Internet Draft
Expires October 23, 1997
draft-ietf-rps-rpsl-02.txt

Cengiz Alaettinoglu
USC/ISI
Tony Bates
Cisco Systems
Elise Gerich
At Home Network
Daniel Karrenberg
RIPE
David Meyer
University of Oregon
Marten Terpstra
Bay Networks
Curtis Villamizer
ANS
April 23, 1997

Routing Policy Specification Language (RPSL)

Status of this Memo

This Internet Draft is the reference document for the Routing Policy Specification Language (RPSL). RPSL allows a network operator to be able to specify routing policies at various levels in the Internet hierarchy; for example at the Autonomous System (AS) level. At the same time, policies can be specified with sufficient detail in RPSL so that low level router configurations can be generated from them. RPSL is extensible; new routing protocols and new protocol features can be introduced at any time.

This document is an Internet Draft, and can be found as draft-ietf-rps-rpsl-02.txt in any standard internet drafts repository. 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. Internet Drafts may be updated, replaced, or obsoleted by other documents at any time. It is not appropriate to use Internet Drafts as reference material, or to cite them other than as a "working draft" or "work in progress."

Please check the I-D abstract listing contained in each Internet Draft directory to learn the current status of this or any other Internet Draft.

Contents

1 Introduction 4

2 RPSL Names, Reserved Words, and Representation 4

3 `mntner` Class 6

4 `person` Class 8

5 `route` Class 8

6 Set Classes 10

6.1 `route-set` Class . . . 10

6.2 `as-set` Class . . . 11

6.3 Predefined Set Objects . . . 13

6.4 Hierarchical Set Names . . . 13

7 `aut-num` Class 13

7.1 `import` Attribute: Import Policy Specification . . . 13

7.1.1 Peering Specification . . . 15

7.1.2 Action Specification . . . 16

7.1.3 Filter Specification . . . 17

7.1.4 Example Policy Expressions . . . 20

7.2 `export` Attribute: Export Policy Specification . . . 21

7.3 Other Routing Protocols, Multi-Protocol Routing Protocols, and Injecting Routes Between Protocols . . . 22

7.4 Ambiguity Resolution . . . 23

7.5 `default` Attribute: Default Policy Specification . . . 25

7.6 Structured Policy Specification 26

8 `dictionary` Class **30**

8.1 Initial RPSL Dictionary and Example Policy Actions and Filters 33

9 Advanced `route` Class **37**

9.1 Specifying Static Routes 37

9.2 Specifying Aggregate Routes 38

10 `inet-rtr` Class **39**

11 `inet-tunnel` Class and Specifying Tunnels **41**

12 Security Consideration **42**

13 Acknowledgements 44

A Routing Registry Sites **46**

B Authors' Addresses **46**

1 Introduction

This Internet Draft is the reference document for the Routing Policy Specification Language (RPSL). RPSL allows a network operator to be able to specify routing policies at various levels in the Internet hierarchy; for example at the Autonomous System (AS) level. At the same time, policies can be specified with sufficient detail in RPSL so that low level router configurations can be generated from them. RPSL is extensible; new routing protocols and new protocol features can be introduced at any time.

RPSL is a replacement for the current Internet de-facto standard routing policy specification language known as RIPE-181 [6] or RFC-1786 [7]. RIPE-81 [8] was the first language deployed in the Internet for specifying routing policies. It was later replaced by RIPE-181 [6].

Through operational use of RIPE-181 it has become apparent that certain policies cannot be specified and a need for an enhanced and more generalized language is needed. RPSL addresses RIPE-181's limitations. RPSL is object oriented; that is, objects contain pieces of policy and administrative information. These objects are registered in the Internet Routing Registry (IRR) by the authorized organizations. The registration process is beyond the scope of this document. Please refer to [2] and [4] for more details on the IRR.

In the following sections, we present the classes that are used to define various policy and administrative objects. The "*mntner*" class defines entities authorized to add, delete and modify a set of objects. The "*person*" class describes technical and administrative contact personnel. Autonomous systems (ASes) are specified using the "*aut-num*" class. Routes are specified using the "*route*" class. Sets of ASes and routes can be defined using the "*as-set*" and "*route-set*" classes. The "*dictionary*" class provides the extensibility to the language. The "*inet-rtr*" class is used to specify routers. Tunnels are specified using "*inet-tunnel*" class. Many of these classes were originally defined in earlier documents [6, 18, 20, 17, 5] and have all been enhanced.

This document is self-contained. However, the reader is encouraged to read RIPE-181 [7] and the associated documents [18, 20, 17, 5] as they provide significant background as to the motivation and underlying principles behind RIPE-181 and consequently, RPSL. They further cover the basic concept of the Internet Routing Registry (IRR) [2, 4], the data repository for storing global RPSL based routing policies and a fundamental component in the application of RPSL. For a tutorial on RPSL, the reader should read the RPSL applications document [4].

2 RPSL Names, Reserved Words, and Representation

Each class has a set of attributes which store a piece of information about the objects of the class. Attributes can be mandatory or optional: A mandatory attribute has to be defined for all objects of the class; optional attributes can be skipped. Attributes can also be single or multiple valued. Each object is uniquely identified by a set of attributes, referred to as the class "key".

The value of an attribute has a type. The following types are most widely used. Note that RPSL is case insensitive.

⟨*object-name*⟩ Many objects in RPSL have a name. An ⟨*object-name*⟩ is made up of letters, digits, the character underscore "_", and the character hyphen "-"; the first character of a name must be a letter, and the last character of a name must be a letter or a digit. The following words are reserved by RPSL, and they can not be used as names:

```
any as-any rs-any peeras
and or not
atomic from to at action accept announce except refine
networks into
```

Names starting with certain prefixes are reserved for certain object types. Names starting with "`as-`" are reserved for as set names. Names starting with "`rs-`" are reserved for route set names.

⟨*as-number*⟩ An AS number x is represented as the string "`AS`x". That is, the AS 226 is represented as `AS226`.

⟨*ipv4-address*⟩ An IPv4 address is represented as a sequence of four integers in the range from 0 to 255 separated by the character dot ".". For example, `128.9.128.5` represents a valid IPv4 address. In the rest of this document, we may refer to IPv4 addresses as IP addresses.

⟨*address-prefix*⟩ An address prefix is represented as an IPv4 address followed by the character slash "/" followed by an integer in the range from 0 to 32. The following are valid address prefixes: `128.9.128.5/32`, `128.9.0.0/16`, `0.0.0.0/0`; and the following address prefixes are invalid: `0/0`, `128.9/16` since `0` or `128.9` are not strings containing four integers.

⟨*date*⟩ A date is represented as an eight digit integer of the form `YYYYMMDD` where `YYYY` represents the year, `MM` represents the month of the year (01 through 12), and `DD` represents the day of the month (01 through 31). For example, June 24, 1996 is represented as 19960624.

⟨*email-address*⟩ is as described in RFC-822[11].

⟨*dns-name*⟩ is as described in RFC-1034[22].

⟨*person*⟩ is either a full name of a person or a uniquely assigned NIC-handle. Its syntax has the following form:

```
  <firstname> [<initials>] <lastname>
| <nic-handle>

 E.g.
    John E Doe
    JED31
```

A NIC handle is an identifier used by routing, address allocation, and other registries to unambiguously refer to people.

⟨*free-form*⟩ is a sequence of ASCII characters.

⟨`X`-*object-name*⟩ is a name of an object of type `X`. That is ⟨`mntner`-*object-name*⟩ is a name of an `mntner` object.

⟨*registry-name*⟩ is a name of an IRR registry. The routing registries are listed in Appendix A.

A value of an attribute may also be a *lists* of one of these types. A list is represented by separating the list members by commas ",". For example, "`AS1, AS2, AS3, AS4`" is a list of AS numbers. Note that being list valued and being multiple valued are orthogonal. A multiple valued attribute has more than one value each of which may or may not be a list depending on the attribute. On the other hand a single valued attribute may have a list value.

An RPSL object is textually represented as a list of attribute-value pairs. Each attribute-value pair is written on a separate line. The attribute name starts at column 0, followed by character ":" and followed by the value of the attribute. The object's representation ends when a blank line is encountered. An attribute's value can be split over multiple lines, by starting the continuation lines with a white-space (" " or tab) character. The order of attribute-value pairs is significant.

An object's description may contain comments. A comment can be anywhere in an object's definition, it starts at the first "#" character on a line and ends at the first end-of-line character. White space characters can be used to improve readability.

3 `mntner` Class

The `mntner` class defines entities that can create, delete and update RPSL objects. A provider, before he/she can create any RPSL object, first needs to create a `mntner` object. The attributes of the `mntner` class are shown in Figure 1. A more complete description of `mntner` class can be found in [18]. Here, we summarize the `mntner` class for completeness.

The `mntner` attribute is mandatory and is the class key attribute. Its value is an RPSL name. The `auth` attribute specifies the scheme that will be used to identify and authenticate update requests from this maintainer. It has the following syntax:

```
auth: <scheme-id> <auth-info>

E.g.
        auth: NONE
        auth: CRYPT-PW dhjsdfhruewf
        auth: MAIL-FROM .*@ripe\.net
```

Attribute	Value	Type
`mntner`	⟨*object-name*⟩	mandatory, single-valued, class key
`descr`	⟨*free-form*⟩	mandatory, single-valued
`auth`	see description in text	mandatory, multi-valued
`upd-to`	⟨*email-address*⟩	mandatory, multi-valued
`mnt-nfy`	⟨*email-address*⟩	optional, multi-valued
`tech-c`	⟨*person*⟩	mandatory, multi-valued
`admin-c`	⟨*person*⟩	mandatory, multi-valued
`remarks`	⟨*free-form*⟩	optional, multi-valued
`notify`	⟨*email-address*⟩	optional, multi-valued
`mnt-by`	⟨`mntner`-*object-name*⟩	mandatory, multi-valued
`changed`	⟨*email-address*⟩ ⟨*date*⟩	mandatory, multi-valued
`source`	⟨*registry-name*⟩	mandatory, single-valued

Figure 1: `mntner` Class Attributes

The `<scheme-id>`'s currently defined are: `NONE`, `MAIL-FROM`, `PGP` and `CRYPT-PW`. The `<auth-info>` is additional information required by a particular scheme: in the case of `MAIL-FROM`, it is a regular expression matching valid email addresses; in the case of `CRYPT-PW`, it is a password in UNIX crypt format; and in the case of `PGP`, it is a PGP public key. If multiple `auth` attributes are specified, an update request satisfying *any* one of them is authenticated to be from the maintainer.

