University of Amsterdam
System & Network Engineering

Research Project 2

A mapping library for the Locator/ID Separation Protocol (LISP)

Author

Marek Kuczyński
MSc. System and Network Engineering
University of Amsterdam
marek.kuczynski@os3.nl

Supervisors

Job Snijders
Network Architect
InTouch NV
job@intouch.eu

Rager Ossel
CEO
InTouch NV
rager@intouch.eu

Abstract

The Locator/ID Separation Protocol (LISP) is a protocol that is currently being developed by a Working Group in the Internet Engineering Task Force (IETF). LISP is a network architecture and a protocol that implements a new approach to addressing and routing on top of regular IP. In a nutshell: LISP separates the 'where' and the 'who' in networking and uses a mapping system to couple the routing location (where) and endpoint identifier (who).

A non-proprietary Linux implementation of LISP is in development, but there currently is no open source control-plane software. This paper gives a brief overview about the control plane of LISP and the Python library that was developed during the research project.

Keywords: LISP, control plane, routing, addressing, Python, library

July 10, 2011

Acknowledgments

I would like to thank Job Snijders, Rager Ossel, Maurits Dijkstra, Daan van der Sanden, Bart de Bruijn and Kevin Tieman for their unwavering support and valuable assistance throughout the research project. I would also like to thank InTouch NV for providing me with the opportunity to work at their office and to get acquainted with their professional environment.

- *Marek*

Contents

1 Introduction **4**

2 Concepts of LISP **5**

2.1 Routing packets 5

2.2 Mapping IDs to locations 5

2.3 Communicating with the regular Internet 6

2.4 Maintaining consistent address space over multiple locations 7

2.5 Summary of benefits 8

3 Previous research **9**

4 LISP control plane **10**

4.1 Control packet structure 10

4.2 Packet types 10

4.3 Control packet encapsulation 11

5 LISP library **12**

5.1 Features and internals 12

6 LISP Internet Gopher **13**

6.1 Features and internals 13

6.2 LIG example 14

7 Conclusion **15**

7.1 Publication of work 15

8 Appendix; source code **18**

8.1 Library 18

8.2 LIG 24

Glossary

Below, a summary of some terms used in this paper is given.

- **AS** - Autonomous System: A collection of routing prefixes that have a consistent routing policy on the Internet.
- **BGP** - Border Gateway Protocol: Protocol responsible for core routing of the Internet. It shares prefixes and how to reach them in a push-based fashion.
- **DFZ** - Default Free Zone: The collection of autonomous systems that do not require a route for their packets to be routed. Each autonomous system has full knowledge of all other autonomous systems around it.
- **EID** - Endpoint ID: An EID is an IPv4 or IPv6 address used in the source and destination address fields of the first (most inner) LISP header of a packet.
- **ETR** - Egress Tunnel Router: An ETR is a router that accepts an IP packet where the destination address in the "outer" IP header is one of its own RLOCs.
- **ITR** - Ingress Tunnel Router: An ITR receives IP packets from site end-systems on one side and sends LISP-encapsulated IP packets toward the Internet on the other side.
- **IXP** - Internet Exchange Point: gathering point where parties exchange network traffic with each other.
- **PETR** - Proxy ETR: A PETR is used for inter-networking between LISP and Non-LISP sites. A PETR acts like an ETR on behalf of LISP sites.
- **PITR** - Proxy ITR: A PITR is used for inter-networking between Non-LISP and LISP sites. A PITR acts like an ITR on behalf of non-LISP sites.
- **RLOC** -Routing Locator: A RLOC is an IPv4 or IPv6 address of an egress tunnel router (ETR). A RLOC is the output of a EID-to-RLOC mapping lookup.
- **xTR**: A xTR is a reference to an ITR or ETR when direction of data flow is not part of the context description. xTR refers to the router that is the tunnel endpoint.

1 Introduction

The LISP protocol[1] is the result of an IETF workshop that was held in 2006[1]. The attendees of the conference agreed on the fact that a new routing architecture for the Internet should be developed, because of scalability issues with the current Border Gateway Protocol (BGP) [2] based model.

Right now, nodes on the Internet use the BGP to share information about the IP address prefixes that are part of their Autonomous System (AS) [3]. Additionally, every node also maintains information about the best path to all other IP prefixes on the Internet. All these characteristics are distributed using a push mechanism that floods available information to all neighbors of the node.

The problems with scale are apparent; the mechanism works well when the total amount of prefixes is small. However, the Internet has grown explosively since 1994 when BGP version 4 was first introduced. It is becoming increasingly harder to maintain all the up-to-date routing information in the memory of a node. Right now, external routing servers are often used in order to store all the routing tables that a node needs to access [4]. This introduces new problems, like for example synchronization of this information and delays caused by looking up the information.

What makes the situation even worse, is that not all prefixes are aggregated in the most optimal way. Sometimes, fractions of a prefix are advertised, which adds to the size of the routing tables. All routes that are part of the default free zone (DFZ) [19] are managed by the parties that operate the autonomous systems individually, meaning that it is difficult to enforce or stimulate the most effective way to spread prefixes.

Splitting IDs from locations

Splitting relation between a prefix ("identity") and a node ("location") can provide a solution for issues like these [20]. It makes it possible to, for example, address multiple prefixes to a single autonomous system, without including all the path information every single time. The concept also applies to routing single IP addresses, since it may be good enough to know the specifics of reaching a very large aggregate somewhere, which will take care of further routing once the IP packet arrives there.

The LISP protocol makes use of this concept, not only to decrease the size of the BGP tables, but also to enable easy to manage OSI level 3 routing [5]. It enables companies or institutions to maintain their own allocated address space without needing to control the AS or the BGP configuration. Additionally, LISP uses a pull mechanism to retrieve information about prefix mappings instead of a push mechanism like BGP. This means that the information is always up to date and that information about prefixes that you are unlikely to ever reach do not need to be stored in the node.

You will read more about the characteristics and possibilities of LISP in the next few chapters. After that, a section follows that describes the control plane library that has been created as part of this project. Finally, the future work and conclusions are listed.

