Internet Draft
Expires September 5, 1996
File: draft-ietf-rsvp-lpm-arch-00.ps

Shai Herzog
USC/ISI
March 1996

Building Blocks for Accounting and Access Control in RSVP

March 5, 1996

Status of Memo

Abstract

This memo describes a set of building blocks for policy based admission control in RSVP. 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

RSVP, by its definition, discriminates between users, by providing some users with better service at the expense of others. Therefore, it is reasonable to expect that RSVP be accompanied by mechanisms for controlling and enforcing access and usage policies. In this document, we refer to such policies as "access control". The term "access control" is quite broad; it ranges from simple access approval to sophisticated accounting and debiting mechanisms (Section 2 describes a few sample scenarios of access control mechanisms). For scaling reasons, we concentrate on policies that follow the bilateral agreements model. The bilateral model assumes that network clouds (providers) contract with their closest point of contact (neighbor) to establish ground rules and arrangements for access control and accounting. These contracts are mostly local and do not rely on global agreements. The bilateral model has similar scaling properties to RSVP and is easier to maintain in distributed environments.

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, but leave the specific policies to local LPM configurations.[1] We begin (Section 2) by describing a few sample scenarios which provide both motivation and demonstration of possible access control policies. Section 3 provides a general description of the RSVP/LPM interface and Section 4 discusses RSVP spec related issues. The appendices describe the detailed interface (object formats, LPM calls, etc.), and provide a peek into some of the more important LPM implementation internals.

[1]We do not advocate specific access control policies since we believe that standardization of specific policies may require significantly more research and better understanding of the tradeoffs.

2 Sample scenarios

In this section, we outline a few sample scenarios for access control; we provide these scenarios as motivation and as needed context for the LPM architecture proposed in this document.

These scenarios, as well as the LPM architecture as a whole, are based on two simple assumptions: (1) RSVP would provide the needed transport service of carrying access control state (POLICY_DATA objects), hop-by-hop. (2) Access control policies are based on bilateral agreements between neighboring providers or users, and are enforced locally by a Local Policy Modules (LPMs). In this document we do not discuss policies based on global agreements or global information because of obvious scalability concerns.

2.1 Simple access control

To provide simple access control, the LPM attempts to match incoming policy objects with one or more of the pre-configured policies or bilateral agreements, in order to accept or reject the reservation.

Consider the following network scenario: one receiver from ISI and two from MIT listen to a PARC seminar. For simplicity of the scenario, let us limit ourselves to a **receiver based** access control scenario.

The bilateral agreements between each two neighboring providers (e.g., R1, R2 with ISI, ISI with LosNettos,... BARRNet with PARC) are simple: the first provider obtains a permission to make reservations over the second provider's network. The notation $PD(cr, uid)$ represents a policy data object of type "cr" (credential) verifying that the flow belongs to *uid*. Credentials can be hierarchical, and may be rewritten on a hop by hop basis through a locally configured conversion table.

Figure 1 illustrates a reservation scenario. An typical example of a bilateral agreement could be between MCI and LosNettos: MCI would allow the LosNettos users to use its backbone. A policy data object $PD(cr, LosNettos)$ would be interpreted by MCI as a green light to accept the reservation. In this scenario, reservations from R1, R2, R3 carry policy data objects that propagate hop-by-hop (encapsulated in reservation messages) toward S1. Assuming all nodes are configured consistently, policy objects are rewritten in nodes B,D,G,I,K,M, which are entry points to clouds).

