UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA
FACULTATEA DE ELECTRONICĂ, TELECOMUNICAŢII ŞI TEHNOLOGIA INFORMAŢIEI
Specializarea: Tehnologii şi Sisteme de Telecomunicaţii

Preliminary Implementation of a LISP MAP Server

Lucrare de licenţă

PREŞEDINTE COMISIE,
Prof. Dr. ing. Virgil Dobrotă

DECAN
Prof. Dr. ing. Marina Ţopa

CONDUCĂTOR
Prof. Dr. ing. Virgil Dobrotă

ABSOLVENT
Cosmin Cobârzan

2010

Contents

Work planning **5**

1 State of the Art **6**
- 1.1 Solution space . . . 6
  - 1.1.1 GSE . . . 6
  - 1.1.2 ENCAPS . . . 6
  - 1.1.3 Proposed solutions - LISP . . . 7
- 1.2 Current implementations . . . 8

2 Theoretical Fundamentals **9**
- 2.1 Background . . . 9
  - 2.1.1 Scalability of the Routing System . . . 9
  - 2.1.2 Growth of the Default Free Zone . . . 9
  - 2.1.3 Overloading the IP Semantics . . . 10
- 2.2 LISP . . . 11
  - 2.2.1 Initial proposal . . . 11
  - 2.2.2 Locator/Identifier Separation principle – Endpoints and Endpoint Names 13
  - 2.2.3 Locator/Identifier Separation Protocol (LISP) . . . 15
- 2.3 LISP Map Server . . . 18
  - 2.3.1 Overview . . . 18
  - 2.3.2 Interactions with other LISP Components . . . 19
  - 2.3.3 Map–Server Processing . . . 20
  - 2.3.4 Map–Resolver Processing . . . 20

3 Implementation **22**
- 3.1 Objectives . . . 22
- 3.2 Map Server architecture . . . 22
  - 3.2.1 Initial proposal . . . 22
  - 3.2.2 Reviewed proposal . . . 23
  - 3.2.3 Map Server components . . . 24
  - 3.2.4 Details of the modules internals . . . 26
- 3.3 Taking a look under the hood – function definitions . . . 37
- 3.4 Giving life to the architecture – choosing a programming language . . . 41
- 3.5 Preparing the testing environment . . . 41

4 Experimental Results **43**
- 4.1 Modules Tests . . . 43

Conclusions **50**
Achievements 50
Further work 50

List of Figures

1 Work planning 5
2 Gantt chart 5

2.1 LISP Topology Example 15

3.1 Map Server initial architecture *[courteousy of UPC Barcelona]* 22
3.2 Map Server architecture 23
3.3 Queue processing – Pushing and popping a message 25
3.4 Packet Receiver processing 26
3.5 Mutex queue 27
3.6 Message Dispatcher 28
3.7 Encapsulated Map Request – field checks 29
3.8 Map Reply – field checks 30
3.9 Map register – field checks 31
3.10 Map Request – field checks 32

4.1 Start_netlink.sh command to configure the Map Server 43
4.2 Map Server start 44
4.3 Searching for a Encapsulated Map Request 46
4.4 Add a Map Reply to the Cache database 47
4.5 Finding a mapping for a Encapsulated Map Request message 47
4.6 Searching for a Map Request 48
4.7 Add a Map Register to the DB database 49
4.8 Finding a mapping for a Map Request message 49

Acknowledgments

First of all I want to thank my parents, my brother and my friend Melinda Fancsal for their constant support and understanding. I would not have succeeded without them!

I also want to thank my supervisors, Prof. Virgin Dobrotã, Ph.D. from The Technical University of Cluj - Napoca and Prof. Jordi Domingo - Pascual, Ph.D. and Albert Cabellos - Aparicio, Ph.D. from Universitat Politécnica de Catalunya, Barcelona, Spain for their help, support and guidance during my research work.

Special thanks are due to my collaborators Loránd Jakab and Florin Coraş for their help, observation and advices.

The work presented in this final paper has been done during an Erasmus scholarship between February-May 2010 at UPC Barcelona, Spain.

Cosmin Cobârzan

Work planning

The LISP Map Server was developed in several stages as the following work planning shows:

ID		Task Name	Duration	Start	Finish
1		LISP documentation	4 days	Tue 2/23/10	Fri 2/26/10
2		Queue implementation	9 days	Tue 3/2/10	Fri 3/12/10
3		Message Flowcharts	5 days	Mon 3/15/10	Fri 3/19/10
4		Messages for errors and function prototyping	5 days	Mon 3/22/10	Fri 3/26/10
5		Databases implementation	15 days	Mon 3/29/10	Fri 4/16/10
6		Database testing	5 days	Mon 4/19/10	Fri 4/23/10
7		Modules linking (Message Dispatcher, Map Server and Map Rezolver with queue and DB)	7 days	Mon 4/26/10	Tue 5/4/10
8		Map Server testing	13 days	Wed 5/5/10	Fri 5/21/10

Figure 1: Work planning

The Gantt chart bellow shows how the work progressed during the three months Erasmus scholarship at UPC Barcelona, Spain:

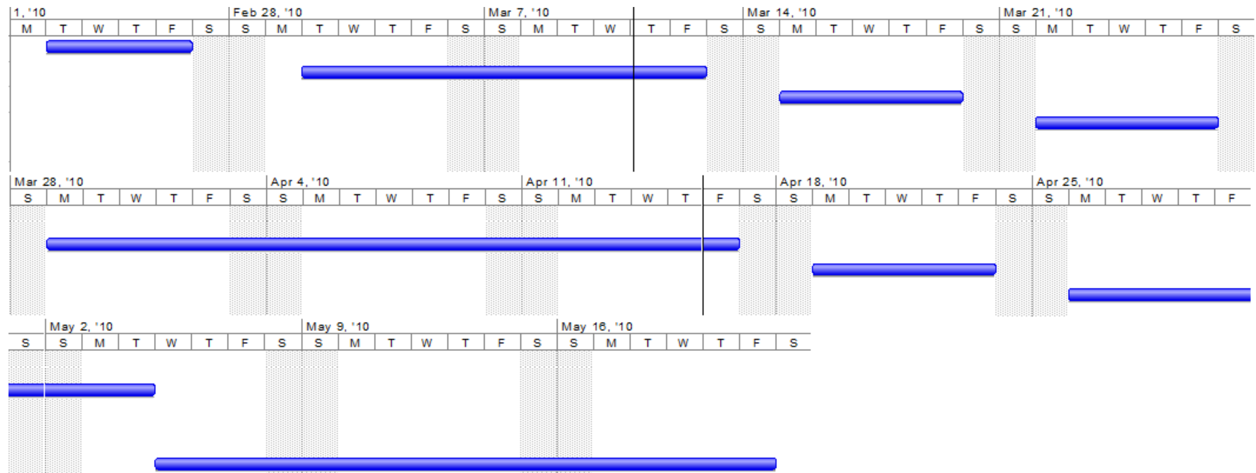

Figure 2: Gantt chart

Chapter 1

State of the Art

In October 2006 the Internet Advisory Board (IAB) held a Routing and Addressing Workshop in Amsterdam, Netherlands with the goal of developing a shared understanding of the problems that the large backbone operators are facing regarding the scalability of the Internet routing system. The outcome of the meeting, their findings and suggestions, have been summed up in a Request For Comments (RFC 4984) [MZF07] and forms the input to the Internet Engineering Task Force (IETF) community which aims to identify the next steps towards effective solutions.

While many aspects of a routing and addressing system were discussed, the participants deemed two as most important and subsequently formulated two problem statements:

1. The scalability of the Routing System
2. The Overloading of the IP Address Semantics

An in depth analysis of the problems (including the two above) affecting the routing system was also provided.

1.1 Solution space

1.1.1 GSE

GSE (Global, Site, and End–system address elements) changes the IPv6 semantics to bear both an identifier and a locator. From the 16 bytes IPv6 address, the first **n** bytes are called Routing Goop (RG) and are used as a locator by the routing system, the last **8** bytes specify the interface of the end–system and the remaining bytes in the middle $\mathbf{(16 - n - 8)}$ specify the site local topology. Address rewriting is used both on outgoing and incoming packets to maintain source provider address aggregation and hide the site's RG from all the internal routers and hosts. GSE proposed to use DNS for providing the mapping service, but it did not offer an effective means for locator failure recovery. Furthermore GSE required changes to host stack in order to use the lower 8 bytes from the IPv6 address.

1.1.2 ENCAPS

This conceptual approach to solve the scalability issues is built on the locator/identifier scheme and provides tunnels between the border routers rather than GSE solution of locator rewriting. The conceived system has a border router that reads the destination and the

source address of a local site originated packet, and does a mapping for these addresses (identifiers) to locators by means of DNS. Finally it encapsulates the initial IP datagram with a new IP header containing the discovered locators and sends it to its intended destination. Between them, the border routers may compute routes using exterior gateway protocols like BGP or interior gateway protocols like OSPF or IS–IS [Hin96]

The ENCAPS solution provides a scalable multihoming solution with no changes to the host stacks; the encapsulation is made at the border routers and it does not imply any changes to the practices of either applications or backbone operators. This proposal was not complete, but it triggered ideas for new sets of solutions.

1.1.3 Proposed solutions - LISP

Amsterdams IAB Routing and Addressing Workshop put the foundation for The Locator/Identifier Separation Protocol which aims to solve the scalability of the routing systems. The workshop concluded that a trade–off between cost/benefits must be made for any solution to the routing scalability. Given the high potential for gains of the indirection approach (locator/identifier split), the map–and–encap idea was chosen as a promising solution space.

The LISP draft [FFML09] describes a simple, incremental, network–based protocol to implement separation of Internet addresses into Endpoint Identifiers (EIDs) and Routing Locators (RLOCs). This mechanism of map–and–encap requires no changes to host stacks and no major changes to existing database infrastructures. Still, it will be necessary to deploy new network equipment namely Ingress Tunnel Routers (ITRs) and Egress Tunnel Routers (ETRs) and a new Mapping System deployment. The ITR finds a destination locator for a local site outgoing packet using the Mapping System, constructs and prepends the LISP header to the original IP datagram and sends the resulting packet to the ETR. If the received packet has as destination address the locator of the ETR, the Egress Tunnel Router strips down the LISP header, and forwards it to the destination EID within its local site. The Mapping System to be used by LISP to find EID–to–RLOC mappings is LISP+ALT. It provides a forwarding path from an Ingress Tunnel Router which requires an EID–to–RLOC mapping to an Egress Tunnel Router which holds that mapping. Using this approach opens the possibility to implement traffic engineering and multihoming in a scalable way.

A number of scaling benefits would be realized by separating current IP addresses into separate spaces for Endpoint Identifiers (EIDs) and Routing Locators (RLOCs):

a) Reduction of table size in the "default free zone" (DFZ). Routing Locators, being a separate numbering space, can be assigned topologically by providers, greatly improving aggregation and reducing the number of globally–visible, routable prefixes.

b) More cost–effective multihoming for sites that connect to different service providers where they can control their own policies for packet flow into the site without using extra routing table resources of core routers.

c) Easing of renumbering burden when clients change providers. Because host Endpoint Identifiers are numbered from a separate, non–provider–assigned and non–topologically–bound space, they do not need to be renumbered when a client site changes its attachment points to the network.

d) Traffic engineering capabilities that can be performed by network elements and do not depend on injecting additional state into the routing system.

e) Existing mobility mechanisms will be able to work in a locator/identifier separation scenario. It will be possible for a host (or a collection of hosts) to move to a different point in the network topology either retaining its home–based address or acquiring new addresses based on the new network location. A new network location could be a physically different point in the network topology or the same physical point of the topology with a different provider.

A key entity in the LISP architecture is the Map Server. It learns EID–to–RLOC mappings and publishes them in the mapping database. This paper will focus on the implementation of the first open source LISP Map Server.

1.2 Current implementations

Development of the LISP software is pushed by Cisco Systems who already has a closed source LISP Map Server implementation running on Cisco IOS and NX–OS platforms.

Chapter 2

Theoretical Fundamentals

2.1 Background

2.1.1 Scalability of the Routing System

The shape of the growth curve of the Default Free Zone Routing Information Base [DFZ RIB] has been the topic of much research and discussion since the early days of the Internet [Hus03]. There have been various hypotheses regarding the sources of this growth. The Routing and Addressing Workshop held in Amsterdam by the IAB identified the following factors as the main driving forces behind the rapid growth of the DFZ RIB:

a) Multihoming,

b) Traffic engineering,

c) Non–aggregatable address allocations (a big portion of which is inherited from historical allocations),

d) Business events, such as mergers and acquisitions.

All of the above factors can lead to prefix de–aggregation and/or the injection of unaggregatable prefixes into the Default Free Zone RIB. Prefix de–aggregation leads to an uncontrolled DFZ Routing Information Base growth because, absent some non–topologically based routing technology (for example, Routing On Flat Labels [CCK+06] or any name–independent compact routing algorithm, e.g., [ANG+04]), topological aggregation is the only known practical approach to control the growth of the Default Free Zone Routing Information Base.

2.1.2 Growth of the Default Free Zone

Todays Internet routing systems have a scalability problem that is growing day by day. The most pressing problem faced today is the growing size of the DFZ routing table (frequently referred to as the Routing Information Base, or RIB) and the consequences that derive from it . Those consequences include the sizes of the Default Free Zone RIB and FIB (the Forwarding Information Base), the cost of recomputing the FIB, concerns about the BGP convergence times in the presence of growing RIB and FIB sizes, and the costs and power (and hence heat dissipation) properties of the hardware needed to route traffic in the core of the Internet.