2 Concepts of LISP

2.1 Routing packets

The LISP protocol operates using OSI layer 3, this layer is used to specify the sending and receiving IP version 4 or 6 address of a packet. In a normal situation when a packet gets sent out, it traverses the local network card and eventually ends up on a node that can route it further towards its destination. This is quite a static procedure, since the entire path the packet will follow is determined based only on the sending and receiving IP addresses and global routing policies that the packet may encounter on the Internet. The sender already needs to have specific information about the destination that it wants to reach.

This is not much of an issue in small environments where for example one IP address represents one server. It gets more difficult when one IP address represents multiple servers or even complete server farms - how do you spread the load over all servers equally? And how do you determine which node contains, for example, the e-mails that a customer has received? There is currently no proper solution for issues like these.

Right now, the load balancing often gets mitigated by the use of round robin DNS [6], where a random IP address from a predefined pool gets returned every time a domain name is looked up. Unfortunately, this setup makes you dependant on the correct functioning higher OSI layers, meaning that you need to send, receive and process a DNS query before your host knows where to address the packet to. Additionally, the method is dependant on application support for DNS and it does not offer decent granularity regarding load balancing. The reachability might partially be solved, but end-user security and network exposure to the outside world are issues that are introduced.

2.2 Mapping IDs to locations

However, DNS can provide consistent addressability since it uses a decentralized, hierarchical database to map labels to IP addresses. The LISP protocol introduced a similar concept, except that identities (in the form of IP addresses or other address types) are looked up and their true IP address location is returned. The LISP mapping database contains the relations between endpoint identifiers (EIDs) and routing locators (RLOCs) and these mappings are spread in a distributed, decentralized fashion.

Figure 1: LISP logo

These routing locators form the gateways between a site or host and the Internet and are announced in the conventional BGP Internet using a normally routeable IPv4 or IPv6

addresses. LISP RLOCs are capable of doing the lookups in the mapping database described above, meaning that they know which RLOC should be addressed if a packet destined for some endpoint identifier (EID) needs to be forwarded. If two LISP sites want to communicate with each other, they can use any type of EID they like. This is not limited to just IP addressing; MAC addresses, GPS coordinates or any type of consistent addressing can be used, as long as you can partition it logically.

After doing the lookup for the EID, the sending RLOC will encapsulate the packet and address it to the IP address of the destination RLOC. The receiving RLOC will decapsulate the packet and send it to whatever destination address was specified in the payload. In most cases, both RLOCs will operate as xTRs (bidirectional tunnel routers), which means they handle both encapsulation and decapsulation from and to a site. This is not a hard requirement though, you can also split these roles over multiple devices for upstream and downstream Internet connectivity.

2.3 Communicating with the regular Internet

The situation illustrated above applies only if both gateways are LISP capable and registered properly. This can be determined by doing a lookup in the mapping database, since there will be no records returned for a site not using LISP. If only one site is LISP enabled, then a packet sent from the LISP site is encapsulated and flows to a a LISP proxy router (called PxTR if it handles both ingress and egress traffic). Once the packet leaves the PxTR, BGP takes over the routing process.

The PxTR does not need to be physically connected to the rest of the LISP site, as long as it is addressable by the xTR router that acts as a gateway for a LISP site. The PxTR will announce the IP address prefixes that it is responsible for in regular BGP, through which it will attract the traffic destined for these sites. It will then encapsulate traffic in between the PXTR and the destination RLOC xTR, which will decapsulate the packet again. An example of these techniques is shown in the next paragraph.

2.4 Maintaining consistent address space over multiple locations

Imagine that the University of Amsterdam owns a /24 IPv4 address block that it wants to share with two American universities. In order to do this, all universities would need a LISP enabled router on the edge of their internal network. These routers have to register with the central mapping database which will keep track of the IP address allocated by their Internet carrier. Additionally, they have to configure which part of the EID space their edge router represents, this information is also registered and stored in the mapping database.

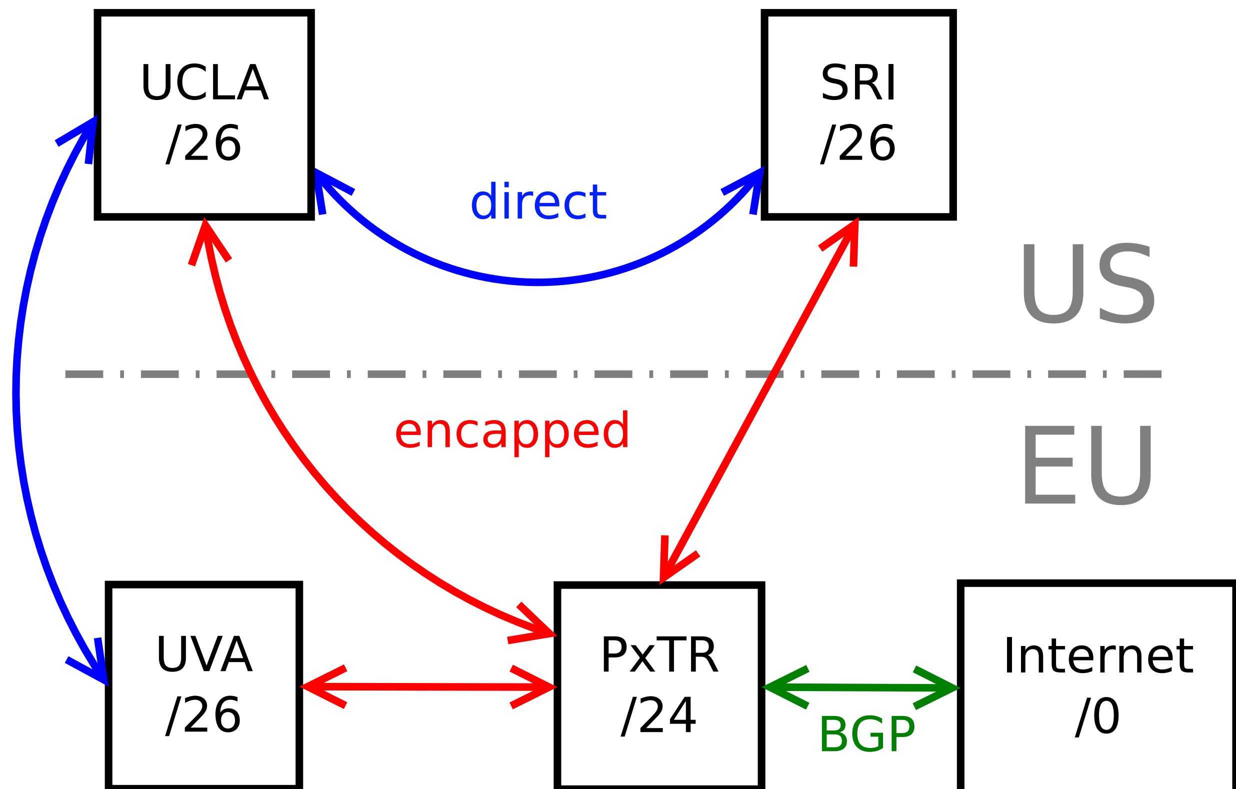


Figure 2: A fictional topology where research institutions can maintain a consistent address space between eachother. Every site gets an allocated subnet out of the public /24. Note; the LISP map resolvers and map servers are not shown on this figure.

