Internet Draft
Expires January 30, 1998
File: draft-ietf-rsvp-policy-oops-01.ps

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

Open Outsourcing Policy Service (OOPS) for RSVP

July 30, 1997

Status of Memo

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].

1 Overview

Reservation protocols function by discriminating between users by providing some users with better service at the expense of others. The utility of reservation protocols is sharply degraded in the absence of mechanisms for restricting access to higher service categories and enforcing network and bandwidth usage criteria. In this document, we refer to such mechanisms as *policy control*. This term is quite broad; it ranges from simple access control to sophisticated accounting and debiting mechanisms.

The policy control component may reside fully within the router (as an add-on module to RSVP). However, it is often advantageous for routers to outsource some of their policy decision making to external entities. Open Outsourcing Policy Service (OOPS) is a protocol for exchanging policy information and decisions between Local Policy Modules (LPMs) located within RSVP-capable routers and one or more external policy servers. OOPS 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. Moreover, the OOPS protocol supports both partial and complete delegation of policy control.

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 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 sub-policy elements and only when things change (otherwise, cache can be used).

Support for different policy interfaces

The OOPS protocol allows clients and servers to negotiate the nature and sophistication of their interaction. For instance, responses from the server to the client may be restricted to allow the server to merely accept, deny or remain neutral on reservation requests, while a more sophisticated implementation may allow the server to respond with preemption priorities or other characteristics of the reservation. The negotiation handshake is simple, and may always fall back onto the lowest level of interaction that must always be supported.

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). Moreover, OOPS functionality is not tied to that of RSVP, and OOPS may be extended to be used by other, non-RSVP, connection setup protocols.

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 Glossary

Policy

Comprehensive set of rules for controlling some aspects of the network.

Sub-policies

Modular building blocks out of which comprehensive policies are compiled.

POLICY_DESC

Data representation of policy information (e.g., POLICY_DATA objects in RSVP).

Sub-policy element

Data representation of sub-policy information, as encapsulated in POLICY_DESC objects.

1.2 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

Policy objects are carried in RSVP messages along the path consisting of four typical node types:

(1) Policy incapable nodes: Node B. (2) Self-sufficient policy node: Node D does not need to outsource policy tasks to external servers since its LPM satisfies its entire policy needs. (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 Node E to PS-2 a direct link or a LAN. (4) "Smart" policy nodes: Nodes A and C include sophisticated LPMs, in that these nodes can process some sub-policy elements, and have the capacity to cache responses from PS-1. In this case, the contact between the clients and server would 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.

The functioning of the policy module at a policy aware router is presented through the following conceptual diagram.

```
+---------+          +----------+
|  RSVP   |          |  Policy  |
|  Module |          |  Server  |
+---------+  OOPS    |          |
|  LPM    |----------|          |
+- - - - -+          +----------+
|PSM| |PSM|          |PSM|  |PSM|
|___| |___|          |___|  |___|
```

Figure 2: Local Policy Modules and Policy Server communications

The policy server and the local policy module provide support for a number of sub-policy elements, each embodied by a policy sub-module (PSM). The policy object forwarded by RSVP may contain a number of elements, each identified by a number, and hence destined to the sub-module that enforces that sub-policy element's number. For instance, some of these sub-objects may deal with authentication, others with security, accounting and so on. The LPM is aware of the sub-modules it is capable of processing locally; After the handshake comes to know the set of sub-policies that are supported by the server. Processing of policy sub-objects can be split between the LPM and the policy server, and responses may be merged back before returning a unified response to RSVP.

2 OOPS Protocol: Basic Features

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 association, so that responses may identify the query to which they correspond.

This section discusses four fundamental concepts of the OOPS protocol: (a) query/response mechanism, (b) flexible division of labor between client and server, and (c) consistent management of client, server and RSVP state.

2.1 Query/Response mechanism

Each OOPS message is uniquely identified by a sequence number; Both client and server begin communication with $Mseq = 0$ (the handshake message), and number consecutive messages in increasing order. These sequence numbers do not imply the order of execution; while the server receives messages in-order, it is free to execute them in any reasonable order.[1]

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

These sequence numbers are mainly used by the `Error-Notification` operation as a means to identify the message that is associated with the reported error.[2]

2.1.1 Associating Queries and Responses

Queries and responses carry a Q_ASSOC object which relates newly received responses to their original query operations. The contents of this object is client-specific and therefore opaque to the server; it is set by the client for each query and is echoed back as-is by the server. The client must store enough information in the Q_ASSOC object to enable its own unique identification of the original query.