The growth of the DFZ RIB over the last few years has been greater than linear. Memory

requirements for Default Free Zone RIB and FIB have increased. The growth driven by prefix de–aggregation also exposes the core of the network to the dynamic nature of the edges, i.e., the de–aggregation leads to an increased number of BGP update messages injected into the DFZ (frequently referred to as "UPDATE churn"). Consequently, additional processing is required to maintain state for the longer prefixes and to update the FIB. The size of the Routing Information Base is bounded by the give address space size and the number of reachable hosts but the amount of protocol activity to distribute dynamic topological changes is not. As a result of this fact, the amount of BGP update churn that the network can experience is unlimited. Current measurements suggest that the update churn is heavy–tailed [HA06], meaning that a relatively small number of Autonomous Systems (ASs) or prefixes are responsible for a disproportionately large fraction of the update churn that we observe today. This situation may be exacerbated by the current Regional Internet Registry (RIR) policy in which end sites are allocated Provider–Independent (PI) addresses. These addresses are not topologically aggregatable, and as such, bring the churn problem described above into the core routing system. [MZF07]

2.1.3 Overloading the IP Semantics

One of the fundamental assumptions underlying the scalability of routing systems was eloquently stated by Yakov Rekhter (and is sometimes referred to as Rekhters Law), namely:

> *"Addressing can follow topology or topology can follow addressing. Choose one."*

The same idea was expressed by Mike O'Dell's design of an alternate address architecture for IPv6 [O'D97], where the address structure was designed specifically to enable aggressive topological aggregation to scale the routing system. Noel Chiappa has also written extensively on this topic (see, e.g., [Chi99]).

It is however difficult to create (and maintain) the kind of congruence envisioned by Rekhters Law in todays Internet. The difficulty arises from the overloading of addressing with the semantics of both "who" (endpoint identifier, as used by transport layer) and "where" (locators for the routing system); some might also add that IP addresses are also overloaded with "how" [Hus06]. However, it is felt, that this kind of overloading had deep implications for the scalability of the global routing system.

The so–called "locator/identifier overload" of the IP address semantics is one of the causes of the routing scalability problem that we see today. Thus, a "split" seems necessary to scale the routing system.

2.2 LISP

LISP is a simple, incremental, network-based protocol to implement the separation of IP addresses into Endpoint Identifiers (EIDs) and Routing Locators(RLOCs). The host stacks are not changed but the infrastructure of the existing databases can suffer minor changes. [FFML09]

2.2.1 Initial proposal

2.2.1.1 Address Rewriting

The idea was originally proposed by Dave Clark and later by Mike O'Dell in his 8+8/GSE [O'D97] specification. The aim was to take advantage of the 16–byte IPv6 address and use the lower 8 bytes as End System Designator (ESD), the top 6 bytes as a routing locator (Routing Goop of RG) and the ones left in between, 2 bytes, as Site Topology Partition (STP).

The model draws a strong distinction between the transit structure of the Internet and a site that contains a rich but private topology which may not leak into the global routing system. Also, the site is defined as the fundamental unit that attaches to the global routing system, being in fact a leaf even if it is multihomed. Interesting to note the above mentioned structure of the address brings also the desired distinction between the identity of end system and its point of attachment to the Public Topology.

The above mentioned insulation provides a site with the flexibility of re-homing and multihoming. And this is because a site's interior hosts and routers are unaware of the Routing Goop and thus if a change in the RG, due to administrative decisions, does occur, the "inner components" do not need to know it. Moreover, this brings forth the possibility of topological aggregation, with the goal of routing scalability, by partitioning the Internet into what O'Dell named "a set of tree-shape regions anchored by 'Large Structures"'. The Routing Goop, in an address, would have the purpose of specifying the path from the root to the tree, or otherwise known as the "Large Structure", to any point in the topology. In the terminal case – that point would be a site. Thus, these "Large Structures" have the goal of aggregating the topology by relational subdivision of the space under them and delegation. It also follows that in the case when no information about the next hop is known, the "Large Structure" could be used as forwarding agents. This significantly limits the minimal-sufficient information required for a router, when performing forwarding. It is also envisioned that additional route information kept is the result of path optimizations from cut-through.

For further details related to the structure within IPv6 addresses and also possible solutions to re–homing and multihoming, one should read the RFC draft [O'D97].

Given the age of the idea (suggested in 1997) and the lack of solutions, at this time, for some of the components proposal, it is only natural that today we see limitations with O'Dell design. In the following some of those limitations will be further detailed. Good overviews of this system and its limitations can be found in [Zha06] and [Mey07].

The main *"flaw"* in the GSE design seems to be the use of DNS when learning about destination hosts. Even if one assumes that root servers will stay relatively stable it must also accept that the ones *"under"* will not. And if considering that a site is multihomed,

one must answer which and how many of its Routing Goops should be returned as reply to a DNS server lookup for that site? Furthermore, given its role, a DNS server must know at all time the RG of the site it currently resides, so that a proper answer can be given for any DNS query. This comes in contradiction with the above stated *"insulation principle"*. Moreover, the support of 2–faced DNS server is brought up, that is, the server must know if the query is remote or from a local site in order to know if the RG should or should not be included in the reply message.

Another issue is handling border link failures. It is possible for the source site to be aware of the status of its border links and choose one of the links which is up. It is impossible however to determine if one of the border routers at the destination has lost connectivity to the site. Thus as a solution, GSE proposed that the border routers for a site be manually configured to form a group and when one looses connectivity to the client site, it should forward the packets to the others still connected. This issue is specific to all solutions that propose a split between the edge and transit domains not only to GSE.

It was anticipated above that the "Large Structure" anchorage of the tree shape regions, with little or none interconnection between lower regions was a wrong assumption. And indeed it is, as the trend in the last decade has shown that the interconnections below the top level are the norm rather than controlled circumventions thus the proposed RG structure needs revisiting.[Zha06]

Also, thought it has scalable support for multihoming, GSE lacks support for traffic engineering. It may be possible to solve this goal too, but the existing proposal does not solve this problem. The same is true for IP tunneling across RG boundaries and given the extensive use of Virtual Private Networks (VPN) a thorough examination of tunneling operations is needed in the GSE context.

2.2.1.2 Map and Encap

The idea, originally proposed by Robert Hinden in his ENCAP scheme [Hin96], speaks about splitting the current single address space in two separate ones: the Endpoint Identifier (EID) space, which covers the hosts, and Routing Locator (RLOC) space which is used for transit between domains. It is believed that, through the decoupling of the EID non–topological aggregatable space from the RLOC (provider owned) space, aggregation can be achieved in the core routing domain and eventually routing scalability.

When a source wants to send a packet to a destination outside its domain, the packet traverses the domain infrastructure to a border router. The packet has as source address the Endpoint Identifier of the initiator and as destination address the EID of the destination host, which could be obtained by means of DNS. The border router needs to map the destination EID to a RLOC which is an entry point in the destination network. The proper means to do this would be a mapping system. Obviously this phase is called the map phase of the map–and–encap. After the mapping is performed, the border router encapsulates the destination packet, by prepending a new outer header, with the destination address the Routing Locator of the destination network and the source its own RLOC. Then, it injects the packet in the routing domain. This is the encap phase of the map–and–encap model.

Thus, the system, in its simplest explanations, consists in the insertion of a new header obtained by means of a mapping process. When the encapsulated packet arrives at the des-

tination border router, the router decapsulates the packet and sends what was the original, source host build, packet to the destination EID in the domain. It is observed that both in source and destination domains, the Endpoint Identifiers need to be routable (in the local scope).

Besides providing a architecture with the goal of obtaining routing scalability, map–and–encap has other advantages as the lack of host stack and no core routing infrastructure changes. This scheme works both with IPv4 and IPv6 addresses and retains the original source address, a feature useful in various filtering scenarios. [Mey08].

Map–and–encap has also downsides, as in address rewriting: the problem of handling border link failure persists and the overhead implied by the encapsulation is a rising controversy.

Both of the above presented approaches, which have inspired new solutions in our current context, seem to give, arguably with a general approach, ways to solve the routing scalability problem. But in doing so, given their reliance on the addition of a new level of indirection to the addressing architecture, practically their intrinsic need to translate EIDs into RLOCs, have created a new problem. The solution to this new problem is a mapping systems. It can be seen that now, the scalability problem has shifted from the routing system to the mapping system and the success or failure of future solution will heavily depend upon the careful design of the mapping system architecture.

2.2.2 Locator/Identifier Separation principle – Endpoints and Endpoint Names

In [Chi99], Noel Chiappa introduces the concept of Endpoint to solve what he believes to be an overloading of the name functionality in the context of networking. There are few fundamental objects in networking, also few names and among these, good examples are: host names and addresses. He has reached the conclusion that the reason for this situation is the following: firstly, as he tracked down the papers to the beginning of networking, he discovered that authors did not make any distinction between the concepts of an object and their associated names. This caused widespread confusions between the properties of the object and those of the name. The second reason would be the lack of a rich set of fundamental objects. When dealing with new problems, difficulties aroused in finding/acknowledging the status of separate entities from previously existing but masked objects.

In the days of ARPANET the address had a straightforward meaning and was build by concatenation the number of the host with a port number. In time, the size of the Internet grew and the tight association between the functions of the term address and this sole instantiation of the name had become confusing. Chiappa and Saltzer [Sal93] explained this by the small number of well defined concepts at hand.

In order to clarify the confusion it was proposed that the term *"address"* should be redefined and used with one of the current implied uses and be limited to this particular one. Also, the fundamental object "endpoint" was defined. Actually, it is acknowledged as previously present in the network but unnamed.

For a better understanding of the technical aspects, the author also defines *"bindings"* and *"namespaces"*. *Binding* refers to the association between a name and an object, but can refer

also to map form one "kind" of name to another. Furthermore, instances of the same object may have more than one name and those names may be from the same class of names, or otherwise *"namespace"*. Depending of the namespace we may have many–to–one bindings or alternatively many one–to–one bindings.

The relation between the structure of the names in a namespace and the function is also observed. This may be explained by the need for ease of use and a good example would be the structure implied by the IP addresses in order to ease routing. Names may also have multiples ways of representation (e.g. Decimal printed notations and the bit representation for IPs).

The namespace together with the possibility to represent names in multiple ways imply the existence of contexts which may help make the distinction between attachment of names to objects and mappings from one namespace to another.

The above theoretical introduction can now be used to analyze the TCP/IP architecture which did not make a clear distinction between the objects and the names of those objects. In fact, it can be observed that namespaces are as old as the NCP protocol architecture, namely the addresses and the host–names.

In TCP/IP architecture the only "names" are IP addresses and can have multiple uses by the routers. When forwarding the user data packets *"the name"* pointed to a place in the network, to the destination of the packets. This is also known as the "network attachment point" or the "interface" used in transport layer identifiers for both of the end–to–end communicating hosts.

The overloading of this single name and "field" with all these functionalities has created problems and a solution would be to split these three functionalities up and have them performed by separate fields.

The other TCP/IP architecture namespace is that of host–names. Host–names are human readable strings and are contained in a hierarchical structure that can be looked up in the distributed and replicated database known as the DNS (Domain Name Server). In fact, the DNS makes a mapping between the human readable characters and one or several IP addresses, which will mediate the TCP *"dialog"* between the hosts, once determined.

As expected, the double function of the IP addresses, that of identifying the interfaces and the hosts, has downsides and an important one is the limitation of host mobility. This is because a TCP connection, as already mentioned earlier, is always identified by one pair IP address–TCP port. It only follows that a change in the IP address, requested by the change of position in the network, will break the TCP connection.

Chiappa tried to solve this problem by proposing a better definition for the term "address", in fact new definitions, or better bounded ones, for all three meanings. He suggested using *"address"* when referring to an interface, *"selector"* when talking about the field used by routers when forwarding packets and introduced a new fundamental object, the *"endpoint"*, to identify and end–to–end communicating host. He was unable, though, to give the endpoint namespace/ namespaces because their uses are multiple, and as stated in the beginning of this section, the forms of names within a namespace are highly dependent on how they will be used. Nevertheless he also gave a rich list of characteristics which he saw useful.

Regarding today's problems, his ideas are taken into consideration to help solve the routing scalability problem by splitting the different functionalities of the term "address: that of *locator* in the core routing system (selector) and *endpoint identifier* (the host interface).

2.2.3 Locator/Identifier Separation Protocol (LISP)

Before taking a closer look at the protocol, we detail some basic LISP terms. The definitions are taken from their source, the Internet Engineering Task Force Draft (IETF) [FFML09]. Moreover, for a better understanding an example of a LISP compliant topology is provided in Figure 2.1.

When a host in a LISP capable domain wants to send a packet, it first finds through DNS lookup the destination Endpoint Identifier. After that, it sets as source address its own EID and as destination address the previously found Endpoint Identifier. If the destination EID is in another domain, the packet traverses the source domain network infrastructure and reaches the Ingress Tunnel Router (ITR) [Mey08].

If the ITR has cached the EID–to–RLOC mapping for the destination Endpoint Identifier, it encapsulates the received packet in a LISP header with its Routing Locator as source address and the mapping provided RLOC as destination address. It then sends the packet over the Internet to the destination Egress Tunnel Router which decapsulates the packet and sends it to the destination computer.

When the Ingress Tunnel Router does not have any knowledge about the destination address, it creates a Map–Request message, injects it in the mapping system and waits for a reply. The mapping system "routes", by implementation specific means, the packet to an authoritative ETR for the searched EID, capable of providing the required EID–to–RLOC mapping. When the Egress Tunnel Router receives the request, it decapsulates the packet and sends through the Internet (not mapping system) a Map–Reply to the source ITR with the required information. Now the Ingress Tunnel Router can proceed to sending the packet to one of the mapping specific ETR as in the above mentioned case.

