Internet Draft
Expires December 12, 1996
File: draft-ietf-rsvp-policy-lpm-00.ps

Shai Herzog
USC/ISI
June 1996

Local Policy Modules (LPM):

Policy Enforcement for Resource Reservation Protocols

June 12, 1996

Status of Memo

Abstract

This memo describes a set of building blocks for policy based admission control in RSVP and similar resource reservation protocols. We describe an interface between RSVP and Local Policy Modules (LPM); this interface provides RSVP with policy related information, and allows local policy modules to support various accounting and access control policies.

1 Introduction

The current admission process in RSVP uses resource (capacity) based admission control; we expand this model to include policy based admission control as well, in one atomic operation. Policy admission control is enforced at border/policy nodes by Local Policy Modules (LPMs). LPMs based their admission decision, among other factors, on the contents of POLICY_DATA objects that are carried inside RSVP messages. LPMs are responsible for receiving, processing, and forwarding POLICY_DATA objects. Subject to the applicable bilateral agreements, and local policies, LPMs may also rewrite and modify the POLICY_DATA objects as the pass through policy nodes.

In this document, we describe the range of policies that **can** be supported, however, we recommend that you read this document along side with its policy reference document [HER96b]. This document describes a generic framework for policy enforcement; we do not advocate any specific access control policies since we believe that standardization of policies (as opposed to the framework) may require significantly more research and better understanding of the tradeoffs.

Section 2 provides a general description of the RSVP/LPM interface, Section 3 specified the internal representation of POLICY_ELEMENT objects, Section 4 describes the detailed interface between RSVP and the LPM, and Section 5 provides a peek into some of the more important LPM implementation internals.

2 The RSVP/LPM interface

Unless we are willing to declare a single monolithic access policy we need to accommodate varying, independent access control mechanisms in RSVP (e.g., over different regions of the Internet, internal accounting vs. inter-provider accounting, quota vs. advanced reservations, etc.). Each mechanism can have its own, type-specific internal format, can be configured for local needs (e.g., policy data rewrite (conversion) table, etc.), and can be added and removed from nodes with little or no impact on other mechanisms.

2.1 POLICY_DATA objects

RSVP messages may carry **optional** POLICY_DATA objects. Policy data objects are a general container for policy related information that could assist local RSVP nodes along the reserved path in their policy decisions. Policy information may originate from end-users, however, it can also be created or converted at the core of the network. POLICY_DATA objects contain an optional list of FILTER_SPEC objects which identify the flows it is associated with: we expect that some access control mechanisms to use **session** POLICY_DATA objects (with wildcard FILTER_SPEC) while others may require the full power of per-flow

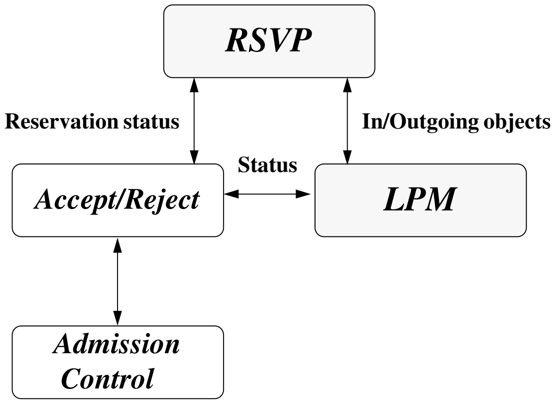

Figure 1: The modular context of access control

object semantics. Generally, we assume that POLICY_DATA objects can be carried by any RSVP message, (e.g., Path, Resv, ResvErr, etc.).