2.2 Division of Labor between Client and Server

The OOPS protocol allows for a flexible division of responsibilities between server and client. First, the client must be able to decide how to distribute the processing and second, it must be able to merge the distributed responses into one unified result.

2.2.1 Distributed Processing

Processing of sub-policies (sub-policy elements within POLICY_DESC objects) can be performed by the server, the client, or by both. The decision on which sub-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).

The client may remove sub-policy elements which are not to be processed by the server. In this case, the client is solely responsible for checking the integrity of the incoming policy object;[3] the client must also set the OP-Code header flag to inform the server to that fact.

During connection establishment, the server may request to have oversight over the clients local decisions; in this case, the client should forward incoming policy objects in their entirety, and consult the server for all RSVP flows, regardless of whether they include POLICY_DATA objects. This oversight is transparent to the client and is therefore post factum.[4]

[2]Senders must be informed about the receiver's failure to process their messages. This is especially critical given that OOPS relies on TCP's reliability and lacks additional reliability mechanisms.

[3]If any portion of the POLICY_DESC object is modified, the digest integrity verification at the server is bound to fail.

[4]The client should not wait for an oversight decision; if the server overrides a local decision, it may notify the client sometime later, even after the local client authorized the RSVP operation.

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.

2.2.2 Unification of Distributed Responses

Division of labor between client and server is only possible to the extent that the client has the capability to unify or merge results; the client must be able to merge the results of queries arriving from servers with its own local results, to produce a single unified response to the underlying protocol (e.g., RSVP).

Results unification is straight-forward for outgoing POLICY_DESC object; since sub-policy elements are independent, their unification is performed by concatenating all local and server elements and packing them in POLICY_DESC objects.[5]

Unification is more complex for status queries, since the various responses must truly be merged to produce a single status result. OOPS defines one basic (default) status response interface (object and unification rules).

However, given that OOPS is an extensible framework, it allows the the client and server to negotiate a more sophisticated interface (see Section 3.1). Additional response interfaces could be described in separate documents which should define the response object format and unification rules.[6]

2.2.3 Default Status Response

The default status response object is of the C-Type 1. C-Type 1 objects may contain two values: a policy admission decision (PAD) and a preemption priority value (PP). It is reasonable to assume that some clients would not be able to utilize the flow preemption priority information; such clients are free to ignore this value and assume that all flows are created equal. (have priority 0).

PADs may have one of three values: ACCEPT, SNUB, and VETO. ACCEPT authorizes the query, SNUB signifies neutrality (neither accept nor reject). A VETO from the server or LPM has a stronger semantics than a snub, since it has the power to forcefully reject a flow regardless of any accept decisions made by the other.

The rules for unification of PAD values A and B are straight-forward:

[5] An oversight sub-policy element would override the locally generated element, if the two are of the same type.

[6] A separate template document and a list of more sophisticated responses should be prepared.

A+B	IF...
SNUB	A=SNUB and B=SNUB
VETO	A=VETO or B=VETO
ACCEPT (+PP value)	Otherwise

A unified result of ACCEPT provides approval for the status query; both SNUB and VETO signal the rejection of the query.

Note that a client and/or server should complete their policy processing even if a veto was cast by some policy.[7]

An ACCEPT response is accompanied by a PP value between 0..255. Lower values describe higher priorities (priority 1 is the highest). The value 0 is reserved for "N/A"; this value is used when preemption priority is not applicable.

The unification of PP values A and B attempts to provide the highest priority (lowest value) which is supported by an ACCEPT decision. The value 0 has no effect on the unified priority:

A+B	IF...
MIN(A,B)	A!=0 and B!=0
A	B=0
B	A=0
0 (n/a)	A=0 and B=0

2.3 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. During normal operations, the state maintained in the client and in the server must remain consistent, and must timeout at roughly the identical times in RSVP, the client, and the server.

