Network Working Group
Internet-Draft
Intended status: Informational
Expires: August 21, 2008

L. Iannone
O. Bonaventure
UC Louvain
February 18, 2008

# OpenLISP Implementation Report
# draft-iannone-openlisp-implementation-00

## Status of this Memo



## Copyright Notice

Abstract

The RRG is working on the design of an alternate Internet Architecture in order solve issues of the current architecture related to scalability, mobility, multi-homing, and inter-domain routing. Among the various proposals, LISP (Locator/ID Separation Protocol) is one of the most advanced. UC Louvain is working on an implementation of this protocol on a FreeBSD platform. The present draft describes the overall architecture of this implementation and its main data structures.

Table of Contents

## 1. Introduction

Very recent activities in the IETF and in the Routing Research Group (RRG) have focused on defining a new Internet architecture, in order to solve issues related to scalability, addressing, mobility, multi-homing, inter-domain traffic engineering and routing ([I-D.iab-raws-report], [I-D.irtf-rrg-design-goals]). It is widely recognized that the approach based on the separation of the end-systems' addressing space (the identifiers) and the routing locators' space is the way to go. This separation is meant to alleviate the routing burden of the Default Free Zone, but it implies the need of distributing and storing mappings between identifiers and locators on caches placed on routers and to perform tunneling or address translation operation.

Among the various proposals presented in various RRG's meeting, LISP (Locator/ID Separation Protocol), based on the map/encap approach [I-D.farinacci-lisp], is one of the most advanced and promising proposals. UC Louvain his currently developing an implementation, called OpenLISP of this protocol in the FreeBSD kernel (version 6.2 - [FreeBSD]). This draft describes the overall architecture of this implementation and its main data structures. The draft is structured as follows. We first describe the kernels' data structures created to store the mappings necessary to perform encapsulation and decapsulation operations. Then, we show the architectural modifications made to the FreeBSD protocol stack in order to support LISP. Finally we describe the new mapping sockets that we have introduced to access the mappings from user space.

### 1.1. Terms Definition

The present draft uses the following terms, which are originally defined in [I-D.farinacci-lisp]. The terms are reported hereafter only as a recall.

Routing Locator (RLOC): the IPv4 or IPv6 address of an egress tunnel router (ETR). It is the output of a EID-to-RLOC mapping lookup. An EID maps to one or more RLOCs. Typically, RLOCs are numbered from topologically-aggregatable blocks that are assigned to a site at each point to which it attaches to the global Internet; where the topology is defined by the connectivity of provider networks, RLOCs can be thought of as PA addresses. Multiple RLOCs can be assigned to the same ETR device or to multiple ETR devices at a site.

Endpoint ID (EID): a 32- or 128-bit value used in the source and destination address fields of the first (most inner) LISP header of a packet. The host obtains a destination EID the same way it obtains an destination address today, for example through a DNS lookup or SIP exchange. The source EID is obtained via existing mechanisms used to set a hosts "local" IP address. An EID is allocated to a host from an EID-prefix block associated with the site the host is attached to. An EID can be used by a host to refer to other hosts. LISP uses PI blocks for EIDs; such EIDs MUST NOT be used as LISP RLOCs. Note that EID blocks may be assigned in a hierarchical manner, independent of the network topology, to facilitate scaling of the mapping database. In addition, an EID block assigned to a site may have site-local structure (subnetting) for routing within the site; this structure is not visible to the global routing system.

EID-prefix: A power-of-2 block of EIDs which are allocated to a site by an address allocation authority. EID-prefixes are associated with a set of RLOC addresses which make up a "database mapping". EID-prefix allocations can be broken up into smaller blocks when an RLOC set is to be associated with the smaller EID- prefix.