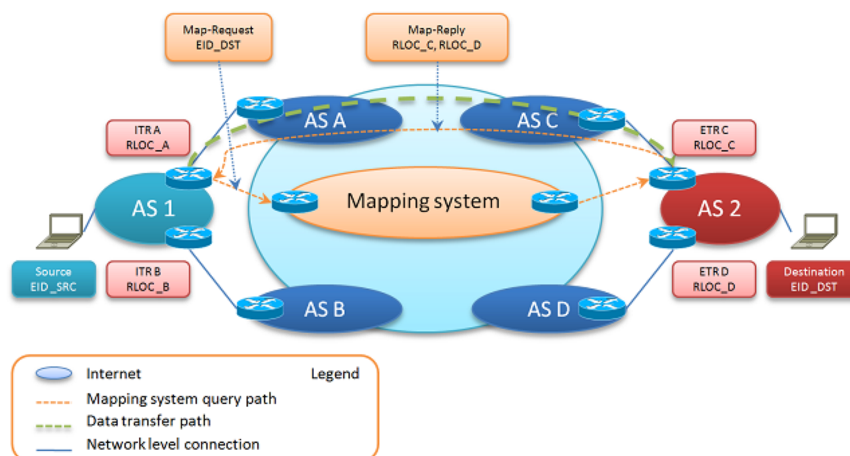

Figure 2.1: LISP Topology Example

The figure represents a generalized model for the steps needed to be performed when intra

domain communication takes place. Consider the source to be the computer connected to AS1, the destination the computer connected to AS2. The source knows about the EID of the destination computer by means of DNS. If a communication is desired, then the source proceeds to creating a packet with its EID (EID_SRC), as source address and EID_DST as the destination address. Because the destinations is in another domain, the packet needs to transverse the AS1 network infrastructure and reach one of the Ingress Tunnel Routers. The figure depicts ITR A as the receiver of the packet. ITR A does not have a mapping for EID_DST and thus LISP–encapsulates the packet, setting as outer header destination address the searched Endpoint Identifier. The source address of ITR A is its Routing Locator. After those fileds have been filled, ITR A injects the packet in the mapping system. When ETR C receives the packet, it decapsulates the request and sends a reply with RLOC_C and RLOC_D to be used as RLOCs, through the Internet, to ITR A. Now, ITR A can send LISP–encapsulated packets, destined for EID_DST, to one of the two RLOCs (priorities may be implied also, not discussed in this example) and they will decapsulate and send them to the destination host.

Ingress Tunnel Router (ITR): a router which accepts an IP packet with a single IP header (more precisely, an IP packet that does not contain a LISP header). The router treats this "inner" IP destination address as an EID and performs an EID–to–RLOC mapping lookup. The router then prepends an "outer" IP header with one of its globally–routable RLOCs in the source address field and the result of the mapping lookup in the destination address field. Note that this destination RLOC may be an intermediate, proxy device that has better knowledge of the EID–to–RLOC mapping closer to the destination EID. In general, 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. Specifically, when a service provider prepends a LISP header for Traffic Engineering purposes, the router that does this is also regarded as an ITR. The outer RLOC the ISP ITR uses can be based on the outer destination address (the originating ITR's supplied RLOC) or the inner destination address (the originating hosts supplied EID).

TE–ITR: is an ITR that is deployed in a service provider network that prepends an additional LISP header for Traffic Engineering purposes.

Egress Tunnel Router (ETR): a router that accepts an IP packet where the destination address in the "outer" IP header is one of its own RLOCs. The router strips the "outer" header and forwards the packet based on the next IP header found. In general, an ETR receives LISP–encapsulated IP packets from the Internet on one side and sends decapsulated IP packets to site end–systems on the other side. ETR functionality does not have to be limited to a router device. A server host can be the endpoint of a LISP tunnel as well.

TE–ETR: is an ETR that is deployed in a service provider network that strips an outer LISP header for Traffic Engineering purposes.

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. Used synonymously with the term "Tunnel Router". For example, "An xTR can be located at the Customer Edge (CE) router", meaning both ITR and ETR functionality is at the CE router.

EID–to–RLOC Cache: a short–lived, on–demand table in an ITR that stores, tracks, and is responsible for timing–out and otherwise validating EID–to–RLOC mappings. This cache

is distinct from the full ”database” of EID–to–RLOC mappings, it is dynamic, local to the ITR(s), and relatively small while the database is distributed, relatively static, and much more global in scope.

EID–to–RLOC Database: a global distributed database that contains all known EID–prefix to RLOC mappings. Each potential ETR typically contains a small piece of the database: the EID–to–RLOC mappings for the EID prefixes ”behind” the router. These map to one of the router's own, globally–visible, IP addresses.

Data Probe: a LISP–encapsulated data packet where the inner header destination address equals the outer header destination address used to trigger a Map–Reply by a decapsulating ETR. In addition, the original packet is decapsulated and delivered to the destination host. A Data Probe is used in some of the mapping database designs to ”probe” or request a Map–Reply from an ETR; in other cases, Map–Requests are used.

For a more detailed description of LISP, the variants, its control and data plane operations the LISP draft [FFML09] is a good reference.

2.3 LISP Map Server

There are two operation modes for a LISP Map–Server: as a Map–Resolver, which accepts Map–Requests from an ITR and *"resolves"* the EID–to–RLOC mapping using the distributed mapping database, and as a Map–Server, which learns authoritative EID–to–RLOC mappings from an ETR and publish them in the database.

Conceptually, LISP Map–Servers share some of the same basic configuration and maintenance properties as Domain Name System (DNS) servers [Moc87] and caching resolvers. With this in mind, this specification borrows familiar terminology (resolver and server) from the DNS specifications.

This paper focuses on the implementation of the first open–source LISP Map–Server that will cover both Map–Resolver and Map–Server operations, as described above. Before going into more detail about the LISP Map Server, a couple of terms need to be defined in order to better understand its interior mechanisms [FF09].

Map–Server: a network infrastructure component which learns EID–to–RLOC mapping entries from an authoritative source (typically, an ETR, though static configuration or another out–of–band mechanism may be used). A Map–Server publishes these mappings in the distributed mapping database.

Map–Resolver: a network infrastructure component which accepts LISP Encapsulated Map–Requests, typically from an ITR, quickly determines whether or not the destination IP address is part of the EID namespace; if it is not, a Negative Map–Reply is immediately returned. Otherwise, the Map–Resolver finds the appropriate EID–to–RLOC mapping by consulting the distributed mapping database system.

Encapsulated Map–Request: a LISP Map–Request with an additional LISP header prepended sent to UDP destination port 4342. The "outer" addresses are globally–routable IP addresses, also known as RLOCs. Used by an ITR when sending to a Map–Resolver and by a Map–Server when sending to an ETR.

Negative Map–Reply: a LISP Map–Reply that contains an empty locator–set returned in response to a Map–Request if the destination EID does not exist in the mapping database. Typically, this means that the "EID" being requested is an IP address connected to a non–LISP site.

Map–Register message: a LISP message sent by an ETR to a Map–Server to register its associated EID prefixes. In addition to the set of EID prefixes to register, the message includes one or more RLOCs to be used by the Map–Server when forwarding Map–Requests (re–formatted as Encapsulated Map–Requests) received through the database mapping system.

2.3.1 Overview

A Map–Server is a device which publishes EID–prefix information on behalf of Egress Tunnel Routers and connects to the LISP distributed mapping database system to help answer LISP Map–Requests seeking the Routing Locators for those EID prefixes. To publish its EID–prefixes, an ETR periodically sends Map–Register messages to the Map–Server. A

Map–Register message contains a list of EID–prefixes plus a set of RLOCs that can be used to reach the Egress Tunnel Router when a Map–Server needs to forward a Map–Request to it.

On the LISP pilot network, which is expected to be a model for deployment of LISP on the Internet, a Map–Server connects to LISP+ALT network and acts as a *"last–hop"* ALT router. Intermediate ALT routers forward Map–Requests to the Map–Server that advertises a particular EID–prefix and the Map–Server forwards them to the owning ETR, which responds with Map–Reply messages.

The LISP Map–Server design also includes the operation of a Map–Resolver, which receives Encapsulated Map–Requests from its client Ingress Tunnel Routers and uses the distributed mapping database system to find the appropriate Egress Tunnel Router to answer those requests. On the pilot network, a Map–Resolver acts as a *"first–hop"* ALT router. It has GRE tunnels configured to other ALT routers and uses BGP to learn paths to ETRs for different prefixes in the LISP+ALT database. The Map–Resolver uses this path information to forward Map–Requests over the ALT to the correct ETRs. A Map–Resolver may operate in a non–caching mode, where it simply decapsulates and forwards the Encapsulated Map–Requests that it receives from ITRs. Alternatively, it may operate in a caching mode, where it saves information about outstanding Map–Requests, originates new Map–Requests to the correct ETR(s), accepts and caches the Map–Replies, and finally forwards the Map–Replies to the original ITRs.

2.3.2 Interactions with other LISP Components

1) **ITR EID–to–RLOC Mapping Resolution**

   An Ingress Tunnel Router is configured with the address of a Map–Resolver. This address is a RLOC which must be routable on the underlying core network; it must not have an Endpoint Identifier which would be resolved through LISP EID–to–RLOC mapping, as that would introduce a circular dependency. When using a Map–Resolver, an ITR does not need other knowledge about any mapping system, does not need to connect to the LISP+ALT infrastructure or implement the BGP and GRE protocols that it uses.

   When it needs an EID–to–RLOC mapping that is not found in its local map–cache, an ITR sends an Encapsulated Map–Request to a configured Map–Resolver. Using the Map–Resolver greatly reduces both the complexity of the Ingress Tunnel Router implementation and the costs associated with its operation. In response to an Encapsulated Map–Request, the ITR can expect one of the following:

   (a) A negative LISP Map–Reply if the Map–Resolver can determine that the requested Endpoint Identifier does not exist. The ITR saves EID prefix returned in the Map–Reply in its cache, marking it as non–LISP–capable and knows not to attempt LISP encapsulation for destinations matching it.

   (b) A LISP Map–Reply from the Egress Tunnel Router that owns the EID–to–RLOC mapping or possibly from a Map–Server answering on behalf of the ETR. Note that the stateless nature of non–caching Map–Resolver forwarding means that the Map–Reply may not be from the Map–Resolver to which the Encapsulated Map–Request was sent unless the target Map–Resolver offers caching.

2) **ETR/Map–Server EID Prefix Registration**

An Egress Tunnel Router publishes its EID prefixes on a Map–Server by sending LISP Map–Register messages. A Map–Register message includes authentication data, so prior to sending a Map–Register message, the ETR and Map–Server must be configured with a secret shared–key. In addition, a Map–Server will typically perform additional verification checks, such as matching any EID–prefix listed in a Map–Register message against a list of prefixes for which the ETR is known to be an authoritative source.

Map–Register messages are sent periodically from an ETR to a Map–Server with a suggested interval between messages of one minute. A Map–Server should time–out and remove an Egress Tunnel Routers registration if it has not received a valid Map–Register message within the past three minutes. When first contacting a Map–Server after restart or changes to its EID–to–RLOC database mappings, an ETR may initially send Map–Register messages at an increased frequency, up to one every 20 seconds. This *"quick registration"* period is limited to five minutes in duration.

After an Egress Tunnel Router publishes its EID–to–RLOC mappings through a Map–Server it does not need to participate further in the mapping system. Using LISP+ALT, means that the ETR does not need to implement GRE or BGP, which greatly simplifies its configuration and reduces its cost of operation.

2.3.3 Map–Server Processing

The operation of a Map–Server, once it has EID–prefixes registered by its client ETRs, is quite simple. In response to a Map–Request (received over the LISP+ALT network), the Map–Server verifies that the destination Endpoint Identifier matches an EID–prefix for which it has one or more registered ETRs, then re–encapsulates and forwards the now Encapsulated Map–Request to a matching Egress Tunnel Router. It does not otherwise alter the Map–Request so any Map–Reply sent by the ETR is returned to the Routing Locator in the Map–Request, not to the Map–Server. Unless also acting as a Map–Resolver, a Map–Server should never receive Map–Replies; any such messages should be discarded without response, perhaps accompanied by logging of a diagnostic message if the rate of Map–Replies is suggestive of malicious traffic.

2.3.4 Map–Resolver Processing

In response to an Encapsulated Map–Request, a Map–Resolver decapsulates the message then checks its local database of mapping entries (statically configured, cached, or learned from associated ETRs). If it finds a matching entry, it returns a non–authoritative LISP Map–Reply with the known mapping.

If the Map–Resolver does not have the mapping entry and if it can determine that the requested IP address does not match an EID–prefix in the mapping database, it immediately returns a negative LISP Map–Reply, one which contains an EID prefix and an empty locator–set. To minimize the number of negative cache entries needed by an Ingress Tunnel Router, the Map–Resolver should return the least–specific prefix which both matches the original query and does not match any EID–prefix known to exist in the LISP–capable infrastructure.

If the Map–Resolver does not have sufficient information to know whether the EID exists, it needs to forward the Map–Request in the LISP+APT mapping database which will provide information about the Endpoint Identifier being requested. This is done in one of two ways:

a) A non–caching Map–Resolver simply forwards the unencapsulated Map–Request, with the original ITR RLOC as the source, on to the distributed mapping database. The Map–Resolver is connected to the ALT network and sends the Map–Request to the next ALT hop learned from its ALT BGP neighbors. The Map–Resolver does not send any response to the ITR; since the source Routing Locator is that of the ITR, the ETR or Map–Server which receives the Map–Request over the ALT and responds will do so directly to the ITR.

b) A caching Map–Resolver queues information from the Encapsulated Map–Request, including the ITR RLOC and the original nonce. It then modifies the Map–Request to use its own Routing Locator, generates a *"local nonce"* (which is also saved in the request queue entry), and forwards the Map–Request as above. When the Map–Resolver receives a Map–Reply, it looks in its request queue to match the reply nonce to a *"local nonce"* entry then de–queues the entry and uses the saved original nonce and ITR RLOC to re–write those fields in the Map–Reply before sending to the ITR. The request queue entry is also deleted and the mapping entries from the Map–Reply are saved in the Map–Resolver's cache.