The `upd-to` attribute is an email address. On an unauthorized update attempt of an object maintained by this maintainer, an email message will be sent to this address. The `mnt-nfy` attribute is an email address. A notification message will be forwarded to this email address whenever an object maintained by this maintainer is added, changed or deleted.

The `descr` attribute is a short, free-form textual description of the object. The `tech-c` attribute is a technical contact person. This is someone to be contacted for technical problems such as misconfiguration. The `admin-c` attribute is an administrative contact person. The `remarks` attribute is a free text explanation or clarification. The `notify` attribute is an email address to which notifications of changes to this object should be sent. The `mnt-by` attribute is a `mntner` object name. The authorization for changes to this object is governed by that maintainer object. The `changed` attribute documents who last changed this object, and when this change was made. Its syntax has the following form:

```
changed: <email-address> <YYYYMMDD>

E.g.
changed: johndoe@terabit-labs.nn 19900401
```

The `<email-address>` identifies the person who made the last change. `<YYYYMMDD>` is the date of

the change. The `source` attribute specifies the registry where the object is registered.

The `descr`, `tech-c`, `admin-c`, `remarks`, `notify`, `mnt-by`, `changed` and `source` attributes are attributes of all RPSL classes. Their syntax, semantics, and mandatory, optional, multi-valued, or single-valued status are the same for for all classes. We do not further discuss them in other sections.

4 `person` Class

A `person` class is used to describe information about people. Even though it does not describe routing policy, we still describe it here briefly since many policy objects make reference to `person` objects. The details of the `person` class can be found in Reference [20].

The attributes of the `person` class are shown in Figure 2. The `person` attribute is the full name of the person. The `phone` and the `fax-no` attributes have the following syntax:

```
   phone: +<country-code> <city> <subscriber> [ext. <extension>]

E.g.:
   phone: +31 20 12334676
   phone: +44 123 987654 ext. 4711
```

Attribute	Value	Type
`person`	⟨*person*⟩	mandatory, single-valued, class key
`address`	⟨*free-form*⟩	mandatory, multi-valued
`phone`	see description in text	mandatory, multi-valued
`fax-no`	same as phone	optional, multi-valued
`e-mail`	⟨*email-address*⟩	mandatory, multi-valued
`nic-hdl`	see description in text	optional, single-valued

Figure 2: `person` Class Attributes

5 `route` Class

Each interAS route originated by an AS is specified using a `route` object. The attributes of the `route` class are shown in Figure 3. The `route` attribute is the address prefix of the route and the `origin` attribute is the AS number of the AS that originates the route into the interAS routing system. The `route` and `origin` attribute pair is the class key.

Attribute	Value	Type
`route`	⟨*address-prefix*⟩	mandatory, single-valued, class key
`origin`	⟨*as-number*⟩	mandatory, single-valued, class key
`withdrawn`	⟨*date*⟩	optional, single-valued
`member-of`	⟨`route-set`-*object-name*⟩ see Section 6	optional, single-valued
`inject-at`	see Section 9	optional, multi-valued
`aggr-by`	see Section 9	optional, single-valued
`export-comps`	see Section 9	optional, single-valued
`holes`	see Section 9	optional, single-valued

Figure 3: `route` Class Attributes

The Figure 4 shows examples of four `route` objects. Note that the last two `route` objects have the same address prefix, namely `128.8.0.0/16`. However, they are different route objects since they are originated by different ASes (i.e. they have different keys).

```
route: 128.9.0.0/16
origin: AS226

route: 128.99.0.0/16
origin: AS226

route: 128.8.0.0/16
origin: AS1

route: 128.8.0.0/16
origin: AS2
withdrawn: 19960624
```

Figure 4: Route Objects

The `withdrawn` attribute, if present, signifies that the originator AS no longer originates this address prefix in the Internet. Its value is a date indicating the date of withdrawal. In Figure 4, the last route object is withdrawn (i.e. no longer originated by `AS2`) on June 24, 1996.

6 Set Classes

To specify policies, it is often useful to define sets of objects. For this purpose we define two classes: `route-set` and `as-set`. These classes define a named set. The members of these sets can be specified by either explicitly listing them in the set object's definition, or implicitly by having `route` and `aut-num` objects refer to the set name in their definitions, or a combination of both methods.

6.1 `route-set` Class

The attributes of the `route-set` class are shown in Figure 5. The `route-set` attribute defines the name of the set. It is an RPSL name that starts with "`rs-`". The `members` attribute lists the members of the set. The `members` attribute is a list of address prefixes or other route-set names.

Attribute	Value	Type
`route-set`	⟨*object-name*⟩	mandatory, single-valued, class key
`members`	list of ⟨*address-prefixes*⟩ or ⟨`route-set-object-names`⟩	optional, single-valued
`members-by-referral`	list of ⟨`mntner`-*object-names*⟩	optional, single-valued

Figure 5: `route-set` Class Attributes

Figure 6 presents some example `route-set` objects. The set `rs-foo` contains two address prefixes, namely `128.9.0.0/16` and `128.9.0.0/16`. The set `rs-bar` contains the members of the set `rs-foo` and the address prefix `128.7.0.0/16`. The set `rs-empty` contains no members.

```
route-set: rs-foo
members: 128.9.0.0/16, 128.9.0.0/24

route-set: rs-bar
members: 128.7.0.0/16, rs-foo

route-set: rs-empty
```

Figure 6: `route-set` Objects

The `members-by-referral` attribute is a list of maintainer names or the keyword `ANY`. If this attribute is used, the route set also includes address prefixes whose `route` objects are registered by one of these maintainers and whose `member-of` attribute refers to the name of this route set.

If the value of a `members-by-referral` attribute is `ANY`, any `route` object referring to the route set name is a member. If the `members-by-referral` attribute is missing, only the address prefixes listed in the `members` attribute are members of the set.

```
route-set: rs-foo
members-by-referral: MNTR-ME, MNTR-YOU

route-set: rs-bar
members: 128.7.0.0/16
members-by-referral: MNTR-YOU

route: 128.9.0.0/16
origin: AS1
member-of: rs-foo
mnt-by: MNTR-ME

route: 128.8.0.0/16
origin: AS2
member-of: rs-foo, rs-bar
mnt-by: MNTR-YOU
```

Figure 7: `route-set` objects.

Figure 7 presents example `route-set` objects that use the `members-by-referral` attribute. The set `rs-foo` contains two address prefixes, namely `128.8.0.0/16` and `128.9.0.0/16` since the `route` objects for `128.8.0.0/16` and `128.9.0.0/16` refer to the set name `rs-foo` in their `member-of` attribute. The set `rs-bar` contains the address prefixes `128.7.0.0/16` and `128.8.0.0/16`. The route `128.7.0.0/16` is explicitly listed in the `members` attribute of `rs-bar`, and the `route` object for `128.8.0.0/16` refer to the set name `rs-bar` in its `member-of` attribute.

Note that, if an address prefix is listed in a `members` attribute of a route set, it is a member of that route set. The `route` object corresponding to this address prefix does not need to contain a `member-of` attribute referring to this set name. The `member-of` attribute of the `route` class is an additional mechanism for specifying the members indirectly.

6.2 `as-set` Class

The attributes of the `as-set` class are shown in Figure 8. The `as-set` attribute defines the name of the set. It is an RPSL name that starts with "`as-`". The `members` attribute lists the members of the set. The `members` attribute is a list of AS numbers, or other `as-set` names.

Figure 9 presents two `as-set` objects. The set `as-foo` contains two ASes, namely `AS1` and `AS2`. The set `as-bar` contains the members of the set `as-foo` and `AS3`, that is it contains `AS1, AS2,`

Attribute	Value	Type
`as-set`	⟨*object-name*⟩	mandatory, single-valued, class key
`members`	list of ⟨*as-numbers*⟩ or ⟨`as-set-object-names`⟩	optional, single-valued
`members-by-referral`	list of ⟨`mntner-`*object-names*⟩	optional, single-valued

Figure 8: `as-set` Class Attributes

`AS3`.

```
as-set: as-foo                    as-set: as-bar
members: AS1, AS2                 members: AS3, as-foo
```

Figure 9: `as-set` objects.

The `members-by-referral` attribute is a list of maintainer names or the keyword `ANY`. If this attribute is used, the AS set also includes ASes whose `aut-num` objects are registered by one of these maintainers and whose `member-of` attribute refers to the name of this AS set. If the value of a `members-by-referral` attribute is `ANY`, any AS object referring to the AS set is a member of the set. If the `members-by-referral` attribute is missing, only the ASes listed in the `members` attribute are members of the set.

```
as-set: as-foo
members: AS1, AS2
members-by-referral: MNTR-ME

aut-num: AS3                      aut-num: AS4
member-of: as-foo                 member-of: as-foo
mnt-by: MNTR-ME                   mnt-by: MNTR-OTHER
```

Figure 10: `as-set` objects.

Figure 10 presents an example `as-set` object that uses the `members-by-referral` attribute. The set `as-foo` contains `AS1`, `AS2` and `AS3`. `AS4` is not a member of the set `as-foo` even though the `aut-num` object references `as-foo`. This is because `MNTR-OTHER` is not listed in the `as-foo`'s `members-by-referral` attribute.

6.3 Predefined Set Objects

In a context that expects a route set (e.g. `members` attribute of the `route-set` class), an AS number `AS`x defines the set of routes that are originated by `AS`x; and an `as-set` `AS-`X defines the set of routes that are originated by the ASes in `AS-`X. A route p is said to be originated by `AS`x if there is a `route` object for p with `AS`x as the value of the `origin` attribute. For example, in Figure 11, the route set `rs-special` contains `128.9.0.0/16`, routes of `AS1` and `AS2`, and routes of the ASes in AS set `AS-FOO`.

```
route-set: rs-special
members: 128.9.0.0/16, AS1, AS2, AS-FOO
```

Figure 11: Use of AS numbers and AS sets in route sets.

The keyword `rs-any` defines the set of all routes registered in IRR. The keyword `as-any` defines the set of all ASes registered in IRR.

6.4 Hierarchical Set Names

Set names can be hierarchical. A hierarchical set name is a sequence of set names and AS numbers separated by colons ":". For example, the following names are valid: `AS1:AS-CUSTOMERS`, `AS1:RS-EXCEPTIONS`, `AS1:RS-EXPORT:AS2`, `RS-EXCEPTIONS:RS-BOGUS`. All set names in an hierarchical `as-set` name should start with "`as-`"; and all set names in an hierarchical `route-set` name should start with "`rs-`".