The MCI cloud is interesting. E is not a border/policy node, but still, it receives the following policy data objects: F→E: $PD(cr, LosNettos)$ and J→E: $PD(cr, NearNet)$. Assuming E has no authority to merge or rewrite these credentials, it must concatenate the two objects and send $PD(cr, LosNettos) + PD(cr, NearNet)$ to D. Let us further assume that D is configured with the following conversion table:

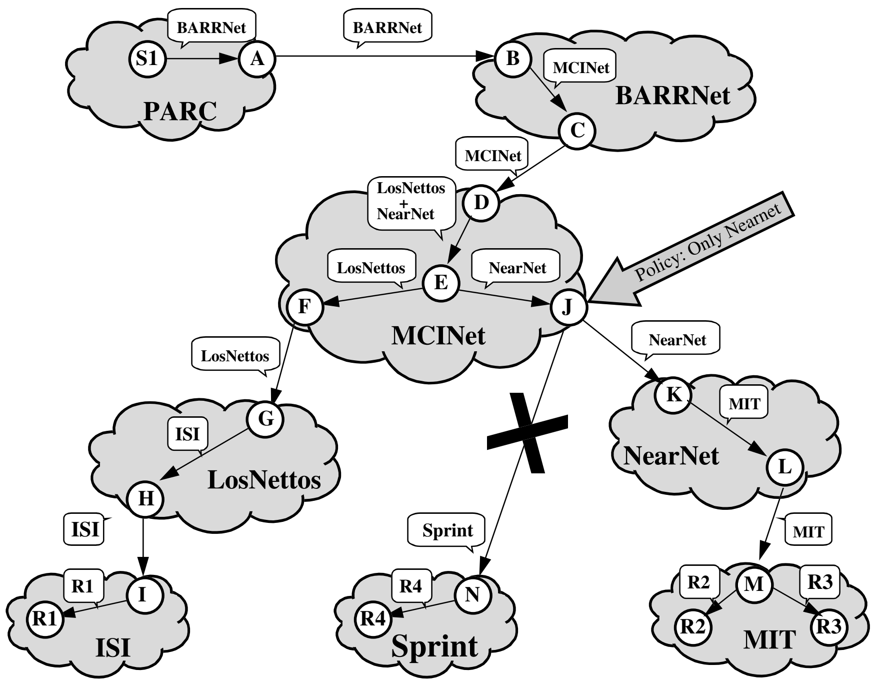

Figure 1: Simple access control

$PD(cr, LosNettos) \implies PD(cr, MCI)$
$PD(cr, NearNet) \implies PD(cr, MCI)$

D's LPM first checks if LosNettos and NearNet are authorized to reserve on their corresponding links and responds accordingly. Assuming authorization is cleared, it merges and rewrites these policy objects as $PD(cr, MCI)$ and forwards the reservation to C.

To complicate the example, assume the conversion table was:

$PD(cr, LosNettos) \implies PD(cr, MCI1)$
$PD(cr, NearNet) \implies PD(cr, MCI2)$

Then D's LPM would forward $PD(cr, MCI1) + PD(cr, MCI2)$ to C instead.

Local policies can also reject reservations:

In figure 1 we see that a reservation made by R4 is rejected because it arrives with insufficient credentials: the local policy in node J accepts only traffic marked as $PD(cr, NearNet)$, and R4's reservation arrives with $PD(cr, Sprint)$.

2.2 Advanced reservation and preemption control

Advanced reservation can be built on top of simple access control: consider the case where every advanced reservation consists of a set of bilateral agreements between different service providers, reserving network capacity at some future period of time. When advanced reservations are not public (i.e., only authorized users can use them), three classes of reservations exist: (1) walk-ins (where the conference itself does not have advanced reservations, (2) advanced reservation with unauthorized users, and (3) advanced reservation with authorized users. These numbers (1..3) can define a "preemption priority" (i.e., walk-ins are preempted first, unauthorized pre-reserved second, and authorized pre-reserved are never preempted).

The advanced reservation scenario is almost identical to the simple access control: let us assume that each bilateral pre-registration is identified by a PRID (Pre-Registration confirmation ID). Policy data objects of type AR (Advanced Reservation) would take the following form: $PD(ar, prid, uid)$. When an AR object arrives, the LPM verifies the existence of pre-reservation $prid$, and checks that uid is permitted to use it. Finally, the flow is classified to one of the above three preemptive priorities and RSVP is notifies.