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 state updates only. To support the

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

hard-state mode, the client must be able to distinguish between repeats (refreshes) and updates; it must also be able to translate the soft-state that is provided by RSVP into 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, split the work between themselves and the server, utilize caching capabilities. 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 separately:

2.3.1 Client State Information Cached at the Server

The client indicates its choice of state management approach by setting (or resetting) the `OOPS_HardState` flag in objects sent to the server. When the client chooses soft-state management, policy state for that specific object ages and expires at the server according to the specified timeout (refresh-period * K). Therefore, the state cached at the server is kept alive by constant refreshing (the client must forward ALL incoming RSVP messages, whether or not they represent refreshes or updates). On the other hand, when indicating a choice of hard-state management, the client 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.[8] The client may refrain from forwarding to the server any repeat policy objects (which represent no updated information).

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

[8] Clearly the channel breaks when either the client or server become disfunctional or die.

2.3.2 Server State Information Cached at the Client

The client indicate its state management capabilities by setting (or resetting) the `OOPS_HardState` flag in queries sent to the server. A choice of soft-state indicates that the client is incapable of caching, and it purges the server responses after usage (one-time, or disposable results). Clearly, without caching, the client must issue a new 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 with the original response to this query. The client may rely on cached results as long as there is no change in RSVP's state (which includes incoming policy objects),[9] 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.

2.3.3 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_DESC objects for Resv-X):

```
                            TYPE            DATA
                            ----            ----
CLIENT ==> SERVER:          NOTIFY:INPUT    RESV-X: PD-1
```

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

```
CLIENT ==> SERVER:          NOTIFY:INPUT    RESV-X: PD-2
```

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

```
                            TYPE             DATA
                            ----             ----
CLIENT ==> SERVER:          QUERY:STATUS    Q_ASSOC=ID1, RESV-X
SERVER ==> CLIENT:          RESP :STATUS    Q_ASSOC=ID1, ACCEPT
```

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

```
SERVER ==> CLIENT:          RESP :STATUS    Q_ASSOC=ID1, REJECT
```

[9] A configurable option may allow the client to use cached results even when some RSVP state changes. There is a clear trade-off here 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.

2.3.4 State Re-synchronization

Both client and server may re-synchronize their respective states at any time during the connection. The reset initiator sends a Bye-Notification with a RESET code, and the receiver responds with a Bye-Notification with the same code. After this exchange, all cached state becomes soft, and a new logical connection is reestablished (beginning with Connection-Initiation-Query,...). New/hard state gradually replaces old/soft state as described in Section 3.2.3.

2.4 Error Handling

We distinguish between two types of possible errors; policy errors and protocol errors.

2.4.1 Protocol Errors

Protocol errors (e.g., missing or bad parameters) do not reveal either positive or negative policy decisions and are therefore neutral (represented as SNUBs).[10]

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 Op-Code.)

Some OOPS protocol errors may only affect the OOPS protocol processing or simply be logged. Other errors may escalate to become policy errors (e.g., a bad POLICY_DESC is reported as a policy error).

2.4.2 Policy Errors

Policy errors are reported in a sub-policy element specific format. These elements are encapsulated in POLICY_DESC objects and are forwarded toward the originator (cause) of the error. In most cases, a negative Status-Response initiates an automatic error response (e.g., RSVP ResvErr or PathErr), however, OOPS allows reporting of other error situations by scheduling an explicit error message (using the Protocol-Message-Notification op-code). (See [Ext] for more about the rules governing error reporting).

[10] This neutrality allows, when appropriate, other valid sub-policy elements to support an accept decision.

Consider a scenario where two receivers R1 and R2 listen to a multicast transmission from S1. A reservation sent by R1 is propagated upstream until it reaches node A, where it encounters a policy rejection.

```
R1------------+
               \
                B-------------A---------- S1
               / \            |
R2------------+   \           |
                   \          |
                   PS1       PS2
```

Figure 3: An Error Reporting Scenario

The following table describes a subset of the relevant signaling which begins with reservation initiation by R1 and R2 and ends by R1 receiving the appropriate error response.