Chapter 3

Implementation

3.1 Objectives

The objective of the author is to implement the first open-source caching Map Server. Until now the only implementation of a LISP Map-Server has been a close one provided by Cisco for its routers using NX OS, IOS and IOS–XE. The resulted Map–Server will be thoroughly tested in a real scenario using LISP message format and information gathered from the Cisco routers that have it already implemented.

3.2 Map Server architecture

3.2.1 Initial proposal

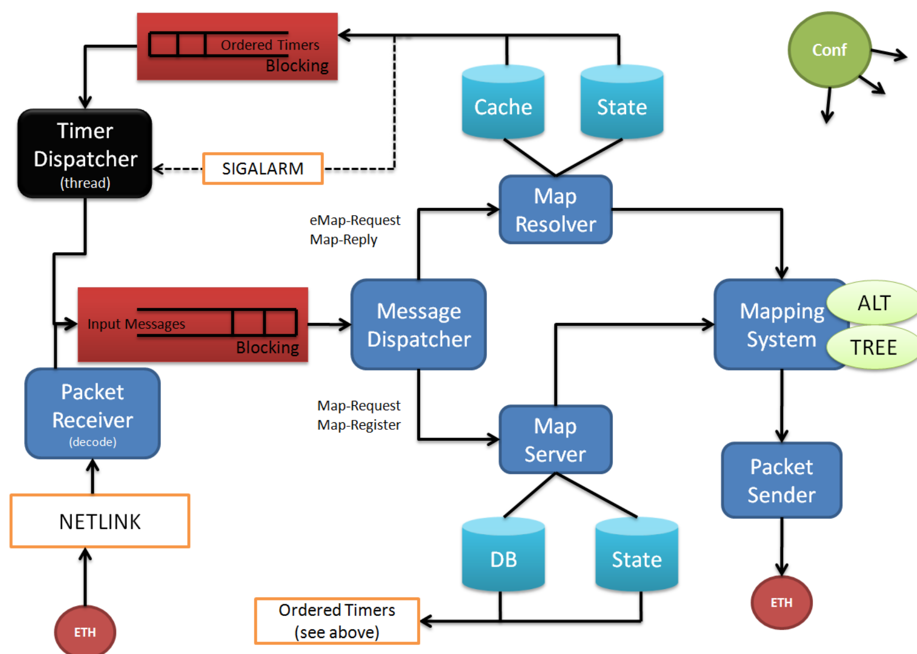

Figure 3.1: Map Server initial architecture *[courteousy of UPC Barcelona]*

In the initial proposal, the Map Server Modules were not so clear defined. Two ordered blocking queues (timers) were though for the Cache module and the DB module. The queues were supposed to contain messages that would have signaled the expiration of a mapping stored in the Cache or DB. With a complex mechanism of queue access (mutex treads), the messages from the ordered queue would go in the queue from the Packet Receiver. The Map Server had also a State Module which would have stored the expiration of the mappings in the DB. By doing this it was thought that the flow of information in the whole Map Server Architecture would be optimum.

After many discussions and research, a viable solution to implement this complex architecture was not found. Many threads had to be handled, making it almost impossible to debug and further develop the Map Server. The main queue, the one in which the Packet Receiver module pushed LISP messages, would contain also other types of messages from the Cache and DB modules. A serious concern arose about the efficiency of the model. It was thought that having mixed information in the queue would overload the Message Dispatcher, slowing the whole Map Server. Special care had to be taken when non–LISP messages would get to the Message Dispatcher. In this situatio nadditional processings needed to be done before any effect wa seen in the Cache or DB databases.

3.2.2 Reviewed proposal

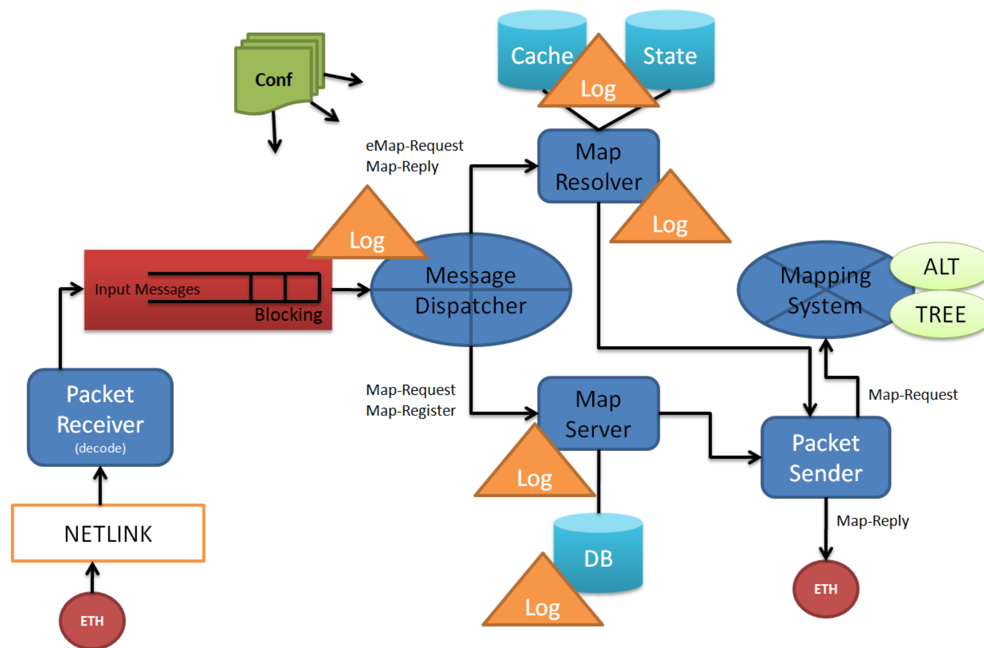

Figure 3.2: Map Server architecture

In the above figure the new Map Server architecture shows a similar approach as in the initial proposal, but the modules have less to worry about when dealing with LISP messages than before. The ordered queues have disappeared and the Time Dispatcher had no use in this reviewed proposal. After many discussions and reviewing of the IETF drafts, the problem of deleting the expired mapping from Cache and DB had a more elegant solution than before, not to say that is also less complex – the expired mapping is deleted when a Map Request or an Encapsulated Map Request gets to the Map Server and a search for the

requested mapping is performed.

A log module has appeared, taking care of all the error messages that would be generated by the operation of the Map Server. New functionalities have been assigned to the Config module, but loading the parameter it contains can be done only at the initialization of the Map Server.

Using this structure, further development and functionality integration are much easier than before.

3.2.3 Map Server components

Because of the modular approach, the Map Server had to be broken down into many independent modules. The communication between them can be uni or bi–directional, depending on the task that they perform. In what follows the general message flow through the Map Server is presented.

The Packet Receiver captures from the network LISP packets and inserts them into an FIFO message queue. It should be noted that the Packet Receiver is not included in the scope of this paper and will not be detailed, as it has been developed at an earlier date, and it is used only to have access at the network layer. The module also broadcasts a condition for the message dispatcher thread when a packet is pushed in the queue. This condition broadcast is made in order to trigger the processing of the current message in the Message Dispatcher module. The Packet Receiver, being the first module to be executed when the Map Server is run, initializes the Config module. All the data from the Config file is stored in a hashtable, making accessing the content of the file more easily and at a speed much greater than it would have when repeated access to the hard disk is made to read data from the Config file. It is known that accessing the hard disk is one of the most time consuming operations and as much as possible it was avoided throughout the whole Map Server architecture.

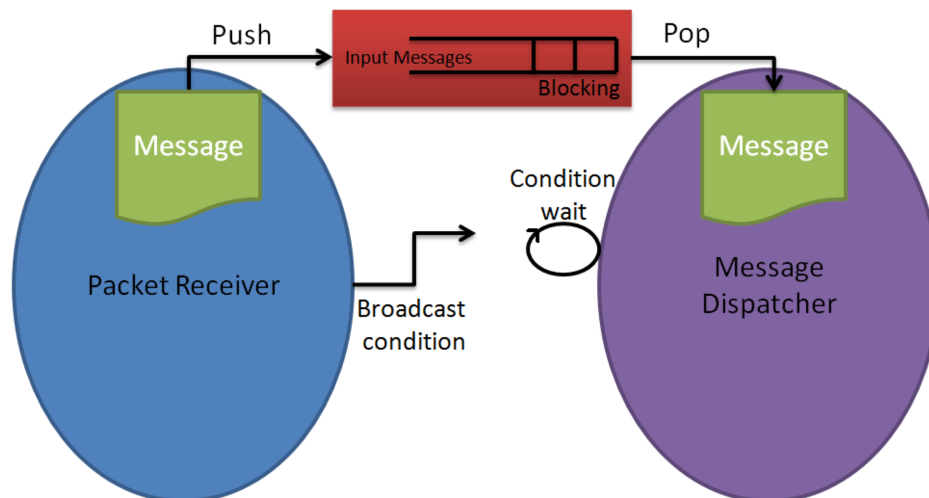

Figure 3.3: Queue processing – Pushing and popping a message

The queue has calls from both Packet Receiver, which pushes – or inserts – messages in it, and the Message Dispatcher, which pops – or extracts – messages from the queue. Implementing the queue required a blocking mechanism, which means that only one thread can access the queue at a given moment. This ensures that the messages are not altered by any other thread. This means that the message pushed in the queue by the Packet Receiver is taken out of the queue by the Message Dispatcher in its original state. The importance of the blocking mechanism used is critical for the application, as all other processing in the following modules depend on the integrity of the message. The Message Dispatcher pops out of the queue a message as soon as the previous message is fully processed.

The Message Dispatcher waits for a new message in a loop. When the Packet Receiver module pushes a message in the queue it also broadcasts a condition that signals the Message Dispatcher to begin processing the message from the queue. The Message Dispatcher extracts the message from the queue and directs the extracted packet towards the Map Resolver or the Map Server depending on the type of the message. Map–Replys and Encapsulated Map–Requests are handled by the Map Resolver. The Map Server receives Map–Requests and Map–Registers.

The Map Server and Map Resolver have fairly the same internal structure, but they operate with different data.

The Map Server module has only a database (DB), organized as a hashtable, in which it stores the mappings received through Map Register messages, from the Egress Tunnel Routers it serves. Map Requests that come to it are resolved in one of the two ways:

a) If the asked Routing Locator corresponds to an ETR which the Map Server serves, the corresponding mapping is sent back to the asking ITR by the Packet Sender;

b) If the RLOC is not one of the ETR served by the Map Server, the request is dropped.

The Map Resolver implemented by the author is a caching one so it needs a Cache database and a State database, both implemented using hashtables. The Encapsulated Map Request

that comes to the Map Resolver is stripped down and the asked mapping is searched in Cache. If the mapping is found, a Map Reply message containing the asked mapping is sent to the originating Ingress Tunnel Router. If the mapping is not found, the Map Resolver stores the nonce from the original Encapsulated Map Request and the ITR RLOC, and then sends a Map Request with its own nonce in the mapping system asking for a Routing Locator that corresponds to the RLOC requested in the Encapsulated Map Request. When a Map Reply message comes to the Map Resolver, the mapping that it carries is first searched in the State module to see if it corresponds to a stored nonce. If it does correspond, the mapping is sent back to the Ingress Tunnel Router that asked for is and it's stored in the Cache; if it does not correspond to an earlier received Encapsulated Map Request, the received mapping is only stored in the Cache.

Separate modules that do not have a definition in the LISP Map Server Draft are the Config (Configure) module and the Log module. The Config module takes care of the administrative part of the implemented Map Server, as it can set variables that are checked during the operation of the Map Server (if the Map Resolver is caching or not, if the Map Server also works as a Map Resolver, etc). The Log module is used to gather all error messages that can be generated by the usage of the Map Server. It can display the information on the screen or store it into a file for analyzing.

The Packet sender and the interaction with the Mapping System do not enter in the scope of the author's paper and will not be discussed.

3.2.4 Details of the modules internals

In what follows, the processing's that each message sufferea and the module that performes it are presented.

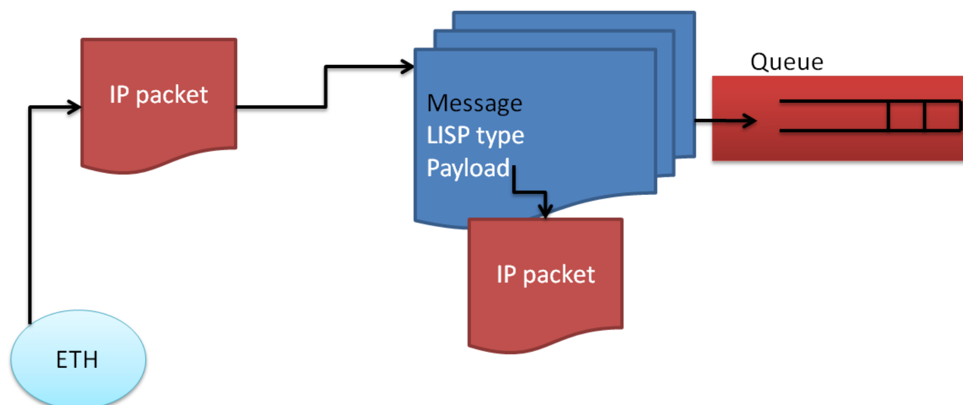

Figure 3.4: Packet Receiver processing

In the Packet Receiver the message is prepared to be inserted in the queue. The LISP message type is situated right after the IP and UDP header of the received IP packet. The headers have to be stripped down so that the LISP type can be stored in the type field of the message structure. The payload field is where the whole message is stored for further processing.

