AALTO UNIVERSITY
School of Science and Technology
Faculty of Information and Natural Sciences
Department of Computer Science and Engineering

A. M. Anisul Huq

Experimental Implementation of a Compact Routing based Mapping System for the Locator/ID Separation Protocol (LISP).

Master's Thesis
Espoo, April 2012

Supervisor: Professor Antti Ylä-Jääski, Aalto University, Finland
Instructor: Hannu Flinck Lic. (Tech.), Nokia Siemens Networks, Finland

AALTO UNIVERSITY
School of Science and Technology
Faculty of Information and Natural Sciences
Degree Programme of Security and Mobile Computing

ABSTRACT OF
MASTER'S THESIS

Author: A. M. Anisul Huq

Title of Thesis:
Experimental Implementation of a Compact Routing based mapping system for the Locator/ID Separation Protocol (LISP).

Date: April 2012 **Pages:** 11 + 126

Professorship: Data Communications Software **Code:** T-110

Supervisors: Professor Antti Ylä-Jääski
Instructor: Hannu Flinck Lic. (Tech.)

It is by now something of a cliché to talk about the "explosive" or "exponential" growth of the Internet. However, the fact remains; the number of participant networks in the Internet has grown phenomenally, far beyond what the initial designers had in mind. This unprecedented growth has caused the existing Internet routing architecture to face serious scalability issues. Single numbering space, multi-homing, and traffic engineering, are making routing tables (RTs) of the default free zone (DFZ) to grow very rapidly. In order to solve this issue, it had been proposed to review the Internet addressing architecture by separating the end-systems identifiers' space and the routing locators' space. Several such proposals exist, among which Locator/Identifier Separation Protocol (LISP) is the only one already being shipped in production routers. As LISP considers two different address spaces, a mapping system is required to provide bindings between the two. Out of several proposed mapping systems, LISP-ALT has emerged as the de-facto one, as it has a complete implementation for the Cisco NX-OS and IOS platforms. LISP-ALT's downsides being, its wide reuse of BGP and assuming a "highly aggregatable" or administratively pre-allocated IP address space; making it unsuitable for solving the routing scalability problem.

The concept of Compact Routing on the other hand, guarantees that the size of the RT will grow sub-linearly, which goes a long way in solving the scalability problem of DFZ RTs. It also puts an upper bound to the latency experienced by a datagram packet. The major drawback here is that, the "label"/address in Compact Routing comprises of three parts which cannot be implemented by any existing addressing scheme. Furthermore, it assumes a static network topology, which is absurd in the current Internet infrastructure.

Therefore, this thesis presents the implementation of an experimental mapping system called CRM that combines the perceived benefits of both Compact Routing and LISP. In this mapping system, the critical functions that affect the scalability of the routing system are grounded to the theory of Compact Routing; so that we might overcome the shortcomings of LISP-ALT. We mitigated Compact Routing's presumption of a static network by reusing LISP's registration messages and choosing landmarks dynamically based on their capability to aggregate. The key objective of this thesis work is to provide proof of concept, to give us first-hand experience regarding the complicacies that arise with the actual development of such a mapping system. Our work also includes a comprehensive comparison between CRM and LISP-ALT. The results suggest that, CRM would be feasible in the current Internet if deployed and it would be far less expensive than LISP-ALT.

Keywords: LISP-ALT, Compact Routing, BGP, scalability, Aggregation
Language: English

Acknowledgements

I am grateful to everyone who supported me throughout my thesis work. I would like to specially thank my supervisor, Prof. Antti Ylä-Jääski for giving me the chance to work on such an interesting topic.

I would like to show special gratitude to my instructor Hannu Flinck from Nokia Siemens Networks (NSN), Finland, for his constant supervision and valuable feedback during the entire thesis work. He is undoubtedly one of the best mentors that I have come in contact with. I would like to thank my co-worker, Tapio Partti from NSN. He had the answers to all my routing problems. A special thanks to my manager at NSN, Jari Lehmusvuori, who have helped me immensely in many administrative tasks.