From/To	Message	Comments
R1->B	Resv [PD1]	
R2->B	Resv [PD2]	
B->PS1	OOPS-Incoming-Policy-Query[PD1,PD2]	;B queries PS1
	OOPS-Status-Query?	
	OOPS-Outgoing-Policy-Query? [Resv]	
PS1->B	OOPS-Status-Response: ACCEPT	
	OOPS-Outgoing-Policy-Response[PD3]	
B->A	Resv [PD3]	;B forwards the Resv to A
A->PS2	OOPS-Incoming-Policy-Query[PD3]	;A queries PS2
	OOPS-Status-Query?	
PS2->A	OOPS-Status-Response: SNUB (reject)	;PS2 reject the reservation
A->PS2	OOPS-Outgoing-Policy-Query? [ResvErr]	;PS2 provides error PD
PS2->A	OOPS-Outgoing-Policy-Response [PD1-E]	
A->B	ResvErr [PD1-E]	;A sends back ResvErr to B
B->PS1	OOPS-Incoming-Policy-Query[PD1-E]	
	OOPS-Outgoing-Policy-Query? [ResvErr]	;PS1 builds error PD
PS1->B	OOPS-Outgoing-Policy-Response[PD1-E'],R1	; (directed to R1 only)
B->R1	ResvErr [PD1-E']	;B sends back ResvErr to R1

Figure 4: Error Reporting Signaling

All error information is carried in POLICY_DESC objects (as sub-policy elements). OOPS server may read and modify this information along the ResvErr path; it may also direct the error responses only to the relevant branches of the reserved tree (in this scenario, the error is associated with R1 but not with R2).

3 Client-Server Connection

The following section describes the fundamentals of client-server connection: establishment, channel, and termination.

3.1 Connection Establishment

OOPS uses a well known port number (OOPS = 3288) for incoming connection requests. Usually, the client would attempt to establish a TCP connection to its preferred policy server, however, both client and server listen to the OOPS port.[11] Regardless of who initiated the TCP connection, once the connection is in place, the OOPS logical connection establishment is always initiated by the client and is performed 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 as well as a list of policy responses that are acceptable to the client. This list is in decreasing order of acceptability, and terminates with the default element.

- 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 sub-policies that the client can send to the server, which one of them should handle default (unrecognized) sub-policies, as well as the format of status responses from server to client. They also establish the Channel-Hold period which is determined as the minimum between the two values declared in the handshake messages, but must be at least 3 seconds.

3.1.1 Reliable Communication

We expect TCP to provide us with reliable, in-order delivery of packets. Given that TCP is responsible for all the time critical network operations, reliability errors are assumed to be virtually nonexistent.

[11] New (or recovering) policy servers are allowed to notify clients on their existence by issuing a TCP connection request to the client's OOPS port number.

3.1.2 Secure Communications

OOPS relies on standard protocols for security of client-server communications. An emerging standard protocol IPSEC [IPSEC] is the mechanism of choice for ensuring either integrity or secrecy. The use of IPSEC and/or other security protocols is transparent to OOPS.

3.2 Connection Termination

This section describes the handling of communication breakdown.

3.2.1 Implicit Termination

The communication channel may be unexpectedly disconnected because of a misbehaving client or server, network split, or for 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 the break and stop debiting this account.

The OOPS protocol relies on Keep-Alive messages to provide application-level communication-channel verification.[12] Implicitly, the communications channel is assumed to be disconnected after it has been idle (no message was received on it) for more than a Channel-Hold period (see Section 3.1). Keep-Alive messages are sent by both client and server as needed[13] to ensure the liveness of the connection (to prevent a Channel-Hold timeout). Keep-Alive messages are not acknowledged.

3.2.2 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 the Channel-Hold period had passed. In between, it should ignore incoming messages (and not reset the Channel-Hold timer).

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

[12] OOPS implementations may utilize system dependent mechanisms for detecting broken TCP connections, but does not rely on them. This is especially important since a server may be in a dysfunctional state while its TCP connection is still open and viable.

[13] When the intermediate period in between two OOPS messages approaches the Channel-Hold time.

3.2.3 Post Termination

Soft-state has an inherent cleanup mechanism; when the channel disconnects, the soft-state begins to age until it eventually expires (using the same mechanism and refresh-period * K used by RSVP).