## 2. Map Tables

LISP defines two different databases to store mappings between EID-prefixes and RLOCs. The "LISP Cache" stores short-lived mappings in an on-demand fashion when new flows start. The "LISP Database" stores all the local mappings, i.e., all the mappings of the EID-Prefixes behind the router. In OpenLISP we merged the two databases in a single radix tree data structure [TCPIP]. This allows to have an efficient indexing structure for all the EID-Prefixes that need to be stored in the system. EID-Prefixes that are part of the LISP Database are marked by a MAPF_LOCAL flag, indicating that they are EID-Prefixes for which the mapping is owned locally. Thus, from a logical point of view the two "databases" are still separated. Actually there are two radix structures in the system, one for IPv4 EID-Prefixes and another for IPv6 EID-Prefixes. In both map tables, each entry has the format depicted in Figure 1.

```
struct mapentry {
   struct radix_node map_nodes[2];  /* tree glue, and other values */

   struct sockaddr_storage *EID;    /* EID value */

   struct locator_chain * rlocs;    /* Set of locators */
   int    rlocs_cnt;                /* Number of rlocs */

   u_long map_flags;                /* up/down?, local */

};
```

The mapentry structure

Figure 1

Besides the fields necessary to build the radix tree itself, the entries contain a pointer to a socket address structure that holds the EID-Prefix to which the entry is related. Furthermore, there is a pointer to a simple list containing all the RLOCs associated to the EID-Prefix. Each element of the list is a socket address structure containing the locator and an rloc_mtx structure. The latter, depicted in Figure 2, contains the priority and weight parameters, whose meaning and use are defined in the original LISP specification.

```
struct rloc_mtx {  /* Metrics associated to the RLOC
                    */

        u_int8_t priority;   /* Each RLOC has a priority.
                              * A value of 255 means that
                              * RLOC MUST not be used.
                              */
        u_int8_t weight;     /* Each locator has a weight.
                              * Used for load balancing
                              * purposes when two or more
                              * locators have the same
                              * priority.
                              */
        u_int16_t flags;     /* Local flags (future use).
                              */
};
```

RLOCs metric data structure.

Figure 2

The number of RLOCs present in the mapping is stored in the rlocs_cnt field, while the map_flags contains the flags that concern the mapping as a whole (e.g., MAPF_LOCAL). The list of RLOCs is always maintained ordered by increasing priority.

The use of a chained list, to store the RLOCs, allows mixing IPv4 and IPv6 RLOCs. This in turn allows to use IPv6 tunneling for IPv4 packets and vice versa. Even more, in this way it is possible, for the same EID, to perform both IPv6 and IPv4 tunneling depending on the RLOC eventually chosen for the encapsulation. This avoids the constraint of having the tunnels toward the same EID either all IPv4 or all IPv6.

## 3. Protocol Stack Modifications

Compared to the original protocol stack implementation of the FreeBSD OS ([TCPIP], [FreeBSD]) four main modules have been added, namely lisp_input(), lisp6_input(), lisp_output(), and lisp6_output(). As should be clear from the names, the first two modules manage incoming IPv4 and IPv6 LISP packets, while the last two modules are responsible for outgoing IPv4 and IPv6 LISP packets. To describe the global architecture, we use the same module representation as in [TCPIP] and show how packets are processed inside the protocol stack.

### 3.1. Incoming Packets

The lisp_input() and lisp6_input() modules are positioned right above respectively the ip_input() and ip6_input() modules, from which they are called, as depicted in Figure 3.

Let's for simplicity assume that an IPv4 LISP packet is received by the system. The packet will be first treated by the ip_input() module. The ip_input() module has been patched in order to recognize LISP packets. The patch consists simply to divert towards lisp_input(), all incoming UDP packets destined to the local machine and having destination port number set to the LISP reserved values 4341 (for encapsulated data packets) or 4342 (for signaling packets). If the UDP packet has neither such a port number it is delivered as usual to the transport layer (i.e., udp_input()). Once the packet reaches the lisp_input(), if the port number is 4342, it is a signaling packet (e.g., Map-Request or Map-reply) and the corresponding action, as defined by LISP, is performed. The complete list of signaling packets and corresponding actions can be found in [I-D.farinacci-lisp]. In the case of an encapsulated data packet (port number 4341), the module strips the UDP header, then it treats the reachability bits and the nonce of the LISP specific header. After having performed with these operations, the LISP header is also stripped. At this point the address family of the IP header of the remaining packet is checked in order to decide to which module to deliver the packet. In practice this means to re-inject the packet in the IP protocol stack, by putting it in the input buffer either of the ip_input() or the ip6_input() module.