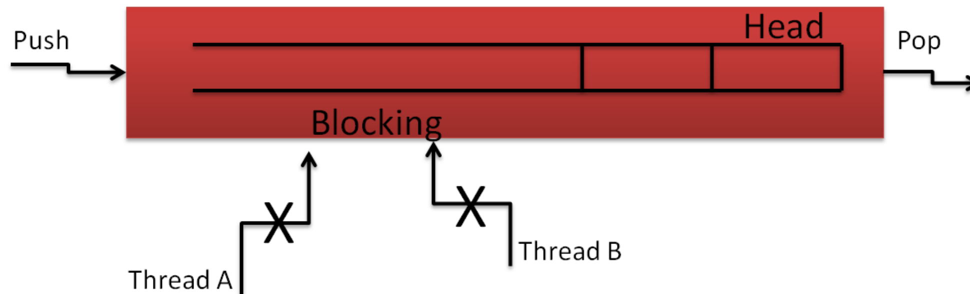

Figure 3.5: Mutex queue

As said before, the queue works on the FIFO principle (First in, First out), it's a simple linked list and it implementes a mutex blocking mechanism. After research it was concluded that a blocking mechanism must be used, as it offers protection of data integrity. At a certain point in time, only one process can access the queue either to insert an element (or "push") or to extract an element (or "pop"). The queue can be accessed only by its first element (known as the "head") for any of the two operations described earlier. The mutex queue is needed as it can't be known when a new LISP message will arrive at the Map Server; it is possible that the Map Server is processing an earlier message and it would naturally put the new message in the queue until it can be process too. By doing so, none of the incoming messages are lost by the Map Server.

Let's take a closer look at the next element in the Map Server architecture. From the queue, a message next stop it's in the Message Dispatcher. Here it is again analyzed and a decision it's taken depending on its type: if the message type is Encapsulated Map Request or Map Reply the message will be directed to the Map Resolver; if the type of the message falls in the Map Request or Map Register categories, the message will be directed on another route towards the Map Server. It must be taken into account that at this point in the processing phase we still have the entire received message, meaning that it has attached the IP and UDP header along with all LISP data. The next two modules, the Map Resolver and the Map Server, will alter the message by removing the IP and UDP header and splitting down the LISP data in records and locators.

The Map Resolver processes Map Replys and Encapsulated Map Requests. For both types of message the IP and UDP header are stripped down and the UDP checksum is compared against 0 to see if the LISP data is valid or not. From this point on, the operations made are different, based on message type:

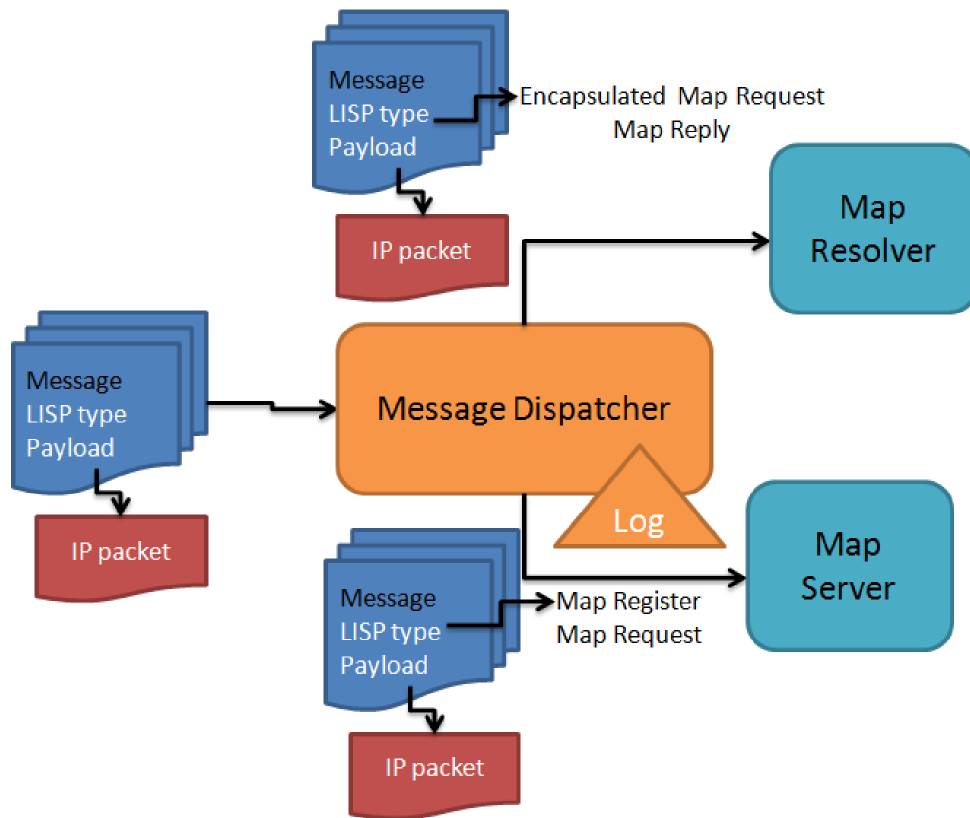

Figure 3.6: Message Dispatcher

a) For an Encapsulated Map Request the next value that is checked is the destination UDP port which must be 4342 as defined in the LISP draft [FFML09]. Further, the message has 32 bits allocated as LISP data from which it can be check if the type corresponds to an Encapsulated Map Request. The family of the message is checked. At this point in the development of the LISP Map Server only IPv4 messages can be processed. Next, the message contains an internal IP and UDP header. As before when dealing with headers, UDP destination port and UDP checksum are checked to see if they comply with the specification – UDP port 4342 and UDP checksum different from 0. After the stripping of the internal IP and UDP header, LISP data is available. Record count is taken into consideration and for each of the records contained in the message a family check is done and a search for a mapping in the cache module. If the search does not give any result, the config module is interrogated to see if the caching state of the Map Resolver is or is not defined. If the Map Resolver is a caching one, it will send a Map Request with its own local generated nonce to the Mapping System asking for a mapping that maches the request from the Encapsulated Request Message. Before doing that, the Map Resolver has to store the original nonce and the ITR RLOC in a database (build as a hashtable), along with the local generated nonce. If the caching state is not defined, the Map Resolver will simply forward the un–encapsulated Map Request to the mapping database.

b) For a Map Reply message, the next value checked is the source UDP port which must be 4342 as defined in the LISP draft [FFML09]. The Map Reply has after the stripping of IP and UDP headers only LISP data. The type is checked again to be sure that the

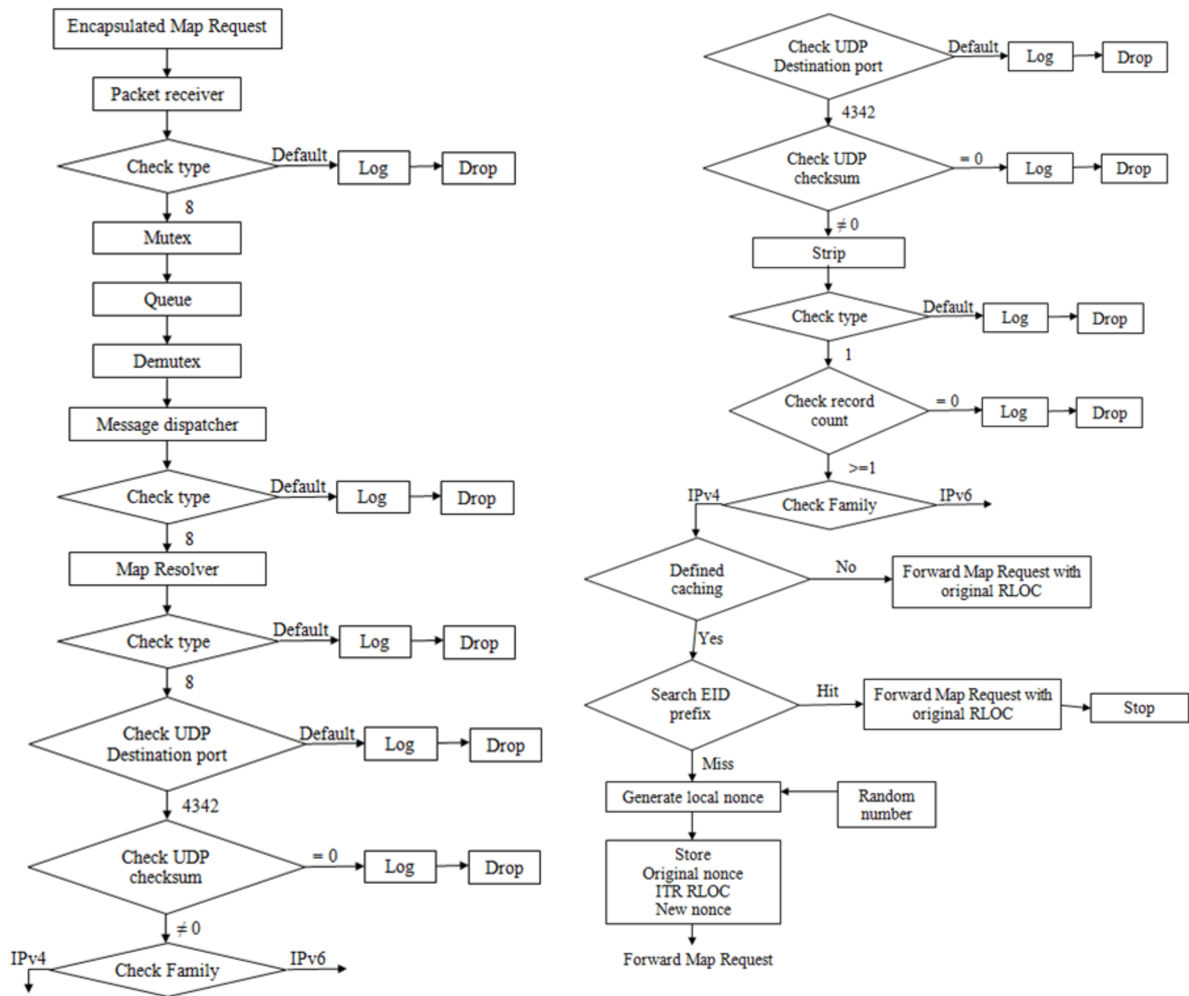

Figure 3.7: Encapsulated Map Request – field checks

correct type of message is processed. Because the Map Resolver can be a caching one, and a former Encapsulated Map Request may have been received but the mapping was not available at that time in the Map Resolver, the state database has to be checked to see if the incoming Map Reply has come as a response to a Map Request sent by the Map Resolver to solve the Encapsulated Map Request. If the Map Reply has come as a response to a Map Request sent by the Map Resolver, after the further processing that any Map Replys has to pass, the locator data has to be sent to the original ITR that has requested the mapping through the Encapsulated Map Request. After checking the State database for a match, regardless of the answer, the record TTL field is checked to see if the received mapping is still valid (record TTL not 0). If the mapping is valid, all the records and locators that it contains are stored in the Cache module.

The Map Server processes Map Registers and Map Requests. As in the previous module, UDP destination port has to be 4342 and UDP checksum must not be 0 in order to have an uncorrupted LISP message. Further processing is specific for each message as follows:

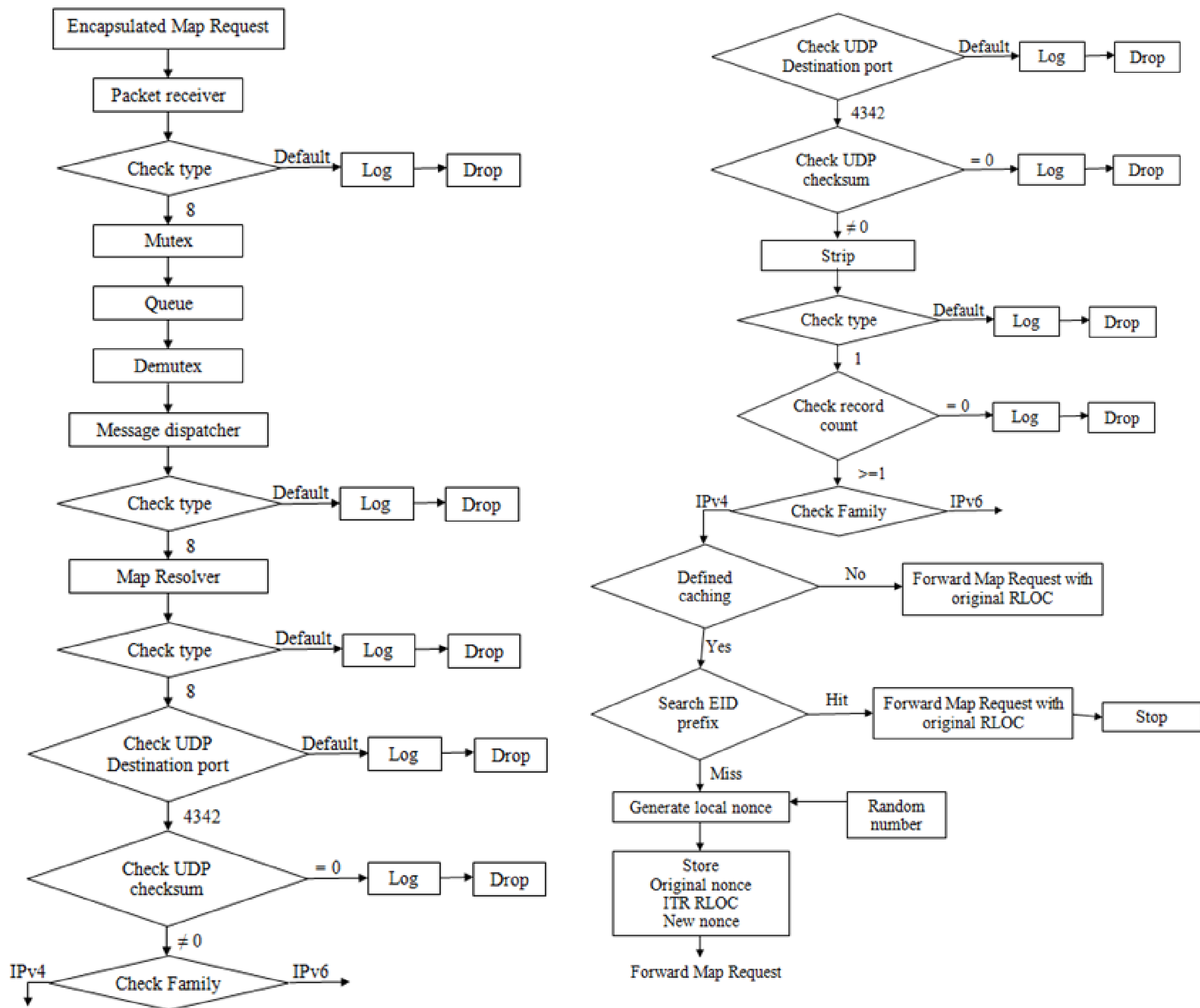

Figure 3.8: Map Reply – field checks

a) Map Registers messages, after their type has been confirmed (type = 3), are searched in the Map Server database to see if an earlier Map Register has not been received with the same mapping data. If the search is not successful, it means that mapping data from the Map Register is unknown for the Map Server and so it's stored in the DB. The record along with its locator is added in the DB.

