Implementing the Locator/ID Separation Protocol: Design and Experience

Luigi Iannone* and Damien Saucez† and Olivier Bonaventure†

*Deutsche Telekom Laboratories AG, Technische Universität Berlin, Berlin, Germany
luigi@net.t-labs.tu-berlin.de

†ICTEAM – Université catholique de Louvain (UCL), Louvain-la-Neuve, Belgium
{first.last}@uclouvain.be

Abstract

During the last few years, the network research community and the industry have been working on the design of an alternate Internet Routing Architecture aiming at solving the issues arising in the current architecture. It is widely accepted that applying a Locator/ID Separation paradigm would result in a more scalable and flexible architecture. As the name suggests, in Locator/ID Separation the identification (ID) and the localization (locator) of end-points is separated, while the link between the ID and the Locator(s) is ensured by what is called the mapping system.

In this paper, we present OpenLISP, an open source implementation of LISP (Locator/ID Separation Protocol). LISP is a Locator/Identifier separation solution based on the map-and-encap approach, which has the merit of being incrementally deployable, hence, falling in the category of *dirty-slate* approaches. OpenLISP is not a merely implementation of the LISP specifications; it also defines the *mapping sockets*, a socket-based abstraction making the data plane and the control plane implementations independent. The evaluation provided in this paper shows the limited impact on the protocol stack performance compared to traditional non-encapsulated traffic.

Key words: Routing, Architecture, FreeBSD, LISP, Future Internet.

1. Introduction

The last few years have witnessed a growing concern on some scalability issues the current Internet Routing Architecture is suffering [1]. Even if the most representative issue is related to the growth of the BGP routing table [2], there are also concerns on addressing, mobility, multi-homing, and inter-domain traffic engineering. This does not mean that the Internet is approaching a hard scalability limit that, once reached, will cause it to collapse. Rather, the key point is that the whole Internet is evolving toward a complex system where the Operational Expenses ("OpEx") cannot be sustained anymore. In other words the Internet is becoming more and more expensive to operate.

As a consequence, the research community and the industry have put a lot of effort on studying what the Internet should be in order to face the new challenges, leading to two approaches that can be followed: *revolutionary* or *evolutionary*. In the revolutionary approach, the Internet is re-thought from scratch and no backward compatibility is required. The opposite approach, the evolutionary approach promotes a smooth transition from the current Internet to the Future Internet. The evolutionary approach is often described as the *dirty slate* approach (in opposition to the *clean slate* revolutionary approach), because the Future Internet will use most of the current technologies. However, even if the two approaches are different, the research community largely agrees on the fact that introducing a separation between the end-systems' addressing space (the identifiers – IDs) and the routing locators' space will solve (or at least alleviate) a large part of the issues the Internet is facing [3, 4, 5]. An implication of the Locator/ID Separation paradigm is the need for distributing and storing mappings between the IDs and the locators, including a mechanism to go from one namespace to the other.

Several evolutionary solutions have already been proposed, but most of them rely on changes at the end-hosts, introducing important modifications in the protocol stack [6, 7, 8, 9, 10]. On the contrary, the Locator/ID Separation Protocol (LISP), proposed by Farinacci et al. [11], has been designed having in mind incremental deployability and the lowest possible level of disruption. Furthermore, it also limits the number of systems in the Internet that will need to be upgraded, since it is meant to be deployed only on border routers of edge networks.

In this paper, we present OpenLISP, our LISP open source implementation. The purpose of such a work is to provide an open and flexible platform for experimentation for both the Data Plane (packets' encapsulation/decapsulation) and the Control Plane (mapping system). The result of this effort is presented in this paper. To the best of our knowledge, OpenLISP is the only publicly available open source implementation of the Locator/ID Separation Protocol.

OpenLISP is more than a LISP implementation. Indeed, while the LISP [11] draft provides the specifications of the LISP Data and Control planes, OpenLISP proposes a technical solution to make the Data Plane and the Control plane implementations independent. This in turn allows the Data Plane to interact with Control Plane protocols that were not initially designed for LISP. To this end, OpenLISP relies on a new socket based solution: the *mapping sockets*. The mapping socket is an API that allows not only control of the Data Plane, but it also offers a Data Plane monitoring solution.

By presenting the OpenLISP architecture and its evaluation we aim at answering the question: "*What would a protocol stack look like and how would it perform if LISP (or similar Loc/ID approaches) would be a fundamental piece of the Internet Routing and Addressing Architecture?*".

The remaining of the paper is organized as follows. In Section 2 we overview the ongoing work related to Locator/ID Separation. In Section 3 we briefly overview the LISP proposal. In Section 4 we describe the architecture of OpenLISP, some of our design choices, and highlight the different components of the Control and Data Plane. In Section 5 we describe the mapping socket API that allows the Control and the Data Plane to communicate in an open and flexible manner. Section 6 presents an evaluation of the performance of our implementation. Finally, Section 7 concludes the paper.

2. Related Work

The idea of improving the Internet routing and addressing architecture with some form of separation between the identity of end-systems and their location in the Internet topology, started back in the mid-90s [12, 13, 6, 14]. More recently, other protocols, based on this separation, have been proposed, to solve other specific issues. Main examples are the Host Identity Protocol (HIP [9]) and Shim6 [10]. The purpose of HIP is to introduce a cryptographic identifier namespace between the IP and the transport layers, while Shim6 targeted the supporting of efficient multi-homing by end-hosts.

After the Routing Research Group (RRG) had been rechartered, several proposals based on Locator/ID Separation have been presented. All of these proposals have the same goal: a new, more scalable, Internet Routing and Addressing Architecture. The differences among them can be found in the fact that some proposals use tunneling (*e.g.*, LISP [11], APT [15], TRRP [16], and IVIP [17]) while others use address rewriting (*e.g.*, GSE [6], Six/One [7], and ILNP [8]). In the tunneling approach a new header containing the locators of the source and destination IDs of the original packet is added. In address rewriting the source and destination IDs of the original packet are substituted with the corresponding locators. In both cases the result is a packet having locators in the outer header, hence that can be forwarded on the routing infrastructure.

Despite the relatively large number of proposals, the only two ongoing implementation activities (other then OpenLISP) concern Six/One [7] (by Ericsson) and LISP [11] (by Cisco). None of these activities is open source.