In contrast, hard-state is assumed to be valid unless explicitly modified. However, when the channel disconnects such an explicit notification is not possible. Clearly, purging all state immediately upon disconnection is not an acceptable approach since should cause disruption of service and would not allow enough time to contact an ALTERNATE server. OOPS uses the following simple rule:

When the communication channel disconnects, all hard state associated with it is assumed to be soft-state that had been refreshed recently.

3.2.4 Switching to An Alternative Server

We assume that as part of their local configuration, clients obtain a list of policy servers and site specific selection criteria. This list can be the basis for server switching decisions.

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[14] is established, the aging cached state from the old server would be gradually replaced by responses from the new server.[15] 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).[16]

After switching to an alternate server, the client may periodically poll its old (preferred) server by attempting a TCP connection to its OOPS port. Similarly, a new (or recovered server) may notify clients about its liveness by attempting to connect to their OOPS port. In the latter case, clients may disconnect the TCP connection or respond with a Connection-

[14] 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.

[15] 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).

[16] 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.

Initiation-Query as if the client initiated the connection in the first place.[17]

4 OOPS Message Format

OOPS messages serve as a wrapper that may include one or more Op-Codes; the message wrapper allows common operation (e.g., MD5 integrity, HOP_DESCs, protocol version, etc.) to be performed and verified in one-shot. All OOPS messages are composed of the following fields:

```
+---------------+---------------+---------------+---------------+
|    Ver        | #Op-Codes     | Flags         |    //////     |
+---------------+---------------+---------------+---------------+
|                 Message Length                                |
+---------------+---------------+---------------+---------------+
|                 Message Sequence Number                       |
+---------------+---------------+---------------+---------------+
|                 OOPS_MSG_AUTH (Optional)                      |
+---------------+---------------+---------------+---------------+
|                 List of Op-Codes...                           |
+---------------+---------------+---------------+---------------+
```

Version: 8 bits

Protocol version number. The current version is 1.

Flags: 8 bits

```
0x01 H_Integrity_Checked      POLICY_DESC Integrity already checked by client
0x02 H_Hops_Checked           Prev/Next HOPs already checked by client
```

#Op-Codes: 8 bits

Number of Op-Codes 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.

OOPS_MSG_AUTH (optional): variable length

This Message Authenticator provides integrity verification based on a shared-keyed message digest. The message digest is calculated over the entire OOPS message.

There is only one object format currently defined is identical to the RSVP INTEGRITY object (defined in [Bak96]).

List of OOPS operation codes (Op-Codes): variable length

Described in the following section.

[17] Future version of this document may include the use of multicast to advertise the liveness of servers.

4.1 OOPS Operation Codes (Op-Codes)

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.

Individual OOPS Op-Codes have the following header:

```
+---------------+---------------+---------------+---------------+
| Operation Code| Op. Subtype   |  Flags        |    //////     |
+---------------+---------------+---------------+---------------+
|                      Length (bytes)                           |
+---------------+---------------+---------------+---------------+
|                      Refresh Period                           |
+---------------+---------------+---------------+---------------+
```

The operation header has the following fields:

operation Code: 8 bits

The type of OOPS operation.

Operation Subtype: 8 bits

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

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 (including header).

Refresh Period

The refresh-period associates with this object (e.g., RSVP's refresh period).

The remainder of this section describes the set of operations that may appear in OOPS messages and their object format. OOPS does not bind itself to a particular protocol (i.e., RSVP) and is built around objects that may belong to different (other) protocols. The current draft is based on the assumption that RSVP would be one (the first) of these protocols and thus, the draft provides the appropriate RSVP objects format.