2.3 Quota enforcement/accounting/debiting

The next step is to allow for more sophisticated access control that is based on usage feedback. Here we add two additional mechanisms which (1) determine **how much** should be debited for a reservation and (2) what **debiting mechanism** should be used (if any). The following scenarios assume a pre-existing set of local accounts. These accounts are established by bilateral agreements that pre-purchase network capacity and set applicable debiting rules. The role of accounting mechanism is to verify the availability of funds/quotas in these accounts for maintaining the reservation. We consider several accounting schemes and briefly describe three: simple debiting, limited debiting, Edge Pricing, and MultiCost (MCost).

1. **Simple debiting**

   Consider the following example: lets assume that LosNettos and Nearnet each have a debit account (pre-purchased capacity) with MCI for their traffic. When E's LPM receives the following $PD(cr, LosNettos)$ and $PD(cr, NearNet)$ for flow f, it must decide the following: (1) How much should be debited for flow f, and (2) how would that debit be shared between the account of LosNettos and NearNet. These are local configuration issues left for service providers. In this scenario, the LPM would attempt to perform the debiting, and would notify RSVP on success or failure. The other aspects of the scenario (Merging policy data objects and forwarding them) is identical to that of simple access control.

2. **Limited debiting (willingness to pay)**

Although we do not have a full understanding of the dynamics of willingness-to-pay and its properties, we can outline the basic scenario, as an extension of the simple debiting model. Willingness to pay is manifested as a limit on the policy object that authorizes the debit. For instance, $PD(crwp, ISI, 10\%\ of\ unicast)$ would represent a policy data object of type *crwp* (Credential, Willingness to Pay), that authorizes debiting the ISI account up to 10% of the unicast cost. Here, the basic idea is that market forces would be the driving force behind what users specify as their willingness to pay.

3. **Edge Pricing**

   Edge Pricing was presented in [SHE95]. This paradigm is based on the assumption that network costs can be estimated and approximated at the edge of the network, based on purely local information. Edge Pricing is an extension of simple debiting: Edge Pricing can determine **how much** is to be debited, and the set of credentials associated with the reservation determines who **(which account)** should be debited.

4. **MultiCost (MCost)**

   MCost is an accounting scheme (and mechanism) that was introduced in [HER95]. MCost has a unique feature: it takes into account the benefits of sharing a multicast tree and distributes these savings among the members of the multicast group, according to configurable policies, basic fairness, and equality.

   MCost computes the cost allocated to each user, and that cost can be the basis for debiting. MCost can be combined with simple debiting in a similar manner to Edge Pricing.

3 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.

3.1 POLICY_DATA objects

RSVP messages may carry **optional** POLICY_DATA objects. Each individual POLICY_DATA object includes a FILTER_SPEC object which identifies the flow it is associated with. We expect some access control mechanisms to use **session** POLICY_DATA objects (with wildcard FILTER_SPEC) while others may require the full power of per-flow object semantics. Generally, we assume that POLICY_DATA objects may be carried by any RSVP message, (e.g., Path, Resv, ResvErr, etc.).

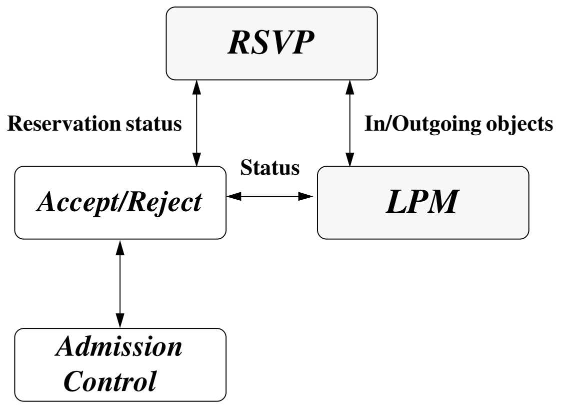

Figure 2: The modular context of access control

3.2 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 2 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). [2]

3.3 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]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.