In particular, Cisco, in collaboration with several other companies and research institutes, has already deployed its implementation on a testbed ([18, 19]) scattered worldwide (*e.g.*, in USA, Japan, Germany, Belgium,...). The testbed is composed of several dozen nodes and supports both IPv4 and IPv6, using LISP+ALT [20] as Mapping Distribution Protocol (*cf.* Section 3.3). Further, and more interestingly, the testbed is inter-operable with the legacy Internet, thanks to the deployment of middle-boxes acting as PTRs (Proxy Tunnel Routers [21]).

3. LISP in a Nutshell

In the present section, we first give a simple example of how end-to-end packet forwarding is performed in the context of LISP. Such an example allows clarifying how the Locator/ID Separation paradigm works, but also, and more importantly, provides an overview of the basic mechanisms of the protocol. Then, we give some details about the LISP's Data Plane and Control Plane.

3.1. End-to-end packet delivery in LISP

LISP is based on a simple IP-over-UDP tunneling approach, implemented typically on border routers whose upstream IP address is used as Routing LOCator (RLOC) for the end-systems of the local domain.[1] End-systems still communicate, sending and receiving packets, using legacy IP addresses, which in the LISP terminology are called Endpoint IDentifiers (EIDs). While EIDs and RLOCs are both IP addresses, on the one hand EIDs have only

[1]Actually the protocol inserts a LISP-specific header between the outer UDP header and the inner (original) IP header. For the sake of simplicity, we will omit dealing with such a header in this paper; however, this just means avoiding describing some further checks on the encapsulation/decapsulation operations and does not modify the basic LISP mechanism. For the interested reader, details on the LISP-specific header, its content, and how to deal with it can be found in the original specifications [11].

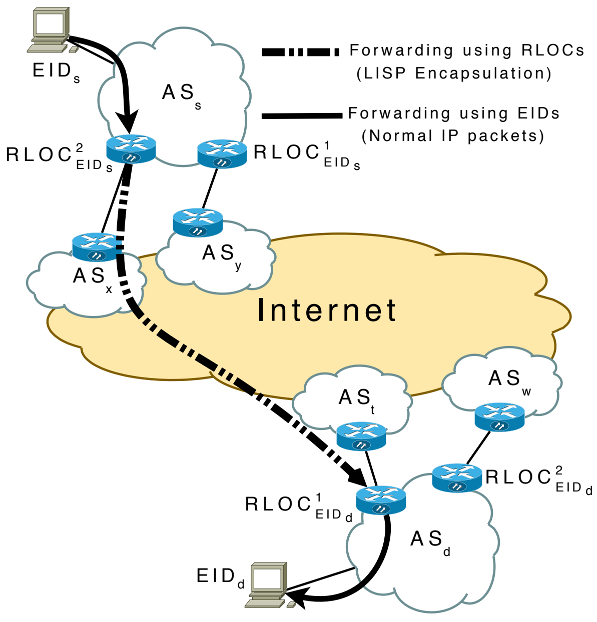

Figure 1: Overview of packet forwarding using LISP.

a local scope and are not routable in the Default Free Zone (DFZ). On the other hand, RLOCs are only used for routing and are not used as endpoint identifiers for host-to-host communications in LISP-enabled domains. Since a domain can be multi-homed, *i.e.*, having several border routers, EIDs can be actually associated with a set of RLOCs. An example of such a scenario is depicted in Fig. 1, where AS$_s$ and AS$_d$ are both multi-homed.

LISP tunnels packets in the core Internet from one of the RLOCs of the source EID to one of the RLOCs of the destination EID. In particular, the Ingress Tunnel Router (ITR) prepends a new LISP header to each packet, while the Egress Tunnel Router (ETR) strips this header before delivering the packet to its final destination. This tunneling approach allows avoiding announcing the EIDs in the core Internet. Only RLOCs are announced in order to correctly deliver packets. This last point allows reducing the size of BGP's routing tables, which is one of the targets of this protocol.[2]

Let us take a closer look at an end-to-end packet delivery, taking the scenario in Fig. 1 as a reference. Assuming that EID$_s$ wants to open a

[2]LISP does not only allow reducing the size of BGP's routing tables, but it also allows address independence, multi-homing, and light weight traffic engineering ([3, 22, 23]).

connection to EID$_d$, the first step is to issue a first IP packet using its ID (EID$_s$) as source address and the destination ID (EID$_d$) as destination address. Note that the destination ID can be obtained by EID$_s$ through a normal DNS query.

Then the packet is routed inside AS$_s$ using legacy IP routing protocols (*e.g.*, OSPF) in order to be delivered to one of EID$_s$'s locators. Assuming the packet reaches RLOC$^2{}_{EID_s}$, this router will act as ITR encapsulating the original packet in a LISP packet. We will explain later how the ITR knows the RLOCs to use for encapsulation; for now, let us assume that the best way to reach EID$_d$ is through RLOC$^1{}_{EID_d}$. Thus, the new header prepended to the original packet contains RLOC$^2{}_{EID_s}$ as source address and RLOC$^1{}_{EID_d}$ as destination address. The packet is then routed in the Internet Default Free Zone (DFZ).

When the destination router acting as an ETR with address RLOC$^1{}_{EID_d}$ receives the packet, it strips the outer LISP header and injects the inner packet in the local domain AS$_d$, where it will be forwarded using legacy IP routing protocols until it finally reaches its destination EID$_d$.

3.2. LISP Data Plane

The operations described in the previous section mainly involve the LISP Data Plane, where packets get encapsulated and decapsulated. In order to perform such operations, the LISP Data Plane needs to know when to encapsulate a packet and what to put exactly in the header, as well as when to decapsulate a packet. To perform these actions, two data structures are needed: the LISP Database and the LISP Cache.