2.2 Modular Context

Before RSVP accepts a reservation it must check for access authorization. This is where local policy modules take effect, verifying access rights to local resources (i.e. links, clouds, etc.). Figure 1 illustrates the context for the proposed design: RSVP interfaces to the LPM to handle input and output of POLICY_DATA objects and to check the status of reservations. Conceptually, a reservation must be accepted both physically and administratively; physically, by traditional admission control (based on congestion) and administratively by the local access policy enforced by the LPM. This dual admission must be atomic and this atomicity is represented by the "accept/reject" module. In this document, we concentrate only on the highlighted modules: the RSVP and the LPM interfaces. The RSVP interface is defined by describing the functionality that is expected from RSVP in order to support access control. It includes the handling of incoming messages, scheduling outgoing messages, and performing status checks. The LPM interface describes the services the LPM provides, through a set of LPM functions. However, we do not define how RSVP should check the status of reservations (it could be done by calling the LPM directly, through an accept/reject module, or in other ways). [1]

[1] The RSVP admission process is unidirectional and does not include upcalls to RSVP, e.g., there is no upcall to notify RSVP that a previously made reservation was canceled or preempted. We do however anticipate that once the initial access control architecture is in place, later changes to the RSVP spec, would define an "accept/reject" module, and associated status update upcalls to RSVP.

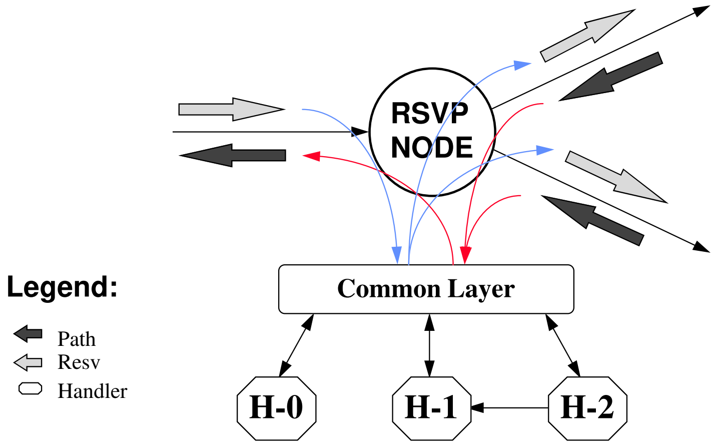

Figure 2: LPM and RSVP: message flow outline

2.3 Local Policy Modules

Local Policy Modules (LPMs) can be configured locally, to a particular access policy. LPMs have three basic functions: first, to receive incoming policy data objects, second, to update the access/accounting status of reservations, and third, to build accounting/policy data objects for outgoing RSVP messages (The LPM message flow outline is illustrated in figure 2). LPMs maintain local access state for supporting the LPM operations, and this state must remain consistent with RSVP's state.

2.3.1 Processing incoming messages

RSVP calls the LPM for object processing each time it receives a POLICY_DATA object. The LPM processes, stores the object's information, and returns a status code to RSVP. The status code reports the success/failure of object processing, but does not reflect the acceptance of the reservation. The status of a reservation must be checked separately (see Section 2.3.3 for more details).

2.3.2 Processing outgoing messages

When RSVP generates an outgoing message it calls the LPM. The LPM assembles the outgoing policy data objects and hands them to RSVP for placing inside the outgoing message.

2.3.3 Reservation status updates

The concept of access control assumes that even previously admitted reservations are conditional, in a sense that changes in access status may trigger some action against the associated reservation (i.e., cancel it, allow its preemption, etc.). Therefore, the access control mechanism must periodically check for reservation status changes (like quota exhaustion) and take the appropriate measures. Reservation status should also be checked when system events require it, (e.g., the arrival of a new policy data object with updated information). Status checks may be limited to the scope of the change (e.g., only the interface from which the new RSVP message arrived).

2.3.4 Optional debiting for Reservations

The simplest form of access control performs a binary task: accept or reject a reservation. More advanced policies may require the LPM to perform book keeping (i.e., usage quota enforcement or even cost recovery). To achieve such tasks, the LPM can be configured to perform debiting. Debiting is not part of the LPM interface, and can be configured as an option into the status update: when RSVP queries the LPM about the status of a reservation, the LPM may perform debiting, and update the status of the reservation according to the debiting result. The debiting process is based on two separate functions: **determining "cost"**, and **actual debiting**. These two functions can be fully independent from each other, and most likely be carried out by different handlers.