When the UVA wants to send traffic to UCLA, the UCLA RLOC address is looked up and traffic can be encapsulated between both RLOC routers directly. The same goes for traffic in between UCLA and SRI, even though the address space is allocated to the UVA in Europe. Traffic in between LISP sites always has a link stretch of 1.0, irregardless of the ISP that connects each site. What should be mentioned as well, is that this minimal stretch requires no site specific configuration, except for the one-time setup mentioned before. The universities can also use other identifiers in their EID space, like for example MAC addresses, to address each other. This can be especially useful when dealing with virtual server environments.

The universities can also communicate with the rest of the world through the PxTR, which will encapsulate the traffic between itself and the sites, while advertising the IP prefixes to the DFZ. Multiple PxTRs can support the topology as long as they are advertised within BGP, so multiple PxTRs could be split up across both continents.

2.5 Summary of benefits

Segregating the "where" from the "who" using LISP provides the following advantages;

- **Consistent and global addressing**
  The university example shows how multiple individual institutions can share address space in order to simplify addressing. The available subnets can be divided in any possible way and assigned to any LISP enabled site. These addresses can either be used only internally, or used as globally routeable addresses as well.

- **Independence from Internet carrier**
  LISP makes use of a mapping system where any IPv4 or IPv6 address can be used as RLOC address. It is therefore irrelevant which Internet service provider is used and what addresses it provides to its endpoints. Multiple endpoint addresses can be associated with one EID space and different metrics like weight and priority can be assigned to them. This increases the reliability of the connectivity a site has to the Internet and also enables traffic engineering to be configured on these links.

- **Layer 2 connectivity using layer 3**
  Layer 2 addresses can be used to address other nodes or networks over layer 3. This means that there is no need for dedicated layer 2 connections between sites, since this can be done over cheaper, easier available and more flexible layer 3 connections. The Instance ID feature of LISP can also guarantee that the connection is secure.

  Keep in mind though that you do lose control over a part of the OSI stack, meaning that you are dependant on another party for layer 2 connectivity. Spreading your connections out over multiple ISPs with different backbones and equipment should provide enough reliability though.

- **IPv6 transition support**
  Many Internet exchange points (IXPs) [21] can route IPv6 traffic at this point, but the connection in between a modem and the IXP is often not ready for IPv6 yet. Using LISP, the traffic in between a xTR and PxTR can be encapsulated with an IPv4 header, meaning that an IPv6 incompatible part of the infrastructure can still be used. Once the path in between the modem and the IXP becomes IPv6 ready, the prefix could be changed using IPv6 router advertisements and LISP could be removed from the network (or it can be used for the other benefits that it offers).

- **Enhanced mobility**
  So far, this paper has described the solutions that LISP offers for network-based flexibility. The same concepts can be applied for individual hosts as well, which offers a whole new scala of opportunities. As an example, mobile phones can register a public IP by contacting the LISP mapping system, irregardless of whether they are using a wireless network or any form of mobile Internet. Connectivity with the outside world goes through the PxTR while connectivity between LISP capable mobile phones has a link stretch of 1. Mobile devices can call each other simply by sending packets to the known public IP address, without the need for a central infrastructure to route the traffic through.

  The software to make this possible on the Android mobile operating system is currently in testing, while Qualcomm (a large manufacturer of mobile phone chips) is considering to implement the protocol in mobile chips. Testing of mobile nodes is also taking place on the public LISP beta network [7].

3 Previous research

The LISP protocol is still under heavy development and it is still in draft status according to the IETF. The documentation that belongs to the protocol development can be found here and here. Over the last few years, the following research has been performed;

- The University of Louvain has released an alpha of OpenLISP[8], which enables encapsulation and decapsulation of LISP packets in the FreeBSD kernel. Their software provides data plane functionality, which can be combined with the control plane through an API.
- Atilla de Groot (former UvA and OS3 student) has evaluated the LISP control plane using OpenLISP and ALT in 2009 [9]. His research focused mostly on the overlay topology that enables LISP sites to share RLOC and EID information. One of his conclusion was that the OpenLISP software was not feature complete yet and that a control plane packet daemon is required.
- A lookup tool called LIG has been developed to query the LISP mapping database [10]. It queries for the location of an EID, after which it receives an RLOC as reply. Later in this paper you will read about the Python implementation that has been created of this tool during the project.
- A LISP beta network [7] has been operational for a couple of years. It enables multi-vendor testing of the protocol and new features, like for example the LISP mobile node extensions. Participants include universities, hardware vendors, but also large Internet companies like Facebook and Google. As of July 2011, it consists of approximately 100 xTRs, over 10 map servers and around 10 PxTRs, all spread over 25 countries [11].
- InTouch NV[12] is currently in the process of deploying LISP on the end-user network. The protocol will replace the current MPLS structure and will enable greater flexibility for customers in various ways. The current implementations of LISP are considered stable enough for a production environment.

4 LISP control plane

The LISP control plane takes care of acquiring, maintaining and sharing RLOC and EID information within the LISP ecosystem. The following list gives a brief overview of the control plane functionality of the LISP protocol. The information is based on `draft-ietf-lisp-13` [13] of the LISP protocol which was released in June 2011, the packets implemented in the library are described in this document as well.