The LISP Database is present on each xTR and consists of all EID-Prefix to RLOC mappings that are "owned locally".[3] An xTR owns a mapping if one (or more) of its upstream interfaces (toward the provider), with a globally routable IP address, are in the set of RLOCs associated with one (or more) EID-Prefixes which are used as addressing space downstream (*i.e.*, inside the local network). For instance, in the example of Fig. 1, the database present in the two xTRs of AS$_s$ contains the entry:

$$\text{EID}_s\text{-Prefix: RLOC}^1{}_{EID_s}, \text{RLOC}^2{}_{EID_s}.$$

The LISP Database is used for outgoing packets to select the source RLOC to use in the outer header (RLOC$^2{}_{EID_s}$ in the example of Fig. 1). It is

[3]By xTR we indicate a system that can be an ITR, an ETR, or both. Like IP addresses, a power of 2 block of contiguous EIDs can be aggregated in a prefix that in this case is called EID-Prefix.

also used for incoming packets to determine if they need to be decapsulated (in our example the packet is decapsulated by $\text{RLOC}^1{}_{EID_d}$ because it is the destination RLOC).[4]

The LISP Cache is a data structure containing mappings for EID-Prefixes that are not owned locally. The purpose of the cache is to provide the information necessary to select the destination RLOC when encapsulating a packet. For instance, in the example of Fig. 1, the ITR of AS_s encapsulating the packet, needs the following mapping present in the cache:

$$\text{EID}_d\text{-Prefix: } \text{RLOC}^1{}_{EID_d},\ \text{RLOC}^2{}_{EID_d}.$$

The actual selection of the RLOC to use in the outer header is done based on the priority and weight that LISP associates to each RLOC. The exact selection process can be found in the original LISP proposal [11]. The mappings contained in the LISP Cache are short-lived and subject to timeout. When a mapping is not used for a certain period, the entry is freed. This means also that the LISP Cache is populated in an on-demand fashion. In particular, the first packet of a flow will trigger a cache miss, this in turn will cause the Data Plane asking the Control Plane to retrieve a mapping for the specific destination EID that triggered the cache miss. This opens the question of what the Data Plane should do with the packet that generated the cache miss, while it is waiting for the mapping. There are three possible options: (i) silently drop the packet; (ii) buffer the packet until the Control Plane provides the needed mapping; (iii) hand over the packet to the Control Plane that will forward it piggybacked on the mapping request. The LISP specifications leave this question open, not enforcing any specific solution.

3.3. LISP Control Plane

As explained in the previous section, the first packet of a new flow may generate a miss in the LISP Cache, depending on whether or not the destination EID is part of a larger EID-Prefix whose mapping is already present in the LISP Cache. Thus the purpose of the LISP Control Plane is to provide those missing mappings on-demand to the Data Plane. Such an objective is achieved by running what is usually called a Mapping Distribution Protocol, which provides a lookup infrastructure for retrieving mappings.

Several Mapping Distribution Protocols have been already proposed for both LISP and non-LISP solutions (*e.g.,* NERD [24], CONS [26], EMACS [27],

[4]Actually, in LISP, besides having to have as destination address the RLOC of the ETR, the packet needs also to have the destination UDP port number set to a specific reserved value. See [11] for further details.

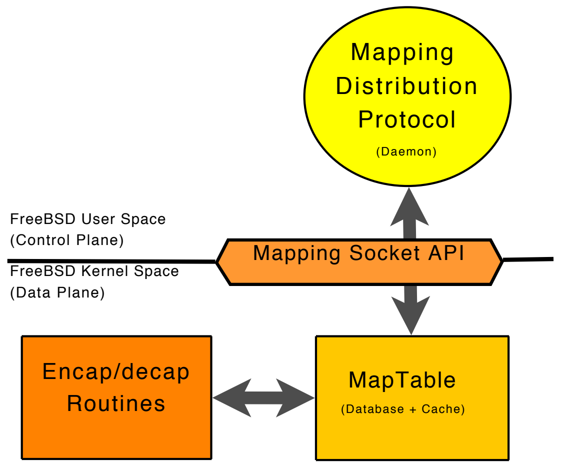

Figure 2: OpenLISP Architecture.

and LISP-DHT [28]). Every Mapping Distribution Protocol has pros and cons and different impact on the LISP Cache [25]; however, it is out of the scope of the present paper to compare and to analyze them. Hereafter we just provide a short overview on the LISP+ALT proposal since it is the solution adopted in the LISP working group and currently deployed in the international testbed ([18, 19]).

The LISP *ALternative Topology* (or LISP+ALT [20]) builds an overlay (the alternative topology), using Generic Routing Encapsulation (GRE) tunnels, among BGP routers advertising EID-Prefixes. These prefixes are pushed toward every node of the overlay. The main idea is to build the overlay in such a way that an aggregation of EID-Prefixes can be aggressively performed. In case of a cache miss, a query (which can also piggyback the packet that triggered the cache miss) is sent on the overlay that will deliver it to the owner of the needed mapping, which in turn will reply sending the mapping to the node that issued the request.

4. OpenLISP

OpenLISP is our implementation of LISP in the FreeBSD [29] operating system. The high-level architecture of OpenLISP is depicted in Fig. 2. In our work we focused on the LISP Data Plane, implemented directly in the kernel space. In the OpenLISP Data Plane there are the routines to perform encapsulation and decapsulation as well as both LISP's cache and database, which are merged in a single data structure called *MapTable*. We describe how we implemented the OpenLISP Data Plane in Section 4.1. Concerning the Control Plane, we purposely did not implement any specific Mapping Distribution Protocol because our aim was not to develop production software. Rather, our aim was to develop a flexible and extensible platform providing support for future experimentation of both new and existing Mapping Distribution Protocols. Nevertheless, we provided OpenLISP with some simple tools (described in Section 4.2) in order to have the possibility from the Control Plane to interact with the Data Plane. Such interaction is possible thanks to the new socket API that we developed in OpenLISP, namely the *mapping sockets*. The mapping socket API offers a flexible and simple communication interface between Control and Data Plane and is described in Section 5.

4.1. OpenLISP Data Plane

When designing new encapsulation and decapsulation features in an OS, the first idea that comes to mind is to define a new virtual interface, like for instance `gif` or `gre` in FreeBSD. Nevertheless, a virtual interface implies an address associated with it, which is not possible in the LISP context. Using RLOCs as virtual interface addresses is not an option since RLOCs are not used only for LISP packets but also for other traffic that could be present in the DFZ. An alternative solution would be to use a modified firewall or NAT (Network Address Translation). Nevertheless, it is our opinion that such an approach would be not sufficiently flexible and dynamic for our purposes. Furthermore, implementing an API toward the control plane would become a major issue.

For the above mentioned reasons, we decided to go for a direct implementation in the kernel protocol stack, also to provide an answer to the question of how would a protocol stack look like if LISP would be a fundamental piece of it. Using this direct approach does not mean that important changes have been introduced in the existing code. We mainly added new code and tried to maintain the changes to original files as small as possible, basically limited to some function calls if specific conditions are met.