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

```
Operation Type = 0, sub-type = 1

<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 3.2.1).

4.1.2 Connection-Initiation-Query

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

<Connection-Initiation-Query> ::=  <Common OOPS header>
                                   <CONNECT_DESC>
                                   <CLASS_ID>
                                   <CLIENT_ID>
                                   <RESP_INT>
                                   <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).

- CONNECT_DESC

  Description of connection parameters.

- CLASS_ID

  The client's class provides an implicit description of the client's capabilities and requirements; the CLASS_ID is an index into the class list maintained by the server; it is used in conjunction with the CLIENT_ID.)

- CLIENT_ID

  The network address of the client. From the combination of CLIENT_ID and CLASS_ID the server can learn about the set of sub-policies it is required to support for this particular client; it can also learn which of these sub-policies are optional and which are mandatory.

- RESP_INT

  A list of possible response interfaces.

- COOKIE

4.1.3 Connection-Accept-Response

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

<Connection-Accept-Response> ::=  <Common OOPS header>
                                  <CONNECT_DESC>
                                  <PLIST>
                                  <RESP_INT>
                                  <COOKIE>
```

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

- CONNECT_DESC
- PLIST

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

  The chosen (agreed upon) status response interface.
- COOKIE

4.1.4 Connection-Reject-Response

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

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

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>
                       <BYE_DESC>
                       [<ERR_DESC>]
```

This message is used by either client or server to terminate the OOPS connection.

4.1.6 Incoming-Policy-Query

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

<Incoming-Policy-Query> ::=  <Common OOPS header>
                             <Q_ASSOC>
                             <PROT_MSG_TYPE>
                             <DST_DESC>
                             <SRC_DESC list>
                             <HOP_DESC>
                             [<ADV_DESC>]
                             <POLICY_DESC list>
```

This operation is used to forward POLICY_DESC 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-Response> ::=  <Common OOPS header>
                                <Q_ASSOC>
                                <ERR_DESC>
```

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).

4.1.8 Outgoing-Policy-Query

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

<Outgoing-Policy-Query> ::=  <Common OOPS header>
                             <Q_ASSOC>
                             <PROT_MSG_TYPE>
                             <DST_DESC>
                             <SRC_DESC list>
                             <HOP_DESC list>
```

This operation queries the server for a set of outgoing policy objects for a set of HOP_DESCs. 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>
                                <Q_ASSOC>
                                { <HOP_DESC>
                                  <ERR_DESC> or <POLICY_DESC>
                                } pairs 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 HOP_DESC (For Path messages, only the LIH part is significant). Each triplet contains a list of policy objects for that hop and an error description.

The OOPS server can block an outgoing RSVP message by replacing the outgoing POLICY_DESC list for a particular HOP_DESC with an <Error-Description> with an appropriate value.

The ability to block outgoing RSVP control messages is especially useful when policy is enforcement is performed at border nodes of a network; RSVP control messages that are allowed through are capable of installing state at internal nodes without being subject to further policy control.

4.1.10 Status-Query

Operation Type = 9, sub-type = 1

```
<Status_Query> ::=  <Common OOPS header>
                    <Q_ASSOC>
                    <PROT_MSG_TYPE>
                    <DST_DESC>
                    <SRC_DESC list>
                    { <HOP_DESC>
                      <QOS_DESC>
                    } triplets list
```

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.[18]

[18] When sub-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.

4.1.11 Status-Response

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

<Status-Response> ::=  <Common OOPS header>
                       <Q_ASSOC>
                       { <HOP_DESC>
                         <STATUS_DESC>
                         [<ERR_DESC>]
                       } triplet list
```

The <Q_ASSOC> 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 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   Q_ASSOC=ID2, Resv-X, LIH={2}
Server ==> Client:  153   Resp :status   Q_ASSOC=ID2, {2,rej}

Two new reservations arrive, carrying new policy data objects:

Client ==> Server:  160   Query:status   Q_ASSOC=ID3, Resv-X, LIH={4,7}
Server ==> Client:  169   Resp :status   Q_ASSOC=ID3, {2,acc;4,acc;7,rej}
```

4.1.12 Delete-State-Notification

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

<Delete-State-Notification> ::=  <Common OOPS header>
                                 <STATE_OP_DESC>
                                 <DST_DESC>
                                 [<PROT_MSG_TYPE>]
                                 [<SRC_DESC list>]
                                 [<HOP_DESC>]
                                 [<ERR_DESC>]
```

- STATE_OP_DESC

  This object describes the type of requested operation (see Appendix A).

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].

Error description is used to provide the server with a reason for the delete (for logging purposes).

4.1.13 Protocol-Message-Notification

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

<Protocol-Message-Notification> ::=  <Common OOPS header>
                                     <PROT_MSG_TYPE>
                                     <DST_DESC>
                                     <SRC_DESC list>
                                     <HOP_DESC>
```

