0
```

```
<Null-Notification> ::= <Common OOPS header>
```

This empty or null notification triggers no operation; thus, can be used as as Keep-Alive signal to test the viability of the communication channel between client and server (see Section 3.2.3).

4.1.2 Connection-Initiation-Query

```
Operation Type = 1, sub-type = 1
```

```
<Connection-Initiation-Query> ::=  <Common OOPS header>
                                   <Ver> <RSVP-K> <Flags>
                                   <Client-ID>
                                   <Max-Pkt-Size> <Keep-Alive period>
                                   <Class Indicator>
                                   <Cookie>
```

The client sends this query to establish a connection with a server. This message is sent following the establishment of a transport connection (TCP).

4.1.3 Connection-Accept-Response

```
Operation Type = 2, sub-type = 1
```

```
<Connection-Accept-Response> ::=  <Common OOPS header>
                                  <Max-Pkt-Size> <Keep-Alive period>
                                  <Policy list>
                                  <Cookie>
```

The server sends this response to accept a client's connection connection request.

4.1.4 Connection-Reject-Response

```
Operation Type = 3, sub-type = 1
```

```
<Connection-Reject-Response> ::=  <Common OOPS header>
                                  <Error-Description>
```

The server sends this response to reject a client's connection initiation. It specifies both reason code and text.

4.1.5 Bye-Notification

```
Operation Type = 4, sub-type = 1
```

```
<Bye-Notification> ::= <Common OOPS header>
```

This message is used by either client or server to terminate the OOPS connection. (Section 3.3.1 includes a description of explicit termination)

4.1.6 Incoming-Policy-Query

```
Operation Type = 5, sub-type = 1
```

```
<Incoming-Policy-Query> ::=  <Common OOPS header>
                             <RSVP MESSAGE TYPE>
                             <SESSION>
                             <FILTER_SPEC list> <RSVP_HOP>
                             <resv_handle> <RESV_FLOWSPEC>
                             <counter (of in P.D.)>
                             <in POLICY_DATA objects>
```

This operation is used to forward POLICY_DATA objects from the client to the server. Selection between hard and soft state management is reflected in the `OOPS_HardState` flag. The other fields are copied from the `PC_InPolicy()` function called by RSVP. (See [Ext]).

4.1.7 Incoming-Policy-Response

```
Operation Type = 6, sub-type = 1

<Incoming-Policy-Query> ::=  <Common OOPS header>
                             <Query Sequence Number>
                             <Error-Description>
```

Incoming-Policy-Response is used ONLY to report protocol errors (e.g., syntax) found with incoming policy objects. (it is not used in the normal operation of the protocol).

The `<Query Sequence Number>` links the response to the original query.

4.1.8 Outgoing-Policy-Query

```
Operation Type = 7, sub-type = 1

<Outgoing-Policy-Query> ::=  <Common OOPS header>
                             <RSVP MESSAGE TYPE>
                             <SESSION>
                             <FILTER_SPEC list>
                             <counter (of RSVP_HOPs)>
                             <RSVP_HOP list>
```

This operation queries the server for a set of outgoing policy objects for a set of RSVP_HOPs. The client can choose between hard and soft state management through the `OOPS_HardState` flag. When hard state is selected, the client caches copies of the outgoing objects and assumes they remain valid unless explicitly modified by the server.

4.1.9 Outgoing-Policy-Response

```
Operation Type = 8, sub-type = 1

<Outgoing-Policy-Response> ::=  <Common OOPS header>
                                <Query Sequence Number>
                                <Counter (of triplets)>
                                { <RSVP_HOP>
                                  <Error-Description>
                                  <out POLICY_DATA objects>
                                } pair list
```

The `<Query Sequence Number>` links the response to the original query.

In the response, the server provides a list of triplets, one for each outgoing RSVP_HOP

(For Path messages, only the LIH part is significant). Each triplet contains a list of policy objects for that hop and an error description.

4.1.10 Status-Query

```
Operation Type = 9, sub-type = 1

<Status_Query> ::=  <Common OOPS header>
                    <RSVP MESSAGE TYPE>
                    <SESSION>
                    <FILTER_SPEC_LIST>
                    <counter (of Triplets)>
                    { <LIH> <resv_handle> <RESV_FLOWSPEC> }
```

This operation queries the server for status results of a list of LIHs. The client can choose between hard and soft state management through the `OOPS_HardState` flag. When hard state is selected, the client caches the status results and assumes they remain valid unless explicitly modified by the server.

In the upstream direction (e.g., Resv) status may need to be checked on multiple LIHs (all reservations for a flow). In such cases, status queries can be perform separately for each LIH, once for all LIHs, or anything in between. Flag `OOPS_FullList` must be set at the last of status query of the series.[13]

4.1.11 Status-Response