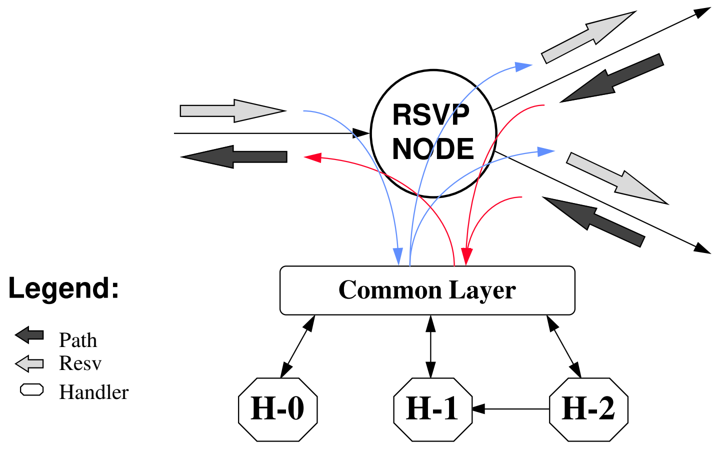

Figure 3: LPM and RSVP: message flow outline

3). LPMs maintain local access state for supporting the LPM operations, and this state must remain consistent with RSVP's state.

3.3.1 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 3.3.3 for more details).

3.3.2 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.

3.3.3 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).

3.3.4 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.[3] 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.

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

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

[3]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.

the logical topology. When policy objects are created at border routers, they are encapsulated in a security envelope (described in Sections A and 3.3.5). The envelop is forwarded as-is over the cloud, and is only removed by the cloud border (exit) node.

3.4 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 appendix C.

4 RSVP spec issues

This section presents changes to the RSVP specifications, required to support the LPM architecture.

4.1 New RSVP message: Reservation Report

The basic building blocks of access control and accounting must be bi-directional in order to allow both source and receiver based policy data objects and both advertising and feedback. Which RSVP messages should encapsulate these upstream and downstream objects? The choice for upstream message is natural; the reservation message. The downstream direction, however, is more problematic: Path messages flow downstream, but are routed according to the multicast group membership, and therefore cannot be accurately delivered to a specific next hop.[4] This makes Path messages less likely to be used for access control, and especially for accounting.

We proposed a new RSVP message type: *Reservation Report* (Rept). Reservation Report messages are sent unicast, downstream, according to the Next_Hop object carried by Resv messages; although Reservation Report messages follow the multicast tree, their unicast delivery provides accurate delivery to the appropriate next hop nodes and only to these nodes[5] Although we propose this new message for supporting the LPM architecture, it may

[4]The same problem existed for the original design of ResvErr, until it was changed to a unicast delivery along the multicast tree.

[5]Consider the case of multipoint links or network clouds: a single copy of a Path message may be delivered to an unknown number of next hops, while the copy of a Report message is guaranteed to reach only the targeted node.

prove useful for other, more general functions of the RSVP protocol. A reservation may have different forms of responses to it: A negative response (ResvErr), a positive response (ack), and a more advanced form of a Reservation Report, like the one proposed here. An integrated approach may incorporate all three responses in the same message type while leaving room for future types.

4.2 List of proposed changes to the RSVP spec

- LPM interface (LPM calls, error codes and response to errors)
- API modifications.
- Reservation Report messages (Either in a general form, or specific to the LPM architecture).
- Default handling of policy data objects.

5 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.

[HER95] S. Herzog, S. Shenker and D. Estrin, Sharing the Cost of Multicast Trees: An Axiomatic Analysis, *Proceedings of ACM/SIGCOMM '95*, Cambridge, MA, Aug. 1995

[SHE95] S. Shenker, D. Clark, D. Estrin, and S. Herzog Pricing in Computer Networks: Reshaping the Research Agenda, *Telecommunications Policy*, Vol. 20, No. 1, 1996 also published in *Proceedings of the Twenty-Third Annual Telecommunications Policy Research Conference*, 1995.