In multicast environments, with upstream merging, it is very likely that a reservation will be debited against multiple network entities that represent the aggregated credentials of the downstream receivers. This raises the issue of the "sharing model". **The sharing model** defines how the reservation is shared among the different policy data objects.[2] The sharing model, and the selection of cost allocation and actual debiting mechanisms is an issue of LPM local configuration, and is **not** discussed in this document.

2.3.5 Security issues

Hop-by-hop authentication mechanism:

The RSVP security mechanism proposed in [BAK96] relies on hop-by-hop authentication. This form of authentication creates a chain of trust that is only as strong as its weakest element (in our case, the weakest router). As long as we believe that all RSVP nodes are policy nodes as well, then RSVP security is sufficient for the entire

[2] Sharing model examples: (1) Each policy object is allocated the full cost, (2) The cost is divided equally between the different objects (3) The cost is attributed to an arbitrary object (4) The cost allocated relative to some criteria like the number of downstream receivers, the size of the organization, the amount of pre-purchased capacity (remaining quota), etc.

RSVP message, including POLICY_DATA objects. This however is not the case when policy is only enforced at boundary nodes.

Security over clouds:

If policies are only enforced at cloud entry and exit points, then RSVP's security is insufficient to protect policy objects, since from a policy enforcement perspective, the in-cloud nodes are unsecured. We propose a "policy data tunneling" approach, where the logical policy topology is discovered automatically, and security is enforced over the logical topology. When policy objects are created at border routers, they are encapsulated in a security envelope (described in Sections 3 and 2.3.5). The envelop is forwarded as-is over the cloud, and is only removed by the cloud border (exit) node.

2.4 Default handling of policy data objects

Because we do not expect (or desire) that every RSVP node will be capable of processing all types of policy data objects, it is essential that RSVP define default handling of such unrecognized objects, and that this default handling be required from any RSVP/LPM implementation. The general concept is that RSVP play the role of a repeater (or a tunnel) by forwarding the received objects without modification. Implementation details are an part of the internal LPM architecture, described in Section 5.

3 POLICY_ELEMENT objects: internal representation

The contents of the POLICY_ELEMENT is opaque to RSVP; the format we describe here is only visible to the LPM. POLICY_ELEMENT objects are made of a list of policy particles. Policy particles have a length, a policy type (PType) and a type specific format.

```
+-------------+-------------+-------------+-------------+
|     Length                |     20      |    CType    |
+---------------------------+-------------+-------------+
|     Policy Particles (list)                           |
+-------------------------------------------------------+
```

Individual policy particle has the following format

```
+---------------------------+---------------------------+
|     Length                |     PType                 |
+---------------------------+---------------------------+
|     Ptype specific format                             |
+-------------------------------------------------------+
```

4 LPM calls

The LPM maintains access control state per flow. This state is complementary to the RSVP state, and both are semantically attached by **flow handles**, for all the LPM calls.

4.1 Success codes

All the LPM calls report success/failure status. This report is made of three components: (1) a return code of the lpm function, that reports the general success of the call (2) a global variable *lpm_errno* that reports specific reason code (similar to the errno in Unix), and (3) a global variable *lpm_eflgs* used for flags set by the LPM call.

4.2 Flow handles (fh)

The LPM uses Flow Handles (fh) to associate RSVP flows with LPM state. RSVP obtains flow handles by calling *lpm_open()*, which is called only once for each session or flow, upon the first arrival of a POLICY_DATA object associated with that flow or session. RSVP obtains the flow handle and stores it in the flow's data structures, for future lpm calls.

When an RSVP message is fragmented, POLICY_DATA objects may be out of order, and may reside in separate packets. The responsibility of associating a POLICY_DATA object with a particular flow (and its flow handles (fh)) lies *always* with RSVP. The FILTER_SPEC object inside the POLICY_DATA object is visible to RSVP, and should be used by it to aid in this classification.[3] It is important to note that under no circumstances should this classification be left to the LPM.