A set object with name `X1:...:Xn-1:Xn` can only be created by the maintainer of the object with name `X1:...:Xn-1`. That is, only the maintainer of `AS1` can create a set with name `AS1:AS-FOO`; and only the maintainer of `AS1:AS-FOO` can create a set with name `AS1:AS-FOO:AS-BAR`.

7 `aut-num` Class

ASes are specified using the `aut-num` class. The attributes of the `aut-num` class are shown in Figure 12. The value of the `aut-num` attribute is the AS number of the AS described by this object. The `as-name` attribute is a symbolic name (in RPSL name syntax) of the AS. The import, export and default routing policies of the AS are specified using `import`, `export` and `default` attributes respectively.

7.1 `import` Attribute: Import Policy Specification

Attribute	Value	Type
`aut-num`	⟨*as-number*⟩	mandatory, single-valued, class key
`as-name`	⟨*object-name*⟩	mandatory, single-valued
`member-of`	⟨`as-set-`*object-name*⟩	optional, single-valued
`import`	see Section 7.1	optional, multi valued
`export`	see Section 7.2	optional, multi valued
`default`	see Section 7.5	optional, multi valued

Figure 12: `aut-num` Class Attributes

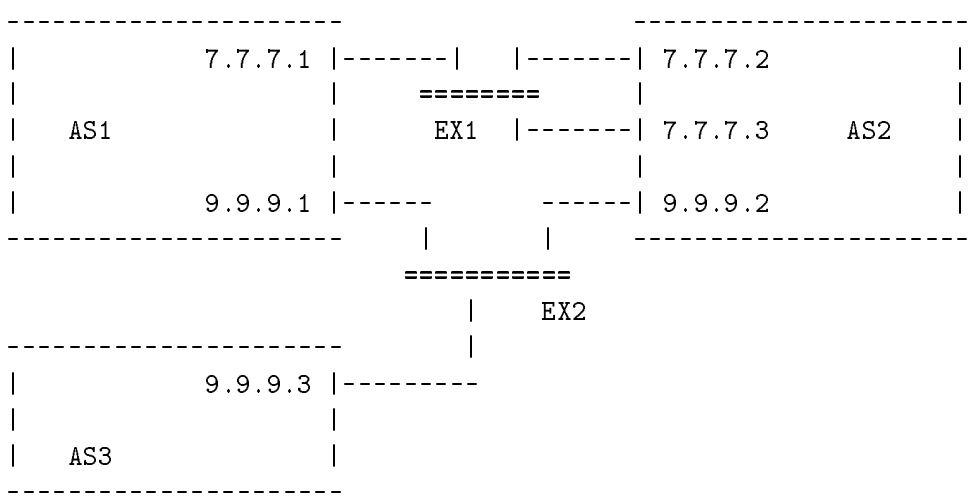

Figure 13: Example topology consisting of three ASes, `AS1`, `AS2`, and `AS3`; two exchange points, `EX1` and `EX2`; and six routers.

A typical interconnection of ASes is shown in Figure 13. In this example topology, there are three ASes, `AS1`, `AS2`, and `AS3`; two exchange points, `EX1` and `EX2`; and six routers. Routers connected to the same exchange point peer with each other, i.e. open a connection for exchanging routing information. Each router would export a subset of the routes it has to its peer routers. Peer routers would import a subset of these routes. A router while importing routes would set some route attributes. For example, `AS1` can assign higher preference values to the routes it imports from `AS2` so that it prefers `AS2` over `AS3`. While exporting routes, a router may also set some route attributes in order to affect route selection by its peers. For example, `AS2` may set the `MULTI-EXIT-DISCRIMINATOR` BGP attribute so that `AS1` prefers to use the router `9.9.9.2`. Most interAS policies are specified by specifying what route subsets can be imported or exported, and how the various route attributes are set and used.

In RPSL, an import policy is divided into import policy expressions. Each import policy expression

is specified using an `import` attribute. The `import` attribute has the following syntax (we will extend this syntax later in Sections 7.3 and 7.6):

```
import: from <peering-1> [action <action-1>]
        . . .
        from <peering-N> [action <action-N>]
        accept <filter>
```

The action specification is optional. The semantics of an import attribute is as follows: the set of routes that are matched by `<filter>` are imported from all the peers specified in `<peerings>`; while importing routes at `<peering-M>`, `<action-M>` is executed.

E.g.

```
aut-num: AS1
import: from AS2 action pref = 1; accept { 128.9.0.0/16 }
```

This example states that the route `128.9.0.0/16` is accepted from `AS2` with preference 1. In the next few subsections, we will describe how peerings, actions and filters are specified.

7.1.1 Peering Specification

Our example above used an AS number to specify peerings. The peerings can be specified at different granularities. The syntax of a peering specification is as follows:

```
  <peer-as> [<peer-router>] [at <local-router>]
| <as-set> [at <local-router>]
```

where `<local-router>` and `<peer-router>` are IP addresses of routers, `<peer-as>` is an AS number, and `<as-set>` is an AS set name. `<peer-as>` must be the AS number of `<peer-router>`. Both `<local-router>` and `<peer-router>` are optional. We first describe the semantics using the first form. If both `<local-router>` and `<peer-router>` are specified, this peering specification identifies only the peering between these two routers. If only `<local-router>` is specified, this peering specification identifies all the peerings between `<local-router>` and any of its peer routers in `<peer-as>`. If only `<peer-router>` is specified, this peering specification identifies all the peerings between any router in the local AS and `<peer-router>`. If neither `<local-router>` nor `<peer-router>` is specified, this peering specification identifies all the peerings between any router in the local AS and any router in `<peer-as>`. If the `<as-set>` form is used, the peering specification identifies all the peerings between `<local-router>` and any of its peer routers in one of the ASes in `<as-set>`. If `<local-router>` is not specified, the peering specification identifies all the peerings between any router in the local AS and any of its peer routers in one of the ASes in `<as-set>`.

We next give examples. Consider the topology of Figure 13 where AS1 has two routers 7.7.7.1 and 9.9.9.1; AS2 has three routers 7.7.7.2, 7.7.7.3 and 9.9.9.2; AS3 has one router 9.9.9.3. 7.7.7.1, 7.7.7.2 and 7.7.7.3 peer with each other; 9.9.9.1, 9.9.9.2 and 9.9.9.3 peer with each other. In example (1) below 7.7.7.1 imports 128.9.0.0/16 from 7.7.7.2.

```
(1) aut-num: AS1
    import: from AS2 7.7.7.2 at 7.7.7.1 accept { 128.9.0.0/16 }

(2) aut-num: AS1
    import: from AS2         at 7.7.7.1 accept { 128.9.0.0/16 }

(3) aut-num: AS1
    import: from AS2                    accept { 128.9.0.0/16 }

(4) as-set: AS-FOO
    members: AS2, AS3

    aut-num: AS1
    import: from AS-FOO      at 9.9.9.1 accept { 128.9.0.0/16 }

(5) aut-num: AS1
    import: from AS-FOO                 accept { 128.9.0.0/16 }

(6) aut-num: AS1
    import: from AS2         at 9.9.9.1 accept { 128.9.0.0/16 }
    import: from AS3         at 9.9.9.1 accept { 128.9.0.0/16 }

(7) aut-num: AS1
    import: from AS2                    accept { 128.9.0.0/16 }
    import: from AS3                    accept { 128.9.0.0/16 }
```

In example (2), 7.7.7.1 imports 128.9.0.0/16 from 7.7.7.2 and 7.7.7.3. In example (3), 7.7.7.1 imports 128.9.0.0/16 from 7.7.7.2 and 7.7.7.3, and 9.9.9.1 imports 128.9.0.0/16 from 9.9.9.2. In example (4), 9.9.9.1 imports 128.9.0.0/16 from 9.9.9.2 and 9.9.9.3. In example (5), 9.9.9.1 imports 128.9.0.0/16 from 9.9.9.2 and 9.9.9.3, and 7.7.7.1 imports 128.9.0.0/16 from 7.7.7.2 and 7.7.7.3. The example (4) and (5) are equivalent to examples (6) and (7) respectively.

7.1.2 Action Specification

Policy actions in RPSL set or modify route attributes, such as assigning a preference to a route, adding a community to the community attribute, or setting the MULTI-EXIT-DISCRIMINATOR attribute. Policy actions can also instruct routers to perform special operations, such as route flap

damping.

The *routing policy attributes* whose values can be modified in policy actions are specified in the RPSL dictionary. Please refer to Section 8 for a list of these attributes. Each action in RPSL is terminated by the character ';'. It is possible to form composite policy actions by listing them one after the other. In a composite policy action, the actions are executed left to right. For example,

```
aut-num: AS1
import: from AS2
        action pref = 10; med = 0; community.append(10250, {3561,10});
        accept { 128.9.0.0/16 }
```

sets pref to 10, and then med to 0.

7.1.3 Filter Specification

A policy filter is a logical expression which when applied to a set of routes returns a subset of these routes. We say that the policy filter matches the subset returned. The policy filter can match routes using any route attribute, such as the destination address prefix (or NLRI), AS-path, or community attributes.

The following policy filters can be used to select a subset of routes:

`ANY`
: The filter-keyword `ANY` matches all routes.

Address-Prefix Set
: This is an explicit list of address prefixes enclosed in braces '{' and '}'. The policy filter matches the set of routes whose destination address-prefix is in the set. For example:

```
{ 0.0.0.0/0 }
{ 128.9.0.0/16, 128.8.0.0/16, 128.7.128.0/17, 5.0.0.0/8 }
{ }
```

An address prefix can be optionally followed by an operator '^-', '^+', '^n', or '^n-m' where `n` and `m` are integers. `^-` operator is the *exclusive more specifics* operator; it stands for the more specifics of the address prefix excluding the address prefix itself. `^+` operator is the *inclusive more specifics* operator; it stands for the more specifics of the address prefix including the address prefix itself. `^n` operator, stands for all the length `n` specifics of the address prefix. `^n-m` operator, stands for all the length `n` to length `m` specifics of the address prefix. For example, the set

```
{ 5.0.0.0/8^+, 128.9.0.0/16^-, 30.0.0.0/8^16, 30.0.0.0/8^24-32 }
```

contains all the more specifics of 5.0.0.0/8 including 5.0.0.0/8, all the more specifics of 128.9.0.0/16 excluding 128.9.0.0/16, all the more specifics of 30.0.0.0/8 which are of length 16 such as 30.9.0.0/16, and all the more specifics of 30.0.0.0/8 which are of length 24 to 32 such as 30.9.9.96/28.