4.1.1. Encapsulation/Decapsulation Routines

Compared to the original protocol stack implementation of the FreeBSD operating system ([30, 29]) four main routines have been added to handle encapsulation and decapsulation operations: `lisp_input()`, `lisp6_input()`, `lisp_output()`, and `lisp6_output()`. As the names suggest, the first two manage incoming IPv4 and IPv6 LISP packets, while the last two are responsible for outgoing IPv4 and IPv6 LISP packets. To describe where

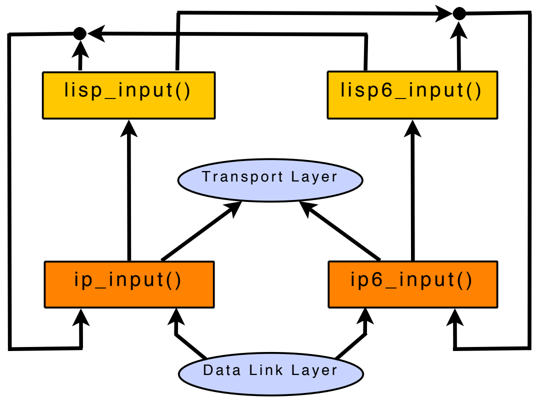

Figure 3: Protocol Stack Modifications for incoming packets.

these routines are positioned in the protocol stack we use the same representation as in [30]. The `lisp_input()`, `lisp6_input()` routines are positioned right above respectively `ip_input()` and `ip6_input()` routines, from which they are called, as depicted in Fig. 3. The `lisp_output()` and `lisp6_output()` routines are positioned right above respectively the `ip_output()` and `ip6_output()` routines, from which they are called, as depicted in Fig. 4. We describe hereafter how the packets are processed in the protocol stack in both encapsulation and decapsulation cases.

Incoming Packets

Let us assume that an IPv4 LISP packet is received by an OpenLISP system. The packet will first be treated by the `ip_input()` routine, which has been patched in order to recognize LISP packets. In the case of a LISP encapsulated data packet, `ip_input()` calls `lisp_input()` function passing the `mbuf` containing the incoming packet. `lisp_input()` strips the outer header.[5] Then, the IP header of the inner packet is checked in order to decide to which routine to deliver the packet, depending on the protocol version. In practice this means re-injecting the packet in the IP layer, by putting it in the input buffer either of `ip_input()` or `ip6_input()`. In the case of an IPv6 LISP packet the overall process is the same.

Once the packet has been re-injected in the protocol stack, it follows the normal process. In other words, if the system that decapsulated the packet is not the final destination the packet is delivered to `ip_forward()`

[5]`lisp_input()` also performs some consistency checks on the LISP specific header.

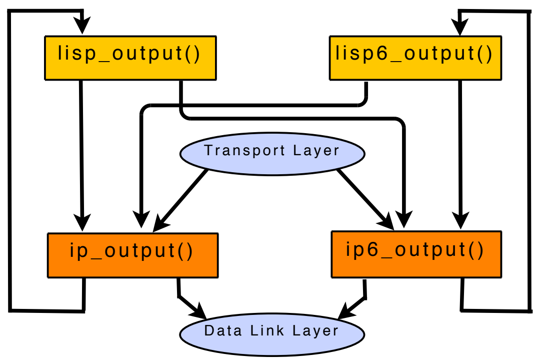

Figure 4: Protocol Stack Modifications for outgoing packets.

or `ip6_forward()`, depending on the IP version number. This will in turn deliver it to the output routine (`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 normal routing table of the system.

Outgoing Packets

Like in the previous section, let us assume that an IPv4 packet is received by the `ip_output()` routine of an OpenLISP system. This packet is not LISP encapsulated and can come either from `ip_forward()` or the transport layer (*i.e.*, `tcp_output()` or `udp_output()`). `ip_output()` has been patched in order to recognize if the packet needs to be encapsulated with a LISP header. The patch consists in calling a function checking if mappings are available in the LISP Database and the LISP Cache in order to build the new outer header. The needed mappings are searched and retrieved from the MapTables structure as detailed in Section 4.1.2.

A first lookup is performed using the source address (source EID) of the packet for a valid mapping in the LISP Database. If no mapping is found, OpenLISP assumes that the packet does not need to be encapsulated, *i.e.*, the system has no RLOCs for the source EID. In this case the packet is normally processed by `ip_output()`. If the mapping exists, OpenLISP assumes that the packet should be encapsulated. Thus, a second lookup is performed using the destination address (destination EID) of the packet for a valid mapping in the LISP Cache. If there is no mapping available, the packet is not encapsulated. Nonetheless, since the packet should be encapsulated, because a mapping exists in the LISP Database, this means that a cache miss has occurred and a message is sent through open mapping sockets in order to notify the Control Plane (details are in Section 5). If a mapping for the destination EID is present, the packet is diverted toward the `lisp_output()` routine, which first performs MTUs checks, then encapsulates the packet selecting the RLOCs to be used conforming to the LISP specifications.

Subsequently the packet is re-injected into the IP layer. This does not absolutely mean that the packet is delivered to `ip_output()`. Indeed, the selected RLOCs can be IPv6 addresses and the final packet can be IPv6 encapsulated, thus, in this case, it is delivered to `ip6_output()`. In the case of an outgoing IPv6 packet the overall process is the same. Once the packet is re-injected in the protocol stack, in both IPv4 and IPv6 cases, the packet follows the normal process.

OpenLISP does not allow recursive encapsulation, in order to protect against bad setups generating loops where a packet is recursively encapsulated until it is dropped due to MTU checks.

4.1.2. Map Tables

In Section 3 we described how LISP defines two different EID-Prefixes to RLOCs mapping storages: the LISP Database and the LISP Cache. OpenLISP merges the two databases in a single radix tree data structure [30] called MapTable. Such an approach allows having 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 tagged with a “database” flag, indicating that the mapping is owned locally. Thus, from a logical point of view the two data structures are still separated. When performing lookups it is possible to limit the scope of the lookup only to entries that have the database flag set to a particular value. In this way we obtain a behavior equivalent to performing a lookup on the LISP Cache or the LISP Database, as needed by the packet’s handling routines described in Section 4.1.1. Actually, there are two radix structures in the system, one for IPv4 EID-Prefixes and another for IPv6 EID-Prefixes.