Protocol Stack Modifications for incoming packets.

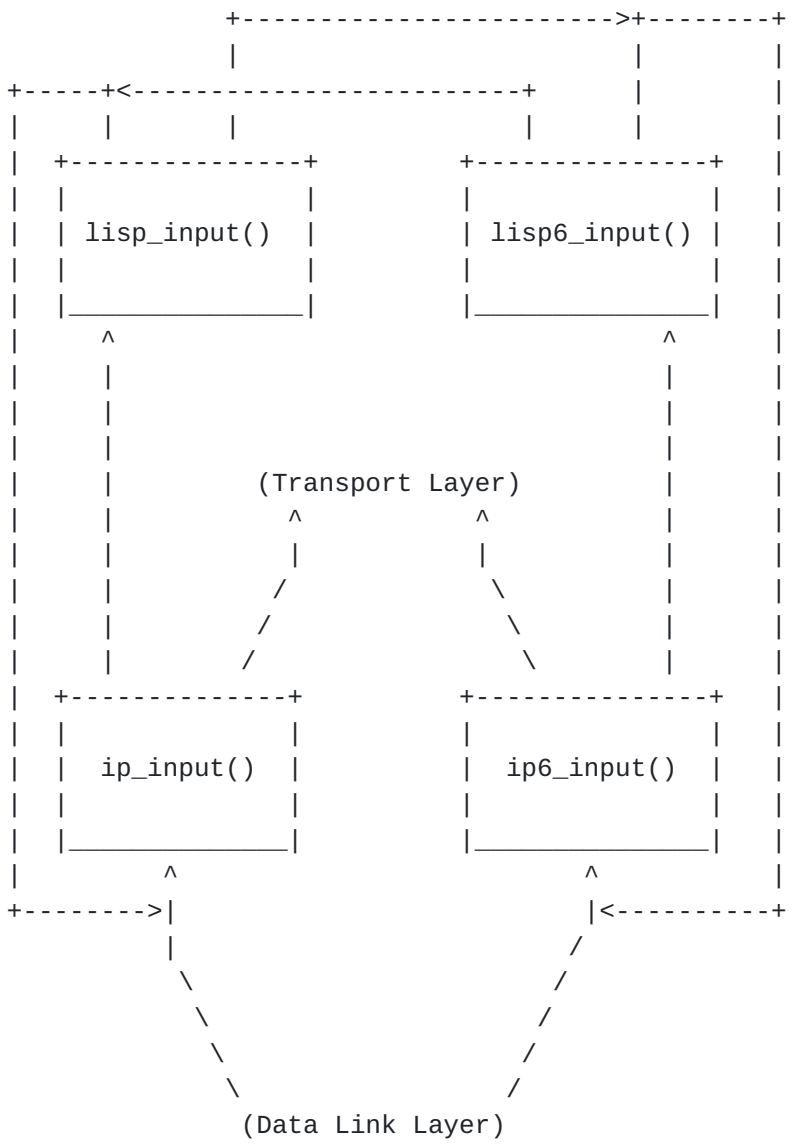


Figure 3

In the case of an IPv6 LISP packet the overall process is the same. The packet is first received by ip6_input(), where if the packet is a locally destined UDP packet with destination port number equal to the LISP reserved 4341 or 4342 values it is delivered to lisp6_input(). The latter module performs the same operations as lisp_input(), with the only difference that it is specialized in treating IPv6 headers. If the packet is a data packet, depending on the address family of the inner header, once decapsulated it is re-injected either in the input buffer of the ip_input() module or the input buffer of ip6_input() module.

Once the packet is re-injected in the protocol stack, in both IPv4 and IPv6 cases, the packet follows the normal process. This means that if the decapsulated packet is not destined to the local host it will be first delivered to the forwarding module (ip_forward() or ip6_forward()) that will in turn deliver it to the output module (ip_output() or ip6_output()) in order to send it down to the data link layer and transmit it toward its final destination. These last actions are driven by the content of the routing table of the system.