Route Set Name

A route set name matches the set of routes that are members of the set. A route set name may be a name of a route-set object, an AS number, or a name of an as-set object (AS numbers and as-set names implicitly define route sets; please see Section 6.3). For example:

```
aut-num: AS1
import: from AS2 action pref = 1; accept AS2
import: from AS2 action pref = 1; accept AS-FOO
import: from AS2 action pref = 1; accept RS-FOO
```

The keyword PeerAS can be used instead of the AS number of the peer AS. PeerAS is particularly useful when the peering is specified using an AS set. For example:

```
as-set: AS-FOO
members: AS2 AS3

aut-num: AS1
import: from AS-FOO action pref = 1; accept PeerAS
```

is same as:

```
aut-num: AS1
import: from AS2 action pref = 1; accept AS2
import: from AS3 action pref = 1; accept AS3
```

A route set name can also be followed by one of the operators '^-', '^+', '^n' or '^n-m'. These operators are distributive over the route sets. For example, { 5.0.0.0/8, 6.0.0.0/8 }^+ equals { 5.0.0.0/8^+, 6.0.0.0/8^+ }, and AS1^- equals all the exclusive more specifics of routes originated by AS1.

AS Path Regular Expressions

An AS-path regular expression can be used as a policy filter by enclosing the expression in '<' and '>'. An AS-path policy filter matches the set of routes which traverses a sequence of ASes matched by the AS-path regular expression. A router can check this using the AS_PATH attribute in the Border Gateway Protocol [27], or the RD_PATH attribute in the Inter-Domain Routing Protocol[25].

AS-path Regular Expressions are POSIX compliant regular expressions over the alphabet of AS numbers. The regular expression constructs are as follows:

ASN where ASN is an AS number. ASN matches the AS-path that is of length 1 and contains the corresponding AS number (e.g. AS-path regular expression AS1 matches the AS-path "1").

The keyword PeerAS can be used instead of the AS number of the peer AS.

`AS-set` where `AS-set` is an AS set name. `AS-set` matches the AS-paths that is matched by one of the ASes in the `AS-set`.

`.` matches the AS-paths matched by any AS number.

`[...]` is an AS number set. It matches the AS-paths matched by the AS numbers listed between the brackets. The AS numbers in the set are separated by white space characters. If a '-' is used between two AS numbers in this set, all AS numbers between the two AS numbers are included in the set. If an `as-set` name is listed, all AS numbers in the `as-set` are included.

`[^...]` is a complemented AS number set. It matches any AS-path which is not matched by the AS numbers in the set.

`^` Matches the empty string at the beginning of an AS-path.

`$` Matches the empty string at the end of an AS-path.

We next list the regular expression operators in the decreasing order of evaluation. These operators are left associative, i.e. performed left to right.

Unary postfix operators `* + ?`

For a regular expression `A`, `A*` matches zero or more occurrences of `A`; `A+` matches one or more occurrences of `A`; `A?` matches zero or one occurrence of `A`.

Binary catenation operator

This is an implicit operator and exists between two regular expressions `A` and `B` when no other explicit operator is specified. The resulting expression `A B` matches an AS-path if `A` matches some prefix of the AS-path and `B` matches the rest of the AS-path.

Binary alternative (or) operator `|`

For a regular expressions `A` and `B`, `A | B` matches any AS-path that is matched by `A` or `B`.

Parenthesis can be used to override the default order of evaluation. White spaces can be used to increase readability.

The following are examples of AS-path filters:

```
<AS3>
<^AS1>
<AS2$>
<^AS1 AS2 AS3$>
<^AS1 .* AS2$>.
```

The first example matches any route whose AS-path contains `AS3`, the second matches routes whose AS-path starts with `AS1`, the third matches routes whose AS-path ends with `AS2`, the fourth matches routes whose AS-path is exactly "`1 2 3`", and the fifth matches routes whose AS-path starts with `AS1` and ends in `AS2` with any number of AS numbers in between.

Composite Policy Filters The following operators (in decreasing order of evaluation) can be used to form composite policy filters:

`NOT` Given a policy filter `x`, `NOT x` matches the set of routes that are not matched by `x`. That is it is the negation of policy filter `x`.

`AND` Given two policy filters `x` and `y`, `x AND y` matches the *intersection* of the routes that are matched by `x` and that are matched by `y`.

`OR` Given two policy filters `x` and `y`, `x OR y` matches the *union* of the routes that are matched by `x` and that are matched by `y`.

Note that an `OR` operator can be implicit, that is '`x y`' is equivalent to '`x OR y`'.

```
E.g.
  NOT {128.9.0.0/16, 128.8.0.0/16}
  AS226 AS227 OR AS228
  AS226 AND NOT {128.9.0.0/16}
  AS226 AND {0.0.0.0/0^0-18}
```

The first example matches any route except `128.9.0.0/16` and `128.8.0.0/16`. The second example matches the routes of `AS226`, `AS227` and `AS228`. The third example matches the routes of `AS226` except `128.9.0.0/16`. The fourth example matches the routes of `AS226` whose length are shorter than 19.

Policy filters can also use the values of other attributes for comparison. The attributes whose values can be used in policy filters are specified in the RPSL dictionary. Please refer to Section 8 for details. An example using the the BGP community attribute is shown below:

```
aut-num: AS1
export: to AS2 announce AS1 AND NOT community.contains(NO_EXPORT)
```

Filters using the routing policy attributes defined in the dictionary are evaluated before evaluating the operators `AND`, `OR` and `NOT`.

7.1.4 Example Policy Expressions

```
aut-num: AS1
import: from AS2 action pref = 1;
        from AS3 action pref = 2;
        accept AS4
```

The above example states that AS4's routes are accepted from AS2 with preference 1, and from AS3 with preference 2 (routes with lower integer preference values are preferred over routes with higher integer preference values).

```
aut-num: AS1
import: from AS2 7.7.7.2 at 7.7.7.1 action pref = 1;
        from AS2                    action pref = 2;
        accept AS4
```

The above example states that AS4's routes are accepted from AS2 on peering 7.7.7.1-7.7.7.2 with preference 1, and on any other peering with AS2 with preference 2.

7.2 export Attribute: Export Policy Specification

Similarly, an export policy expression is specified using an export attribute. The export attribute has the following syntax:

```
export: to <peering-1> [action <action-1>]
        . . .
        to <peering-N> [action <action-N>]
        announce <filter>
```

The action specification is optional. The semantics of an export attribute is as follows: the set of routes that are matched by <filter> are exported to all the peers specified in <peerings>; while exporting routes at <peering-M>, <action-M> is executed.

E.g.

```
aut-num: AS1
export: to AS2 action med = 5; community .= 70;
        announce AS4
```

In this example, AS4's routes are announced to AS2 with the med attribute's value set to 5 and community 70 added to the community list.

Example:

```
aut-num: AS1
export: to AS-FOO announce ANY
```

In this example, AS1 announces all of its routes to the ASes in the set AS-FOO.

7.3 Other Routing Protocols, Multi-Protocol Routing Protocols, and Injecting Routes Between Protocols

The syntax of the `import` and `export` attributes are indeed the following:

```
import: [protocol <protocol-1>] [into <protocol-2>]
        from <peering-1> [action <action-1>]
        . . .
        from <peering-N> [action <action-N>]
        accept <filter>
export: [protocol <protocol-1>] [into <protocol-2>]
        to <peering-1> [action <action-1>]
        . . .
        to <peering-N> [action <action-N>]
        announce <filter>
```

Where the optional protocol specifications can be used for specifying policies for other routing protocols, or for injecting routes of one protocol into another protocol, or for multi-protocol routing policies. The valid protocol names are defined in the dictionary. The `<protocol-1>` is the name of the protocol whose routes are being exchanged. The `<protocol-2>` is the name of the protocol which is receiving these routes. Both `<protocol-1>` and `<protocol-2>` default to the Internet Exterior Gateway Protocol, currently BGP.

In the following example, all interAS routes are injected into RIP.

```
aut-num: AS1
import: from AS2 accept AS2
export: protocol BGP into RIP
        to AS1 announce ANY
```

In the following example, `AS1` accepts `AS2`'s routes including any more specifics of `AS2`'s routes, but does not inject these extra more specific routes into OSPF.

```
aut-num: AS1
import: from AS2 accept AS2^+
export: protocol BGP into OSPF
        to AS1 announce AS2
```

In the following example, `AS1` injects its static routes (routes which are members of the set `AS1:RS-STATIC-ROUTES`) to the interAS routing protocol and appends `AS1` twice to their AS paths.

```
aut-num: AS1
```

```
import: protocol STATIC into BGP
        from AS1 action aspath.prepend(AS1, AS1);
        accept AS1:RS-STATIC-ROUTES
```

In the following example, AS1 imports different set of unicast routes for multicast reverse path forwarding from AS2:

```
aut-num: AS1
import: from AS2 accept AS2
import: protocol IDMR
        from AS2 accept AS2:RS-RPF-ROUTES
```

7.4 Ambiguity Resolution

It is possible that the same peering can be covered by more that one peering specification in a policy expression. For example:

```
aut-num: AS1
import: from AS2 7.7.7.2 at 7.7.7.1 action pref = 2;
        from AS2 7.7.7.2 at 7.7.7.1 action pref = 1;
        accept AS4
```

This is not an error, though definitely not desirable. To break the ambiguity, the action corresponding to the first peering specification is used. That is the routes are accepted with preference 2. We call this rule as the specification-order rule.

Consider the example:

```
aut-num: AS1
import: from AS2                     action pref = 2;
        from AS2 7.7.7.2 at 7.7.7.1 action pref = 1; dpa = 5;
        accept AS4
```

where both peering specifications cover the peering `7.7.7.1-7.7.7.2`, though the second one covers it more specifically. The specification order rule still applies, and only the action "`pref = 2`" is executed. In fact, the second peering-action pair has no use since the first peering-action pair always covers it. If the intended policy was to accept these routes with preference 1 on this particular peering and with preference 2 in all other peerings, the user should have specified:

```
aut-num: AS1
```

```
import: from AS2 7.7.7.2 at 7.7.7.1 action pref = 1; dpa = 5;
        from AS2                    action pref = 2;
        accept AS4
```

It is also possible that more than one policy expression can cover the same set of routes for the same peering. For example:

```
aut-num: AS1
import: from AS2 action pref = 2; accept AS4
import: from AS2 action pref = 1; accept AS4
```

In this case, the specification-order rule is still used. That is, AS4's routes are accepted from AS2 with preference 2. If the filters were overlapping but not exactly the same:

```
aut-num: AS1
import: from AS2 action pref = 2; accept AS4
import: from AS2 action pref = 1; accept AS4 OR AS5
```

the AS4's routes are accepted from AS2 with preference 2 and however AS5's routes are also accepted, but with preference 1.