Each entry of the MapTables, along with the fields necessary to build the radix tree itself, contain a pointer to a socket address structure that holds the EID-Prefix to which the entry is related, a flags field, and a chained list of RLOCs data structures containing the RLOCs addresses as well as their related metrics. The flags field contains general flags that apply to the whole mapping. For instance, this field contains the flag that tags a mapping as part of the LISP Database. In OpenLISP, when a mapping is tagged as part of the LISP Database, it is mandatory that at least one RLOC is a local address, *i.e.*, an address of one of the system’s interfaces; otherwise, an error is returned during insertion. This is because the mapping contains

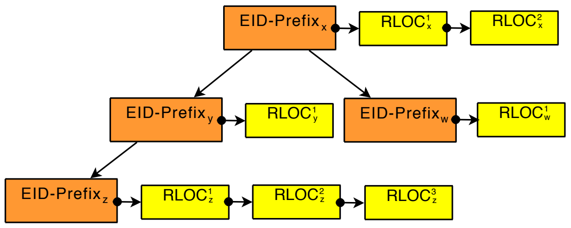

Figure 5: Example of MapTable data structure layout.

all the RLOCs of the domain, which can be distributed on several xTRs of the domain, thus not physically present on the system. Furthermore, when OpenLISP performs encapsulation, it only selects source RLOCs that are addresses of the system. The risk of doing otherwise would be to have encapsulated packets filtered in upstream routers because they are sent with a source address that does not belong to the system performing the encapsulation operation. An example of the layout of MapTables is presented in Fig. 5.

The fact that OpenLISP uses a chained list to store the RLOCs, allows the great flexibility of mixing IPv4 and IPv6 RLOCs, enabling IPv6 encapsulation for IPv4 packets and vice versa, depending on the RLOCs eventually chosen for the encapsulation. One can think that the use of a chained list is not an efficient choice. Nonetheless, the list is always maintained ordered following the criteria described in [11], with the most preferable RLOCs at the head. This means that during the encapsulation operation, when RLOCs are selected, the list is never scanned. The list is explored only during management operations, hence, normal encapsulation operations are not entailed.

To each RLOC in the chained list is also associated an MTU field, used to check if the size of the LISP-encapsulated packet fits the MTU of the outgoing interface. OpenLISP automatically fills this field when a local mapping (*i.e.*, part of the database) is added. More specifically, OpenLISP checks all the RLOCs of the local mapping, if it is an address belonging to the system it copies the MTU of the interface associated with the address.

4.2. OpenLISP Control Plane

As explained in the previous sections, we did not implement any specific Mapping Distribution Protocol, since our aim is to provide an open platform. Nevertheless, we developed two simple tools, namely `map` and `mapstat`, in order to have access to the OpenLISP Data Plane from a shell terminal.

The `map` utility provides a command-line interface to manipulate the networks' MapTables. This utility has similar functionalities to the `route`

```
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
        RLOC Addr: inet  10.1.0.0
        flags: <UP,STATIC,DONE>
```

Figure 6: Example of `map` usage.

```
freebsd% mapstat -X
 Mapping tables

 Internet:
 EID           Flags    Refs   # RLOC(s)
 10.0.0.0/16   US          1   1 2001::1
                               2 10.1.0.0
```

Figure 7: Example of `mapstat` usage.

utility, present in UNIX systems, for manipulating routing tables. The `map` utility supports several general options and commands, enabling the user to specify any arbitrary request that could be delivered via the API described in the next section. The manual of `map` can be found in the appendix of [31]. Fig. 6 shows an example of usage of the `map` tool to perform a lookup for the EID 10.0.0.1.

The `mapstat` command allows retrieving and displaying various contents of network-related LISP data structures. It is similar to the existing `netstat` command, hence, offering similar features but specific to the OpenLISP Data Plane. For instance, it is able to show a complete dump of the content of the MapTables and also a large set of statistics concerning encapsulation and decapsulation operations. Like for `map`, the manual of `mapstat` can be found in the appendix of [31]. Fig. 7 shows an example of usage of the `mapstat` tool to dump the content of the MapTables.