4.1 Control packet structure

The general structure of the packets is as follows;

<table>
<tr><td colspan="2">IPv4 header</td><td colspan="2">IPv6 header</td></tr>
<tr><td colspan="4">UDP</td></tr>
<tr><td>MapRequest</td><td>MapReply</td><td>MapRegister</td><td>MapNotify</td></tr>
<tr><td colspan="4">Nonce</td></tr>
<tr><td>Source EID + RLOC</td><td></td><td colspan="2">Authentication Data</td></tr>
<tr><td colspan="4">N x LISP records</td></tr>
</table>

The contents of the LISP records vary per packet type.

4.2 Packet types

- **MapRequest**
  The map request is sent out in order to retrieve an RLOC address for an EID prefix. A host can send this packet to a map server which will send back a reply message. The map request contains several flags denoting what kind of request is made. The packet also contains RLOC information about the requesting node and the address plus address family that is being requested.

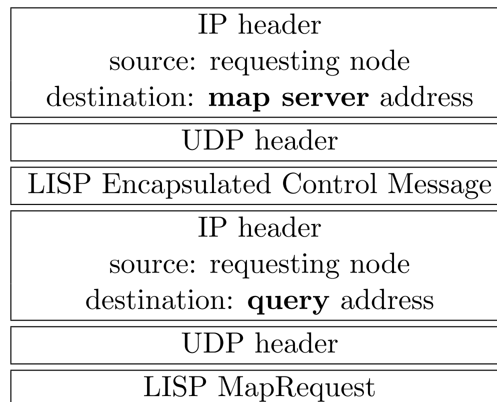
- **MapReply**
  This packet is sent back to the requester, containing the EID allocation plus its size. It also contains TTL information for the record, RLOC addresses and information about the metrics of the space.

- **MapRegister**
  The register packet announces that a xTR is going to be responsible for an EID prefix. The packet contains the addresses of the prefix, the sizes of them and authentication data in order to validate the request.

- **MapNotification**
  The map notification is used to notify a remote LISP site about local changes in the RLOC or EID space. The packet contents are very similar to the map register packet format.

4.3 Control packet encapsulation

Encapsulated Control Message
The ECM is not really a packet type of its own, but rather a wrapper around another LISP control message. It adds an additional IP, UDP and LISP header in front of a regular LISP packet, through which two machines can be addressed at once. As an example, when making a map request, the following structure is followed;



The packet will first be received by the map server, which will strip the packet from the top three layers. After that, the packet is forwarded to the RLOC IP address responsible for the EID. This RLOC will send a map reply back to the original requester. All involved nodes are now up to date of each others status after this packet triangle has completed.

5 LISP library

Programming language: Python

Python [14] was chosen as the programming language for the creation of the library. Reasons include easy portability across operating systems and an extensive amount of libraries that can be used. Additionally, it is a very accessible and widely used programming language that can easily be used in a modular fashion. Python will not perform as great on high data throughput as C will, but it is a good choice for lower throughput functions like control planes.

Packet crafting library: Scapy

Scapy [15] is a well-known library for packet manipulation in Python. It uses the concept of 'layers' in order to craft packets, where every layer stands for a network protocol like IP or UDP. These layers can be chained together and special fields like checksums over payload can automatically be calculated. The research group could therefor focus completely on the design and coding of the new LISP layer, which will be submitted to the 'contribute' code base of Scapy in the near future.

5.1 Features and internals

The final version of the library can craft all the control plane packets of LISP. After initializing a packet layer class, the user can specify the parameters that the packet should contain. Special fields like record count fields, address identifier fields and the varying bit length of these fields are taken care of as well, they are automatically calculated when the packet is built.

The library works fully synchronous, meaning that it can both generate and dissect LISP packets. The library can also handle both IPv4 and IPv6 addresses and destinations while any new address families for EIDs can be added easily. The library has been verified to work with existing networking hardware and it can also dissect packet captures or live traffic sent by these devices.

6 LISP Internet Gopher

Once the library was finalized, some basic tools were built in order to interact with LISP nodes. The first and finalized piece of software is a Python implementation of the LISP Internet Gopher (LIG). The original implementation of this tool has been written in C by David Meyer [10].

LIG can query a LISP map server for the RLOC belonging to a certain EID. This happens by sending out a encapsulated map request to the map server, in a similar fashion as described in paragraph 4.3. After this, the map reply is processed and displayed by the requesting node.

6.1 Features and internals

The Python implementation of LIG extends on the original by storing the requests and replies in Python objects. This means that the library and the Python LIG can be used together in order to create a daemon or other type of service.

LIG uses the following syntax to request RLOCs;

```
1 $ ./lig.py <map-server> <EID query>
```

The following steps are taken in order to send the request and to process the reply.

1. A check is done whether arguments are supplied on the command line, or whether LIG should drop to the Python shell for user input.
2. A check is done to see whether the input values are IP addresses or DNS names. DNS names are resolved and stored as the corresponding IP address.
3. Information about the sending node is collected. Checks are done to see whether the node has an IPv4 and/or IPv6 address, this is important since LISP topologies can use either one address family or both. IPv6 is preferred as the sending address, but then there obviously has to be an IPv6 address for the destination as well.
4. The request packet is built with all the information available. Checksums for the IP headers are calculated.
5. A socket is opened on the local machine. This socket will capture all traffic for a short period (default 1 second) and filter out the LISP map replies.
6. The correct map reply is verified by checking the packet type and by comparing the nonce from the sender and receiver. This is done by iterating through the array containing the network card capture.
7. The output of the capture is printed to the terminal.

