Internet Draft
Expires May 22, 1997
File: draft-ietf-rsvp-policy-lpm-01.ps

Shai Herzog
IBM T.J. Watson Research Center
November 1996

Local Policy Modules (LPM): Policy Control for RSVP

November 22, 1996

Status of Memo

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

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

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

Abstract

This memo details a generic framework for policy enforcement based on the RSVP/Policy Control interface described in [Ext].

Contents

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. We introduce a framework named "Local Policy Modules" (LPM) that is based on the RSVP/Policy Control interface described in [Ext]. Policy admission control is enforced at border/policy nodes by LPMs; LPMs provide RSVP with information about the status of reservations, based on the contents of incoming policy objects, applicable bilateral agreements, and local policies. They are also responsible for constructing outgoing POLICY_DATA objects, by either copying, modifying or entirely rewriting the POLICY_DATA objects that pass through them.

This document suggests a generic framework for policy control which is based on RSVP's policy extensions [Ext] and its policy reference [Arch] documents. Interoperability considerations suggest the need for some standardized policies; the appendix to this document provides a list of such policies, however, it leaves a wide range of policies for local and proprietary usage.

The LPM architecture governs the following aspects of policy control: policy element format (Section 2 and Appendix A), policy multiplexing (Section 3), flows association (Section 4), PC services (Section 5), state maintenance (Section 6), fragmentation (Section 7, security considerations (Section 8), and configuration (Section 9).

2 Policy Elements

The format of policy data objects of C-type 1 is defined in [Ext] as including a list of policy elements. Policy elements relate policy specific information and have the following format:

```
+----------------------------+----------------------------+
|     Length                 |       P-type               |
+----------------------------+----------------------------+
|     Policy                                              |
+---------------------------------------------------------+
```

Length: 16 bits

The length (in bytes) of the policy element

P-type: 16 bits

The type of the policy element

Policy: variable length (multiples of 32 bits)

A description of the policy itself. Each policy element has its own P-type specific format. (See Appendix A for current type list).

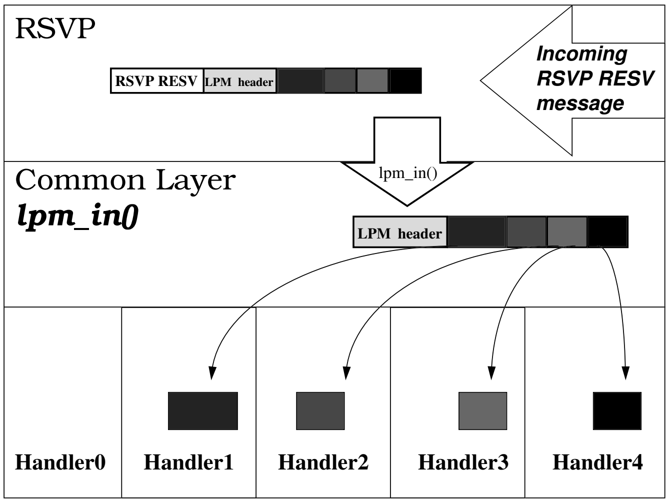

Figure 1: Demultiplexing an incoming Resv message with POLICY_DATA objects

3 The LPM Policy Multiplexer

We have contended in [Arch] that the exact nature of the usage policies is a local matter between service providers and their users, or other neighboring providers. Successful development and deployment of usage policies would greatly depend on the ability of ISPs to experiment and develop their desired policies. The LPM architecture was design to provide such capability by supporting a wide range of local, flexible, and plug-n-play policies.

Modularity is achieved by dividing the policy space into 65535 independent types of policy elements, identified by the P-type field. Moreover, multiple independent policies could simultaneously be in effect when multiple policy elements are included in a single POLICY_DATA object. The LPM is divided into two layers: a policy-specific layer and a common layer (Figure 1). The policy-specific layer contains the set of locally configured handlers, one for each P-type supported by the local node. Unrecognized objects are processed by a handler of the reserved P-type 0; this handler performs the default handling as specified in [Ext] (forwarding the objects as-is with the appropriate outgoing RSVP messages). The common layer provides the glue between RSVP and the policy-specific layer by demultiplexing RSVP's LPM calls into individual policy-specific calls.

On input, the common layer disassembles the incoming POLICY_DATA object, dispatches each policy element to its policy-specific handler, and aggregates the return code status from all handlers (Figure 1). On output, the common layer collects the policy elements from all active handlers, and assembles them into a single POLICY_DATA object (Figure 2).