4.3 Associating source and receiver objects

The access status of a reservation may depend on policy data objects originating from the source, receivers or both. For instance, a lecture can be sponsored by the source that would provide the necessary credentials. If the LPM architecture is to support source based policies, it must be able to associate source objects with reservation state. Some associations are trivial (like in the case of fixed filter (FF) reservation style) but some are more complicated (as in WF reservations). Since the LPM architecture associates flow handles with individual source state, it is the responsibility of RSVP to map reservations to their list of associated sources. The list takes the form of a list of flow handles, and can be passed on to LPM functions through a pair of parameters, *int fh_num* and *int *fn_vec*).

[3] The FILTER_SPEC object is opaque to the LPM and the only reason it is included inside the POLICY_DATA object is to allow RSVP to associate the object with its corresponding flow.

4.4 LPM calls format

lpm_open (int *fh)

When RSVP first encounters POLICY_DATA objects, it calls the LPM's *lpm_open* routine. The LPM builds internal control blocks and places the flow handle value in fh, for future reference.

All incoming POLICY_DATA objects are passed by RSVP to the LPM:

lpm_in (int fh_num, int *fh_vec, int vif, RSVP_HOP *hop, int mtype, POLICY_DATA *polp, int ttd)

Parameter *vif* describes the input virtual interface[4] from which the RSVP message was received, *hop* describes the node that sent the RSVP message (previous hop/next hop), and *mtype* describes the type (and implicitly, the direction) of the RSVP message (i.e., Path, Resv etc.). Parameter *polp* points to the policy data object, and *ttd* provides a timeout (time to die) value for the policy data object.

When RSVP is ready for output, it queries the LPM:

lpm_out (int fh_num, int *fh_vec, int vif, RSVP_HOP *hop, int mtype, POLICY_DATA **polp)

The parameters are similar to those for *lpm_in*. A successful call places a pointer to the outgoing POLICY_DATA object in *polp*; Notice that the output process is performed separately for each outgoing RSVP message, but is required to maintain consistency and atomicity even if some LPM status had changed in between outputs of different outgoing RSVP messages. Notice that there is no formal limit on the size of the resulting POLICY_DATA object. If the resulting object is too large to be sent in a single RSVP message it is RSVP's responsibility to perform semantic fragmentation because it has the unique knowledge about available message space. An alternative solution would be to provide an $lpm\_fragment()$ service to help RSVP in this task.

Checking the status of an existing reservation is done by calling:

lpm_status (int fh_session, int fh_num, int *fh_vec, int vif, int cur_time, int phy_resv_handle, Object_header *phy_resv_flwspec, int ind)

[4] The term Virtual Interface (vif) is borrowed from DVMRP terminology, although, for LPM purposes it can be any integer index that RSVP associates with specific interfaces, independently from any routing protocol.

Status is checked individually for each outgoing (reserved) link. Parameter *fh_session* specifies the flow handle associated with the session, *phy_resv_handle* identifies the physical reservation (e.g., ISPS, etc.), and *phy_resv_flwspec* describes the current, merged FlowSpec of the reservation. The value of *cur_time* describe the current RSVP time, which allows the LPM to timeout old state (state with earlier time to die values). Parameter *ind* is used to have different flavors of status checks:
LPM_STATF_AGE: setting this flag ages (and times out) LPM state associated with the specified fh. Status checks may be periodic or event driven; this flag is set only for periodic status checks. *LPM_STATF_RECALC*: Status checks may involve calculations over multiple outgoing interfaces, and thus need only be done once for all interfaces before individual per-interface status is reported. This bit is set on for the first vif checked and is reset for the rest.[5] Status checks with *ind* set to 0 simply report values that were already calculated before and do not age the LPM state.

If RSVP prunes branches from the reservation tree, it must notify the LPM by calling:

lpm_prune (int fh_num, int *fh_vec, int vif, RSVP_HOP *hop, int mtype)

(The details of this call is described in Section 4.5).