5. Mapping Sockets API

As previously explained, LISP operates in both the Data Plane and the Control Plane, hence, there is the need to make these two parts communicating with each other. The original LISP specifications do not define any API for this purpose. To this end, in line with the UNIX philosophy, we defined a new type of sockets that we called "*mapping sockets*". Mapping sockets are based on raw sockets in the newly defined `AF_MAP` domain and are in principle very similar to the well-known routing sockets ([30, 32]). On the one hand, mapping sockets allow Mapping Distribution Protocols running in the user space to send messages to the kernel space in order to perform operations and modify the kernel's data structure (*e.g.*, MapTables) and receive confirmation messages. On the other hand, mapping sockets offer also signaling functionality, allowing the kernel to notify daemons running in user space of specific events related to LISP (*e.g.*, cache miss). Like routing sockets,

```
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 8: Mapping socket message header.

mapping sockets are broadcast, meaning that messages sent from the kernel to the user space are delivered to all open sockets. This enables all processes dealing with LISP to be notified on specific events or changes performed by one of these processes.

The operations that can be performed on the MapTables are the following:

`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. The result consists in the same message sent back with a specific flag set to indicate that the operation has been done.

`delete`: Used to delete a mapping. It works in the same way as `add`.

`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 result is a message containing the requested mapping (if present).

When performing such operations it is possible to specify whether they concern the LISP Cache or the LISP Database by a flag indicating if the mapping is local or not (*cf.* Section 4.1.2).

The messages sent across mapping sockets are all composed of a common header, the `map_msghdr{}`, whose definition, for the sake of clarity, is depicted in Fig. 8. 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 the `rt_msghdr{}` structure for routing sockets; details about them and their use can be found in [30]. The `map_type` field obviously contains types valid only for mapping sockets, as defined in [31]. The `map_flags` field is used to set some general flags that concern the whole mapping entry of the message.

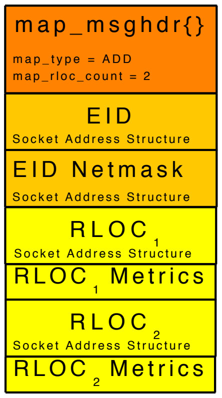

Figure 9: Example of mapping socket message structure.

The `map_addrs` field is a bitmask identifying the nature and number of data structures present in the message right after the header. These can be the EID, its netmask in case of a prefix, and the list of RLOCs. Nevertheless, the `map_addrs` field does not contain all the data structures, in particular for what concerns RLOCs. Indeed, the `map_addrs` field just states whether at least one RLOC is present; the exact number of RLOCs is contained in the `map_rloc_count` field. An example of the final structure of a mapping socket message for adding a mapping is depicted in Fig. 9. As can be seen, the EID and its mask are simple socket address structures, while an RLOC is composed of a socket address structure followed by a data structure (described in [31]) containing the metrics of that specific RLOC. An RLOC cannot be followed by a netmask since RLOCs are full addresses by definition.

Further details on how to open a mapping socket, read and write from it, the detailed message structure, the possible operations, and the signaled events can be found in [31].

6. Evaluation

In the previous sections we described the LISP protocol and how it can be implemented on a common UNIX system, namely our OpenLISP implementation for the FreeBSD operating system. In the present section, we provide an evaluation of the OpenLISP Data Plane, in order to measure the impact that the Locator/ID Separation paradigm has on traffic. Since LISP is a map-and-encap approach, relying on tunneling, in this study, we first evaluate the cost in terms of added latency in the forwarding operation when performing encapsulation and decapsulation (presented in Section 6.1).

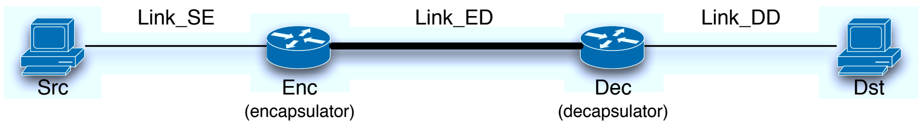

Figure 10: Testbed topology.

Then, we explore the impact of this increased latency on TCP flows (presented in Section 6.2). All experiments are performed for both IPv4 and IPv6 end-to-end traffic.

Fig. 10 presents the topology of the testbed we used in our evaluation. This testbed is composed of four identical dual processor PCs (Intel Xeon 5500 Quad-core 64-bit), equipped with two Intel PRO/1000 network interface cards. The four machines are connected in a linear topology (*cf.* Fig. 10) using Gigabit links. `Src` and `Dst` are respectively, the source and the destination of the generated traffic. These two machines run vanilla FreeBSD 7.3 kernel (*i.e.*, not enhanced with OpenLISP) throughout all the experiments. The two machines in the middle perform forwarding operations. In particular, when enabled, `Enc` performs encapsulation, hence operating as an ITR, while `Dec` performs decapsulation, hence acting as an ETR. Both machines run FreeBSD version 7.3 as well, and depending on the experiment performed, the kernel is enhanced with OpenLISP.

6.1. Packet forwarding latency

As a first set of measurements, we evaluate the additional latency that packets experience during the forwarding operation, due to encapsulation and decapsulation operations. This can be done by capturing, using `tcpdump`, all packets on the input and output interfaces. Then, by matching the packets in the two traces and subtracting their timestamp, it is possible to calculate the latency for each single packet.

As a baseline for comparison to LISP we first measured the latency experienced by packets when no encapsulation is used and normal routing is performed. We measured the latency with and without a LISP enabled stack on `Enc` and `Dec` for both IPv4 and IPv6.

As a second type of test we measured the latency when Generic Routing Encapsulation (GRE [33]) is used. We performed this measurement in order to compare LISP encapsulation/decapsulation operations with other protocols that perform the same kind of operations. We chose GRE because, like LISP, it introduces a shim header between the outer and the inner IP header. However, differently from LISP, the addresses in the outer header