We next give the general specification order rule for the benefit of the RPSL implementors. Consider two policy expressions:

```
aut-num: AS1
import: from peerings-1 action action-1 accept filter-1
import: from peerings-2 action action-2 accept filter-2
```

The above policy expressions are equivalent to the following three expressions where there is no overlap:

```
aut-num: AS1
import: from peerings-1 action action-1 accept filter-1
import: from peerings-3 action action-2 accept filter-2 AND NOT filter-1
import: from peerings-4 action action-2 accept filter-2
```

where peerings-3 are those that are covered by both peerings-1 and peerings-2, and peerings-4 are those that are covered by peerings-2 but not by peerings-1 ("filter-2 AND NOT filter-1" matches the routes that are matched by filter-2 but not by filter-1).

Example:

```
aut-num: AS1
import: from AS2 7.7.7.2 at 7.7.7.1
        action pref = 2;
        accept {128.9.0.0/16}
import: from AS2
        action pref = 1;
        accept {128.9.0.0/16, 75.0.0.0/8}
```

Lets consider two peerings with AS2, 7.7.7.1-7.7.7.2 and 9.9.9.1-9.9.9.2. Both policy expressions cover 7.7.7.1-7.7.7.2. On this peering, the route 128.9.0.0/16 is accepted with preference 2, and the route 75.0.0.0/8 is accepted with preference 1. The peering 9.9.9.1-9.9.9.2 is only covered by the second policy expressions. Hence, both the route 128.9.0.0/16 and the route 75.0.0.0/8 are accepted with preference 1 on peering 9.9.9.1-9.9.9.2.

7.5 `default` Attribute: Default Policy Specification

Default routing policies are specified using the `default` attribute. The `default` attribute has the following syntax:

```
default: to <peering> [action <action>] [networks <filter>]
```

The `<action>` and `<filter>` specifications are optional. The semantics are as follows: The `<peering>` specification indicates the AS (and the router if present) is being defaulted to; the `<action>` specification, if present, indicates various attributes of defaulting, for example a relative preference if multiple defaults are specified; and the `<filter>` specifications, if present, is a policy filter. A router chooses a default router from the routes in its routing table that matches this `<filter>`.

In the following example, AS1 defaults to AS2 for routing.

```
aut-num: AS1
default: to AS2
```

In the following example, router 7.7.7.1 in AS1 defaults to router 7.7.7.2 in AS2.

```
aut-num: AS1
default: to AS2 7.7.7.2 at 7.7.7.1
```

In the following example, AS1 defaults to AS2 and AS3, but prefers AS2 over AS3.

```
aut-num: AS1
default: to AS2 action pref = 1;
default: to AS3 action pref = 2;
```

In the following example, AS1 defaults to AS2 and uses 128.9.0.0/16 as the default network.

```
aut-num: AS1
default: to AS2 networks { 128.9.0.0/16 }
```

7.6 Structured Policy Specification

The import and export policies can be structured. We only reccomend structured policies to advanced RPSL users. Please feel free to skip this section.

The BNF for a structured policy specification is the following:

```
<import-factor> ::= <import-expression> |
                    from <peering-1> [action <action-1>]
                    . . .
                    from <peering-N> [action <action-N>]
                    accept <filter>;

<import-term> ::=  <import-factor> |
                   LEFT-BRACE
                   <import-factor>
                   . . .
                   <import-factor>
                   RIGHT-BRACE

<import-expression> ::= <import-term>                              |
                        <import-term> EXCEPT <import-expression> |
                        <import-term> REFINE <import-expression>
```

Please note the semicolon at the end of an <import-factor>. The syntax and semantics for an <import-factor> is already defined in Section 7.1.

An <import-term> is either a sequence of <import-factor>'s enclosed within matching braces (i.e. '{' and '}') or just a single <import-factor>. The semantics of an <import-term> is the union of <import-factor>'s using the specification order rule. Here is an example of an <import-term>:

```
{
   from AS1 accept AS1;
```

```
      from AS2 accept AS2;
   }
```

where AS1's routes are imported from AS1 and AS2's routes are imported from AS2. A nice way to view an <import-term> is as a set of import policies.

An <import-expression> is either an <import-term> or two <import-term>'s separated by keywords "except" or "refine". Without going into semantics, for example:

```
   {
      from AS1 accept AS1;
      from AS2 accept AS2;
   } except {
         from AS3 accept {128.9.0.0/16};
     }
```

is an <import-expression>. A nice way to view except and refine keywords is as policy-set operators. To provide nesting, an <import-expression> is also an <import-factor>. Hence there can be exceptions to exceptions, refinements to refinements, or even refinements to exceptions, and so on.

The semantics for the except policy-set operator is as follows: The result of an except operation is another <import-term>. The resulting policy set contains the policies of the right hand set unmodified. The policies of the left hand set are also contained but their filters are modified to exclude the filters of the right hand set. That is, the policies in the right hand set are exceptions to the policies in the left hand set. When there are multiple levels of nesting, the operations (both except and refine) are performed right to left.

We next modify the syntax of the import attribute to:

```
import: [protocol <protocol1>] [into <protocol2>]
        <import-expression>
```

If the <import-expression> in an import attribute is not structured, i.e. contains a single <import-factor>, the semicolon after it can be omitted.

Consider the following example:

```
import: from AS1 accept as-foo;
        except {
           from AS2 accept AS226;
           except {
              from AS3 accept {128.9.0.0/16};
```

```
            }
         }
```

where the route `128.9.0.0/16` is originated by `AS226`, and AS226 is a member of the as set `as-foo`. In this example, the route `128.9.0.0/16` is accepted from AS3, any other route (not `128.9.0.0/16`) originated by AS226 is accepted from AS2, and any other ASes' routes in as-foo is accepted from AS1.

We can come to the same conclusion using the algebra defined above. Consider the inner exception specification:

```
from AS2 accept AS226;
except {
   from AS3 accept {128.9.0.0/16};
}
```

is equivalent to

```
{
 from AS3 accept {128.9.0.0/16};
 from AS2 accept AS226 AND NOT {128.9.0.0/16};
}
```

Hence, the original expression is equivalent to:

```
import: from AS1 accept as-foo;
        except {
           from AS3 accept {128.9.0.0/16};
           from AS2 accept AS226 AND NOT {128.9.0.0/16};
        }
```

which is equivalent to

```
import: {
          from AS3 accept {128.9.0.0/16};
          from AS2 accept AS226 AND NOT {128.9.0.0/16};
          from AS1 accept as-foo AND NOT
               (AS226 AND NOT {128.9.0.0/16} OR {128.9.0.0/16});
        }
```

In the case of the `refine` policy-set operator, the resulting set is constructed by taking the cartasian product of the two sets as follows: for each policy l in the left hand set and for each policy r in the

right hand set, the peerings of the resulting policy are the peerings common to both r and l; the filter of the resulting policy is the intersection of l's filter and r's filter; and action of the resulting policy is l's action followed by r's action. If there are no common peerings, or if the intersection of filters is empty, a resulting policy is not generated.

Consider the following example:

```
import: { from AS-ANY action pref = 1; accept community.contains({3560,10});
          from AS-ANY action pref = 2; accept community.contains({3560,20});
        } refine {
           from AS1 accept AS1;
           from AS2 accept AS2;
           from AS3 accept AS3;
        }
```

Here, any route with community {3560,10} is assigned a preference of 1 and any route with community {3560,20} is assigned a preference of 2 regardless of whom they are imported from. However, only AS1's routes are imported from AS1, and only AS2's routes are imported from AS2, and only AS3's routes are imported form AS3, and no routes are imported from any other AS. We can reach the same conclusion using the above algebra. That is, our example is equivalent to:

```
import: {
  from AS1 action pref = 1; accept community.contains({3560,10}) AND AS1;
  from AS1 action pref = 2; accept community.contains({3560,20}) AND AS1;
  from AS2 action pref = 1; accept community.contains({3560,10}) AND AS2;
  from AS2 action pref = 2; accept community.contains({3560,20}) AND AS2;
  from AS3 action pref = 1; accept community.contains({3560,10}) AND AS3;
  from AS3 action pref = 2; accept community.contains({3560,20}) AND AS3;
}
```

Note that the common peerings between "from AS1" and "from AS-ANY" are those peerings in "from AS1". Even though we do not formally define "common peerings", it is straight forward to deduce the definition from the definitions of peerings (please see Section 7.1.1).

Consider the following example:

```
import: {
  from AS-ANY action med = 0; accept {0.0.0.0/0^0-16};
  } refine {
       from AS1 at 7.7.7.1 action pref = 1; accept AS1;
       from AS1            action pref = 2; accept AS1;
    }
```

where only routes of length 0 to 16 are accepted and med's value is set to 0 to disable med's effect for all peerings; In addition, from AS1 only AS1's routes are imported, and AS1's routes imported at 7.7.7.1 are preferred over other peerings. This is equivalent to:

```
import: {
  from AS1 at 7.7.7.1 action med=0; pref=1; accept {0.0.0.0/0^0-16} AND AS1;
  from AS1            action med=0; pref=2; accept {0.0.0.0/0^0-16} AND AS1;
}
```

The above syntax and semantics also apply equally to structured `export` policies with "from" replaced with "to" and "accept" is replaced with "announce".

8 `dictionary` Class

The dictionary class provides extensibility to RPSL. Dictionary objects define *routing policy attributes*, *types*, and *routing protocols*. Routing policy attributes, henceforth called rp-attributes, may correspond to actual protocol attributes, such as the BGP path attributes (e.g. `community`, `dpa`, and `AS-path`), or they may correspond to router features (e.g. BGP route flap damping). As new protocols, new protocol attributes, or new router features are introduced, the dictionary object is updated to include appropriate rp-attribute and protocol definitions.

An rp-attribute is an abstract class; that is their data representation is not available. Instead, they are accessed through access methods. For example, the rp-attribute for the `BGP AS-path` attribute is called `aspath`; and it has an access method called `prepend` which stuffs extra AS numbers to the `AS-path` attributes. Access methods can take arguments. Arguments are strongly typed. For example, the method `prepend` above takes AS numbers as argument.

Once an rp-attribute is defined in the dictionary, it can be used to describe policy filters and actions. Policy analysis tools are required to fetch the dictionary object and recognize newly defined rp-attributes, types, and protocols. The analysis tools may approximate policy analyses on rp-attributes that they do not understand: a filter method may always match, and an action method may always perform no-operation. Analysis tools may even download code to perform appropriate operations using mechanisms outside the scope of RPSL.