6.2 LIG example

An example of the in and output of LIG is displayed below. The server and query in this example are DNS names that are mapped to IP addresses representing a map server and EID (`85.184.2.42` and `85.184.3.77` respectively). The output of 'Ethernet' and 'IP' has been omitted.

```
1  marek@lisp-dev:~/py-lispnetworking$ sudo ./lig.py ms.marek.asia marek.asia
2  WARNING: No route found for IPv6 destination :: (no default route?)
3  Begin emission:
4  ...
5
6  ###[ Ethernet ]###
7  ...
8
9  ###[ IP ]###
10 ...
11
12 ###[ UDP ]###
13         sport     = 4342
14         dport     = 62337
15         len       = 48
16         chksum    = 0xb1c8
17 ###[ LISP ]###
18 ###[ LISP Map-Reply packet ]###
19                 ptype     = 2L
20                 reply_flags=
21                 p2        = 0L
22                 reserved  = 0L
23                 map_count = 1
24                 nonce     = 0x39ffd3ec9
25                 \map_records\
26                  |###[ LISP Map-Reply Record ]###
27                  |   record_ttl= 1440L
28                  |   locator_count= 1
29                  |   eid_prefix_length= 29
30                  |   action    = no_action
31                  |   authoritative= 1L
32                  |   reserved  = 0L
33                  |   map_version_number= 0L
34                  |   record_afi= 1
35                  |   record_address= '85.184.3.72'
36                  |   \locators  \
37                  |    |###[ LISP Locator Records ]###
38                  |    |   priority  = 50
39                  |    |   weight    = 50
40                  |    |   multicast_priority= 255
41                  |    |   multicast_weight= 0
42                  |    |   reserved  = 0L
43                  |    |   locator_flags= local_locator+route
44                  |    |   locator_afi= 1
45                  |    |   address   = '82.136.213.75'
```

7 Conclusion

Both the library and the Python implementation of LIG lay the groundwork for the following;

- **Registration of hosts in LISP**
  The library can interact with the LISP mapping database and it should be able to successfully finish the transaction of a registration. The library plus some additional software should be able to register an OpenLISP node as a valid participant, since OpenLISP has the kernel support to process the packets generated by LISP. The research team is excited about the possibilities that further development of the LISP mobile node can bring in the near future.
- **Testing of LISP control plane**
  The LISP control plane specifications are still work in progress and some design decisions haven't been made yet. Using the library, the correct functioning of prototype designs can be verified and diagnostic information can be retrieved about possible failures during the processing of control plane packets.
- **Packet handling for the creation of a daemon**
  With Python LIG, it is already possible to perform lookups in the LISP database system. The packet handling of the library could be combined with the necessary logic to automatically receive and reply to status requests.
- **Debugging and hacking of routing hardware**
  At the moment, it is difficult to debug routing equipment since you are limited to interaction using other hardware. It would be very useful to test the current LISP implementations in routers, for example by sending malformed or illegal packets. Right now, these kind of issues are still easy to fix, since LISP is not deployed on a massive scale yet.
- **Other creative solutions**
  The development of the Python LIG implementation proves that it is not difficult to create tools which can interact with the LISP control plane. Much like with the draft definitions, the exact business models for LISP are still on the drawing table. Simple control plane tools could enable management, status updates or basic configuration to be done by third parties or customers.

7.1 Publication of work

The source code of the programs produced during the research project can be downloaded of Github[16] or found in the appendix. The source code falls under the GPL2 license[18]. This document and the presentation slides used to present it can be downloaded from the research project website [17].

References

[1] *LISP protocol information*
http://www.lisp4.net/

[2] *Border Gateway Protocol*
http://en.wikipedia.org/wiki/Border_Gateway_Protocol

[3] *Autonomous Systems*
http://en.wikipedia.org/wiki/Autonomous_system_(Internet)

[4] *Route reflectors*
http://en.wikipedia.org/wiki/Route_reflector

[5] *OSI Layer 3*
http://en.wikipedia.org/wiki/OSI_Layer_3

[6] *Round-robin DNS*
http://en.wikipedia.org/wiki/Round-robin_DNS

[7] *LISP Beta network*
http://www.lisp4.net/lisp-site/

[8] *OpenLISP*
http://www.openlisp.org/

[9] *Implementing OpenLISP with LISP + ALT, Atilla de Groot, 2009*
http://cees.delaat.net/sne-2008-2009/p06/report.pdf

[10] *LIG C source code*
https://github.com/davidmeyer/lig

[11] *Overview of LISP beta network*
http://www.lisp4.net/images/lisp-alt.pdf

[12] *InTouch NV*
http://www.intouch.eu

[13] *LISP draft 13*
http://tools.ietf.org/id/draft-ietf-lisp-13.txt

[14] *Python*
http://en.wikipedia.org/wiki/Python_(programming_language)

[15] *Scapy*
http://www.secdev.org/projects/scapy/

[16] *RP2 project source code*
https://github.com/intouch/py-lispnetworking

[17] *RP2 page containing this document and presentation*
http://cees.delaat.net/sne-2010-2011/index.html

[18] *GPL2 information*
http://www.gnu.org/licenses/gpl-2.0.html

[19] *Default Free Zone*
http://en.wikipedia.org/wiki/Default-free_zone

[20] *Locator/ID split*
http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=4685199

[21] *Internet Exchange Point*
http://en.wikipedia.org/wiki/Internet_exchange_point

All references have been accessed in July 2011.

8 Appendix; source code

Last updated; July 10, 2011. Visit Github [16] for updated versions.