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

4.1.14 Client-Status-Notification

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

<Client-Status-Notification> ::=  <Common OOPS header>
                                  <Q_ASSOC>
                                  <STATUS_DESC>
```

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 Error-Notification

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

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

Message-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 the message).

`<Message-Sequence-Number>` identified the message that triggered the error. It uses identical format to the one used by the OOPS message header.

Message-Error-Notification is not acked.

5 Acknowledgment

This document reflects feedback from Paul Amsden, Fred Baker, Lou Berger, Bob Braden, Ron Cohen, Deborah Estrin, Steve Jackowski, Tim O'Malley, Claudio Topolcic, Raj Yavatkar, and many other IPC and RSVP collaborators,

6 Authors' Address

Shai Herzog Phone: (917) 318-7938
IP*Highway* Email: herzog@iphighway.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

7 References

References

[IPSEC] R. Atkinson, Security Architecture for the Internet Protocol, *RFC1825*, Aug. 1997.

[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 ReSer-Vation 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.

A Appendix: OOPS Objects

This section describes objects that are used within OOPS OP-Codes. All objects have a common header:

```
+---------------+---------------+---------------+---------------+
|            Length             |     Class     |  C-Type       |
+---------------+---------------+---------------+---------------+
```

Length describes the length of the entire object, in bytes. Class describes the type of object and C-Type describes the a class sub-type.

- CLASS_ID class
  - Class = 1, C-Type = 1

```
+---------------+---------------+---------------+---------------+
|   ASCII String ........ 0 Padded to multiples of 32 bits      |
+---------------+---------------+---------------+---------------+
```

- CLIENT_ID class
  - Class = 2, C-Type = 1
    A Network Address.

```
+---------------+---------------+---------------+---------------+
|   IPv4 Address                                                |
+---------------+---------------+---------------+---------------+
```

  - Class = 2, C-Type = 2

```
+---------------+---------------+---------------+---------------+
|   IPv6 Address                                                |
|                                                               |
|                                                               |
|                                                               |
+---------------+---------------+---------------+---------------+
```

From the combination of Client-ID and Class-Indicator the server can learn about the set of sub-policies it is required to support for this particular client; it can also learn which of these sub-policies are optional and which are mandatory.

- RESP_INT class
  - Class = 3, C-Type = 1

```
+---------------+---------------+---------------+---------------+
| Most-Prefered |.....          |               |               |
+---------------+---------------+---------------+---------------+
|               | Least-Pref.   |...0 Padded to 32 bit multiples|
+---------------+---------------+---------------+---------------+
```

- COOKIE class

  - Class = 4, C-Type = 1
    Currently, no values are defined.

- PLIST class

  - Class = 5, C-Type = 1

```
+---------------+---------------+---------------+---------------+
|          Number (or pairs)    |             //////            |
+---------------+---------------+---------------+---------------+
|          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 sub-policies that the server is willing to support.

- ERR_DESC class

  - Class = 6, C-Type = 1

```
+---------------+---------------+---------------+---------------+
| Error-Code    |     //////    |         Reason Code           |
+---------------+---------------+---------------+---------------+
|   Error ASCII String .... 0 Padded to multiples of 32 bits    |
+---------------+---------------+---------------+---------------+
```

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

- Q_ASSOC class

  - Class = 7, C-Type = 1

```
+---------------+---------------+---------------+---------------+
|                      Client-Specific Semantics                |
//                        (Variable Length)                    //
|                                                               |
+---------------+---------------+---------------+---------------+
```

    The client-specific contents of this object is opaque to the server; it is set by the client for a query and is echoed by the server as-is. The client must store enough information there that will enable it to uniquely identify the original query when the response arrive. This must at least include a counter to identify the version of the latest query.[19]

- PROT_MSG_TYPE class

[19] A simple association could be the combination of a pointer to an internal client (router) control-block that describes the query, and a query version counter.

– Class = 8, C-Type = 1

```
+---------------+---------------+---------------+---------------+
|               RSVP MSG TYPE                                   |
+---------------+---------------+---------------+---------------+
```

Values specified in [RSVPSP].