We next describe the syntax and semantics of the dictionary class. This description is not essential for understanding dictionary objects (but it is essential for creating one). Please feel free to skip to the RPSL Initial Dictionary subsection (Section 8.1).

The attributes of the dictionary class are shown in Figure 14. The `dictionary` attribute is the name of the dictionary object, obeying the RPSL naming rules. There can be many dictionary objects, however there is always one well-known dictionary object "`RPSL`". All tools use this dictionary by default.

Attribute	Value	Type
`dictionary`	⟨*object-name*⟩	mandatory, single-valued, class key
`rp-attribute`	see description in text	optional, multi valued
`typedef`	see description in text	optional, multi valued
`protocol`	see description in text	optional, multi valued
`encapsulation`	see Section 11	optional, multi valued

Figure 14: `dictionary` Class Attributes

The `rp-attribute` attribute has the following syntax:

```
rp-attribute: <name>
   <method-1>(<type-1-1>, ..., <type-1-N1> [, "..."])
   ...
   <method-M>(<type-M-1>, ..., <type-M-NM> [, "..."])
```

where `<name>` is the name of the rp-attribute; and `<method-i>` is the name of an access method for the rp-attribute, taking `Ni` arguments where the `j-th` argument is of type `<type-i-j>`. A `method` name is either an RPSL name or one of the operators defined in Figure 15. The operator methods can take only one argument.

```
operator=          operator==
operator<<=        operator<
operator>>=        operator>
operator+=         operator>=
operator-=         operator<=
operator*=
operator/=
operator.=
```

Figure 15: Operators

An rp-attribute can have many methods defined for it. Some of the methods may even have the same name, in which case their arguments are of different types. If the argument list is followed by "...", the method takes a variable number of arguments. In this case, the actual arguments after the Nth argument are of type `<type-N>`.

Arguments are strongly typed. A type of an argument can be one of the *predefined types* or one of the `dictionary defined types`. The predefined type names are listed in Figure 16. The `integer` and the `real` types can be followed by a lower and an upper bound to specify the set of valid values of the argument. The range specification is optional. We use the ANSI C language conventions

for representing `integer`, `real` and `string` values. The `enum` type is followed by a list of RPSL names which are the valid values of the type. The `boolean` type can take the values `true` or `false`. `as_number`, `ip_address`, `address_prefix` and `dns_name` types are as in Section 2. `filter` type is a policy filter as in Section 7.

```
integer[lower, upper]          as_number
real[lower, upper]             ipv4_address
enum[name, name, ...]          address_prefix
string                         dns_name
boolean                        filter
```

Figure 16: Predefined Types

The `typedef` attribute specifies a *dictionary defined type*. Its syntax is as follows:

```
typedef: <name> <type-1> ... <type-N>
```

where `<name>` is the name of the type being defined and `<type-M>` is another type name, either predefined or dictionary defined. The type defined by a typedef is either of the types 1 through N (analogous to unions in C[19]).

A dictionary defined type can also be a *list* type, specified as:

```
list [<min_elems>:<max_elems>] of <type>
```

where the list elements are of `<type>` and the list contains at least `<min_elems>` and at most `<max_elems>` elements. The size specification is optional. In this case, there is no restriction in the number of list elements. A value of a list type is represented as a sequence of elements separated by the character "," and enclosed by the characters "{" and "}".

A `protocol` attribute of the `dictionary` class defines a protocol and a set of peering options for that protocol (which are used in `inet-rtr` class in Section 10). Its syntax is as follows:

```
protocol: <name>
   MANDATORY | OPTIONAL <option-1>(<type-1-1>, ..., <type-1-N1> [, "..."])
   ...
   MANDATORY | OPTIONAL <option-M>(<type-M-1>, ..., <type-M-NM> [, "..."])
```

where `<name>` is the name of the protocol; `MANDATORY` and `OPTIONAL` are keywords; and `<option-i>` is a peering option for this protocol, taking `Ni` many arguments. The syntax and semantics of the arguments are as in the `rp-attribute`. If the keyword `MANDATORY` is used the option is mandatory

and needs to be specified for each peering of this protocol. If the keyword `OPTIONAL` is used the option can be skipped.

The `encapsulation` attribute defines a valid encapsulation name for `inet-tunnel` objects. Please refer to Section 11 for details.

8.1 Initial RPSL Dictionary and Example Policy Actions and Filters

The Figure 18 shows the initial `RPSL` dictionary. It has eight rp-attributes: `pref` to assign local preference to the routes accepted; `med` to assign a value to the `MULTI_EXIT_DISCRIMINATOR` BGP attribute; `dpa` to assign a value to the `DPA` BGP attribute; `aspath` to prepend a value to the `AS_PATH` BGP attribute; `community` to assign a value to or to check the value of the `community` BGP attribute; `flap_damp` to enable or disable routing flap damping feature of the routers; `next-hop` to assign next hop routers to static routes; and `cost` to assign a cost to static routes. The dictionary defines two types: `community_elm` and `community_list`. `community_elm` type is either a 4-byte unsigned integer, or one of the keywords `no_export` or `no_advertise` (defined in [9]), or a list of two 2-byte unsigned integers in which case the two integers are concatenated to form a 4-byte integer. (The last form is often used in the Internet to partition the community space. A provider uses its AS number as the first two bytes, and assigns a semantics of its choice to the last two bytes.) The rp-attributes `ttlscope`, `dvmrp-metric`, `boundary` are for specifying tunnel characteristics and are described in Section 11.

The initial dictionary (Figure 18) defines only options for the Border Gateway Protocol: `asno` and `flap_damp`. The mandatory `asno` option is the AS number of the peer router. The optional `flap_damp` option instructs the router to damp route flaps when importing routes from the peer router.

The initial dictionary (Figure 18) defines the following encapsulation types: IPinIP [28], IPMOBILITY [23], DVMRP [24], GRE [15], and IPv6 [10].

Policy Actions and Filters Using RP-Attributes

The syntax of a policy action or a filter using an rp-attribute `x` is as follows:

```
x.method(arguments)
x ``op'' argument
```

where `method` is a method and ``op'' is an operator method of the rp-attribute `x`. If an operator method is used in specifying a composite policy filter, it evaluates earlier than the composite policy filter operators (i.e. `AND`, `OR`, `NOT`, and implicit or operator).

The `pref` rp-attribute can be assigned a positive integer as follows:

```
dictionary:   RPSL
rp-attribute: pref   # preference, smaller values represent higher preferences
              operator=(integer[0, 65535])  # assign an integer
rp-attribute: med    # BGP multi_exit_discriminator attribute
              operator=(integer[0, 65535])  # assign an integer
              operator=(enum[igp_cost])     # assign the IGP metric
rp-attribute: dpa    # BGP destination preference attribute (dpa)
              operator=(integer[0, 65535])   # assign an integer
rp-attribute: aspath # BGP aspath attribute
              prepend(as_number, ...)       # prepend the AS numbers
                                            # from last to first order
typedef:      community_elm # needed for the community attribute
              integer[1, 4294967200],       # 4 byte community value
              enum[internet, no_export, no_advertise] # defined in RFC 1997
              list[2:2] of integer[0, 65535] # construct a 4 byte integer
                                            # by concating two 2-byte integers
typedef:      community_list # needed for the community attribute
              list of community_elm
rp-attribute: community # BGP community attribute
              operator=(community_list)     # assign a list of communities
              operator==(community_list)    # true if equals the argument
                                            # order independent comparison
              operator.=(community_elm)     # append just one community
              append(community_elm, ...)    # append the listed communities
              delete(community_elm, ...)    # delete the listed communities
              contains(community_elm, ...)  # true if one of comms is contained
rp-attribute: flap_damp # flap_damping router feature
              enable()                      # enable flap_damping
              disable()                     # disable flap_damping
rp-attribute: next-hop # next hop router in a static route
              operator=(ipv4_address)       # assign a router address
              operator=(enum[self])         # assign router's own address
rp-attribute: cost # cost of a static route
              operator=(integer[0, 65535])  # assign an integer
rp-attribute: ttlscope # IP time-to-live, useful for tunnels
              operator=(integer[0, 65535])  # assign an integer
rp-attribute: dvmrp-metric # A DVMRP metric, useful for tunnels
              operator=(integer[0, 65535])  # assign an integer
rp-attribute: boundary # for admin scoped multicast
              operator=(list of address_prefix) # assign address regions
```

Figure 17: RPSL Dictionary (cont.)

```
   pref = 10;
```

```
encapsulation: IPinIP
encapsulation: IPMOBILITY
encapsulation: DVMRP
encapsulation: GRE
encapsulation: IPv6
protocol:     BGP # Border Gateway Protocol
              MANDATORY asno(as_number)   # as number of the peer router
              OPTIONAL flap_damp()        # enable flap damping
protocol:     OSPF
protocol:     RIP
protocol:     IGRP
protocol:     IS-IS
protocol:     STATIC
protocol:     RIPv6
protocol:     DVMRP
protocol:     PIM-DM
protocol:     PIM-SM
protocol:     CBT
protocol:     MOSPF
```

Figure 18: RPSL Dictionary

The `med` rp-attribute can be assigned either a positive integer or the word "`igp_cost`" as follows:

```
med = 0;
med = igp_cost;
```

The `dpa` rp-attribute can be assigned a positive integer as follows:

```
dpa = 100;
```

The BGP `community` attribute is list-valued, that is it is a list of 4-byte integers each representing a "community". The following examples demonstrate how to add communities to this rp-attribute:

```
community .= 100;
community .= NO_EXPORT;
community .= {3561,10};
```

In the last case, a 4-byte integer is constructed where the more significant two bytes equal 3561 and the less significant two bytes equal 10. The following examples demonstrate how to delete communities from the `community` rp-attribute:

```
community.delete(100, NO_EXPORT, {3561,10});
```

Filters that use the `community` rp-attribute can be defined as demonstrated by the following examples:

```
community.contains(100, NO_EXPORT, {3561,10});
```

The community rp-attribute can be set to a list of communities as follows:

```
community = {100, NO_EXPORT, {3561,10}, 200};
community = {};
```

In this first case, the community rp-attribute contains the communities 100, `NO_EXPORT`, {3561,10}, and 200. In the latter case, the community rp-attribute is cleared. The community rp-attribute can be compared against a list of communities as follows:

```
community == {100, NO_EXPORT, {3561,10}, 200};
```

To influence the route selection, the BGP `as_path` rp-attribute can be made longer by prepending AS numbers to it as follows:

```
aspath.prepend(AS1);
aspath.prepend(AS1, AS1, AS1);
```

Flap damping can be turned on or off as follows:

```
flap_damp.enable();
flap_damp.disable();
```

The following examples are invalid:

```
med = -50;                      # -50 is not in the range
med = igp;                      # igp is not one of the enum values
med.assign(10);                 # method assign is not defined
community.append({AS3561,20}); # the first argument should be 3561
```

Figure 19 shows a more advanced example using the rp-attribute `community`. In this example, `AS3561` bases its route selection preference on the community attribute. Other ASes may indirectly affect `AS3561`'s route selection by including the appropriate communities in their route announcements.

```
aut-num: AS1
export: to AS2 action community.={3561,10};
        to AS3 action community.={3561,20};
        announce AS1

as-set: AS3561:AS-PEERS
members: AS2, AS3

aut-num: AS3561
import: from AS3561:AS-PEERS
        action pref = 10;
        accept community.contains({3561,10})
import: from AS3561:AS-PEERS
        action pref = 20;
        accept community.contains({3561,20})
import: from AS3561:AS-PEERS
        action pref = 30;
        accept ANY
```

Figure 19: Policy example using the `community` rp-attribute.