On status queries, policy-specific handlers can vote to accept or snub a reservation, but may also cast a veto. The common layer collects responses from all active handlers, and combines them 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 accepts it it, and no other that vetoed

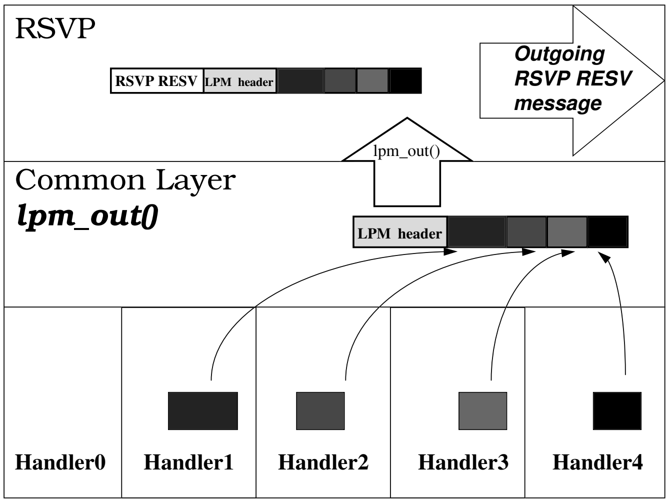

Figure 2: Constructing POLICY_DATA objects for an outgoing Resv message

it.[1] Consider the case where an internet service provider (ISP) admits flows either under flat-rate service or under pay-per-use arrangement. Several handlers can be active: [ID] for verifying the flow's owner/group ID, [FR] for verifying a flat-rate account, and [PPU] for performing pay-per-use transactions.

Let us examine some representative cases:

- Unknown user: REJECT: [ID]: veto. It doesn't matter what other handlers say.
- Flat-rate user: ADMIT : [ID]: snub, [FR]: accept, and [PPU]: snub.
- Pay-per-use user:
  Broke: REJECT: [ID]: snub, [FR]: snub, and [PPU]: snub.
  Rich : ADMIT: [ID]: snub, [FR]: snub, and [PPU]: accept.

4 Associating Policies to Flows

The LPM uses RSVP's criteria for identifying flows; Each policy element is associated with a specific [SESSION, RSVP_HOP, FILTER_SPEC list] combination.

For every POLICY_DATA object, the SESSION information is provided by RSVP (as parameter in the PC_xxxx() calls) and the RSVP_HOP and FILTER_SPEC list are embedded in the POLICY_DATA object itself. When no FILTER_SPECs are provided, the object is assumed to be associated with all the flows of the session. When no RSVP_HOP is provided, the POLICY_DATA object is assumed to have been assembled by the RSVP_HOP listed in the RSVP message. (i.e., the neighboring RSVP next/previous hop).

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

5 LPM Policy Control (PC) Functions

In this section we provide a rough outline of the basic operations required for each service supported by the LPM. We use the notation *xxxx_rtn()* to represent a call to function rtn() for element of P-type xxxx.

- Process a received POLICY_DATA object

```
Call:  PC_InPolicy   (session, lih, rsvp_hop, message_type,
                      in_policy_objects, resv_handle,
                      resv_flowspec, timeout)
                      -> RCode
```

```
PD_List = PC_Reassemble(session, rsvp_hop, in_policy_objects->OID, ...);
Flowp = Locate_state(session, PD_List->filter_spec_list);
for (each PD in PD_List) {
    if (!Integrity_check(PD)
        error();
    for (each P_element_xxxx in PD)
        if (!xxxx_InPolicy(..., Flowp, PA, rsvp_hop, message_type, timeout)
            error();
}
return PC_AuthCheck(session, lih, message_type,
                    PD_List->filter_spec_list,
                    resv_handle, resv_flowspec);
```

- Request an outgoing POLICY_DATA object

```
Call:  PC_OutPolicy (session, filter_spec_list,
                      lih, rsvp_hop, message_type,
                      out_policy_objects,
                      max_pd, avail_pd)
                      -> RCode
```

```
Flowp = locate_state(session, filter_spec_list);
element_list = NULL;
for (each active element handler of P-type xxxx) {
    policy_element = xxxx_OutPolicy(..., flowp, rsvp_hop, message_type,... )
    add_element(element_list, policy_element);
}
PD_List = PC_Fragment(element_list, max_pd, avail_pd)
return (PD_List);
```