```
Operation Type = 10, sub-type = 1

<Status_Response> ::=  <Common OOPS header>
                       <Query Sequence Number>
                       <Counter (of triplets)>
                       { <LIH>
                         <Status Result>
                         <Error-Description>
                       } pair list
```

The `<Query Sequence Number>` links the response to the original query.

In the response, the server provides a list of triplets, each of which contains an LIH, status, and any applicable error results. The set of LIHs is an attribute of the results and not of

[13] When policies are interdependent across LIHs (as when the cost is shared among downstream receivers), flag `OOPS_FullList` notifies the server that the list of reserved LIH is complete and that it can safely compute the status of these reservations.

the query; the server is allowed to respond with a superset of LIHs specified in the original query, as in the following example:

```
                    Seq#  Type            Data
                    ---   ----            ----
Client ==> Server:  150   Query:status    Resv-X, LIH={2}
Server ==> Client:  153   Resp :status    #150:{2,rej}

Two new reservations arrive, carrying new policy data objects:

Client ==> Server:  160   Query:status    Resv-X, LIH={4,7}
Server ==> Client:  169   Resp :status    #160:{2,acc;4,acc;7,rej}
```

4.1.12 Delete-State-Notification

```
Operation Type = 11, sub-type = 1

<Delete-State-Notification> ::=  <Common OOPS header>
                                 <RSVP MESSAGE TYPE>
                                 <SESSION>
                                 <FILTER_SPEC_LIST>
                                 <RSVP_HOP>
                                 <Op-type>
```

This operation informs the sender about an immediate RSVP teardown of state caused by PATH_TEAR, RESV_TEAR, routes change, etc. As a result, the server should ignore the described state as if it was never received from the client.

Despite its name, this operation can be used to switch between blockaded and non-blockaded state.

The semantics of this operation is described for `PC_DelState()` in [Ext].

4.1.13 Schedule-RSVP-Notification

```
Operation Type = 12, sub-type = 1

<Schedule-RSVP-Notification> ::=  <Common OOPS header>
                                  <RSVP MESSAGE TYPE>
                                  <SESSION>
                                  <FILTER_SPEC list>
                                  <RSVP_HOP>
```

The operation results in the generation of an outgoing RSVP message (Path, Resv, etc.) in the client's RSVP. RSVP should schedule the requested message to the specified RSVP_HOP.

4.1.14 Client-Status-Notification

```
Operation Type = 13, sub-type = 1

<Client-Status-Notification> ::=  <Common OOPS header>
                                  <Query Sequence Number>
                                  <Status Result>
```

The Client notifies the server about the status results computed at the client (that may also include results from other servers, if policy computation is spread among several servers).

The overall status of an RSVP flow is computed by merging the client's status report with the server's. The server should not commit a transaction (e.g., charge an account) before knowing its final status. The Client-Status-Results operation can be sent with the query, if the client computed its status prior to making the query. It can also be sent later, after the server sent its response to the status query.

4.1.15 Resend-Notification

```
Operation Type = 14, sub-type = 1

<Resend-Notification> ::= <Common OOPS header>
                          <Counter (of missing messages)>
                          <Message sequence number> list
```

Both client and server may issue a `Resend-Messsage` request when they detect missing or out-of-order messages. The Resend-Notification has message sequence number 0. The message explicitly lists the sequence numbers of all missing messages. Notice that since OOPS uses a reliable transmission protocol this list should never be long. (See Section 3.2).

4.1.16 Error-Notification

```
Operation Type = 6, sub-type = 1

<Error-Notification> ::=  <Common OOPS header>
                          <Message Sequence Number>
                          <Error-Description>
```

Error-Notification can be used by either client or server to report errors associated with an entire message (as opposed to a specific operation). Error-Notification may be triggered by both syntax or substantive errors (e.g., failure to verify the integrity of a previous message).

`<Message Sequence Number>` identified the message that triggered the error.

Error-Notification is not acked.

4.2 Fields format

- <Ver><RSVP-K><Flags>

```
+---------------+---------------+---------------+---------------+
|   Version     | RSVP-K        |   Flags       |       0       |
+---------------+---------------+---------------+---------------+
```

  Ver: Currently, version 1.

  RSVP-K: The K value used by RSVP as a refresh-period multiplier.

  Flags:

```
0x01  OOPS_CONNECT_DefaultC     Client handles default policies.
```

- <Max-Pkt-Size><Keep-Alive period>

```
+---------------+---------------+---------------+---------------+
| Max-Pkt-Size (in KBytes)      | Keep-Alive period (in seconds)|
+---------------+---------------+---------------+---------------+
```

- <Class Indicator>