9 Advanced `route` Class

9.1 Specifying Static Routes

The attribute `inject-at` can be used to specify static routes. Its syntax is as follows:

```
inject-at: <router> [action <action>]
```

where `<router>` is an IP address of a router and `<action>` is as in the `aut-num` class. `<router>` executes the `<action>` and injects the route to the interAS routing system. `<action>` may set certain route attributes such as a next-hop router or a cost.

In the following example, the router `7.7.7.1` injects the route `128.7.0.0/16`. The next-hop routers (in this example, there are two next-hop routers) for this route are `7.7.7.2` and `7.7.7.3` and the route has a cost of 10 over `7.7.7.2` and 20 over `7.7.7.3`.

```
route:     128.7.0.0/16
origin:    AS1
inject-at: 7.7.7.1 action next-hop = 7.7.7.2; cost = 10;
inject-at: 7.7.7.1 action next-hop = 7.7.7.3; cost = 20;
```

9.2 Specifying Aggregate Routes

The attributes `aggr-by`, `inject-at`, `export-comps`, and `holes` are used for specifying aggregate routes [13].

The `aggr-by` attribute defines what component routes are used to form the aggregate. Its syntax is as follows:

```
aggr-by: [atomic] <filter>
```

A router in the origin AS forms the aggregate route if there is at least one route in its routing table that matches `<filter>`. If the keyword `ATOMIC` is specified, the aggregation is done atomically, otherwise the BGP path attributes of the matching routes are used to form the BGP path attributes of the aggregate route. For example, if atomic aggregation is done, the aggregate route would have an AS-path that starts from the aggregating AS [13]. Otherwise, the aggregate route would have an AS-path containing AS-sets formed from the AS-paths of the matching routes.

Figure 20 shows some example aggregate `route` objects. The aggregate `128.9.0.0/16` is generated if there is a route that matches the filter "`128.9.0.0/16^- AND <^AS226>`" (this filter matches more specifics of 128.9.0.0/16 that are received form `AS226`). The BGP path attributes of the matching routes are used to form the BGP path attributes of the route `128.9.0.0/16`. Similarly, the aggregate `128.8.0.0/16` is generated if there is a route that matches the filter "`128.8.0.0/16^- AND <^AS226>`". However, its path attributes are generated using the atomic aggregation rules [13]. The aggregate `128.7.0.0/16` is always and atomically generated since the policy filter "`ANY`" matches any route in the routing table.

```
route: 128.9.0.0/16
origin: AS1
aggr-by: {128.9.0.0/16^-} AND <^AS226>

route: 128.8.0.0/16
origin: AS1
aggr-by: ATOMIC {128.8.0.0/16^-} AND <^AS226>

route: 128.7.0.0/16
origin: AS1
aggr-by: ATOMIC ANY
inject-at: 7.7.7.1
inject-at: 9.9.9.1
export-comps: {128.7.9.0/24}
```

Figure 20: Aggregate `route` objects.

The `inject-at` attribute lists the routers in the originating AS that inject this route to the interAS routing system. That is, these routers are configured to perform the aggregation. If the `inject-at` attribute is missing, all routers in the originating AS perform the aggregation. The route `128.7.0.0/16` in Figure 20 is injected by routers `7.7.7.1` and `9.9.9.1` in `AS1`.

When a set of routes are aggregated, the intent is to export only the aggregate route and suppress the exporting of the component routes to the outside world. However, to satisfy certain policy and topology constraints (e.g. a multi-homed component), it is often required to export some of the components. The `export-comps` attribute equals an RPSL filter that matches the routes that need to be exported to the neighboring ASes. If this attribute is missing, no component route needs to be exported to the neighboring ASes. The `export-comps` attribute can only be specified if an `aggr-by` attribute is specified for the route object. The component `128.7.9.0/24` of route `128.7.0.0/16` in Figure 20 needs to be exported to other ASes.

The `holes` attribute lists the component address prefixes which are not reachable through the aggregate route (perhaps that part of the address space is unallocated). Figure 21 shows a `route` object whose two components, namely `128.9.0.0/16` and `128.7.0.0/16`, are not reachable via the aggregate. That is, if a data packet destined to a host in `128.9.0.0/16` is sent to `AS1`, `AS1` can not deliver it to its final destination (i.e. it is black-holed).

```
route: 128.0.0.0/12
origin: AS1
aggr-by: {128.0.0.0/12^-}
holes: 128.7.0.0/16, 128.9.0.0/16
```

Figure 21: The route `128.0.0.0/12` does not lead to destinations in `128.9.0.0/16`.

10 `inet-rtr` Class

Routers are specified using the `inet-rtr` class. The attributes of the `inet-rtr` class are shown in Figure 22. The `inet-rtr` attribute is a valid DNS name of the router described. Each `alias` attribute, if present, is a canonical DNS name for the router. The `local-as` attribute specifies the AS number of the AS which owns/operates this router.

The value of an `ifaddr` attribute has the following syntax:

```
<ipv4-address> masklen <integer>
[tunnel <inet-tunnel-object-name>]
[action <action>]
```

The IP address and the mask length are mandatory for each interface. If the interface is a tunnel, and if there is an `inet-tunnel` object describing the tunnel, the tunnel's name can also be specified.

Attribute	Value	Type
`inet-rtr`	⟨*dns-name*⟩	mandatory, single-valued, class key
`alias`	⟨*dns-name*⟩	optional, multi-valued
`local-as`	⟨*as-number*⟩	mandatory, single-valued
`ifaddr`	see description in text	mandatory, multi-valued
`peer`	see description in text	optional, multi-valued

Figure 22: `inet-rtr` Class Attributes

(An example `inet-rtr` object with tunnels is presented in Section 11.) Optionally an action can be specified to set other parameters of this interface.

Figure 23 presents an example `inet-rtr` object. The name of the router is "`amsterdam.ripe.net`". "`amsterdam1.ripe.net`" is a canonical name for the router. The router is connected to 4 networks. Its IP addresses and mask lengths in those networks are specified in the `ifaddr` attributes.

```
inet-rtr: Amsterdam.ripe.net
alias:    amsterdam1.ripe.net
localas:  AS3333
ifaddr:   192.87.45.190 masklen 24
ifaddr:   192.87.4.28   masklen 24
ifaddr:   193.0.0.222   masklen 27
ifaddr:   193.0.0.158   masklen 27
peer:     BGP 192.87.45.195 asno(AS3334), flap_damp()
```

Figure 23: `inet-rtr` Objects

Each `peer` attribute, if present, specifies a protocol peering with another router. The value of a `peer` attribute has the following syntax:

```
<protocol> <ipv4-address> <options>
```

where `<protocol>` is a protocol name, `<ipv4-address>` is the IP address of the peer router, and `<options>` is a comma separated list of peering options for `<protocol>`. Possible protocol names and attributes are defined in the dictionary (please see Section 8). In the above example, the router has a BGP peering with the router `192.87.45.195` in `AS3334` and turns the flap damping on when importing routes from this router.

11 `inet-tunnel` Class and Specifying Tunnels

Tunneling is a fundamental networking technology that is used in a variety circumstances. A common use of tunneling is to incrementally deploy a new network layer protocol. The approach is to encapsulate ("tunnel") the new protocol through the existing network layer protocol, usually IP. Examples of this approach include include the multicast backbone [3], where multicast packets are encapsulated in IP packets using protocol 4 (IP in IP), and IPv6 backbone [1], where IPv6 packets are encapsulated in IP packets using IP protocol 41 [14].

Another use of tunneling is to force congruence between the existing (IP unicast) topology and some new topology. Due the special requirements of IP multicast routing, the MBONE is also an example of this use of tunneling.

This section describes general tunneling specification in RPSL. Both point-to-point and point-to-multipoint tunnels of encapsulation types, including DVMRP, GRE, and IPv6, are supported. In addition to the encapsulation, a protocol to run inside the tunnel can also be specified.

Tunnels are specified using the `inet-tunnel` class. The attributes of the `inet-tunnel` class are shown in Figure 24. The `inet-tunnel` attribute is a valid RPSL name for the tunnel described. The `tunnel-source` attribute is the IP address of the source end point of the tunnel. The `inet-tunnel` and the `tunnel-source` attributes form the class key. That is, a point-to-point tunnel is specified using two tunnel object, one for each end point of the tunnel. The `tunnel-sink` attribute is the IP address of other end points of the tunnel. If the tunnel is a multi-point tunnel, multiple `tunnel-sink` attributes can be used to list each end point. The `tunnel-encap` attribute is an encapsulation name. Valid encapsulation names are defined in the dictionary and include IPinIP [28], IPMOBILITY [23], DVMRP [24], GRE [15], and IPv6 [10]. The `tunnel-protocol` attribute is a protocol name to run "inside" the tunnel. Valid protocol names are defined in the dictionary and include BGP, RIPv6, DVMRP. See [26] for an application that uses BGP tunneled in GRE. The `tunnel-mcast-tree` attribute is used to describe the multicast tree construction mechanism used on the tunnel. Examples include PIM-DM and PIM-SM.