Many thanks to Johanna Heinonen, Petteri Poyhonen and Jarno Rajahalme of NSN for providing me valuable advice and much needed support during the writing process of this thesis.

Finally, this thesis is dedicated to my adoring parents. They are the only two people who have supported and inspired me with their unconditional love throughout my life.

Espoo, April 2012

A. M. Anisul Huq

Contents

List of Tables

List of Figures

Abbreviations and Acronyms

ALT	Alternate Topology
API	Application Programming Interface
AS	Autonomous System
BGP	Border Gateway Protocol
CIDR	Classless Inter-Domain Routung
DA	Destination Address
DFZ	Default Free Zone
DNS	Domain Name System
eBGP	external BGP
EID	Endpoint Identifier
ETR	Egress Tunnel Router
FIB	Forwarding Information Base
IAB	Internet Architecture Board
IANA	Internet Assigned Numbers Authority
iBGP	internal BGP
IEEE	Institute of Electrical and Electronics Engineers
IETF	Internet Engineering Task Force
IGP	Internal Gateway Protocol
ILNP	Identifier-Locator Network Protocol
IP	Internet Protocol
IPv4	Internet Protocol version 4
IPv6	Internet Protocol version 6
IRTF	Internet Research Task Force
ISP	Internet Service Provider
ITR	Ingress Tunnel Router
LISP	Locator Identifier Separation Protocol
LM	Landmark
MPLS	Multi-Protocol Label Switching
MRAI	Minimum Route Advertisement Interval
PA	Provider Aggregated
PI	Provider Independent
RIB	Routing Information Base
RIR	Regional Internet Registry

RLOC	Routing LOcator
SA	Source Address
SQL	Structured Query Language
TCP	Transmission Control Protocol
TE	Traffic Engineering
UDP	User Datagram Protocol
VA	Virtual Aggregation
VM	Virtual Machine
VP	Virtual Prefix
VPN	Virtual Private Network

Chapter 1

Introduction

1.1 Motivation

The Internet routing infrastructure provides connectivity to millions of computers around the world. As the Internet experienced an explosive growth during the last three decades, its routing system has encountered numerous challenges brought about by the unprecedented scale of the system. In addition to the relentless growth in the number of customer networks, there have been increasing trends of multihoming to facilitate load balancing and fail-proofing, a desire to assign provider-independent (PI) addresses over provider-allocated (PA) addresses; so that internal renumbering can be avoided when shifting to a different provider and deployment of Virtual Private Networks (VPNs) to support business and enterprise users. Unfortunately, this brisk user growth compounded with multihoming, PI addressing and VPN provisioning has led to a fast growth of the global routing systems. At the same time, Internet service providers (ISPs) are faced with economical constraints that might prevent them from upgrading to the latest technologies to meet the demands. Most recently, the Internet routing architecture was confronted with two new challenges: firstly, the IPv4 address space became exhausted which in turn, leads to the wide deployment of IPv6 in the foreseeable future and secondly, the emerging mobile access to Internet from billions of hand-held devices. The latter further drives the demands for IPv6 to be rolled out and yet the sheer size of the IPv6 address space presents a great scaling concern for the routing system. In essence, it is this natural evolution of the Internet that has lead us into the problematic situation regarding routing scalability. Therefore, it has now become imperative that we come up with a solution for the routing scalability problems which would enable the Internet to grow in an unimpeded manner and will allow ISPs to operate within feasible upgrade intervals and costs. This need happens to be the primary motivation of this thesis work. The aim is to design and implement a backward-compatible, evolutionary scheme that solves the predicaments caused by the rapidly growing global routing system.