```
+---------------+---------------+---------------+---------------+
|             Length (total)    |         Class Code            |
+---------------+---------------+---------------+---------------+
|   ASCII String ........ 0 Padded to multiples of 32 bits      |
+---------------+---------------+---------------+---------------+
```

- <Client-ID>

  Client address, uses the same format as RSVP's FILTER_SPEC objects.

  From the combination of Client-ID and Class-Indicator the server can learn about the set of policies it is required to support for this particular client.

- <Cookie>

```
+---------------+---------------+---------------+---------------+
|           Length (total)      |    Type       |       0       |
+---------------+---------------+---------------+---------------+
|   Octet String ........ 0 Padded to multiples of 32 bits      |
+---------------+---------------+---------------+---------------+
```

Currently, no values are defined.

- <Policy list>

```
+---------------+---------------+---------------+---------------+
|         Number (or pairs)     |              0                |
+---------------+---------------+---------------+---------------+
|         From Policy 1         |         To Policy 1           |
+---------------+---------------+---------------+---------------+
....
+---------------+---------------+---------------+---------------+
|         From Policy n         |         To Policy n           |
+---------------+---------------+---------------+---------------+
```

Each "From Policy m" and "To Policy m" pair represent a range of policies that the server is willing to support.

- <Error-Description>

```
+---------------+---------------+---------------+---------------+
| Length (*)    | Error-Type    |        Reason Code            |
+---------------+---------------+---------------+---------------+
|   Error ASCII String .... 0 Padded to multiples of 32 bits    |
+---------------+---------------+---------------+---------------+
```

(*) Length of the overall <Error-Description> in 4 bytes increments (i.e., length value of X should be interpreted as $X * 4$ bytes description and an $(X - 1) * 4$ bytes Error ASCII String.

No errors are reported by setting the length to 1 (4 bytes) and setting the Error-Type to 0.

Detailed Error-Types and Reason-Codes would be defined in future versions of this document.

- <resv_handle>

```
+---------------+---------------+---------------+---------------+
|               IntServ or Client-Specific Semantics            |
+---------------+---------------+---------------+---------------+
```

The server may use the `<resv_handle>` to obtain IntServ and other low-level information about the reservation.

The current version of this document does not define the semantics of this field. It may be a pointer into some router specific data structures (proprietary) or an index into mib records obtainable through SNMP.

- <Query Sequence Number> (and internally, <Message Sequence Number>)

```
+---------------+---------------+---------------+---------------+
|               <Message Sequence Number>                       |
+---------------+---------------+---------------+---------------+
| Obj. Seq. Num.|                     0                         |
+---------------+---------------+---------------+---------------+
```

- <Counter>

```
+---------------+---------------+---------------+---------------+
|                          <Counter>                            |
+---------------+---------------+---------------+---------------+
```

- <Status Result>

```
+---------------+---------------+---------------+---------------+
| Results       |                     0                         |
+---------------+---------------+---------------+---------------+
```

Results may have one of the following values:

1 : `Accept`
2 : `Snub`
3 : `Veto`

- <Op-Type>

```
+---------------+---------------+---------------+---------------+
| Mod-Type      |                      0                        |
+---------------+---------------+---------------+---------------+
```

Op-Type values:

1 : `Delete State`
2 : `Block State`
3 : `Unblock State`

5 Acknowledgment

This document reflects feedback from many other RSVP collaborators.

References

[Bak96] F. Baker. RSVP Cryptographic Authentication *Internet-Draft*, draft-ietf-rsvp-md5-02.txt, 1996.

[RSVPSP] R. Braden, L. Zhang, S. Berson, S. Herzog, and S. Jamin, Resource ReSerVation Protocol (RSVP) Version 1 Functional Specification. *Internet-Draft*, draft-ietf-RSVPSP-14.[ps,txt], Nov. 1996.

[Arch] S. Herzog Accounting and Access Control Policies for Resource Reservation Protocols. *Internet-Draft*, draft-ietf-rsvp-policy-arch-01.[ps,txt], Nov. 1996.

[LPM] S. Herzog Local Policy Modules (LPM): Policy Enforcement for Resource Reservation Protocols. *Internet-Draft*, draft-ietf-rsvp-policy-lpm-01.[ps,txt], Nov. 1996.

[Ext] S. Herzog RSVP Extensions for Policy Control. *Internet-Draft*, draft-ietf-rsvp-policy-ext-02.[ps,txt], Apr. 1997.

Authors' Address

Shai Herzog	Phone: (914) 784-6059
	Email: herzog@watson.ibm.com
Dimitrios Pendarakis	Phone: (914) 784-7536
	Email: dimitris@watson.ibm.com
Raju Rajan	Phone: (914) 784-7260
	Email: raju@watson.ibm.com
Roch Guérin	Phone: (914) 784-7038
	Email: guerin@watson.ibm.com

IBM T. J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598