When RSVP deletes an entire flow state, it must notify the LPM:

lpm_close (int fh)

Upon this notification, the LPM finishes its accounting for this reservation (final debits/credits) and deletes all internal state associated with fh.

Initializing the LPM is done once only, in the initialization phase of RSVP, by calling.

lpm_config (void)

4.5 State Maintenance

LPM state must remain consistent with the corresponding RSVP state. State is created when POLICY_DATA objects are passed to the LPM and can be updated or removed through several possible mechanisms that correspond to RSVP's state management mechanisms:

[5]This is an optimization. While useless, there should be no harm in recalculating status parameters, for each outgoing interface.

Timeout:

When new POLICY_DATA objects cease to arrive (as a result of either change of policy or fragmentation loss) the locally stored state begins to age. Each POLICY_ELEMENT/FILTER_SPEC pair is subject to a timer, and when the timer goes off, the state should be deleted. The timer mechanism should be similar to that of RSVP and both should remained synchronized in the following way: each time RSVP hands over a policy object to the LPM (*lpm_in*()) it provides the LPM with time-to-die value (*current-timer + time-to-live)* . Each time RSVP verifies the status of a reservation (*lpm_status*()), it provides the current timer value, forcing all pieces of information with an earlier timeout value to be purged.

Teardown

From a network security standpoint, creating new policy state requires the similar integrity protection as tearing it down. We propose a very simple mechanism for tearing down state: the state created by sending POLICY_ELEMENT Pe_i is torn down by sending $-Pe_i$ (the same object marked as teardown). In this case, the LPM would locate the original state, compare it with the teardown object, if a match is found, tear it down. We define each POLICY_ELEMENT as a pair of two CTypes, thus effectively splitting the CType range of POLICY_ELEMENT objects in two. Given a POLICY_ELEMENT i, Pe_i represents an updated state, while Pe_{i+1} represents teardown state of CType i ($-Pe_i$).

Pruning When the shape of the reserved tree changes due to routing updates or RSVP teardown messages, RSVP purges the state of the pruned link, and must also call *lpm_prune()* to purge the corresponding LPM state.

Closing: The call *lpm_close(fh)* purges all the state associated with the handle fh. Closing a flow handle is done when RSVP no longer maintains any state associated with that flow (a sender quits, the session is over, etc.).

5 LPM internals

This section describes the current internal design of the LPM. While this design is not part of the mandatory specification we recommend following it.

5.1 LPM configurations

LPM configuration can be general, for all handlers, but can also be type/handler specific. (e.g., a specific handler's rewrite conversion table for policy data objects). Configuration may be expressed in a simple configuration file or even through a configuration language.

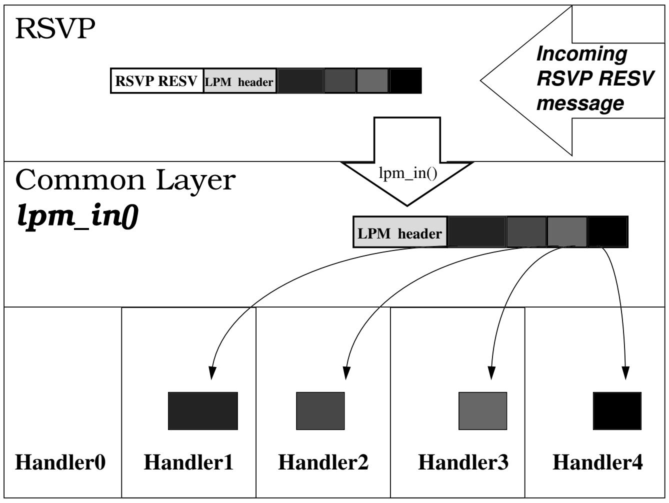

Figure 3: Disassembly of an incoming Resv message with POLICY_DATA objects

5.2 The LPM layered Design

The internal format of POLICY_DATA objects is PType specific, allowing up to 65535 independent types. Our design allow each specific PType to be handled by a separate handler, and allow such handlers to be added and configured independently. Clearly, handlers are allowed to handler more than one PTypes.