Attribute	Value	Type
`inet-tunnel`	⟨*rpsl-name*⟩	mandatory, single-valued, class key
`tunnel-source`	⟨*ipv4-address*⟩	mandatory, single valued, class key
`tunnel-sink`	⟨*ipv4-address*⟩	mandatory, multi-valued, class key
`tunnel-encap`	⟨*encapsulation-name*⟩	mandatory, single-valued
`tunnel-protocol`	⟨*protocol-name*⟩	mandatory, single valued
`tunnel-mcast-tree`	⟨*protocol-name*⟩	optional, single valued
`tunnel-in`	see description in text	mandatory, multi-valued
`tunnel-out`	see description in text	mandatory, multi-valued

Figure 24: `inet-tunnel` Class Attributes

The `tunnel-in` and `tunnel-out` attributes have the following format:

```
tunnel-in:  from <ipv4-address> [action <action>] accept   <filter>
tunnel-out: to   <ipv4-address> [action <action>] announce <filter>
```

where ⟨*action*⟩ and ⟨*filter*⟩ are as in the `aut-num` class. The possible actions are defined in the dictionary and include

`ttlscope` The minimum IP time-to-live required for a packet to be forwarded to the specified endpoint (in the case of multipoint tunnels, there may be per endpoint scopes).

`boundary` A boundary attribute describes an administratively defined class of packets that will not be forwarded through the tunnel [21].

`dvmrp-metric` A DVMRP metric.

These attributes are particularly relevant to multicast routing. Attributes for other tunnels can later be defined in the dictionary. The ⟨*filter*⟩ specifications describe filters that are appropriate for the tunnel's routing protocol. In the case of DVMRP, the filter specification can be the list of network prefixes accepted or advertised.

Figure 25 has two examples of `tunnel` objects. In the first example, the router `eugene-isp.nero.net` has two tunnels: a DVMRP tunnel to `dec3800-2-fddi-0.SanFrancisco.mci.net` and a GRE tunnel to `eugene-isp.nero.net`. The DVMRP tunnel object is called `MBONE-TUNNEL-EUG.eugene-isp.nero.net` will accept any routes and forward packets to the DVMRP tunnel if the packet's time-to-live is greater than or equal to 64. In addition, `eugene-isp.nero.net` will not pass any packets that match the administrative scope boundary filter (in this case, `239.254.0.0/16`). The GRE tunnel is named `GRE-TUNNEL-EUG`.

12 Security Consideration

This document describes RPSL, a language for expressing routing policies. As such, it does not itself have (or need) a security architecture. However, any registry that implements this language should provide a mechanism for:

1. Data Integrity and Origin Authentication. Both data origin and integrity can be provided by associating cryptographically generated digital signatures with each object in a IRR. There may be a single private key that signs for all objects associated with a given `MAINTAINER` object, or there may be finer grained control. As is common, it is expected that an implementation will keep the `MAINTAINER` private key off-line and it will be used to re-sign all objects for a given `MAINTAINER`.

2. Public Key Distribution. It is expected that any IRR implemeting RPSL will use the Group Key Management Protocol (GKMP) [16]. The IETF IP Security Working Group is actively

```
inet-rtr: eugene-isp.nero.net
loacalas: AS4600
ifaddr:   166.48.14.6 masklen 30 tunnel MBONE-TUNNEL-EUG
ifaddr:   166.48.14.6 masklen 30 tunnel GRE-TUNNEL-EUG
admin-c:  DMM65
tech-c:   DMM65
notify:   nethelp@ns.uoregon.edu
mnt-by:   MAINT-AS3582
changed:  meyer@ns.uoregon.edu 961122
source:   RADB

inet-tunnel:     MBONE-TUNNEL-EUG
tunnel-source:   166.48.14.6   # eugene-isp.nero.net
tunnel-sink:     204.70.158.61 # dec3800-2-fddi-0.SanFrancisco.mci.net
tunnel-encap:    DVMRP
tunnel-protocol: DVMRP
tunnel-in:       from 204.70.158.61 accept ANY
tunnel-out:      to 204.70.158.61
                 action
                    ttlscope=64;
                    boundary={239.254.0.0/16};
                    dvmrp-metric=1;
                 announce AS-NERO-TRANSIT
...

inet-tunnel:       GRE-TUNNEL-EUG
tunnel-source:     166.48.14.6
tunnel-sink:       206.42.19.240
tunnel-protocol:   DVMRP
tunnel-mcast-tree: PIM-DM
tunnel-encap:      GRE
tunnel-in:         from 206.42.19.240 accept ANY
tunnel-out:        to 206.42.19.240
                   action
                     ttlscope=64;
                   announce ANY
...
```

Figure 25: `inet-tunnel` Objects

working on GKMP extensions to the standards-track ISAKMP key management protocol being developed in the same working group.

3. Transaction Security. When a user is querying a registry for policy objects, to eliminate snooping and to eliminate third parties injecting objects, the server and the client may optionally use authentication and encryption techniques [12].

13 Acknowledgements

We would like to thank Jessica Yu, Randy Bush, Alan Barrett, David Kessens, Bill Manning, Sue Hares, Ramesh Govindan, Kannan Varadhan, Satish Kumar, Craig Labovitz, Rusty Eddy, David J. LeRoy, David Whipple, Jon Postel, Deborah Estrin, Elliot Schwartz, Joachim Schmitz, and the participants of the IETF RPS Working Group for various comments and suggestions.

References

[1] 6BONE. See http://www-6bone.lbl.gov/6bone/.

[2] Internet Routing Registry. Procedures. http://www.ra.net/RADB.tools.docs/, http://www.ripe.net/db/doc.html.

[3] MBONE. See http://www.best.com/ prince/techinfo/misc.html.

[4] C. Alaettinoğlu, D. Meyer, and J. Schmitz. Application of Routing Policy Specification Language (RPSL) on the Internet. Internet Draft draft-ietf-rps-appl-rpsl-00.ps, March 1997. Work in progress.

[5] T. Bates. Specifying an 'Internet Router' in the Routing Registry. Technical Report RIPE-122, RIPE, RIPE NCC, Amsterdam, Netherlands, October 1994.

[6] T. Bates, E. Gerich, L. Joncheray, J-M. Jouanigot, D. Karrenberg, M. Terpstra, and J. Yu. Representation of IP Routing Policies in a Routing Registry. Technical Report ripe-181, RIPE, RIPE NCC, Amsterdam, Netherlands, October 1994.

[7] T. Bates, E. Gerich, L. Joncheray, J-M. Jouanigot, D. Karrenberg, M. Terpstra, and J. Yu. Representation of IP Routing Policies in a Routing Registry. Technical Report RFC-1786, March 1995.

[8] T. Bates, J-M. Jouanigot, D. Karrenberg, P. Lothberg, and M. Terpstra. Representation of IP Routing Policies in the RIPE Database. Technical Report ripe-81, RIPE, RIPE NCC, Amsterdam, Netherlands, February 1993.

[9] R. Chandra, P. Traina, and T. Li. BGP Communities Attribute. Request for Comment RFC-1997, Network Information Center, August 1996.

[10] A. Conta and S. Deering. Generic Packet Tunneling in IPv6. Technical Report draft-ietf-ipngwg-ipv6-tunnel-04.txt, October 1996.

[11] D. Crocker. Standard for the format of ARPA Internet text messages. Request for Comment RFC-822, Network Information Center, August 1982.

[12] D. Eastlake and C. Kaufman. Domain Name System Security Extensions. Technical Report RFC2065, January 1997.

[13] V. Fuller, T. Li, J. Yu, and K. Varadhan. *Classless Inter-Domain Routing (CIDR): an Address Assignment and Aggregation Strategy*, 1993.

[14] R. Gilligan and E. Nordmark. Transition Mechanisms for IPv6 Hosts and Routers. Technical Report RFC1933, April 1996.

[15] S. Hanks, T. Li, D. Farinacci, and P. Traina. Generic Routing Encapsulation (GRE). Technical Report RFC1701, October 1994.

[16] H. Harney. Group Key Management Protocol (GKMP). Technical Report draft-harney-gkmp-arch-01.txt, draft-harney-gkmp-spec-01.txt, August 1996. Informational RFC publication is pending.

[17] D. Karrenberg and T. Bates. Description of Inter-AS Networks in the RIPE Routing Registry. Technical Report RIPE-104, RIPE, RIPE NCC, Amsterdam, Netherlands, December 1993.

[18] D. Karrenberg and M. Terpstra. Authorisation and Notification of Changes in the RIPE Database. Technical Report ripe-120, RIPE, RIPE NCC, Amsterdam, Netherlands, October 1994.

[19] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice-Hall, 1978.

[20] A. Lord and M. Terpstra. RIPE Database Template for Networks and Persons. Technical Report ripe-119, RIPE, RIPE NCC, Amsterdam, Netherlands, October 1994.

[21] D. Meyer. Administratively Scoped IP Multicast. Technical Report draft-ietf-mboned-admin-ip-space-01.txt, December 1996.

[22] P. V. Mockapetris. Domain names - concepts and facilities. Request for Comment RFC-1034, Network Information Center, November 1987.

[23] C. Perkins. Minimal Encapsulation within IP. Technical Report RFC2004, October 1996.

[24] T. Pusateri. Distance Vector Multicast Routing Protocol. Technical Report draft-ietf-idmr-dvmrp-v3-03, September 1996.

[25] Y. Rekhter. Inter-Domain Routing Protocol (IDRP). *Journal of Internetworking Research and Experience*, 4:61–80, 1993.

[26] Y. Rekhter. Auto route injection with tunnelling, October 1996. NANOG, See http://www.academ.com/nanog/oct1996/multihome.html.

[27] Y. Rekhter and T. Li. A Border Gateway Protocol 4 (BGP-4). Request for Comment RFC-1654, Network Information Center, July 1994.

[28] W. Simpson. IP in IP Tunneling. Technical Report RFC1853, October 1995.

A Routing Registry Sites

The set of routing registries as of November 1996 are `RIPE`, `RADB`, `CANet`, `MCI` and `ANS`. You may contact one of these registries to find out the current list of registries.

B Authors' Addresses

Cengiz Alaettinoglu
USC Information Sciences Institute
4676 Admiralty Way, Suite 1001
Marina del Rey, CA 90292
email: cengiz@isi.edu

Tony Bates
Cisco Systems, Inc.
170 West Tasman Drive
San Jose, CA 95134
email: tbates@cisco.com

Elise Gerich
At Home Network
385 Ravendale Drive
Mountain View, CA 94043
email: epg@home.net

Daniel Karrenberg
RIPE Network Coordination Centre (NCC)
Kruislaan 409
NL-1098 SJ Amsterdam
Netherlands
email: dfk@ripe.net

David Meyer
University of Oregon
Eugene, OR 97403
email: meyer@antc.uoregon.edu

Marten Terpstra
c/o Bay Networks, Inc.
2 Federal St
Billerica MA 01821
email: marten@BayNetworks.com

Curtis Villamizar
ANS
email: curtis@ans.net