During the early stages of the Internet, the primary goal was to interconnect all packet switched networks so that packets could be delivered from any IP box to any other IP boxes. IP Gateways were invented to interconnect networks with different underlying communication technologies. All these gateways ran in a single routing domain and they were expected to forward packets for all their neighbors. Afterwards when the Internet started to expand rapidly, this flat routing architecture was abandoned to keep up with the increasing network scale and management complexity. When Exterior Gateway Protocol (EGP) was introduced, the concept of Autonomous System (AS) was also developed. Later, Border Gateway Protocol (BGP) replaced EGP to accommodate routing policies and more complex peering structure. At present, the pervasive practice of multihoming is having a negative impact on the scalability of routing and addressing architecture. Though considered essential for network operations, multihoming is destroying topology-based prefix aggregation [39].

Another fundamental problem in scaling involves the equal treatment of every AS; even when customer networks (e.g. university campuses, company sites etc.) and provider networks (e.g. AT&T) have different business models, different growth trends and different goals in network operations. A routing flap [1] to any destination triggers routing updates to be propagated to the entire Internet, even when no one communicates with that particular destination before its connectivity recovers. Both *Huston* [25] and *Oliveira et al.* [48] have shown that, overwhelming majority of BGP updates are generated by a very small number of sources, most of them originating from small edge networks. Failing to accommodate the distinction between customer networks (CNs) and provider networks (PNs) is the root cause of the scalability problem facing today's global routing architecture [39]. [2]

Up until now, the proposed solutions to overcome the difficult problem of routing scalability includes address aggregation and hierarchical partitioning of the network domains. Address aggregation (RFC1518) is a method of representing a series of network addresses through a single summary address. The advantage of route aggregation lies in the conservation of network resources. Advertising fewer routes saves bandwidth, and CPU cycles are conserved by processing fewer routes. Most importantly, memory consumption is reduced by the decreased size of route tables. [3]

In the mid 90's, the Internet's routing system adopted strong address aggregation using Classless Inter-Domain Routing (CIDR) to combat the address scaling problem. CIDR allows better aggregation of IP address space through variable length IP prefixes. It allows address aggregation at several levels. The idea here is that the allocation of addresses has a topological significance which in turn enables us to recursively aggregate addresses at various points within the hierarchy of the Internet's topology. However, to achieve highly efficient address aggregation and relatively small routing tables, a CIDR network must have: a tree-like graph structure and addresses should be assigned following this structure. With CIDR, backbone routers need not maintain forwarding information at the network level. Instead, the forwarding information at the level of arbitrary aggregates of networks. This recursive address aggregation reduces the number of entries in the forwarding table of the backbone routers. By utilizing CIDR, we can represent the network numbers from 208.12.16/24 through 208.12.31/24 with a single entry of 208.12.16/20. Because from the binary representation, it is evident that, the leftmost 20 bits of all the addresses in this range are the same (i.e. 11010000 00001100 0001). Thus, we can aggregate these 16 networks into one "super network" represented by the 20-bit prefix of 208.12.16/20 [54]. CIDR is also accompanied by a suggested prefix allocation policy that creates opportunities for aggregation. Unfortunately the benefits of CIDR are counteracted by disincentives to aggregate, which leads to the announcement of more specific prefixes in addition to, or instead of, aggregated prefixes. In particular, *Bu et al.* [6] has shown that, multihoming, inbound traffic engineering, fragmented address space and failure to aggregate are the principal contributors of BGP routing table growth and in turn causing CIDR to fail [35, 62].

According to the suggestions of Internet Research Task Force (IRTF), partitioning the address space into two may provide a way out. Here, an address space will be used for hosts residing in the edges networks; while another separate address space will be utilized for routing across the core network. One proposal for implementing this suggestion is the Locator/Identifier Separation Protocol (LISP) [13]. It is a simple IP-over-UDP tunneling protocol aimed at giving a network layer support for partitioning the address space. LISP is incrementally deployable without disrupting the current Internet architecture and/or without the need to make heavy changes

[1] A route flap is any routing change that would cause a change in the BGP table. `http://www.nanog.org/meetings/nanog3/notes/route-flapping.php`

[2] This distinction is in terms of global data delivery service. CNs serve directly to end users and are consumers of the global data delivery service. PNs on the other hand, have the sole purpose of delivering packets for a charge. In return, they are contractually obligated to the customer networks for providing packet delivery service.