## 3.2. Outgoing Packets

The lisp_output() and lisp6_output() modules are positioned right above respectively the ip_output() and ip6_output() modules, from which they are called, as depicted in Figure 4.

Let's for simplicity assume that an IPv4 is received by the ip_output() module, coming either from the ip_forward() module or the transport layer (i.e., either tcp_output() or udp_output()). Note that we refer to a normal IPv4 packet, not a LISP encapsulated packet. The ip_output() module has been patched in order to recognize if the packet needs to be encapsulated with a LISP header. The patch consists in checking if there is a valid mapping in the LISP database. This means to perform a search in the map table using the source address (source EID) of the packet. If the lookup returns an entry with the MAPF_LOCAL flag set (recall Section 2) then the packet is diverted toward the lisp_output() module. The lisp_output(), will first prepend to the packet the LISP header (i.e. reach bits and nonce). Then a second lookup using the destination address (destination EID) of the original packet is performed on the map table in order retrieve a valid mapping. If a mapping is found, the first RLOC of the list is used, along with the mapping found from the previous lookup on the source EID, to build the IP+UDP header to be prepended to the packet. If no mapping is found, the LISP 1 variant encapsulation is used, i.e., the original destination EID is used also in the outer header. Subsequently the packet is sent again to the IP layer in order to ship it to the data-link layer. This does not mean that the packet is delivered to ip_output(). Indeed, the mapping for the destination address can have an IPv6 RLOC as a first element of the list of locators, meaning that the prepended header is IPv6+UDP and that the packet is delivered to the ip6_output() module.

Protocol Stack Modifications for outgoing packets.

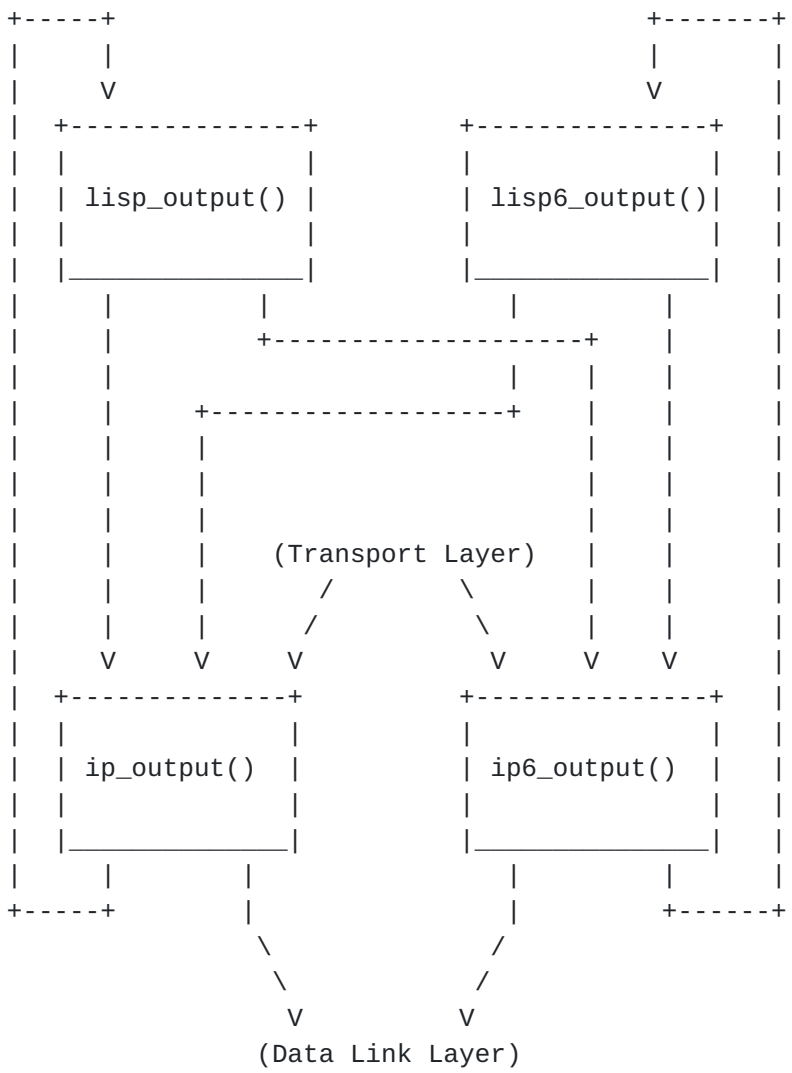


Figure 4

In the case of an outgoing IPv6 packet the overall process is the same. The packet, if a mapping exists for the source EID, is first diverted toward lisp6_output(), which prepends the correct headers to the packet and, depending of the RLOC used, delivers the packet either to the ip_output() module or the ip6_output() module.