8.1 Library

```
1  #!/usr/bin/env python
2  # scapy.contrib.description = Locator ID Separation Protocol
3  # scapy.contrib.status = loads
4  """
5      This file is part of a toolset to manipulate LISP control-plane
6      packets "py-lispnetworking".
7
8      Copyright (C) 2011 Marek Kuczynski <marek@intouch.eu>
9      Copyright (C) 2011 Job Snijders <job@intouch.eu>
10
11     This file is subject to the terms and conditions of the GNU General
12     Public License. See the file COPYING in the main directory of this
13     archive for more details.
14 """
15
16 import socket,struct,random,netifaces,sys
17 from string import ascii_letters
18 from scapy import *
19 from scapy.all import *
20
21 """ GENERAL DECLARATIONS """
22
23 _LISP_TYPES = {
24     0 : "reserved",
25     1 : "maprequest",
26     2 : "mapreply",
27     3 : "mapregister",
28     4 : "mapnotify",
29     8 : "encapsulated_control_message"
30 }
31
32 _LISP_MAP_REPLY_ACTIONS = {
33     0 : "no_action",
34     1 : "native_forward",
35     2 : "send_map_request",
36     3 : "drop"
37 }
38
39 _AFI = {
40     """ An AFI value of 0 used in this specification indicates an unspecified
           encoded address where the length of the address is 0 bytes following the
           16-bit AFI value of 0. See the following URL for the other values:
41     http://www.iana.org/assignments/address-family-numbers/address-family-numbers
           .xml """
42
43     "zero" : 0,
44     "ipv4" : 1,
45     "ipv6" : 2,
46     "lcaf" : 16387
47 }
48
```

```
49 """ nonce_max determines the maximum value of a nonce field. The default is set
       to 18446744073709551615, since this is the maximum possible value (>>> int('f
       '*16, 16)). TODO - see about the entropy for this source"""
50 nonce_max = 16777215000
51 nonce_min = 15000000000
52
53 """CLASS TO DETERMINE WHICH PACKET TYPE TO INTERPRET
54 scapy is designed to read out bytes before it can call another class. we are
       using the ugly conditional construction you see below to circumvent this,
       since all classes must have the length of one or more bytes. improving and
       making this prettier is still on the TODO list """
55
56 class LISP(Packet):
57     name = "LISP"
58
59     def guess_payload_class(self, payload):
60         # read the payload (non interpreted part of the packet string) into a
               variable
61         a = payload[:1]
62                 # put the hex from the packet remainder into an attribute
63         b = struct.unpack("B", a)
64                 # shift the value from the attribute for 4 bits, so that we have
                       only the 4 bit type value that we care about in the form of a
                       byte. this means that flags are not taken into account in this
                        value, which makes enumeration much cleaner and easier.
65         c = b[0] >> 4
66
67                 # compare the integer from the value to the packettype and
                       continue to the correct class
68         if c == 1:
69             return LISP_MapRequest
70         elif c == 2:
71             return LISP_MapReply
72         elif c == 3:
73             return LISP_MapRegister
74         elif c == 8:
75             return LISP_Encapsulated_Control_Message
76         else:
77             return payload
78
79
80 """ the class below reads the first byte of an unidentified IPv4 or IPv6 header.
       it then checks the first byte of the payload to see if its IPv4 or IPv6 header
       . the IPv4 header contains a byte to describe the IP version, which is always
       hex45. IPv6 has a 4 bit header, which is harder to read in scapy. maybe this
       can be done in a prettier way - TODO """
81
82 class LCAF_Type(Packet):
83     def guess_payload_class(self, payload):
84         a = payload[:1]
85         b = struct.unpack("B", a)
86         c = b[0] >> 4
87
88         if c == 4:
89             return IP
90         elif c == 6:
91             return IPv6
92         elif c == 16387:
93                 print "LCAF, WIP"
94         else:
```

```
95                  return payload
96
97  """
98  LISPAddressField, Dealing with addresses in LISP context, the packets often
        contain (afi, address) where the afi decides the length of the address (0, 32
        or 128 bit). LISPAddressField will parse an IPField or an IP6Field depending
        on the value of the AFI field.
99  """
100
101 class LISP_AddressField(Field):
102     def __init__(self, fld_name, ip_fld_name):
103         Field.__init__(self, ip_fld_name, '0')
104
105         self.fld_name=fld_name
106         self._ip_field=IPField(ip_fld_name, '127.0.0.1')
107         self._ip6_field=IP6Field(ip_fld_name, '::1')
108
109     def getfield(self, pkt, s):
110         if getattr(pkt, self.fld_name) == _AFI["ipv4"]:
111             return self._ip_field.getfield(pkt,s)
112         elif getattr(pkt, self.fld_name) == _AFI["ipv6"]:
113             return self._ip6_field.getfield(pkt,s)
114
115     def addfield(self, pkt, s, val):
116         if getattr(pkt, self.fld_name) == _AFI["ipv4"]:
117             return self._ip_field.addfield(pkt, s, val)
118         elif getattr(pkt, self.fld_name) == _AFI["ipv6"]:
119             return self._ip6_field.addfield(pkt, s, val)
120
121
122 """RECORD FIELDS, PART OF THE REPLY, REQUEST, NOTIFY OR REGISTER PACKET CLASSES
        """
123
124 """ LISP Address Field, used multiple times whenever an AFI determines the length
         of the IP field. for example, IPv4 requires 32 bits of storage while IPv6
        needs 128 bits. This field can easily be extended once new LISP LCAF formats
        are needed, see the LISP_AddressField class for this. """
125 class LISP_AFI_Address(Packet):                    # used for 4 byte fields that
         contain a AFI and a v4 or v6 address
126     name = "ITR RLOC Address"
127     fields_desc = [
128         ShortField("afi", int(1)),
129         LISP_AddressField("afi", "address")
130     ]
131
132     def extract_padding(self, s):
133         return "", s
134
135 """ Map Reply LOCATOR, page 28, paragraph 6.1.4, the LOCATOR appears N times
        dependant on the locator count in the record field """
136 class LISP_Locator_Record(Packet):
137     name = "LISP Locator Records"
138     fields_desc = [
139         ByteField("priority", 0),
140         ByteField("weight", 0),
141         ByteField("multicast_priority", 0),
142         ByteField("multicast_weight", 0),
143         BitField("reserved", 0, 13),
144         FlagsField("locator_flags", 0, 3, ["local_locator", "probe", "route"]),
145         ShortField("locator_afi", int(1)),
```

```
146             LISP_AddressField("locator_afi", "address")
147         ]
148 
149         # delimits the packet, so that the remaining records are not contained as '
            raw' payloads
150         def extract_padding(self, s):
151             return "", s
152 
153 """ Map Reply RECORD, page 28, paragraph 6.1.4, the RECORD appears N times
        dependant on Record Count """
154 class LISP_MapRecord(Packet):
155         name = "LISP Map-Reply Record"
156         fields_desc = [
157             BitField("record_ttl", 0, 32),
158             FieldLenField("locator_count", None, "locators", "B", count_of="locators
                ", adjust=lambda pkt,x:x/12),
159             ByteField("eid_prefix_length", 0),
160             BitEnumField("action", 0, 3, _LISP_MAP_REPLY_ACTIONS),