[3] IP Subnetting & Address Aggregation. `http://www.netstreamsol.com.au/networking/notes/routing/ip_subnetting_address_aggregation.html`

in the protocol stack of end-systems. End-systems will still send and receive packets using IP addresses in the edges networks. But these IP addresses (belonging to the edge network) will not be routable in the core network. When end-systems need to send/receive packets to/from outside the local AS, they will do so through one or more associated Tunnel Routers. Tunnel Routers will have globally routable IP addresses through which packets will be routed in the core network. Now, this separation of host addresses from the ones used for routing will require the use of a mapping between the two. There are a number of proposed systems; but LISP-ALT (LISP Alternative Topology) [17] has emerged as the de-facto mapping system, as it is currently being developed and experimented on by Cisco. The intention here is to solve the scalability problem of BGP. LISP-ALT widely reuses BGP and assumes a "highly aggregatable" host address space. For achieving "highly aggregatable" address space, the IP address assignment process has to be along network topological lines [18].

However, in our opinion, the presumption of a "highly aggregatable" host address space is a glaring limitation of LISP-ALT. Consequently, LISP-ALT will suffer from the same problem of address erosion as the current solution provided by Internet Assigned Numbers Authority (IANA) and Regional Internet Registries (RIRs). When the address space erodes, it will lead to a "not so" aggregatable address space, which in turn will increase the size growth of routing tables (RTs). Our current experience also shows that, BGP does not cope well with the rapid growth of RTs. Eventually, LISP-ALT will have the same drawback and limitations as the prevailing hierarchical distribution based resolution. A part of this failure is caused by the fact that, IANA does not have any actual authority; it only keeps authoritative records concerning various numbers for other organizations. In other words, IANA merely serves as a bookkeeper in recording the address assignments that were made. It has no influence over the already assigned addresses i.e. IANA cannot "police" over how the addresses are reassigned to different customer level organizations. Same can be said for RIRs also. Therefore, the allocated address boundaries degenerate as the network evolves to meet the real life needs; e.g. multihoming, traffic-engineering etc. And after a while, the original "optimally" allocated address space will start to weigh down the network or in extreme cases may fail altogether [4] [5]. LISP data plane does not suffer from the scaling problem caused by multi-homing. Instead this problem is shifted to the mapping system (i.e. LISP-ALT) that tries to route (in the ALT network) based on EIDs and this address space will erode due to normal business dynamics (e.g. organizations shifting from one service provider to another).

For convenience, we will refer to our proposed "Compact routing based mapping system for the locator identifier separation protocol (LISP)" as "CRM" from hereafter [36]. The question then arises is why do we need another unique mapping system based on Compact routing? The answer lies in the short comings of LISP-ALT. In our CRM, we did not take the liberty of imagining a "highly aggregatable" address space. Our system's aggregation is not based on any administratively pre-allocated IP address space, rather through learning about the network reachability. It will deal with the "orphan" addresses (i.e. non-aggregatable addresses) either by delegating or by generating virtual prefixes [6]. Also stated previously is the fact that, LISP-ALT extensively uses BGP. Down the road, this strategy will lead to catastrophe. Because the very system that is supposed to solve the scalability of BGP, cannot itself be heavily dependent on the functionality of BGP. That is why, in our opinion, using LISP-ALT will not resolve the scalability issue of BGP and after a while we might find ourselves in the same ominous situation as we are right now. Conversely, while designing our CRM, we made an conscious effort so that it utilizes BGP in a minimal fashion and at the same time, the critical functions that affect the scalability of the routing system are grounded to the theory of compact routing.