Once the packet is re-injected in the protocol stack, in both IPv4 and IPv6 cases, the packet follows the normal process. This means that the encapsulated packet will be delivered to the data-link layer.

## 4. Mapping Sockets

In line with the UNIX philosophy and to give the possibility for future mapping distribution systems running in the user space to access the kernel's map tables a new type of socket, namely the "mapping sockets", has been defined.

Mapping sockets are based on raw sockets in the new AF_MAP domain and are very similar to the well known routing sockets ([TCPIP], [NetProg].) A mapping socket is easily created in the following way:

```
#include <net/maptables.h>

int s = socket(PF_MAP, SOCK_RAW, 0);
```

Note that <net/maptables.h> is the header file containing all the useful data structures and definitions.

Once a process has created a mapping socket, it can perform the following operations by sending messages across it:

- MAPM_ADD: used to add a mapping. The process writes the new mapping to the kernel and reads the result of the operation on the same socket.

- MAPM_DELETE: used to delete a mapping. It works in the same way as MAPM_ADD.

- MAPM_GET: used to retrieve a mapping. The process writes on the socket the request of a mapping for a specific EID and reads on the same socket the result of the query.

The messages sent across mapping socket for the above operations all use the same data structure, namely map_msghdr{}, depicted in Figure 6.

The field map_type can be set only to the type listed above. The fields map_msglen, map_version, map_pid, map_seq, and map_errno have the same meaning and are used in the same way as for the rt_msghdr{} structure for routing sockets. Details about these fields and their use can be found in [TCPIP]. The map_flags field is used to set some general flags that concern the whole mapping entry or the message, as described in Table 1.

Mapping Message Header.

```
struct map_msghdr {             /* From maptables.h
                                 */
        u_short map_msglen;     /* to skip over non-understood
                                 * messages
                                 */
        u_char  map_version;    /* future binary compatibility
                                 */
        u_char  map_type;       /* message type */
        int     map_flags;      /* flags, incl. kern & message,
                                 *  e.g. DONE
                                 */
        int     map_addrs;      /* bitmask identifying sockaddrs
                                 *  in msg
                                 */
        int     map_rloc_count; /* Number of rlocs appended to
                                   the msg */
        pid_t   map_pid;        /* identify sender
                                 */
        int     map_seq;        /* for sender to identify action
                                 */
        int     map_errno;      /* why failed
                                 */
};
```

Figure 6

| Constant | Value | Description |
|---|---|---|
| MAPF_UP | 0x1 | Mapping usable. |
| MAPF_LOCAL | 0x2 | Mapping is local. This means that it should be considered as part of the LISP Database. |
| MAPF_STATIC | 0x4 | Mapping manually added. |
| MAPF_DONE | 0x8 | Message confirmed. |

Table 1: General mapping flags

As can be noted, there is a flag (MAPF_LOCAL) that indicates whether the mapping is part of the LISP cache or the LISP database as defined in [I-D.farinacci-lisp]. From a logical perspective these are different data structures. However, as explained in Section 2, they

are merged in the radix data structure in order to have an efficient lookup mechanism for all possible EIDs.

The map_addrs field is a bitmask identifying the nature and number of data structures present in the message right after the header. The possible values and related descriptions can be found in Table 2.

| Constant | Value | Description |
|---|---|---|
| MAPA_EID | 0x1 | EID socket address present. |
| MAPA_EIDMASK | 0x2 | EID netmask socket address present. |
| MAPA_RLOC | 0x4 | At least one RLOC is present. The exact number of RLOCs can be found in the map_rloc_count field. |

Table 2: Data structure bitmask

The map_addrs field does not contain exactly all the data structures, in particular, for RLOCs, a bit just states if at least one RLOC is present. The exact number of RLOCs present is contained in the map_rloc_count field. While EID and its mask, if present, are simple socket address structures, an RLOC is composed of a socket address structure followed by an rloc_mtx structure containing the metrics of that specific RLOC. The rloc_mtx data structure has been described in Section 2, and is depicted in Figure 2 with a description of each metric.