b) Map Requests messages have their type checked and after that, if the check is passed, a search is performed in the DB to see if a mapping for the requested EID–prefix is found for each record that it may contain. Depending on the outcome of the search, if it's a successful one, the mapping for the EID–prefix is sent back to the asking ITR and if the search fails the packet is dropped.

Last, but not least, the modules that use in their implementation a hashtable (Cache, State, DB) deserve more words on their internal workings. The hashtable concept was chosen to store mapping data because it offers quick search functions and (theoretically) unique hash codes for keys and values. A hashtable is a data structure that uses a hash function to efficiently map certain identifiers or keys to associated values. The hash function is used to transform the key into the index (the hash) of an array element (the slot or bucket) where the corresponding value is to be sought [Wik]. The key in this implementation is a structure that contains the EID AFI, EID prefix and EID mask length from a Map Reply/Map

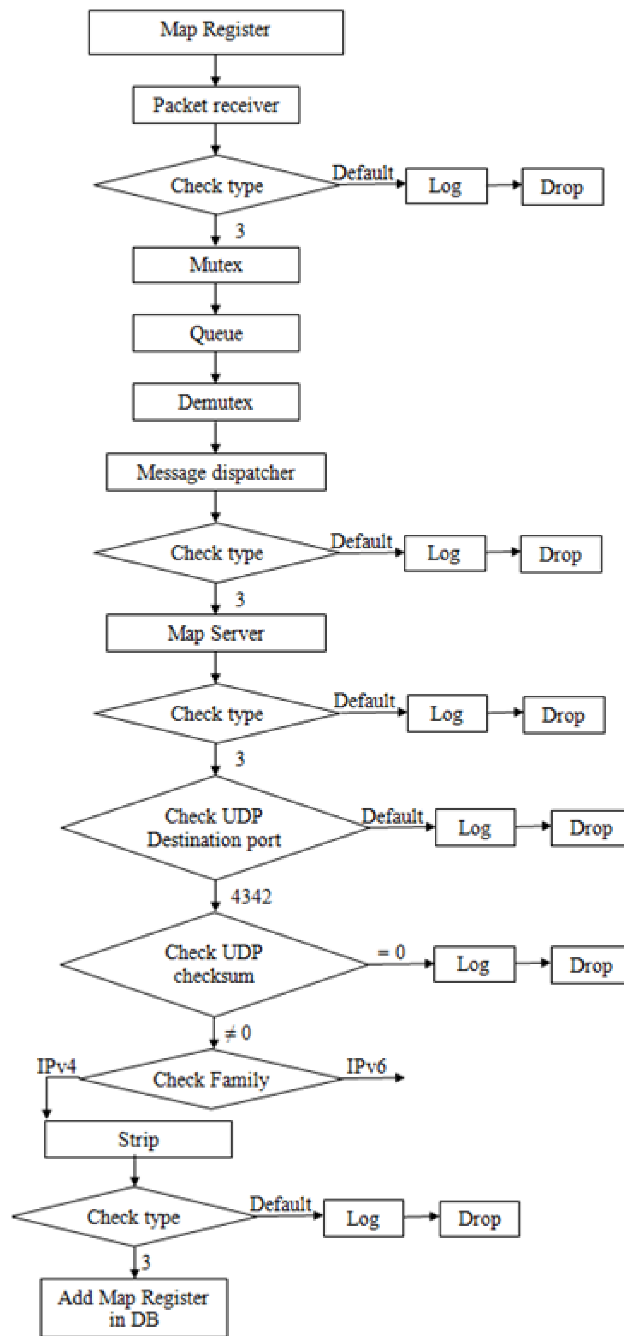

Figure 3.9: Map register – field checks

Request Record and the value is a structure that contains the Map Reply Record and Map Reply Locator, a timestamp, a length field and a pointer to the next element used when the record has more than one Map Reply Locators to link them in between, in order to eliminate hashtable collisions. The internal working of the hashtable (hash functions, add and delete an entry, expanding of the hashtable to accommodate more entries) has been taken from [Cla05]. Hashtables rely on hash functions to generate unique codes so that collisions will not appear if data is different. For the Cache, DB and State modules, the same hash function was used:

```
unsigned int hashfromkey(void *ky)
{
```

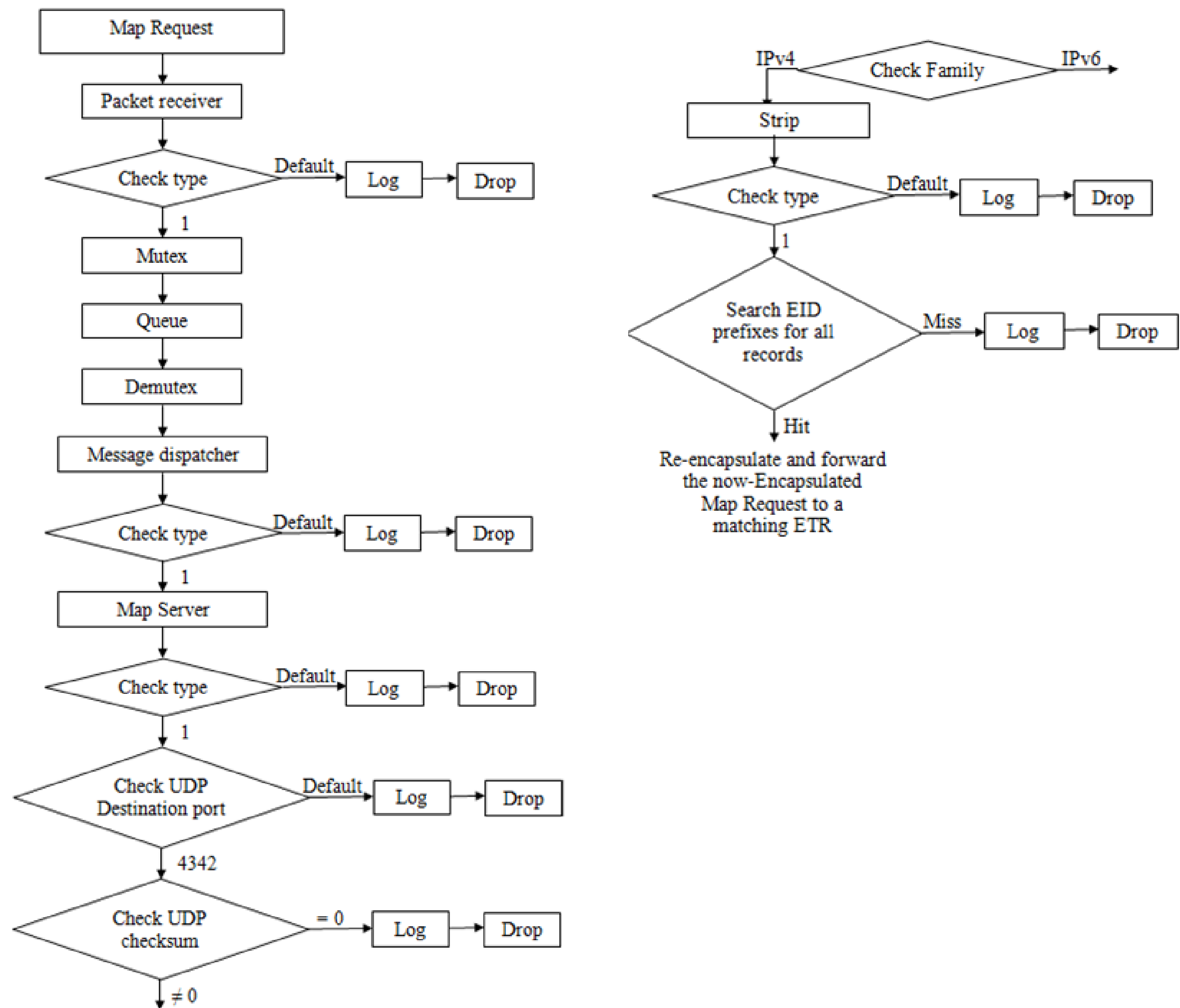

Figure 3.10: Map Request – field checks

```
    struct key *k = (struct key *)ky;
    return ((((k->eid_prefix << 17) | (k->eid_prefix >> 15)) ^ k->eid_afi)
+(k->eid_afi * 17) + (k->eid_mask_len *13 * 29));
}
```

Naturally, collisions appear when the hash function is applied on the same data. The problem has been solved by *linking in between* the values that share the same key. This observation is critical for the working of the Map Server as it gives a reliable way to store different mappings for the same Routing Locator with little effort.

Specific operations on the LISP data have been done separately in the following modules. Cache and DB modules share the same infrastructure and contain the same information. Minor variations have been implemented because the two hashtables are situated near different modules in the Map Server Architecture and have to deal with different types of messages when a search is done; those variations will be explained at the appropiate moment.

In what follows, the functions responsible for the interaction between the LISP messages

and the hashtable that effectively stores the information are presented. Important checks have to be made before storing the information in the hashtable. Memory allocation is an important part in the good working of all the Map Server and so, a great interest in correctly judging the amount needed has been taken. It will be pointed out later, during the presentation of the function where and how memory was allocated.

a) *db_add* – receives as parameters the hashtable where the adding is done and a Map Reply Record. From the Map Reply Record, the EID mask length field is checked to assure that only IPv4 mask lengths are computed further in the function. Any record that has a length greater than 32 will be dropped and a log message will be sent towards the log module. Next, memory for the key is allocated and from the Map Reply Record the values necessary for the fields present in the key structure are computed – information regarding EID AFI, EID mask length. The EID prefix is computed from the record EID and the EID mask length by applying a mask with EID mask length (32 bit value with a EID mask length of "1"s on the most significant bit position) over the record EID (logical AND). The key processing is made by a "private" function – `db_key`, which can be accessed only from this module.

```
rrec=(struct request_rec*)rec;
... ...
/*compute the prefix for an IPv4 address according to the eid_mask_len*/
    for (i=0; i<=rrec->eid_mask_len; i++)
       {
          pref=pref+(uint32_t)pow(2,32-i);
       }
    /*store only the prefix for the key with & between the eid_prefix
and eid_mask_len*/
    k->eid_prefix=(rrec->eid_prefix&pref);
}
```

After the key computation, a search is performed in the hashtable to see if previous records with the same key are found. If the search is successful, the locator is linked to the values already stored for the same key, but if the key is not found, then the pair key–value is stored in the hashtable creating a new entry. For both situations, a time stamp is applied for the value, to be able to see if a mapping stored is still valid or not when a Map Request comes at the Map Resolver. Knowing the time when the new mapping is stored in the hashtable eliminates the need for other complex tools of determining is a mapping is valid or not.

b) *db_search* – receives the hashtable where the search will be made, a *Map Request record* and a Map Reply locator, in which, if the search is successful, a mapping will be present, ready to be sent to the requesting Ingress Tunnel Router. From the received Map Request record the key is computed as before using `db_key`. As in the search function all EID prefixes have an EID mask length equal with 32 (when using IPv4 addresses) it is needed to be performed a longest prefix match to see which entry in

the hashtable is more specific for the requested mapping. The search starts with EID mask length 32 and the original Endpoint Identifier, but if the search is not successful, the EID mask length is decreased with 1 and a new EID is computed by applying the new mask over the original EID.

```
/*Always get a key with /32 prefix from db_key; if not found, we go
lower, with /31, /30 and so on until we find the prefix that matches
the eid*/
    for (i=0;i<32;i++)
      div=div+(uint32_t)pow(2,32-i);
/*The value in div corresponds to the IP mask 255.255.255.255*/
    i=1;
    while (((v = (struct value *)hashtable_search(hash,k))== NULL)
    &&!(k->eid_prefix==0))
     {
        k->eid_prefix=k->eid_prefix&div;
        k->eid_mask_len=k->eid_mask_len-1;
        k->eid_afi=1;
        div=div-(uint32_t)pow(2,i);
        i=i+1;
     }
```

When a value is found, before sending the mapping to the requesting ITR, some more tests have to be done. Firstly, a TTL check has to be done to see if the stored mapping is still valid. The `ttl_update` function, present in the "private" function file db_private, checks if the current time is inside the record TTL present in the value stored in the hashtable. For a Map Request the record TTL field specifies the number of minutes that a mapping is valid. If the mapping is not valid anymore, the entry is deleted from the hashtable and a log message is sent to the log module. On the other hand, if the mapping is still valid, the locators stored with the value have to be extracted. After doing that, the mapping data is available to be sent to the requesting ITR.

c) *db_serv_search* – has the same functionality as `db_search`, but the it receives a *Map Reply record* from which the key is computed. This function is called by the Map Server module when a Map Request comes to the Map Server. Although the two records differ in the number of fields, in the hashtable only the information present in both records is used. This observation had a big impact on the design of the LSIP Map Server, as it reduced the number of functions needed and it made possible function reuse for the hashtables in both Map Resolver and Map Server modules.

d) *db_delete* – removes an entry, received as a Map Reply record parameter, from the hashtable, received as parameter. From the Map Reply record the key for the hashtable is computed using db_key. Before any memory dealocation, a search is made to see

if the computed key has indeed an associated value. If the search is successful, the function can delete a record in two different ways

i) the key and the values associated with the key are deleted form the hashtable and memory is freed

ii) only a value associated with the key is deleted if the record TTL has expired, but the rest of the values remain intact

In the db_private file there exists functions that can export to a file all the values stored in the hashtable, if there is the need to examine the values (mappings) stored in the hashtable.