[4] Abuse Issues and IP Addresses. `http://www.iana.org/abuse/faq.html`

[5] Lisp Archive. `http://answerpot.com/showthread.php?1552501-EID%20Allocation%20/%20ALT%20Base/Page2`

[6] To be precise, virtual prefixes start to generate when the system grows quickly; otherwise there might be situations when "orphan" addresses are advertised.

Since our mapping system is based on Compact routing, we can guarantee that the size of the routing table will grow sub-linearly. LISP-ALT does not provide with any such assurances.

LISP-ALT does not provide any upper bound for the latency. Because, this system cannot ensure how much delay an ALT datagram will encounter while it travels through the ALT network. A Compact routing based mapping system like ours is however has a fixed upper bound for the path stretch. This in turn, makes the delay very much predictable.

Theoretically, our proposed CRM has the aforementioned clear advantages over LISP-ALT. This lead us to its experimental implementation. Though the idea of a CRM looked solid on paper, we wanted to gain first-hand experience regarding the complicacies that arise with the actual development.

1.2 Objective and scope

The key objective of this thesis work is to provide proof of concept. Our implementation of a CRM will be the first of its kind. The idea is to look closely at the existing routing scalability issues and design/implement such a system that mitigates these problems. We are effectively "fusing" ideas from both Compact routing and LISP to come up with a mapping system that has the best of both worlds. Our main goal is to discover whether such a system is feasible or not.

At the initial stages, our intention is to figure out how much of a "new" system (i.e. in addition to the existing standard Unix/Linux based routing systems) we need to construct to fulfill the functionality requirements of a CRM. At the same time, we wanted know, to what extent we could be able to use the existing routing facilities. Because reusing routing functionalities would allow us to develop a less complicated system.

Afterwards, we move on to determine how "deep" or "loose" the integration of the functionalities need to be. For example, to have a working system, would we require to make modifications to the network stack or can we develop a mapping system that is completely modular from the underlying system (OS kernel, IP stack etc.). Answer to this question is rather crucial; because we do not want get involved with intricacies of the kernel if possible; which might result in a "costly" system specific CRM and hamper the deployment.

If the aforementioned questions are answered then we will try to discover whether there are any missing parameters that need to be taken into account. For instance, we need to determine exactly which of the BGP attributes we will be using to discover the network topology.

One of the prime objectives from the beginning, is to minimize dependencies. Our goal is to deliver an independent system that will be compatible with any system (i.e. routing software, OS etc.) by making minimal changes to the interface.

Our ultimate aim is to come up with a mapping system that limits the role of BGP; so that we can later on retire it and replace it with a simpler topology discovery protocol. To achieve this, our CRM should use BGP functionalities (e.g. attributes, path selection etc.) in the least possible way; so that if we can come up with a suitable topology discovery protocol in the future then our CRM can be plugged into it by making minimal changes.

We have not confined the scope of this thesis work only to the prototype development. Performance analysis, possible future optimizations and comparisons (between LISP-ALT and CRM) are also part of our current work. However, the primary focus of this research work is to learn about the pitfalls experienced and adjustments made while developing such a system and also to integrate the algorithm that decides whether virtual EID-prefixes should be used or not, developed by *Flinck et al.* [14].

1.3 Thesis Outline

The thesis is logically structured to provide the reader with suitable background knowledge before diving deep into the details of implementation and subsequent analysis. After introducing the work in Chapter 1, an overview on topics such as, routing scalability issues, compact routing, LISP and BGP are presented in Chapter 2. To provide a better understanding of the external tools used in our implementation, brief overviews on Quagga, MySQL and MD5 message-digest algorithm are provided in Chapter 3. This concludes the background work. The original work is presented in Chapters 4 and 5, where details regarding the software implementation of our work is presented, and a comprehensive comparison between LISP-ALT and CRM is provided. The thesis ends with a very short overview of the research questions that we have answered through this dissertation. Appendices A and B provide additional material concerning the outcome and definitions of CRM's database tables; while appendix - C presents C(GCC)-code that implements the main sub-tasks of our prototype.