A Appendix: object format

POLICY_DATA objects are built from basic building blocks (sub-objects), with the following format:

```
+-------------+-------------+-------------+-------------+
|           length          |           PType           |
+-------------+-------------+-------------+-------------+
|  PType specific format                                |
+-------------------------------------------------------+
```

The header of the POLICY_DATA object is defined by the RSVP spec, while the body format is hidden from RSVP, and is only known to the LPM. POLICY_DATA object include the standard RSVP object header, with Class = class_POLICY_DATA, and a CType value. Currently, the CType value selects from three versions of POLICY_DATA objects: *POLICY_SIMPLE*, *POLICY_INTEGRITY*, and *POLICY_ENCAP*.

POLICY_SIMPLE:

```
+-------------+-------------+-------------+-------------+
|           length          | POLICY_DATA |      1      |
+-------------+-------------+-------------+-------------+
|  policy data sub-object 1                             |
+-------------------------------------------------------+
 ....
+-------------------------------------------------------+
|  policy data sub-object n                             |
+-------------------------------------------------------+
```

POLICY_INTEGRITY:

The object format is similar to POLICY_SIMPLE, with the added integrity envelope.

```
+-------------+-------------+-------------+-------------+
|           length          | POLICY_DATA |      2      |
+-------------+-------------+-------------+-------------+
|  RSVP_HOP object                                      |
+-------------------------------------------------------+
|  INTEGRITY object                                     |
+-------------------------------------------------------+
|  policy data sub-object 1                             |
+-------------------------------------------------------+
 ....
+-------------------------------------------------------+
|  policy data sub-object n                             |
+-------------------------------------------------------+
```

Encapsulation provides an optional security envelope for policy data objects; it ensures that all the policy sub-objects were created by the node described by the RSVP_HOP object, and were not compromised. In this document, we do not define how the INTEGRITY object is to be computed. However, we would like to note that it may be computed over other RSVP objects like SESSION, SCOPE etc., in order to guarantee that the POLICY_DATA object is associated with the right flow/reservation.

POLICY_ENCAP:

This object is the external (visible) representation of POLICY_DATA object, representing the full format.

```
+------------+------------+------------+------------+
|          length         | POLICY_DATA |    255    |
+------------+------------+------------+------------+
|  FILTER_SPEC object                               |
+------------+--------------------------------------+
|  Flags     |          Reserved                    |
+------------+--------------------------------------+
|  POLICY_ENCAP or POLICY_SIMPLE object 1           |
+---------------------------------------------------+
 ....
+---------------------------------------------------+
|  POLICY_ENCAP or POLICY_SIMPLE object n           |
+---------------------------------------------------+
```

Note: The FILTER_SPEC object is opaque to the LPM, however, it is included in the POLICY_OBJECT to assist RSVP with fragmentation.

There is currently only one flag in the flags field, `POLICYD_FLAG_REPORT`. This flag can be specified only for Resv messages, and tells the LPM (and RSVP) that the reservation requires a Reservation Report message.[6]

B Appendix: 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.

B.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.

B.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

[6]This may become obsolete if/when a Report Request bit is added to the Resv message format.

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.[7] It is important to note that under no circumstances should this classification be left to the LPM.

B.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*).

B.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)

[7] 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.

Parameter *vif* describes the input virtual interface[8] 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.

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.

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 phy_resv_handle, Object_header *phy_resv_flwspec, int ind)

Status is checked individually for each outgoing (reserved) link. Parameter *fh_session* specifies the flow handle associated with the session, and *phy_resv_handle* identifies the physical reservation (e.g., ISPS, etc.), *phy_resv_flwspec* describes the current, merged FlowSpec of the reservation. *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.[9] 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)

[8]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.