State module implements a simpler hashtable regarding the values stored. The values associated to keys store only the old nonce and the new nonce, information necessary when a Map Reply is processed by the Map Resolver. The operations done by the state module are not different from the ones made by the other two modules. A state key–value pair can be added in the state hashtable as before. The search algorithm remains unchanged. A great part of the written code has been reused for this module.

In fact, at a closer analysis, the information handeled by the modules that have a hashtable as storing facility is the same. All keys have EID prefix, EID mask length and EID AFI as this information is available regardless the LISP message processed. It was also observed that for the values stored it is much simpler to store a Map Reply Record and a Map Reply Locator in a single value than to make many processing on the LISP message to extract the needed information. The difference in memory from the whole Map Reply Record and Map Reply Locator to only the information needed was insignificant compared to the effort needed to extract the information.

If any of the above tests fails, an error number and message are sent to the log module and further operations are halted and the message is dropped. The log module is initialized when the Map Server starts. There are three possible ways it can work:

a) write the errors in a file

b) write the errors on screen

c) do not write any errors

Regardless of the way the log module is configured, the errors are sent in the same way to the module: error number, error message. The errors that the Map Server can generate vary from: *"out of memory"* errors when trying to allocate memory for a key, value or locator, *"out of range"* errors for EID prefix, *"incorrect values"* error for UDP destination port, UDP header checksum or message types to *"TTL"* errors when a Map Reply expires.

Last, but not least, the Config module was implemented also using a hashtable. In concept is a simple idea – put some configuration options in a file, read the file and then fill the hashtable with a key–value combination. When it came to implement it, some difficulties arouse with the parsing of the file. After many attempts to extract the configuration options it was concluded that a smaller than 3 character configuration key cannot be stored in the hashtable. This is because the internal hash function algorithm that generates a unique hash code did not have sufficient data to generate such a code with less than 3 characters. In the configuration file the options had to be given in a predefined pattern: option="value". Comments can be inserted in the configuration file, using "#" before a line, to explain how and what options are available without interfering with the normal functioning of the Config module or Map Server.

3.3 Taking a look under the hood – function definitions

The following presentation is strictly based on alphabetical order and does not imply any relations between the functions. After this presentation, there will be a diagram with all the functions and their dependencies from one another. The files with _private in their name have functions that are only for internal use of the Map Server and can not be used from other functions, as it will be showed in the diagram.

Configure module

conf_private.h

```
/*Adds an configure element (key and value) in the hashtable*/
int config_add(struct hashtable *hash, struct conf_elem *rec);

/*Searches a configure key in the hashtable and returnes the value asociated with the key*/
int config_search(struct hashtable *hash, struct conf_elem *rec, struct conf_elem *value);
```

config.h

```
/*Takes the key that is searched from par, sends it to config_search, and puts the result in
rez*/
int conf_parameter(struct hashtable *test,char *par, char *rez);

/*Reads the configure file, process its information and sends to config_add an element that
will be added in the hashtable*/
int conf_init(struct hashtable *hash,char *s);

/*Opens the config file and reads its contents.  It sends each readed line to conf_init*/
int config(struct hashtable *conf);
```

Hashtable database

db.h

```
/*Adds an element in the cache or DB hashtable and alocates memory for it*/
int db_add(struct hashtable *hash, struct mrp_rec *rec);
```

```
/*Search for a locator in the Map Resolver hash table*/
int db_search(struct hashtable *hash, struct request_rec *rec, struct reply_locator *l);

/*Search for a locator in the Map Server hash table*/
int db_serv_search(struct hashtable *hash, struct mrp_rec *rec, struct reply_locator *l);

/*Deletes a key from the hashtable, along with all the values that the key contains*/
int db_delete(struct hashtable *hash, struct mrp_rec *rec, int flag);

/*Export to file all the databases valid values*/
int db_export_file(struct hashtable *hash);
```

db_private.h

```
/*Computing of the hash key*/
int db_key(void *rec, struct key *k, int type);

/*Update de ttl associated with a key*/
int ttl_check(struct value *v);

/*Export the valid values from the hashtable*/
int db_export(struct hashtable *h);
```

Log module

log.h

```
/*Choose the mode you want to print errors:  1=file 2=screen 3=none*/
int log_init(int mod);

/*Prints the errors and messages in the mode selected in log_init*/
void log_print(int err, char *msg);

/*Close the log file in case mode=1*/
void log_finish();
```

Map server and Map Dispatcher modules

map_server.h

```
/*Processes Encapsulated Map Requests and Map Replys according to the flow charts presented
before*/
int lm_map_resolver(struct lm_struct *str,struct lm_message *msg);

/* Processes Map Registers and Map Requests according to the flow charts presented before */
int lm_map_server(struct lm_struct *str,struct lm_message *msg);

/*Message dispatcher -- thread that takes out an element from the queue and varying on the
type directs it to lm_map_resolver or lm_map_server*/
void *lm_msgdsp(void *lm_str)
```

Queue module

queue.h

```
/*Insert an element in the back of the queue*/
int push(struct lm_queue *head, struct lm_message msg);

/*Gets the element from the head of the queue*/
int pop(struct lm_queue *head, struct lm_message *msg);
```

queue_private.h

```
/*Insert an int at the end of the list*/
int insert_queue(struct lm_queue *queue, struct lm_message val);

/*Destroys the list -- queue*/
int destroy_queue(struct lm_queue *queue);

/*Delete an element from the head of the list--queue*/
int delete_queue_item(struct lm_queue *queue);
```

State module

state.h

```
/*Adds an element in the state module of the Message Dispatcher and alocates memory for it*/
int state_add(struct hashtable *hash, struct map_request *rec);

/*Search for a nonce in the hash table*/
int state_search(struct hashtable *hash, struct map_reply *rec);

/*Deletes a key from the hashtable, along with all the values that the key contains*/
int state_delete(struct hashtable *hash, struct map_reply *rec);
```

state_private.h

```
/*Computing of the hash key*/
int state_db_key(void *rec, struct state_key *k, int type);

/*Generate local nonce for Map Request*/
uint64_t gen_nonce();
```

3.4 Giving life to the architecture – choosing a programming language

The goal of this paper is to develop a open source implementation of a Map Server. With that in mind, hunting for a programming language had a target. Also considered were the following characteristics:

a) portability

b) easy integration with current platforms

c) low level access for both memory and network

d) low runtime demand on system resources

After those major lines were drawn, the obvious answer was ANSI C on a Linux environment. The pair satisfies all the above consideration. The C programming language was familiar to the author, from both high-school and university courses. The familiarity with the programming language and the experience gathered contributed to a good understanding and implementation of the Map Server Architecture. Also, when problems arose during the implementations of the modules, a large collection of books and articles was available to look for solutions. Choosing the Ubuntu Linux implementations was a purely personal choise. The functionality was not affected in any way, the code being tested also on other distributions (Fedora Core 11), without needing any other change to the system. The only demand is that any Linux system should be up to date with the latest C compiler and libraries.

3.5 Preparing the testing environment

In the late stages of development of the Map Server, the need to test the modules working together has become a driven factor. At the beginning, the Map Server was tested with captured LISP messages from a Cisco router running Cisco IOS installed at UPC Barcelona. The messages were a Map Register message and a Map Reply message. Encapsulated Map Requests were sent with LISP Internet Groper (LIG) – written by D. Farinacci and D. Meyer [FM10]. It remained only one more message to be sent in order to fully test the Map Server for testing the correctness of the processing made. Where to find a Map Request message? This question plus the need of random data in the messages that arrive at the Map Server pushed the author to write its own testing solution.

The starting point was the LISP Internet Groper which was modified. It is written in C and is an open source project, meaning that the author could have easy access to the

code. The original LIG could send Encapsulated Map Requests and wait for an Map Reply message to arrive. Following the Encapsulated Map Request message model, the other messages were written. The Map Request message is in fact an Encapsulated Map Request message without the inner UDP and IP headers. This type of message was easy to configure as the internal structure was already made. The other two messages posed problems concerning the dimensions of the fields and their order in the sent message. With the help of a modified Wireshark program, that can identify correctly all LISP messages [Jak10], the LISP messages were fined tuned. It turned out that the order of the fields needed to be in the reverse order as in the LISP draft [FFML09] to be able to correctly receive and decode the LISP message sent at the Map Server. There were also other typical values for the message fields observed from the captured LISP messages that were formerly used to test the Map Server. For example the priority of the Locators was always 50 or the M priority was 255. Those values were taken into account when the building of the LISP messages was done.

Getting as close as possible to a real scenario needed a random number of records and locators in each packet. Also, the Routing Locators needed to have a wide distribution in the IP numbering space. The burden of complying with all those requirements was not so easy. It may seem strange to say this, but the ordering of the records and locators inside the LISP message was difficult to achieve as it needed a good pointer arithmetic and precise memory allocation.

After all the messages where correctly built and sent, the testing of the Map Server entered the final part. Now, with this tool at hand, the pointer arithmetic needed at the Map Server side could be done. If from LIG, the messages needed to have a reverse order of the fields as normal, in the Map Server everything arrived as it should. It can be concluded that when sending LISP messages special care has to be taken regarding the order of the fields inside the message.

Chapter 4

Experimental Results

4.1 Modules Tests

The experiments were performed trying to simulate an as close-to-real environment as possible. The following tables provide test, descriptions and results details for the major modules in the architecture.

The Map Server needs to be configure before usage with the `.\start_netlink.sh` script as showed in the figure. The start_nelink.sh script makes a connection between the kernel–space and the user–space. The script intercepts UDP packets with LISP control port destination and sends them through the netlink interface to the user–space in a queue. From that queue the Map Server reads the LISP messages.

```
[root@AchileM LISPMapServer]# ./start_netlink.sh
IPv4:
Chain PREROUTING (policy ACCEPT)
target     prot opt source               destination
LOG        udp  --  anywhere             anywhere             udp dpt:lisp-contro
l LOG level debug prefix `lispmapper: '
LOG        udp  --  anywhere             anywhere             udp spt:lisp-contro
l LOG level debug prefix `lispmapper: '
NFQUEUE    udp  --  anywhere             anywhere             udp dpt:lisp-contro
l NFQUEUE num 0
NFQUEUE    udp  --  anywhere             anywhere             udp spt:lisp-contro
l NFQUEUE num 0

Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination

IPv6:
Chain PREROUTING (policy ACCEPT)
target     prot opt source               destination
LOG        udp      anywhere             anywhere             udp dpt:lisp-contro
l LOG level debug prefix `lispmapper: '
LOG        udp      anywhere             anywhere             udp spt:lisp-contro
l LOG level debug prefix `lispmapper: '
NFQUEUE    udp      anywhere             anywhere             udp dpt:lisp-contro
l NFQUEUE num 0
NFQUEUE    udp      anywhere             anywhere             udp spt:lisp-contro
l NFQUEUE num 0

Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination
```

Figure 4.1: Start_netlink.sh command to configure the Map Server

After the .\start_netlink.sh script is runned, the Map server can be started:

```
[root@AchileM LISPMapServer]# ./test
opening library handle
unbinding existing nf_queue handler for AF_INET (if any)
binding nfnetlink_queue as nf_queue handler for AF_INET
unbinding existing nf_queue handler for AF_INET6 (if any)
binding nfnetlink_queue as nf_queue handler for AF_INET6
binding this socket to queue '0'
setting copy_packet mode
```

Figure 4.2: Map Server start

Map Server Tests

Test	Description	Result
Add a Map Register	A Map Register is processed in the Map Server and if all tests have an positive outcome based on the flowchart for Map Register the record is added in the DB	Successful add when data comply with requirements.
Add a Map Register with TTL expired	A Map Register comes with an expired TTL. Processing fails	Log error generated.
Search an existing mapping with valid TTL	A Map Request asks for an existing mapping in the DB.	If mapping is present in the DB the mapping is returned as search result.
Search a existing mapping with invalid TTL	A Map Request asks for an existing mapping in the DB, but the TTL is expired	The entry is deleted from the DB hashtable and a log error is generated.
Search for an non-existent mapping	A Map Request asks for a non existing mapping in the DB.	Log error generated.