Chapter 2

Background Information

The Internet community is currently facing an important challenge regarding the scalability of the global routing system. In order to get a deeper understanding of this problem, we have to look into the architecture of the Internet. The Internet is divided into thousands of autonomous systems (ASs), each of which is formed of networks of hosts or routers administrated by a single organization. Hosts and routers are primarily identified with 32-bit IP addresses (128-bit long IPv6 addresses are in the process of getting deployed in the current Internet infrastructure). To ensure scalability, IP addresses are aggregated into contiguous blocks, called prefixes that consist of 32-bit IP address and mask lengths (e.g. 1.2.3.0/24 represents an IP block ranging from 1.2.3.0 to 1.2.3.255). Routers exchange reachability information for each prefix using the Border Gateway Protocol (BGP). In other words, each entry in the BGP routing table contains the reachability information for a particular prefix [6, 37]. The size of a BGP routing table refers to the number of prefixes contained in that table. But before going in to the details of BGP routing table growth, we would like to explain the related technical terminologies to achieve better understanding. In the context of Internet routing, the default-free zone (DFZ) refers to the collection of all ASs that do not require a default route. Theoretically, DFZ routers contain a "complete" BGP table, sometimes referred to as the global routing table (GRT) or global BGP table. Hence, GRT is the set of all Internet address prefixes announced into the DFZ and therefore comprises of the entire "public Internet". In essence, it houses all the advertized BGP routes.

The ASs on the Internet can be classified in a loose tier hierarchy, where only a few tier-1 ASs exist. These tier-1 ASs frequently "peer" with each other. The term "peer" refers to the transport of traffic between two networks for "free", however in some cases "paid" peering is also practiced. "Free" traffic transportation occurs only when an equivalent amount is exchanged; otherwise "paid" peering takes place. The network receiving most incoming traffic will generally receive payment. Tier-2 and tier-3 ASs on the other hand, pay for some or all routes accordingly and seek peering agreements when possible and advantageous for both parties. Traditionally DFZ consisted of mainly tier-1 ASs, but the rapid growth of multihoming has meant that, DFZ may now have to reach even tier-3 networks [49].

In our opinion, the biggest challenge faced by the Internet community since the '90s, is the already depleted IPv4 address space. IPv6 is, to a large extent, is ready to fill this breach. However, experience with IPX suggests that it might take a decade to get IPv6 fully deployed in the current Internet infrastructure. As stated at the beginning of this section, the other serious agenda is regarding the rapid growth of routing tables (RTs) and it is at the focal point of our work. Figure 3 depicts how the number of active BGP entries (FIB) in IPv4 DFZ has increased since 1994 (until mid-2011), showing a stable super-linear growth pattern since 2002. Already there are close to 400,000 prefixes in the DFZ [26] and if the current trend continues then it will reach the milestone of 0.5 million prefixes by 2015. The number of RIB entries in IPv4 DFZ is

already gone over eleven million [49].