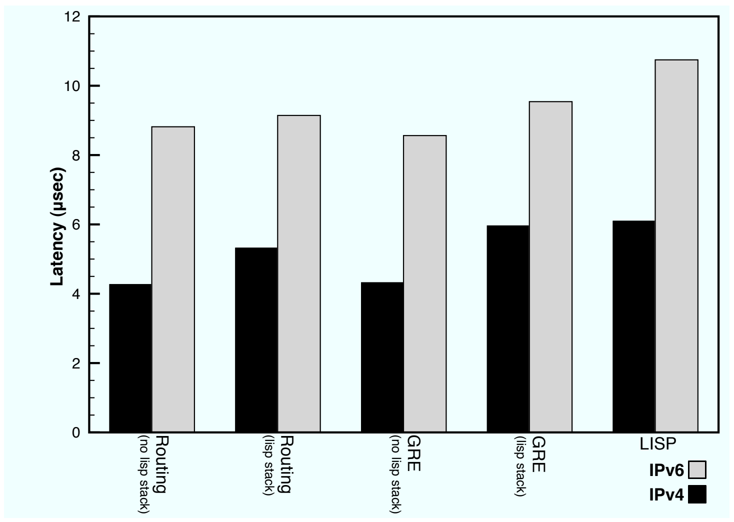

Figure 11: Packet Forwarding latency on `Enc`.

are statically configured, thus compared to LISP, which needs to lookup the LISP Database and the LISP Cache, the operation is simpler. Again, we performed the measurements with and without a LISP enabled stack on `Enc` and `Dec` for both IPv4 and IPv6. It is important to remark that when using IPv6 traffic, this is encapsulated in an IPv4 GRE tunnel, since there is a lack of support for GRE tunnels over IPv6 in FreeBSD.

In a third type of experiment we measured the latency introduced by LISP on `Enc` and `Dec`. To this end we installed one static mapping in the LISP Cache and LISP Database using the `map` utility described in Section 4.2. As for the previous experiments we performed measurements for both IPv4 and IPv6. In order to have a fair comparison with the GRE experiment, we set up the mappings in a way that when using IPv6, the flow is LISP encapsulated between `Enc` and `Dec` using IPv4 locators. It is important to notice that, when GRE or LISP encapsulation is used, the reverse path (between `Dec` and `Enc`) also uses the same encapsulation.

Figs. 11 and 12 present the results obtained, respectively, for `Enc` and `Dec` for both IPv4 and IPv6 traffic. These are obtained as an average of 100,000 UDP packets. As can be observed, even when LISP is not used for encapsulation, a LISP protocol stack has a higher latency, due to additional checks introduced in the input/output routines of the IP layer, as explained in Section 4.1. However, the impact in all cases is limited in the order of

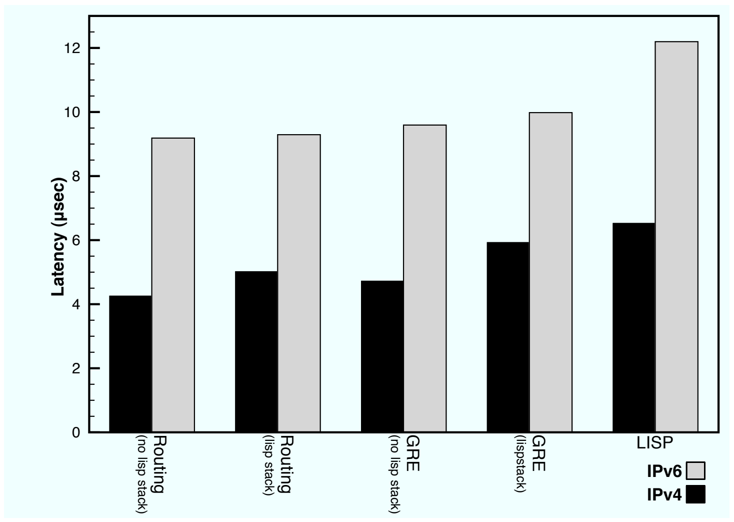

Figure 12: Packet forwarding latency on `Dec`.

1 μsec in both IPv4 and IPv6 cases.[6] Only in the case of LISP encapsulation for IPv6 packets, the increase is around 3 μsec (3.11 μsec, equivalent to $\sim$33%) for `Dec`, when comparing Routing without LISP stack and LISP. This is most probably due to the fact that the larger IPv6 header takes more time to be handled.

In general, the fact that the forwarding latency has increased was expected, since we are performing additional computation. Nevertheless, in comparison to the benefits offered by LISP, we consider the latency increase as acceptable. If very high forwarding speed has to be sustained, we would recommend hardware-accelerated solutions.

6.2. TCP throughput

In the previous section we proposed measurements concerning the forwarding latency internal to routers. Here we try to understand what is the impact of the increased latency on an end-to-end TCP connection. We chose TCP since it automatically adapts to the Round Trip Time (RTT) of the path.

We used the same experiment setup as for the latency measurements, thus with and without LISP protocol stack, with and without GRE tunneling, and

[6]The higher average latency observed for IPv6 is probably due to the fact that the header is much bigger compared to IPv4.

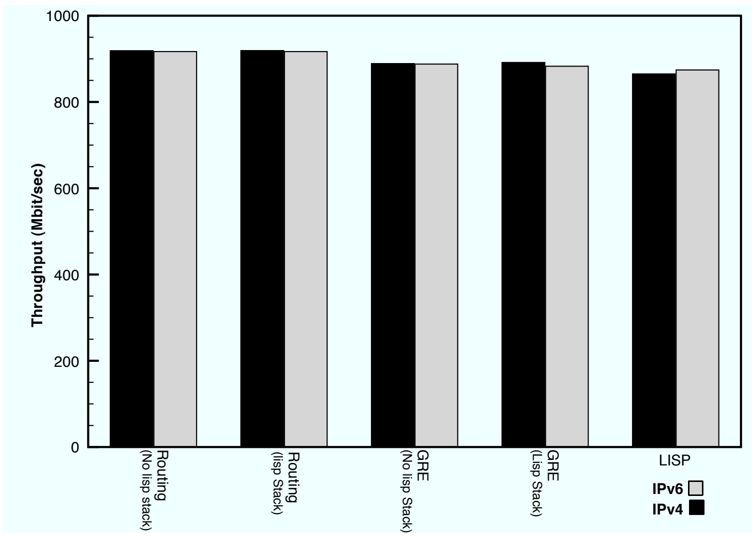

Figure 13: Average throughput for TCP flow.

for both IPv4 and IPv6. A single TCP flow was generated each time, ending after having successfully transferred 100 GB of data. For each experiment we measured the average throughput in Mbit/sec, and the total time needed to complete the transfer.

The results concerning the average throughput are summarized in Fig. 13. Obviously, the throughput at transport layer is on average lower for IPv6 since there is a larger overhead due to the bigger IP header. Also, when using LISP protocol stack the throughput lowers, and it becomes even lower when using LISP encapsulation. While, on the one hand, this was expected from the latency measurements, on the other hand, it can be observed that the reduction in throughput is equivalent to less than 5% (less than 45 Mbit/sec).

The exact values are also presented in Table 1 for IPv4 and Table 2 for IPv6. In the same tables there is the total transfer time needed for the 100 GB. As can be observed the transfer takes at least 900 seconds (*i.e.*, 15 min). IPv4 represents the worst case, where the use of LISP (compared to routing without LISP protocol stack) increases the transfer time by 58 seconds, representing an increase of 6.2%.

The presented results are promising, since they prove that introducing LISP has a limited impact. Further, what must be also taken into account is the fact that average flows in the Internet are much shorter and experience much higher delays, thus reducing the impact of LISP being almost negligible.

Table 1: TCP performances for IPv4 traffic (100GB transferred)

Forwarding Method	Transfer Time (sec)	Avg Throughput (Mbit/sec)
Routing (no LISP stack)	935.0	918.574
Routing (LISP stack)	934.7	918.967
GRE (no LISP stack)	966.3	888.925
GRE (LISP stack)	963.6	891.449
LISP	993.2	864.850

Table 2: TCP performances for IPv6 traffic (100GB transferred)

Forwarding Method	Transfer Time (sec)	Avag Throughput (Mbit/sec)
Routing (no LISP stack)	936.9	916.856
Routing (LISP stack)	936.9	916.830
GRE (no LISP stack)	967.5	887.892
GRE (LISP stack)	973.0	882.822
LISP	982.7	874.158

7. Conclusion

The present paper describes the overall architecture and the development work of OpenLISP, an open source implementation of the LISP proposal in the FreeBSD OS. OpenLISP provides complete Data Plane support for both encapsulation and decapsulation operations, both IPv4 and IPv6, as well as EID-to-RLOC mapping storage and efficient lookup.

The performance evaluation shows that the cost of running LISP in terms of forwarding latency is acceptable. Further, the impact on end-to-end flows is minimal. Indeed, the TCP measurements presented in this paper show less than 6.2% transfer time increase for 100 GB. Such a difference is exacerbated by the high bandwidth, low latency setup of our testbed. In more common scenarios the impact of LISP will be much lower. Nevertheless, we plan to perform kernel profiling in order to point out exactly which operation introduces the highest latency and possibly understand how performance can be further improved.