Map Resolver Tests

Test	Description	Result
Add a Map Reply with no previous Encapsulated Map Request sent	A simple Map Reply is directly added in the Cache hashtable, after resolving the flowchart requirements	Successful add when data comply with requirements. Log error generated otherwise.
Search a existing mapping with valid TTL	An Encapsulated Map Request asks for an existing mapping in the Cache.	If mapping is presend in the Cache the mapping is returned as search result
Search a existing mapping with invalid TTL	An Encapsulated Map Request asks for an existing mapping in the Cache but the TTL of the entry is expired	Delete the mapping from Cache hashtable and generate log error.
Search for an inexistent mapping	An Encapsulated Map Reques asks for a inexistent mapping in the Cache	Log error generated.

In order to see how much processing power the Map Server needs to run smoothly and what loads is the system needs to bear, some tests were performed on the hashtable by adding and searching random elements. The tests were done on a system with a Core 2 Duo processor (T5470 – 1.6 Ghz) and 3 Gb of RAM installed.

Number of entries added	Number of entries searched	Time needed for add(sec)	Time needed for search(sec)	Average CPU load	Average memory load
5000	10.000	~0	~0	~20%	~0.4%
10.000	100.000	~0	~4	~50%	~17%
100.000	1.000.000	~5	~42	~60%	~45%

Examples of the running of the Map Server are ilustrated below.

a) Searching a Encapsulated Map Request in a empty database or in a database that does not contain the requested mappinng. (Cache database)

```
LISP ENCAPSULATED CONTROL MESSAGE
Eid prefix 168852354
EID afi 1
EID mask len 32
EID prefix 168852354
EID searched: 168852354
EID mask length for EID searched 31
EID searched: 168852352
EID mask length for EID searched 30
EID searched: 168852352
EID mask length for EID searched 29
EID searched: 168852352
EID mask length for EID searched 28
EID searched: 168852352
EID mask length for EID searched 27
EID searched: 168852352
EID mask length for EID searched 26
EID searched: 168852352
EID mask length for EID searched 25
EID searched: 168852224
EID mask length for EID searched 24
EID searched: 168851968
EID mask length for EID searched 23
EID searched: 168851456
EID mask length for EID searched 22
EID searched: 168851456
EID mask length for EID searched 21
EID searched: 168849408
EID mask length for EID searched 20
EID searched: 168845312
EID mask length for EID searched 19
EID searched: 168837120
EID mask length for EID searched 18
EID searched: 168820736
EID mask length for EID searched 17
EID searched: 168820736
EID mask length for EID searched 16
EID searched: 168820736
EID mask length for EID searched 15
EID searched: 168820736
EID mask length for EID searched 14
EID searched: 168820736
EID mask length for EID searched 13
EID searched: 168820736
EID mask length for EID searched 12
EID searched: 167772160
EID mask length for EID searched 11
EID searched: 167772160
EID mask length for EID searched 10
EID searched: 167772160
EID mask length for EID searched 9
EID searched: 167772160
EID mask length for EID searched 8
EID searched: 167772160
EID mask length for EID searched 7
EID searched: 134217728
EID mask length for EID searched 6
EID searched: 134217728
EID mask length for EID searched 5
EID searched: 0
EID mask length for EID searched 4
EID found 0
Mapping not found
```

Figure 4.3: Searching for a Encapsulated Map Request

b) Addind in the Cache database a Map Reply that has a mapping.

```
LISP MAP REPLY
Eid Map Reply: 168852354
Eid Mask lengthEID added 168849408
EID mask length 20
EID AFI 1
Succcess at map reply
EID added 168849408
EID mask length 20
EID AFI 1
Succcess at map reply
```

Figure 4.4: Add a Map Reply to the Cache database

c) Searching again the Encapsulated Map Request from a) and finding the mapping stored before from the Map Reply message.

```
LISP ENCAPSULATED CONTROL MESSAGE
Eid prefix 168852354
EID afi 1
EID mask len 32
EID prefix 168852354
EID searched: 168852354
EID mask length for EID searched 31
EID searched: 168852352
EID mask length for EID searched 30
EID searched: 168852352
EID mask length for EID searched 29
EID searched: 168852352
EID mask length for EID searched 28
EID searched: 168852352
EID mask length for EID searched 27
EID searched: 168852352
EID mask length for EID searched 26
EID searched: 168852352
EID mask length for EID searched 25
EID searched: 168852224
EID mask length for EID searched 24
EID searched: 168851968
EID mask length for EID searched 23
EID searched: 168851456
EID mask length for EID searched 22
EID searched: 168851456
EID mask length for EID searched 21
EID searched: 168849408
EID mask length for EID searched 20
EID found 168849408
Mapping found for the Encapsulated Map Reply, have to send further the locator
```

Figure 4.5: Finding a mapping for a Encapsulated Map Request message

d) Searching a Map Request in a empty database or in a database that does not contain the requested mappinng. (DB database)

```
LISP MAP REQUEST
EID afi 1
EID mask len 32
EID prefix 168852354
EID searched: 168852354
EID mask length for EID searched 31
EID searched: 168852352
EID mask length for EID searched 30
EID searched: 168852352
EID mask length for EID searched 29
EID searched: 168852352
EID mask length for EID searched 28
EID searched: 168852352
EID mask length for EID searched 27
EID searched: 168852352
EID mask length for EID searched 26
EID searched: 168852352
EID mask length for EID searched 25
EID searched: 168852224
EID mask length for EID searched 24
EID searched: 168851968
EID mask length for EID searched 23
EID searched: 168851456
EID mask length for EID searched 22
EID searched: 168851456
EID mask length for EID searched 21
EID searched: 168849408
EID mask length for EID searched 20
EID searched: 168845312
EID mask length for EID searched 19
EID searched: 168837120
EID mask length for EID searched 18
EID searched: 168820736
EID mask length for EID searched 17
EID searched: 168820736
EID mask length for EID searched 16
EID searched: 168820736
EID mask length for EID searched 15
EID searched: 168820736
EID mask length for EID searched 14
EID searched: 168820736
EID mask length for EID searched 13
EID searched: 168820736
EID mask length for EID searched 12
EID searched: 167772160
EID mask length for EID searched 11
EID searched: 167772160
EID mask length for EID searched 10
EID searched: 167772160
EID mask length for EID searched 9
EID searched: 167772160
EID mask length for EID searched 8
EID searched: 167772160
EID mask length for EID searched 7
EID searched: 134217728
EID mask length for EID searched 6
EID searched: 134217728
EID mask length for EID searched 5
EID searched: 0
EID mask length for EID searched 4
EID found 0
Mapping not found!
```

Figure 4.6: Searching for a Map Request

e) Adding in the DB database a Map Register that contains a mapping.

```
LISP MAP REGISTER
Eid prefix 168852354
Search EID: 168852352
Search EID: 168852352
Search EID: 168852352
Search EID: 168852352
Search EID: 168852352
Search EID: 168852352
Search EID: 168852352
Search EID: 168852224
Search EID: 168851968
Search EID: 168851456
Search EID: 168851456
Search EID: 168849408
Search EID: 168845312
Search EID: 168837120
Search EID: 168820736
Search EID: 168820736
Search EID: 168820736
Search EID: 168820736
Search EID: 168820736
Search EID: 168820736
Search EID: 167772160
Search EID: 167772160
Search EID: 167772160
Search EID: 167772160
Search EID: 167772160
Search EID: 134217728
Search EID: 134217728
Search EID: 0
EID found 0
EID added 168852352
EID mask length 29
EID AFI 1
Succcess at map register
```

Figure 4.7: Add a Map Register to the DB database

f) Search again the Map Request from d) and finding the mapping stored before from the Map Register message.

```
LISP MAP REQUEST
EID afi 1
EID mask len 32
EID prefix 168852354
EID searched: 168852354
EID mask length for EID searched 31
EID searched: 168852352
EID mask length for EID searched 30
EID searched: 168852352
EID mask length for EID searched 29
EID found 168852352
Mapping found, need to send the locator further
```

Figure 4.8: Finding a mapping for a Map Request message

Conclusions

Achievements

At the end of the three months period spent at UPC Barcelona, the Map Server architecture was implemented as far as the initial plan was concerned. The modules suffered changes during this period and the implemented versions are described above. All the IEFT draft requirements concerning the LISP Map Server have been fulfilled. Other features, as the log and Config module have been attached to the Map Server offering extended functionality and configurability for the end user.

Further work

Taking a look at the Map Server architecture (fig. 3.2) and considering all the above chapters, one may see that two components miss from this paper - the interaction with the Mapping System and the Packet Sender. Those two modules are to be completed at a later date by one of the members of the UPC Barcelona team that have LISP development under their care. In order for the Map Server to work in a real network and to achieve its role, the Packet Sender and the interaction with the Mapping System need to be implemented.

Another concern that arose after work was finished was the optimization of the implemented architecture. The main optimization that needs to be performed is the memory allocation and management. As for now, two copies of the same LISP message are needed - one at the Packet Receiver and one in the Cache or DB hashtables - for the Map Server to run. In future, only one copy of the message should suffice for the running of the Map Server.

LISP drafts suggest that the new protocol can and will interact with both IPv4 and IPv6. The implementation of the Map Server so far has focused on the IPv4 packets and all the operations done on the received LISP messages are on IPv4 packets only. However it will be easy to make the needed adjustments in the Map Server implementation so that it would process also IPv6 packets. The statement has in mind the modular architecture and the ease with which a module can be replaced without altering the good working of the whole Map Server. The hashtables can store any kind of data, regardless of its length. Modifications

will be made in the structures so that they can accommodate 128 bit addresses.

Bibliography

[ANG+04] Ittai Abraham, Noam Nisan, Cyril Gavoille, Dahlia Malkhi, and Mikkel Thorup. Compact Name-Independent Routing with Minimum Stretch. In *Proceedings of the 16th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 20–24. ACM Press, 2004.

[CCK+06] Matthew Caesar, Tyson Condie, Jayanthkumar Kannan, Karthik Lakshminarayanan, Ion Stoica, and Scott Shenker. ROFL: Routing on Flat Labels. In *SIGCOMM*, 2006.

[Chi99] J. Noel Chiappa. Endpoints and Endpoint Names: A Proposed Enhancement to the Internet Architecture, 1999.

[Cla05] Christopher Clark. Source code for a hash table data structure in C, jan 2005.

[FF09] Vince Fuller and Dino Farinacci. LISP Map Server, oct 2009.

[FFML09] Dino Farinacci, Vince Fuller, David Meyer, and Darrel Lewis. Locator/ID Separation Protocol (LISP), sep 2009.

[FM10] D. Farinacci and D. Meyer. LISP Internet Groper (LIG), feb 2010.

[HA06] Geoff Huston and Grenville Armitage. Projecting Future IPv4 Router Requirements from Trends in Dynamic BGP Behaviour. *ATNAC 2006, Melbourne, Australia*, 2006.

[Hin96] R. Hinden. New Scheme for Internet Routing and Addressing (ENCAPS) for IPNG, jun 1996.

[Hus03] Geoff Huston. Analyzing the Internet's BGP Routing Table. In *The Internet Protocol Journal - Volume 4, Number 1*, 2003. http://www.potaroo.net/papers/ipj/2001-v4-n1-bgp/bgp.pdf.

[Hus06] Geoff Huston. Whither Routing? *ISP Column*, 2006.

[Jak10] Lorànd Jakab. LISP dissector for wireshark (BETA), may 2010.

[Mey07] David Meyer. Update on Routing and Addressing at IETF 69. *IETF Jurnal*, 3(2), 2007.

[Mey08] David Meyer. The locator identifier separation protocol (LISP). *The Internet Protocol Jurnal*, 11(1):23–26, 2008.

[Moc87] P. Mockapetris. Domain Names - Implementation and Specification, nov 1987.

[MZF07] D. Meyer, L. Zhang, and K. Fall. Report from the IAB Workshop on Routing and Addressing, sep 2007.

[O'D97] Mike O'Dell. GSE - An Alternate Addressing Architecture for IPv6, 1997.

[Sal93] J. Saltzer. On the Naming and Binding of Network Destinations, aug 1993.

[Wik] Wikipedia. Hash Table - en.wikipedia.org/wiki/hash_table.

[Zha06] Lixia Zhang. An overview of multihoming and open issues in GSE. *IETF Jurnal*, 2(2), 2006.

Abbreviations

ALT	Alternative Topology
ANSI	American National Standards Institute
ARPANET	Advanced Research Projects Agency Network
AS	Autonomous Systems
BGP	Border Gateway Protocol
CE	Customer Egde
DB	Database
DFZ	Default Free Zone
DNS	Domain Name Server
EID	Endpoint Identifier
EID AFI	Endpoint Identifier Address Family Indicator
ENCAPS	Encapsulation
ESD	End System Designator
ETR	Egress Tunnel Router
FIB	Forwarding Information Base
FIFO	First in first out
GRE	Generic Routing Encapsulation
GSE	Global, Site, and End-system address elements
IAB	Internet Advertising Board
IETF	Internet Engineering Task Force
IOS	Internetwork Operating System
IP	Internet Protocol
IPv4	Internet Protocol Version 4
IPv6	Internet Protocol Version 6
IS-IS	Intermediate system to intermediate system
ISP	Internet Service Provider
ITR	Ingress Tunnel Router
LIG	LISP Internet Groper
LISP	Locator/Identifier Separation Protocol
OSPF	Open Shortest Path First

PI	Provider Independent
RFC	Request for Comments
RG	Rooting Goop
RIB	Routing Information Base
RLOC	Routing Locator
SHA-1	Secure Hash Algorithm-1
STP	Site Topology Partition
TCP/IP	Transmission Control Protocol/Internet Protocol
TE	Trafic Engineering
TTL	Time to live
UDP	User Datagram Protocol
VPN	Virtual Private Network
xTR	Tunnel Router