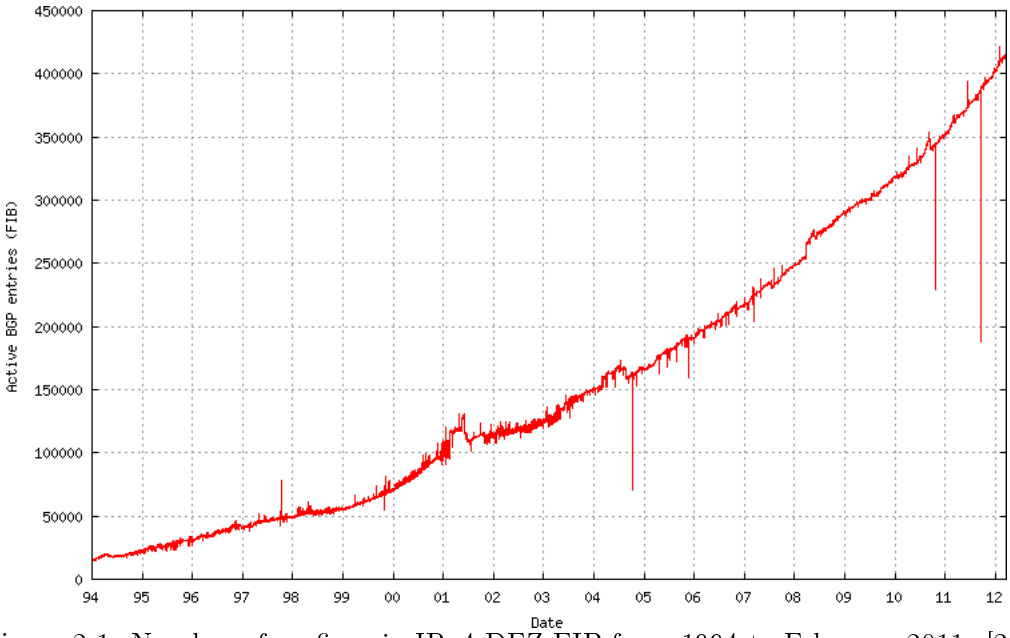

Figure 2.1: Number of prefixes in IPv4 DFZ FIB from 1994 to February 2011. [26]

Moore's law directly impacts on the capability to support large RTs, as these are subject to greater resource requirements. As indicated by this law, the computation power and memory requirements for a router that is capable of sustaining the growing Internet can be met by the advent of multi-core processors and large memories. However, even if the technology can cope with the scalability challenges, the necessity to make periodic upgrades due to the dramatic increase of RTs may significantly increase Internet operator's expenditures on new equipment and thus challenge the economic viability of the Internet [37].

Many factors contribute to the growth of active BGP entries (FIB) in IPv4 DFZ and will continue to do so; while some new trends may further accelerate this process [49]. In our opinion, the main reasons behind global RT growth are:

- Multi-homing,
- Traffic Engineering,
- Address fragmentation and
- Failure to aggregate.

These reasons are introduced and discussed in the following sub-sections to give a better understanding of the specific factors contributing to the problem and to elaborate the overall situation. It must be noted that, though the first two are considered as legitimate reasons, the latter two are treated as indefensible ones.

Multi-homing Multihoming is the process of having multiple connections to different ISPs. It can be divided into two parts: host and AS multi-homing. Host multi-homing is where an end user is connected to multiple ISPs while AS multi-homing is where an AS is connected to multiple upstream provider ISPs. It is basically a business practice with the primary goal to increase the AS's network durability, provide redundancy, load balancing and connectivity in case of a link failure. However, this practice increases RT size in the DFZ.