161             BitField("authoritative", 0, 1),
162             BitField("reserved", 0, 16),
163             BitField("map_version_number", 0, 12),
164             ShortField("record_afi", int(1)),
165             LISP_AddressField("record_afi", "record_address"),
166             PacketListField("locators", None, LISP_Locator_Record, count_from=lambda
                pkt: pkt.locator_count + 1)
167         ]
168 
169         # delimits the packet, so that the remaining records are not contained as '
            raw' payloads
170         def extract_padding(self, s):
171             return "", s
172 
173 """ Map Request RECORD, page 25, paragraph 6.1.2, the 'REC', appears N times
        depending on record count """
174 class LISP_MapRequestRecord(Packet):
175         name= "LISP Map-Request Record"
176         fields_desc = [
177             ByteField("reserved", 0),
178                     # eid mask length
179             ByteField("eid_mask_len", 24),
180                     # eid prefix afi
181             ShortField("request_afi", int(1)),
182                     # eid prefix information + afi
183             LISP_AddressField("request_afi", "request_address")
184         ]
185 
186         def extract_padding(self, s):
187             return "", s
188 
189 """PACKET TYPES (REPLY, REQUEST, NOTIFY OR REGISTER)"""
190 
191 class LISP_MapRequest(Packet):
192         """ map request part used after the first 16 bits have been read by the
            LISP_Type class"""
193         name = "LISP Map-Request packet"
194         fields_desc = [
195             BitField("ptype", 0, 4),
196             FlagsField("request_flags", None, 6, ["authoritative", "
                map_reply_included", "probe", "smr", "pitr", "smr_invoked"]),
197             BitField("p1", 0, 6),
```

```
198             # right now we steal 3 extra bits from the reserved fields that are
                    prior to the itr_rloc_records
199             # the lambda you see below, checks for the length of the '
                    itr_rloc_records' by going from the largest possible IP + AFI
                    record (IPv6 = 18 bytes) to the smallest one (IPv4 = 6 bytes). The
                     entry in the middle (%12) takes care of dual IPv4 records.
200             # TODO - get the 2 record limitation worked out.
201         FieldLenField("itr_rloc_count", None, "itr_rloc_records", "B", count_of="
                itr_rloc_records", adjust=lambda pkt,x:((not (x%18) and (x/18-1)) or
                ((not (x%12) and (x/12-1)) or ((not x%6) and (x/6-1))))),
202         FieldLenField("request_count", None, "request_records", "B", count_of="
                request_records", adjust=lambda pkt,x:x/8),
203         XLongField("nonce", random.randint(nonce_min, nonce_max)),
204             # below, the source address of the request is listed, this occurs
                    once per packet
205         ShortField("request_afi", int(1)),
206             # the LISP IP address field is conditional, because it is absent if
                    the AFI is set to 0
207         ConditionalField(LISP_AddressField("request_afi", "address"), lambda pkt:
                pkt.request_afi != 0),
208         PacketListField("itr_rloc_records", None, LISP_AFI_Address, count_from=
                lambda pkt: pkt.itr_rloc_count + 1),
209         PacketListField("request_records", None, LISP_MapRequestRecord,
                count_from=lambda pkt: pkt.request_count)
210     ]
211
212 class LISP_MapReply(Packet):
213     """ map reply part used after the first 16 bits have been read by the
            LISP_Type class"""
214     name = "LISP Map-Reply packet"
215     fields_desc = [
216         BitField("ptype", 0, 4),
217         FlagsField("reply_flags", None, 3, ["probe", "echo_nonce_alg", "security"
                ]),
218         BitField("p2", 0, 9),
219         BitField("reserved", 0, 8),
220         FieldLenField("map_count", 0, "map_records", "B", count_of="map_records",
                 adjust=lambda pkt,x:x/16 - 1),
221         XLongField("nonce", random.randint(nonce_min, nonce_max)),
222         PacketListField("map_records", 0, LISP_MapRecord, count_from=lambda pkt:
                pkt.map_count + 1)
223     ]
224
225 class LISP_MapRegister(Packet):
226     """ map reply part used after the first 16 bits have been read by the
            LISP_Type class"""
227     name = "LISP Map-Register packet"
228     fields_desc = [
229         BitField("ptype", 0, 4),
230         FlagsField("register_flags", None, 1, ["proxy_map_reply"]),
231         BitField("p3", 0, 18),
232         FlagsField("register_flags", None, 1, ["want-map-notify"]),
233         FieldLenField("register_count", None, "register_records", "B", count_of="
                register_records", adjust=lambda pkt,x:x/16 - 1),
234         XLongField("nonce", random.randint(nonce_min, nonce_max)),
235         ShortField("key_id", 0),
236         ShortField("authentication_length", 0),
237             # authentication length expresses itself in bytes, so no
                    modifications needed here
```

```
238            StrLenField("authentication_data", None, length_from = lambda pkt: pkt.
                   authentication_length),
239            PacketListField("register_records", None, LISP_MapRecord, count_from=
                   lambda pkt:pkt.register_count + 1)
240        ]
241
242    class LISP_MapNotify(Packet):
243        """ map notify part used after the first 16 bits have been read by the
               LISP_Type class"""
244        name = "LISP Map-Notify packet"
245        fields_desc = [
246            BitField("ptype", 0, 4),
247            BitField("reserved", 0, 12),
248            ByteField("reserved_fields", 0),
249            FieldLenField("notify_count", None, "notify_records", "B", count_of="
                   notify_records"),
250            XLongField("nonce", random.randint(nonce_min, nonce_max)),
251            ShortField("key_id", 0),
252            ShortField("authentication_length", 0),
253                # authentication length expresses itself in bytes, so no
                       modifications needed here
254            StrLenField("authentication_data", None, length_from = lambda pkt: pkt.
                   authentication_length),
255            PacketListField("notify_records", None, LISP_MapRecord, count_from=lambda
                    pkt: pkt.notify_count)
256        ]
257
258    class LISP_Encapsulated_Control_Message(Packet):
259        name = "LISP Encapsulated Control Message packet"
260        fields_desc = [
261            BitField("ptype", 0, 4),
262            FlagsField("ecm_flags", None, 1, ["security"]),
263            BitField("p8", 0, 27)
264        ]
265
266        """ Bind LISP into scapy stack
267
268        According to http://www.iana.org/assignments/port-numbers :
269        lisp-data        4341/tcp    LISP Data Packets
270        lisp-data        4341/udp    LISP Data Packets
271        lisp-cons        4342/tcp    LISP-CONS Control
272        lisp-control     4342/udp    LISP Data-Triggered Control """
273
274        # tie LISP into the IP/UDP stack
275    bind_layers( UDP, LISP, dport=4342 )
276    bind_layers( UDP, LISP, sport=4342 )
277    bind_layers( LISP_Encapsulated_Control_Message, LCAF_Type, )
278
279    """ start scapy shell """
280    if __name__ == "__main__":
281        interact(mydict=globals(), mybanner="lisp debug")
```