- Check the status of an existing reservation

```
Call:  PC_AuthCheck (session, filter_spec_list,
                     lih, message_type, resv_handle,
                     resv_flowspec, ind)
                     -> RCode

Flowp = locate_state(session, filter_spec_list);
for (each active element handler of P-type xxxx) {
    curr_status = xxxx_AuthCheck(Flowp, lih, message_type,
                                 resv_handle, resv_flowspec,...)
    merge_status(status, curr_status);
}
purge_old_state();
return (status);
```

- Initialize Policy Control services

```
Call:  PC_Init (void) -> RCode

for (each active element handler of P-type xxxx)
    xxxx_Init()
```

- Synchronize RSVP and policy control state (see Section 6.3).

```
Call:  PC_Branch     (session, filter_spec_list,
                     rsvp_hop, op_type)
                     -> RCode

Flowp = locate_state(session, filter_spec_list);
for (each active element handler of P-type xxxx)
    xxxx_Branch(..., flowp, rsvp_hop, op_type,... )
```

- Delete policy control state (see Section 6.4)

```
Call:  PC_Close (session, filter_spec_list) -> RCode

Flowp = locate_state(session, filter_spec_list);
for (each active element handler of P-type xxxx)
    xxxx_Close(flowp)
```

5.1 Error Signaling

As described in [Ext], policy errors are handled by RSVP in two phases: (1) RSVP is notified about the error (a return codes from either `PC_AuthCheck()` or `PC_InPolicy()`). (2) RSVP prepares a standard error message (PathErr or ResvErr), queries `PC_OutPolicy()` for an outgoing error policy object, and embeds the object in the outgoing error message.

Error Signaling is opaque to the LPM common-layer as well; it simply demultiplexes the `PC_OutPolicy()` call to the active handlers. When `xxxx_OutPolicy()` is called with a message type of either PathErr or ResvErr, individual policy handlers use a last-error cache to generate outgoing error policy elements. These elements are assembles by the LPM common-layer into a single POLICY_DATA object that is returned to RSVP.

6 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 imitate RSVP's state management mechanisms:

6.1 Time-out:

When new POLICY_DATA objects cease to arrive (either as a result of a change of policy or a fragment loss) the locally stored state begins to age. Each policy-element 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 (*PC_InPolicy*()) it provides a time-out value. Each time RSVP verifies the status of a reservation (*PC_AuthCheck*()), the LPM examines its internal state, purging old state.

6.2 Instantaneous Policy Replacement

In some cases, policies must be replaced or purged immediately.[2] Instantaneous replacement is especially critical to avoid over-charging when accounting or other debiting policies are in effect. We propose a very simple rule:

> Only one instance of a policy element type (P-type) is allowed for any given flow.

Following this rule, a modified policy-element would immediately purge the old one. Purging an old policy-element without installing a new one can be done simply by sending an empty policy-element (with the 32bit header only). As added benefit, this rule guarantees that the receiving LPM would never face multiple, contradictory provisions of the same policy-type.[3]

[2] As opposed to stopping refreshes and waiting for the time-out mechanism to purge old state.

[3] "There can be only one", Highlander.

6.3 Tree/Branch maintenance:

When the shape of the session (multicast) tree changes due to route changes, teardown messages, or blockade state, RSVP must notify the LPM about the change. (See Section 5, *PC_Branch()* for more details.)

6.4 Closing:

The call *PC_Close(session, filter-spec list)* purges all the state linked to the session and filter-spec list. Closing a Policy Association is done when RSVP no longer maintains any state associated with that flow (all senders quit). Notice that on-going operations (e.g., accounting) must be shut-down in an orderly manner before the state is purged.

7 Syntactic Fragmentation of Large Policy Data Objects

Document [Ext] describe syntactic fragmentation of large POLICY_DATA objects from RSVP's view point. In this section, we describe the LPM support for this syntactic fragmentation.

7.1 Fragmentation

When RSVP queries the LPM for outgoing policy objects (`PC_OutPolicy()`) it provide the LPM with two size parameters: *max_pd* (desired maximal object size), and *avail_pd* (available space in the outgoing RSVP message). The general fragmentation rules for a POLICY_DATA (PD) object is:

```
PC_Fragment(session, ..., pd_list, max_pd, avail_pd)

if (avail_pd < MIN_POL_OBJ)          /* Minimal object size */
   return NULL;

/* Assemble the full PD */

if (size of PD <= avail_pd)          /* No need for fragmentation */
   pd_list = PD;
else {
   oid=Pick_Oid(session);
   pd_list=Fragment(max_pd, PD, oid);
   Add_PD_Token(pd_list, oid);
}
return pd_list;
```

Remarks:

- Pick_Oid() picks an object ID for the outgoing POLICY_DATA object. (See [Ext] for OID selection criteria).
- Fragment() fragments the PD object to the desired size (max_pd). Notice that if it cannot fulfill the desired size it should attempt to get as close to it as it can, and let IP fragmentation handle if from there.[4] All the fragments are marked by including the Fragmentation option, and the OID is placed in their header.
- Add_PD_Token() completes the syntactic fragmentation by creating a token object with minimal header of size MIN_POL_OBJ. The selected oid is placed in the token's header, and the token is concatenated to the end of the pd_list. This token object will be embedded in the standard outgoing RSVP message.