### 4.1. An example of mapping sockets usage

Hereafter is described an example using mapping sockets. Along with the code in the kernel, a small utility called "map" has been written. This utility has similar functionalities to the "route" utility present in UNIX systems. It allows to manually manage map tables. Assuming we want to retrieve the mapping for the EID 10.0.0.1, we can type:

```
freebsd% map get -inet 10.0.0.1
```

The map tools first builds a buffer containing a map_msghdr{} structure, followed by a socket address structure containing the EID for the kernel to look up, as depicted in Figure 8. The map_type is set to MAPM_GET and the map_addrs is set to MAPA_EID. The entire buffer is written to a mapping socket previously open.

Data sent to the kernel across mapping socket for MAP_GET command.

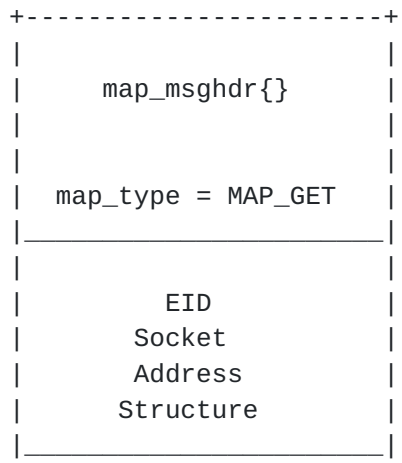


Figure 8

Afterwards, map reads from the socket the reply of the kernel. Assuming that the kernel has a mapping for 10.0.0.0/16 associated to two locators, the kernel will reply with a message which has the format depicted in Figure 10.

The first part of the message is a map_msghdr{} structure, with the map_type unchanged, the map_addrs set to 0x07, which is equivalent to MAPA_EID, MAPA_EIDMASK, and MAPA_RLOC all set, and finally the map_rloc_count set to 2. Right after the map_msghdr{} there is a first socket address structure containing the EID prefix, which is 10.0.0.0 in this example. The second socket address structure contains the netmask, 255.255.0.0 in this case. The third socket address structure contains the first RLOC. RLOCs are returned ordered by increasing priority. After the first RLOC there is an rloc_mtx structure containing the metrics associated to the first RLOC. The message ends with the socket address structure for the second RLOC and the rloc_mtx structure for its metrics.

When using the map utility a possible output for the get request for EID 10.0.0.1 can be:

```
freebsd% map get -inet 10.0.0.1
   Mapping for EID: 10.0.0.1
     EID: 10.0.0.0
     EID mask: 255.255.0.0
     RLOC Addr: inet6 2001::1     P 255     W 100
     RLOC Addr: inet  10.1.0.0    P 255     W 100
```

Data sent from the kernel across mapping socket for MAP_GET command.