Since OpenLISP runs on FreeBSD, it is able to work on both routers and end-hosts, providing a wide range of test and deployment scenarios. We hope that OpenLISP will provide the research community with the right tool for exploring the Locator/ID Separation paradigm, gaining knowledge through experimentation. We designed OpenLISP to provide a flexible support for developing new Control Planes (*i.e.*, Mapping Distribution Protocols), which are the critical issue for any Future Internet Routing and Addressing Architecture based on the Locator/ID Separation paradigm. The *mapping sockets*

introduced by OpenLISP, allow storing only the essential information for the Data Plane in the MapTables, while any other information and procedures are maintained in the user space with a specific daemon. This approach is similar to what can be found in current routers where the Control Plane maintains an extended routing table, namely the RIB – Routing Information Base, while the Data Plane contains a reduced routing table with the minimal information needed to correctly perform packet forwarding, namely the FIB – Forwarding Information Base.
OpenLISP can be freely downloaded from: `http://www.openlisp.org`

Acknowledgement

This work is partially funded by Trilogy, a research project (ICT-216372) supported by the European Community under its Seventh Framework Program and a CISCO URP grant. The views expressed here are those of the authors only.

References

[1] D. Meyer, L. Zhang, K. Fall, Report from the IAB Workshop on Routing and Addressing, *RFC 4984*, IETF Network Working Group, September 2007.

[2] BGP Routing Table Analysis Report, Available Online at: *http://bgp.potaroo.net/*.

[3] B. Quoitin, L. Iannone, C. de Launois, O. Bonaventure, Evaluating the Benefits of the Locator/Identifier Separation, Proceedings 2nd ACM SIGCOMM Workshop on Mobility in the Evolving Internet Architecture (MobiArch), August 2007.

[4] T. Li, Design Goals for Scalable Internet Routing, Internet Draft *draft-irtf-rrg-design-goals-03.txt*, IRTF Internet Research Task Force, October 2010.

[5] T. Li, Recommendation for a Routing Architecture, Internet Draft *draft-irtf-rrg-recommendation-14.txt*, IRTF Internet Research Task Force, September 2010.

[6] M. O'Dell, GSE - An Alternate Addressing Architecture for IPv6, Internet Draft *draft-ietf-ipngwg-gseaddr-00.txt*, IETF Network Working Group, February 1997.

[7] C. Vogt, Six/One: A Solution for Routing and Addressing in IPv6, Internet Draft *draft-vogt-rrg-six-one-01.txt*, IETF Network Working Group, November 2007.

[8] R. Atkinson, ILNP - Identifier/Locator Network Protocol, Internet Draft *draft-rja-ilnp-intro-06.txt*, IRTF Internet Research Task Force, August 2010.

[9] R. Moskowitz, P. Nikander, Host Identity Protocol (HIP) Architecture, *RFC 4423*, IETF Network Working Group, May 2006.

[10] E. Nordmark, M. Bagnulo, Shim6: Level 3 Multihoming Shim Protocol for IPv6, Standards Track *RFC 5533*, IETF Network Working Group, June 2009.

[11] D. Farinacci, V. Fuller, D. Meyer, D. Lewis, Locator/ID separation protocol (LISP), Internet Draft *draft-ietf-lisp-09.txt*, IETF Network Working Group, October 2010.

[12] J. Saltzer, On the Naming and Binding of Network Destinations, RFC 1498, IETF Network Working Group, August 1993.

[13] R. Hiden, New Scheme for Internet Routing and Addressing (ENCAPS) for IPNG, RFC 1955, IETF Network Working Group, June 1996.

[14] N. Chiappa, Endpoints and Endpoint Names: A Proposed Enhancement to the Internet Architecture, Available Online at: *http://ana.lcs.mit.edu/ jnc/tech/endpoints.txt*, 1999.

[15] D. Jen, M. Meisel, D. Massey, L. Wang, B. Zhang, L. Zhang, APT: A Practical Transit Mapping Service, Internet Draft draft-jen-apt-01.txt, IETF Network Working Group, November 2007.

[16] W. Herrin, Tunneling Route Reduction Protocol (TRRP), Available Online at: *http://bill.herrin.us/network/trrp-rrg.html*, 2008.

[17] R. Whittle, Ivip - a new scalable routing and addressing architecture for the Internet, Available Online at: *http://www.firstpr.com.au/ip/ivip/*, 2008.

[18] Cisco - LISP Testbed for IPv4, Available Online at: *http://www.lisp4.net/*.

[19] Cisco - LISP Testbed for IPv6, Available Online at: *http://www.lisp6.net/*.

[20] D. Farinacci, V. Fuller, D. Meyer, D. Lewis, LISP Alternative Topology (LISP+ALT), Internet Draft *draft-ietf-lisp-alt-05.txt*, IETF Network Working Group, October 2010.

[21] D. Lewis, D. Meyer, D. Farinacci, V. Fuller, Interworking LISP with IPv4 and IPv6, Internet Draft *draft-ietf-lisp-interworking-01.txt*, IETF Network Working Group, August 2010.

[22] D. Saucez, B. Donnet, L. Iannone, O. Bonaventure, Interdomain Traffic Engineering in a Locator/Identifier Separation Context, Proceedings of IEEE Internet Network Management Workshop (INM'08), October 2008.

[23] M. Menth, D. Klein, M. Hartmann, Improvements to LISP Mobile Node, Prooceedings of 22^{nd} International Teletraffic Congress (ITC), September 2010.

[24] E. Lear, NERD: A Not-so-novel EID to RLOC Database, Internet Draft *draft-lear-lisp-nerd-08.txt*, IETF Network Working Group, March 2010.

[25] L. Iannone, O. Bonaventure, On the Cost of Caching Locator/ID Mappings, in: Proceedings of the 3rd International Conference on Emerging networking EXperiments and Technologies (CoNEXT'07), ACM, USA, December 2007.

[26] S. Brim, D. Farinacci, V. Fuller, D. Lewis, D. Meyer, LISP-CONS: A Content distribution Overlay Network Service for LISP, Internet Draft *draft-meyer-lisp-cons-04.txt*, IETF Network Working Group, April 2008.

[27] S. Brim, D. Farinacci, D. Meyer, J. Curran, EID Mappings Multicast Across Cooperating Systems for LISP, Internet Draft *draft-curran-lisp-emacs-00.txt*, IETF Network Working Group, November 2007.

[28] L. Mathy, L. Iannone, LISP-DHT: Towards a DHT to map identifiers onto locators, in: Proceedings of ReArch'08 - Re-Architecting the Internet., December 2008.

[29] The FreeBSD Project, "FreeBSD: the power to serve", Available Online at: *http://www.freebsd.org/*.

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

[31] L. Iannone, D. Saucez, O. Bonaventure, OpenLISP Implementation Report, Internet Draft *draft-iannone-openlisp-implementation-01.txt*, IETF Network Working Group, July 2008.

[32] W. Stevens, B. Fenner, A. Rudoff, UNIX Network Programming, Volume 2 The Sockets Networking API, 3rd Edition, Professional Computing Series, Addison-Wesley, 2004.

[33] D. Farinacci, T. Li, S. Hanks, D. Meyer, P. Traina, Generic Routing Encapsulation (GRE), *RFC 2784*, IETF Network Working Group, March 2000.