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

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

B.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:

Atomic object management: Every new POLICY_DATA object is self contained and its content overrides all previous state: existing state that is not listed in the newly arriving POLICY_DATA object is purged.

Aging: When new POLICY_DATA objects cease to arrive (either because RSVP messages cease to arrive, or because they arrive without policy data objects), the stored state begins to age. Aging is done in a similar manner to the way RSVP ages reservations: When a policy data object arrives, a timer is set to TTL_FACTOR. Every call to *lpm_status* decreases the timer by 1. When the timer reaches 0, the state is purged.

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.).

C Appendix: LPM internals

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

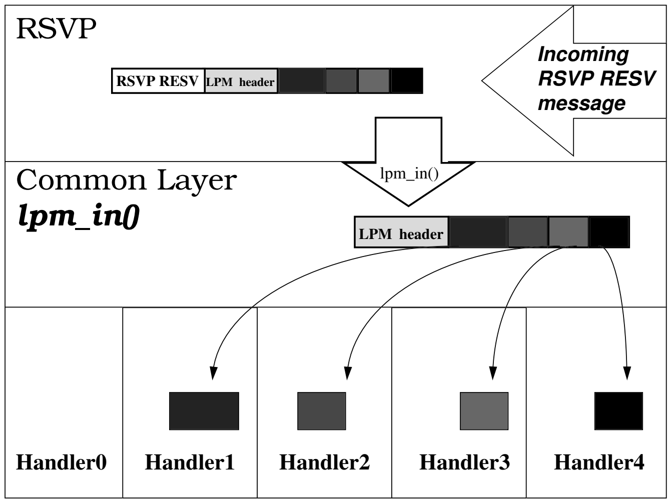

Figure 4: Disassembly of an incoming Resv message with POLICY_DATA objects

C.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.

C.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 4). 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 4). On output, it collects the internal objects from all active handlers, and assembles them into a single POLICY_DATA object (figure 5).

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

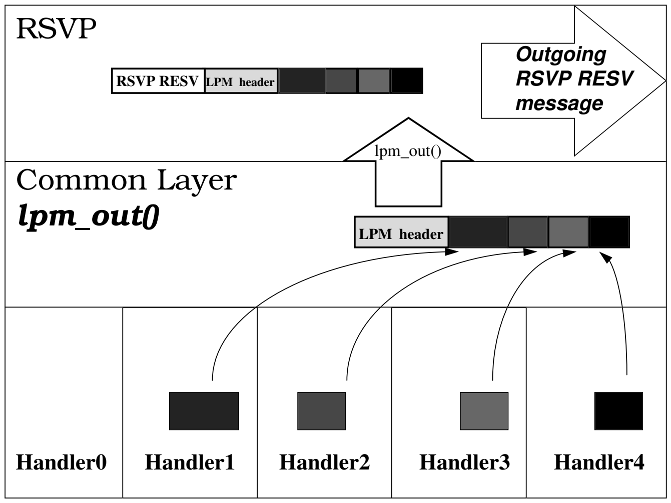

Figure 5: Assembly of POLICY_DATA objects for an outgoing Resv message

responses.[10]

C.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. (See the scenario 3 Section 2.3). 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.

C.4 Default handling of policy data objects

Default handling of policy data objects is needed in two cases: first, when the RSVP node is **not** a policy node at all, and second, when the arriving POLICY_DATA object includes objects of an unknown type. Both cases are handled in a similar manner: the policy

[10] 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.

object is stored and forwarded without modification, merging or any other operation. In our implementation we dedicate PType 0 for default handling: Unrecognized objects are handled by handler of PType 0. In a non-policy node, all objects are unrecognized, and thus all are handled as PType 0, regardless of their actual PType. PType 0 is regarded as a reserved type.

Notice that the internal format of POLICY_DATA objects is a list of objects; If a node is a merging point in the multicast tree, the default handler output is simply a concatenation of the lists of incoming objects encapsulated in a single POLICY_DATA object, of type POLICY_ENCAP.