8.2 LIG

```
1  #!/usr/bin/env python2.6
2  """
3      This file is part of a toolset to manipulate LISP control-plane
4      packets "py-lispnetworking".
5
6      Copyright (C) 2011 Marek Kuczynski <marek@intouch.eu>
7      Copyright (C) 2011 Job Snijders <job@intouch.eu>
8
9      This file is subject to the terms and conditions of the GNU General
10     Public License. See the file COPYING in the main directory of this
11     archive for more details.
12 """
13 # query a mapserver for the RLOC of the given EID space
14 # note that this does _not_ work over NAT
15 # you also need root for the sockets, might fix this in the future
16
17 from lisp import *
18
19         # define the timeout here, just in case no reply is received
20 timeout = 1
21         # define the interface to send out on, FIXME
22 interface = 'eth0'
23         # define the use message
24 use = "USAGE: ./pyLIG.py <mapserver> <eid-query>"
25 afi_error = "ERROR: the AFI (IPv4 / IPv6) you're trying to use is not available.
       check ifconfig"
26
27         # class to resolve FQDN addresses using Google DNS. it sends out a DNS
              packet and returns the reply IP. right now, qtype is set to A (= IPv4)
              , gonna fix this for AAAA (= IPv6) soon.
28 def resolveFQDN(host):
29     dns=DNS(rd=1,qd=DNSQR(qname=host,qtype='A'))
30     response=sr1(IP(dst='8.8.8.8')/UDP()/dns)
31     if response.haslayer(DNS):
32         ans = response.getlayer(DNS).an
33         return ans.rdata
34
35         # check if an input is a FQDN or IP record, since they both appear as
              strings
36 def checkFQDN(string):
37     if re.match("[A-Za-z]", string):
38         return resolveFQDN(string)
39     else:
40         return string
41
42 def sendLIG(map_server, query):
43         # an alternative approach to retrieve the hosts ip is by using socket.
              gethostbyname(socket.gethostname()), but this unfortunatly often
              returns only a loopback on LINUX systems. the method below appears to
              work
44     source_ipv4 = netifaces.ifaddresses(interface)[socket.AF_INET][0]['addr']
45     source_ipv6 = netifaces.ifaddresses(interface)[socket.AF_INET6][0]['addr']
46         # warn the user there is no IPv6 connectivity
47     if not source_ipv6:
48         print "NOTIFY: you have no IPv6 connectivity"
49
50         # generate a random source port, this seems to be an OK range
51     sport1 = random.randint(60000, 65000)
```

```
52      sport2 = random.randint(60000, 65000)
53      map_server_afi = int(0)
54      query_afi = int(0)
55      source_afi = int(0)
56      source = int(0)
57          # let scapy open a socket already, so that the first packet will be
                captured too
58      server_socket = L2ListenSocket()
59
60          # check if the map server specified is IPv4 or IPv6, this is important
                for python field lengths
61          # could implement it in a method, but we use it just once anyway
62      c = map_server.count(':')
63      if c == 0:
64                  map_server_afi = 4
65      elif c > 0:
66                  map_server_afi = 6
67
68          # the same for the query, check for IPv4 or IPv6
69      d = query.count(':')
70      if d == 0:
71                  query_afi = 1
72                  eid_mask_len = 32
73      elif d > 0:
74                  query_afi = 2
75                  eid_mask_len = 128
76
77          # determine whether to use an IPv4 or IPv6 header and set some values.
                initiate the 'packet' too here.
78      if source_ipv6 and map_server_afi == 6:
79                  source_afi = 2
80                  source = source_ipv6
81                  packet = IPv6(dst=map_server)
82                  socket_afi = socket.AF_INET6
83      elif source_ipv4 and map_server_afi == 4:
84                  source_afi = 1
85                  source = source_ipv4
86                  packet = IP(dst=map_server)
87                  socket_afi = socket.AF_INET
88      else:
89                  print afi_error
90
91          # build the packet with the information gathered. flags are set to smr +
                probe (equals 12)
92      packet /= UDP(sport=sport1,dport=4342)/LISP_Encapsulated_Control_Message(
            ptype=8)
93
94          # check whether to use IPv4 or IPv6 for the second IP header
95      if query_afi == 1 and source_ipv4:
96          packet /= IP(src=source_ipv4, dst=query, ttl=255)
97      elif query_afi == 2 and source_ipv6:
98          packet /= IPv6(src=source_ipv6, dst=query)
99
100         # build the packet, uncomment the debig command below to see its
                structure
101     packet /= UDP(sport=sport2,dport=4342)/LISP_MapRequest(request_afi=0, address
            =source, ptype=1, itr_rloc_records=[LISP_AFI_Address(address=source,afi=
            source_afi)],request_records=[LISP_MapRequestRecord(request_address=query,
             request_afi=query_afi, eid_mask_len=eid_mask_len)])
102
```

```
103         # debug
104     # packet.show2()
105 
106         # send packet over layer 3
107     send(packet)
108 
109         # start capturing on the source port. initiate count value f
110     f = 0
111         # use the earlier opened socket to capture traffic on UDP port 4342
112     capture = sniff(filter='udp and port 4342', timeout=timeout, opened_socket=
            server_socket)
113     for i in range(len(capture)):
114         try:
115                 if capture[i].nonce == packet.nonce and capture[i].ptype == 2:
116                         capture[i].show2()
117                         f = 1
118                         break
119         except AttributeError:
120                 pass
121 
122         # print message if no reply received
123     if f == 0:
124         print "ERROR: no reply received, are you sure you're not behind NAT and
                that your connectivity is OK?"
125 
126         # close the socket, else it'll stay alive for a while
127     server_socket.close()
128 
129         # check command line arguments
130 if len(sys.argv) == 3:
131         map_server = sys.argv[1]
132         query = sys.argv[2]
133         map_server = checkFQDN(map_server)
134         query = checkFQDN(query)
135         sendLIG(map_server, query)
136                 # if no arguments specified, drop to CLI
137 elif len(sys.argv) == 1:
138         print use
139         if __name__ == "__main__":
140                 interact(mydict=globals())
141 else:
142                 # if a weird amount of arguments is given, display usage
                        information
143         print use
```