The LPM is divided into two layers: a PType specific layer and a common layer (figure 3). The PType specific layer provides a set of locally configured independent handlers, one for each PType supported by the local node. The common layer provides the glue between RSVP and the PType specific layer by multiplexing RSVP's lpm calls into individual, PType specific calls.

On input, the common layer disassembles the incoming POLICY_DATA object, dispatches the internal objects to their PType specific handlers, and aggregates the return code status (figure 3). On output, it collects the internal objects from all active handlers, and assembles them into a single POLICY_DATA object (figure 4).

On status queries, the common layer queries all the active handlers, and combines their individual status responses into a single status result. We use the following rule: a reservation is approved by the common layer, if there is at least one handler that approves it, and none other rejects it. PType specific handlers can accept, reject or be neutral in their responses.[6]

[6] A policy data object that determines cost is a good example for a neutral handler. It provide information about how much the flow costs, but does not perform actual debiting.

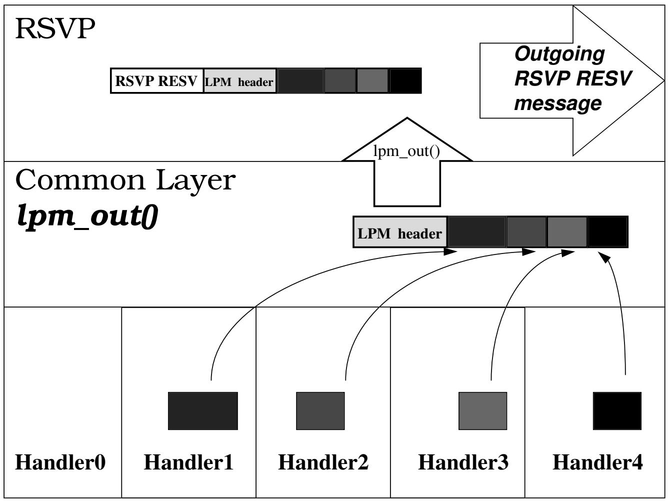

Figure 4: Assembly of POLICY_DATA objects for an outgoing Resv message

5.3 Interaction between handlers

It is reasonable to assume that independent PTypes may require some interaction between their handlers. Consider the case where policy object type-1 is a credential type (defines a user identity) and a type-2 is an accounting type (determines cost), a possible interaction could be to let type-2 determine the cost, and let type-1 perform the actual debiting according to the user identity. Such interaction has two basic requirements: order dependency and export capability. Order dependency is required because type-2 must calculate the cost before type-1. Export capability is needed to allow type-2 to export the calculation results to type-1. Our implementation allows the ordering or handlers to be expressed as part of local LPM configuration. It also provides internal support for function calls between independent handlers (in order to obtain exported state).

Consider the case where type-3 and type-4 also perform accounting. The proposed architecture is flexible enough to allow local configuration to select the handler that determines the debited cost: type-2, type-3 or type-4.

5.4 Default handling of policy data objects

In [HER96c] we define the default handling of unrecognized POLICY_DATA objects. If an RSVP node is LPM capable, it may be more beneficial for the LPM to take that burden off from RSVP and perform it itself. We propose the use of CType 0 for default handling: In a policy node, only unrecognized objects would be handled by handler PType 0. In a non-policy node, all objects are unrecognized, and therefore should all are handled as PType 0, regardless of their actual PType. PType 0 is regarded as a reserved type.

6 Acknowledgment

This document incorporates inputs from Deborah Estrin, Scott Shenker and Bob Braden and feedback from RSVP collaborators.

References

[BAK96] F. Baker. RSVP Cryptographic Authentication *Internet-Draft*, draft-ietf-rsvp-md5-02.txt, 1996.

[HER96c] RSVP Extensions for Policy Control. *Internet-Draft*, draft-ietf-rsvp-policy-ext-00.[ps,txt].

[HER96b] Accounting and Access Control Policies for Resource Reservation Protocols. *Internet-Draft*, draft-ietf-rsvp-policy-arch-00.[ps,txt].