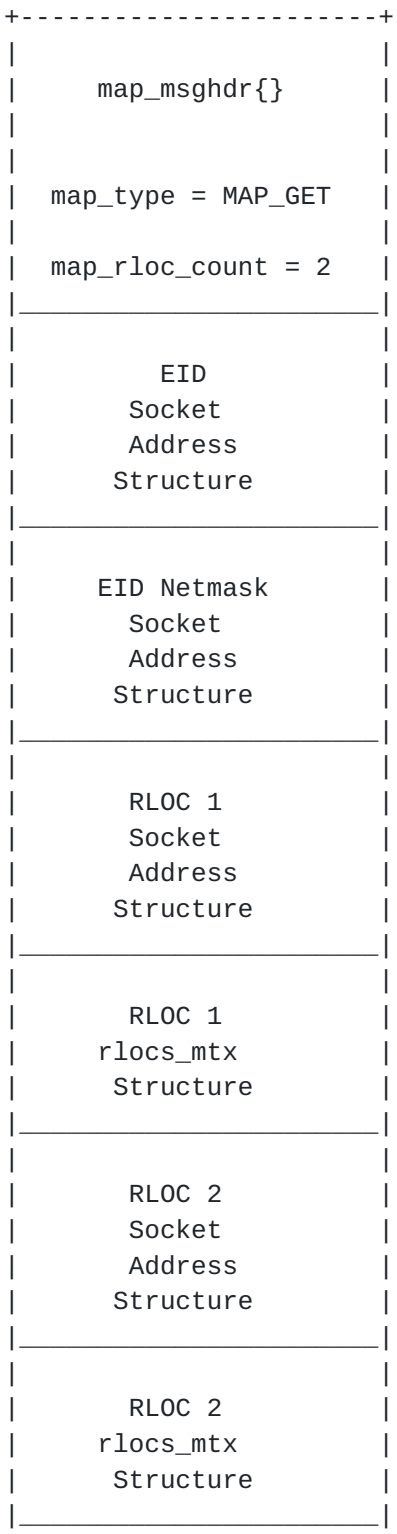


Figure 10

## 5. Conclusion

The present memo describes the overall architecture of OpenLISP, an implementation of the LISP proposal in the FreeBSD OS. OpenLISP provides support for encap/decap operations and EID-to-RLOC mappings storage in the kernel space. It can work as both a router and end-host, thus providing a wide range of test scenarios. The code will be publicly released as soon as the main debugging phase has ended and the code shows very stable behavior. However, people interested in this software can already contact the authors. We think that the mapping sockets introduced by OpenLISP is a great tool for easy development of Mapping Distribution Protocols in the user space. People working in this area can contact authors. We believe that a complete working system composed by OpenLISP and a mapping distribution protocol would provide very helpful insights, leading to important improvements for both OpenLISP and the mapping distribution protocol.

## 6. Acknowledgements

The work described in the present memo has been partially supported by the European Commission within the IST AGAVE Project.

## 7. IANA Considerations

This memo includes no request to IANA.

## 8. Security Considerations

The present memo does not introduce any new security issue that is not already mentionned in [I-D.farinacci-lisp] and [I-D.bagnulo-lisp-threat].

## 9. Informative References

[FreeBSD] The FreeBSD Project, "FreeBSD, the power to serve", <http://www.freebsd.org>.

[I-D.bagnulo-lisp-threat] Bagnulo, M., "Preliminary LISP Threat Analysis", draft-bagnulo-lisp-threat-01 (work in progress), July 2007.

[I-D.farinacci-lisp] Farinacci, D., "Locator/ID Separation Protocol (LISP)", draft-farinacci-lisp-05 (work in progress), November 2007.

[I-D.iab-raws-report] Meyer, D., "Report from the IAB Workshop on Routing and Addressing", draft-iab-raws-report-02 (work in progress), April 2007.

[I-D.irtf-rrg-design-goals] Li, T., "Design Goals for Scalable Internet Routing", draft-irtf-rrg-design-goals-01 (work in progress), July 2007.

[NetProg] Stevens, W., Fenner, B., and A. Rudoff, "UNIX Network Programming, The Sockets Networking API.", Addison-Wesley Professional Computing Series Volume 1 - Third Edition, 2004.

[TCPIP] Wright, G. and W. Stevens, "TCP/IP Illustrated Volume 2, The Implementation.", Addison-Wesley Professional Computing Series, 1995.

## Authors' Addresses

Luigi Iannone
UC Louvain
Place St. Barbe 2
Louvain la Neuve, B-1348
Belgium

Phone: +32 10 47 87 18
Email: luigi.iannone@uclouvain.be
URI: http://inl.info.ucl.ac.be

Olivier Bonaventure
UC Louvain
Place St. Barbe 2
Louvain la Neuve, B-1348
Belgium

Email: Olivier.Bonaventure@uclouvain.be
URI: http://inl.info.ucl.ac.be

## Full Copyright Statement



## Intellectual Property



## Acknowledgment

Funding for the RFC Editor function is provided by the IETF Administrative Support Activity (IASA).