- DST_DESC class
  - Class = 9, C-Type = 1
    The RSVP SESSION object as defined in [RSVPSP].
- SRC_DESC class
  - Class = 10, C-Type = 1
    The RSVP FILTER_SPEC object as defined in [RSVPSP].
- HOP_DESC class
  - Class = 11, C-Type = 1
    The RSVP_HOP object as defined in [RSVPSP].
- ADV_DESC class
  - Class = 12, C-Type = 1
    The RSVP ADSPEC object as defined in [RSVPSP].
- QOS_DESC class
  - Class = 13, C-Type = 1
    The RSVP FLOWDESC object as defined in [RSVPSP].
- POLICY_DESC class
  - Class = 14, C-Type = 1
    The RSVP POLICY_DATA object as defined in [Ext] and [RSVPSP].
- OOPS_MSG_AUTH class
  - Class = 15, C-Type = 1
    The RSVP INTEGRITY object as defined in [RSVPSP] and [Bak96].
- STATUS_DESC class
  - Class = 16, C-Type = 1

```
+---------------+---------------+---------------+---------------+
| Results       |   Priority    |             //////            |
+---------------+---------------+---------------+---------------+
```

Results may have one of the following values:

1 : `Accept`

2 : `Snub`

3 : `Veto`

Priority ranges between 1..255 (see 2.2.3).

- CONNECT_DESC class
  - Class = 17, C-Type = 1

    This object describes the OOPS connection parameters; in the Connection-Accept-Response, the refresh-multiplier is an echo of the value received with the Connection-Initiation-Query.

```
+---------------+---------------+---------------+---------------+
|  Version      |  Flags        | Refresh-Mult. |    //////     |
+---------------+---------------+---------------+---------------+
| Max-Msg-Size (in KBytes)      | Channel-Hold period (in sec.) |
+---------------+---------------+---------------+---------------+
```

Ver: 8 bits

Currently, version 1.

Flags:

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

Refresh-Mult.:

The refresh-period multiplier (e.g., RSVP's K value).

Max-Msg-Size: Upper limit on the length of an OOPS message

Channel-Hold period: Implicit disconnection timeout

- BYE_DESC class
  - Class = 18, C-Type = 1

    BYE_DESC provides details about the Bye-Notification request.

```
+---------------+---------------+---------------+---------------+
|   Bye-Flags   |  //////       |      BYE_DELAY (seconds)      |
+---------------+---------------+---------------+---------------+
```

Bye-Flags:

`0x01     An echo (response) to a received Bye-Notification`

The BYE_DELAY could provide both sides with some time delay to be better prepared to a pending bye.[20] The delay value is determined by the originator of the bye-notification, and is echoed in the bye response. The delay effect should be as if the Bye-Notification was sent BYE_DELAY seconds later with a delay timer value of 0.

- STATE_OP_DESC class

[20] Similar to the delayed shutdown command known in Unix.

– Class = 19, C-Type = 1

```
+---------------+---------------+---------------+---------------+
|   Op-Type     |                      //////                   |
+---------------+---------------+---------------+---------------+
```

Op-Type values:

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

B Appendix: Error Codes

This appendix describes an initial list of error codes available in OOPS, as well as the set of Reason Codes for each error code. (Reason Code of 0 must be used when Reason Codes are not applicable). This list should evolve and not be considered conclusive.[21]

- Code = 1, Connection Management
  1: Connection Reject: Server does not support client version.
  2: Bye: Reset due to routine state re-synchronization
  2: Bye: Reset due to connection problems (Bad message formats)
- Code = 2, Protocol problems
  1: Syntax: Bad OOPS message
  2: Syntax: Bad OOPS Op-Code
  3: Syntax: Bad POLICY_DESC format
- Code = 3, Policy Decisions
  1: Don't forward: refrain from forwarding an outgoing message
  2: Policy Reject: cancel protocol operation (Reservation, path, etc.)
- Code = 4, State Management
  1: Delete State: Reservation Canceled
  2: Delete State: route change
  3: Delete State: State Timeout
  4: Blockade State
  5: Unblock State

[21] Not even close to be conclusive at this point in time!