7.2 Reassembly

```
PC_Reassemble(Session, rsvp_hop, ..., in_PD, time-out)

pd_list = Get_PDList(session, rsvp_hop, in_PD->oid);
if (IS_FRAGMENT(in_PD)) {
   Add_PD_Fragment(pd_list, in_PD, time-out);
   return;
}
Reassemble(pd_list);  /* no need to add PD to list */
Reset_Fragments(...);
```

[4]Because of syntactic fragmentation, the RSVP control message contains only a small policy token, and therefore, do not incur significant added loss risk. The impact of a failed IP fragmentation on a large policy object is limited to the loss of the policy object itself.

Remarks:

- Get_PDList locates or creates a fragment list associated with a session/rsvp_hop/oid triplet.
- Add_PD_Fragment() adds a fragment to the pd_list.
- Reassemble() processes the received list of fragments for that triplet.
- Reset_Fragments(): The management of fragments is one of the issues at local discretion. For example, fragments that arrive after the token object may be (1) purged immediately or (2) combined with earlier fragments to generate a more complete POLICY_DATA object. If (1) is chosen, *Reset_Fragments*() simply purges all the previously received fragments each time. If (2) is chosen, a timeout mechanism must be provided to purge old fragments.

7.3 IP Style vs. Semantic Fragmentation

The actual fragmentation method is determined by the Fragment/Reassembly; it is therefore orthogonal to the syntactic fragmentation mechanism.

Two common fragmentation policies are:

- IP and IP style fragmentation:

  The simplest approach could be to use IP fragmentation. First, the large POLICY_DATA object would be sent by a single vacuous RSVP message, that would undergo IP fragmentation. A lost fragment would result in loosing the entire POLICY_DATA object, however RSVP would not be adversely effected. Immediately following the vacuous message, a standard RSVP message is sent with a minimal policy token embedded in it. The LPM can also perform IP style fragmentation itself if/when the limitations of IP fragmentation (e.g., maximum of 64K bytes message) become a problem.

- Semantic Fragmentation:

  Semantic fragmentation is highly context sensitive and at least in the case of the RSVP protocol [RSVPSP] was proven to be a formidable problem. In Appendix B we outline a possible approach to semantic fragmentation.

8 LPM Security

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; 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 POLICY_DATA objects.

However, when policy is enforced only at border nodes (cloud entry and exit points), RSVP's hop-by-hop security is insufficient to protect policy objects; from a policy control perspective, the in-cloud nodes are unsecured, and might be unlawfully manipulating policy objects that pass through them. The solution is to have a secure "policy tunnel", that creates logical policy topology, on top of which security is enforced.

The secure, automatic tunnel is created by adding an INTEGRITY object to each policy data object assembled by a border node. When the policy object is received by the next border/policy node, the integrity envelope guarantees that none of the intermediate non-policy-aware (and unsecured) RSVP nodes have modified the object's contents.

One of the advantages of automatic tunneling is that it can use the same or similar key distribution mechanisms as advocated for RSVP in [Bak96] since it complies with the hop-by-hop security model. Here, the previous/next hops are the policy-capable (as opposed to directly connected) neighboring RSVP nodes.

9 LPM Configuration

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. Because of the early stages of this work, we believe it is too early to provide specific configuration details.

9.1 Interaction Between Handlers

Independent element types may require some interaction between their handlers. Consider the case where policy type-1 computes the cost of a flow, while type-2 performs actual debiting of a user/group account based on the this computed cost (e.g., credit card account). Such interaction has two basic requirements: order dependency and export capability. In our example, type-1 must calculate the cost before type-2 is activated (such partial ordering may be set as part of the local configuration process). Export capability is required, in this case, to allow type-1 to export the calculation results to type-2. The simplest approach could be to allow inter-handler function calls.

In some cases, a single element handler may be capable of interacting with multiple equivalent peer handlers. In our example, once type-1 determined the cost, there could be several accounts available for debiting (Visa, MasterCard, AmEx etc.) each handled by a different element type (type-2, type-3, type-4). Local configuration may enforce the use of a certain card by binding type-1 with a particular card handler, e.g., AmEx/type-4. Configuration may also set a certain order such that the lower cards on the list would be debited only after the previous ones have been attempted and failed.

10 Acknowledgment

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

References

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

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

[Her95] S. Herzog, S. Shenker, and D. Estrin. Sharing the Cost of Multicast Trees: An Axiomatic Analysis. *Proceedings of ACM SIGCOMM '95*, Aug. 1995.

[Ext] S. Herzog RSVP Extensions for Policy Control. *Internet-Draft*, draft-ietf-rsvp-policy-ext-01.[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.

Author's Address

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

Phone: (914) 784-6059
Email: herzog@watson.ibm.com

A A List of Currently Defined Policies

Usage policies are assumed to be local by nature, however, interoperable framework implies the need to standardize the format and contents of inter-provider policies. The policy element space is partitioned accordingly into two ranges; the lower range is reserved for globally meaningful policies, while the upper range is set aside for purely local policies. The following list details the currently defined policy elements:

0: NULL/default

This is a reserved type that is used for unrecognized objects.

1: String Credentials (Sender or Receiver)

A single character string.
The string must be NULL-terminated and padded to multiple of 32 bits.

2: Reservation Ack

In its simplest form, this policy requires only a header and no actual policy information.

If included in Resv messages, it requests an ack of the current reservation. When included in the Path message, it confirms the Reservation succeeded all the way to the source (end-to-end).

3: MultiCost (Cost Allocation) [Her95]

Upstream Format:
FLOW_SPEC object,
32bit unsigned counter: the number of downstream members for FLOW_SPEC.

Downstream Format:
FLOW_SPEC object,
32bit unsigned counter: cost unit type
Double Precision Float: allocated cost (units)

Local Policies:

All P-types values that are not defined in this document are available for local use. We recommend that you choose local policy P-types starting at 65535 and going down, to prevent conflicts when the list in this appendix grows.

B Semantic Fragmentation

IP style fragmentation is best suited for cases where nothing but the complete set of fragments will do. Policies enjoy a different semantics. They are compiled from the start as a list of smaller, independent elements, which makes it ideal for semantic fragmentation. When policies are fragmented into independent elements, the loss of some elements does not invalidate others that were received properly. Moreover, the received elements can be incrementally added to form a workable (even if not complete) policy. The consequences are that there is no need for fragmentation negotiation between the sender and receiver; the sender may fragment the object into its desired level of details. The receiver may use its preferred reassembly policy. (i.e., what to do when fragments are missing).

Semantic fragmentation imposes an added burden on state management since the absence of a policy-element is ambiguous. Consider the case were a new policy element P_i is introduced but an older policy element P_j is lost. One option would be to apply only the new policy in P_i, but another could be to use the previously received P_j along with the new P_i to maintain consistency (Assuming the state in P_j had not timed out yet). This implies the need for a time-out (and possibly a teardown) mechanism for each {policy-element, FILTER_SPEC} object pair.

B.1 Fragmentation Example

Semantic fragmentation is context sensitive and therefore can only be performed by the same handlers that assemble specific policy elements and understand their internal semantics. Clearly, the following example is not universal since it assumes specific policy semantics.

Consider the following fragmentation example where S_i denotes a FilterSpec for sender i, $P_j[i..j]$ represents a policy element that is associated with individual senders from the set $(S_i, ..., S_j)$, and $*$ is a wildcard, all senders, operator. Let us further assume that the original POLICY_DATA object is:

(1) $S_1, ..., S_n, P_1[i..j], P_2[k..l], P_3[*]$

First, we can separate the different policy elements since each of them is an independent unit. (1 into 2.1 + 2.2 + 2.3)

(2.1) $S_1, ..., S_n, P_1[i..j]$
(2.2) $S_1, ..., S_n, P_2[k..l]$
(2.3) $S_1, ..., S_n, P_3[*]$

Now, we can compress the source list by eliminating irrelevant sources:

(3.1) $S_i, ..., S_j, P_1[i..j]$
(3.2) $S_k, ..., S_l, P_2[k..l]$

(3.3) $S_1, ..., S_n, P_3[*]$

Finally, we can break each non-wildcard policy element and attach it to its corresponding filter: (3.1 into 4.1.i..4.1.j)

(4.1.i) $S_i, P_1[i]$
...
(4.1.j) $S_j, P_1[j]$

We could do the same for 3.2, however, P_3 could not be broken to smaller semantic pieces since is a wildcard policy.