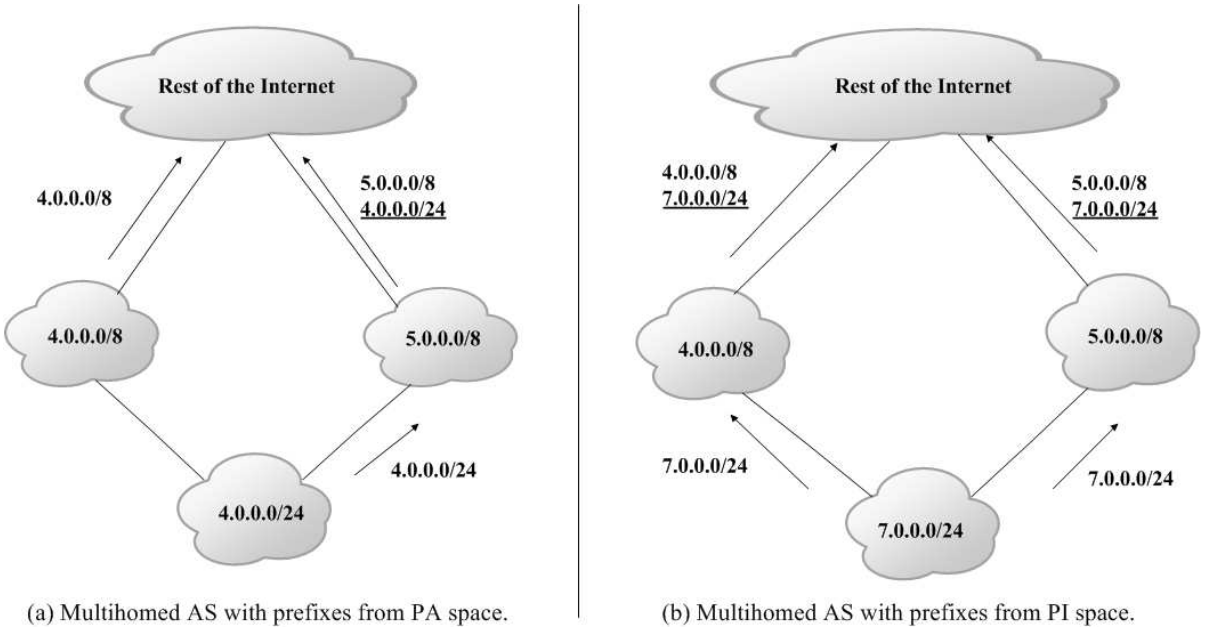

Figure 2.2: (a) Multi-homed AS acquires its addresses from PA space, resulting in one additional advertisement (only if multi-homing is used for redundancy, otherwise 2 advertisements are propagated) (b) Multi-homed AS acquires its addresses from PI space, resulting in two additional advertisements. The underlined prefixes refer to the extra entries that needs to be advertised [45].

A multi-homed AS requires all its providers to advertise the prefixes it serves to allow traffic to be routed via any of its providers. The number of additional advertisements and routing table entries depend on the type of address allocation for the multi-homed AS, as well as on the type of service required by that AS. The Internet Assigned Numbers Authority (IANA) allow two types of address to be allocated to an AS: provider aggregatable (PA) and provider independent (PI). PA space is assigned from a provider AS to its customers; enabling the provider AS to advertise one aggregated prefix using BGP that permits traffic to reach both it and its customers. Conversely, PI space is assigned independent of an AS's provider, and as a result may be from a different prefix range, making it non-aggregatable by this AS's provider; requiring the provider to advertise both its own and its customers prefixes. A multi-homing AS with prefixes assigned from PA space, can result in either n or n - *1* additional advertisements, where n is the number of providers. If the AS's reason for multi-homing is redundancy, n - *1* advertisements are required (one for every other provider). However, if the AS wants to be able to send and receive data over all its links at once then n advertisements are required to ensure that one path is not selected over another due to that path having a longer prefix. Should the AS's prefixes be assigned from PI space, there is always n advertisements, regardless of type of service, due the each provider having to advertise a prefix that is out with its own prefix range. The aforementioned figure 2.2 provides illustration of the differences between a PA and PI prefix allocation with respect to advertisements when an AS multi-homes [45].

Back in 2002, *Bu et al.* [6] examined the BGP RTs of Oregon route server and quantified (in percentage) the contributors to the growth (of RTs). According to their measurements, multi-homing introduces around 20 - 30% extra prefixes.

Traffic-engineering Traffic engineering (TE) is "concerned with the performance optimization of networks" [66]. Its (i.e. TE's) policies cause the expansion of the RTs. TE is delivered either by advertising longer prefixes or by prepending AS-path. In essence, such de-aggregation influences traffic flows. In the figure below, in both of the

configurations, traffic originates from AS 702 and AS15169 is applying TE to manage its ingress traffic [1].

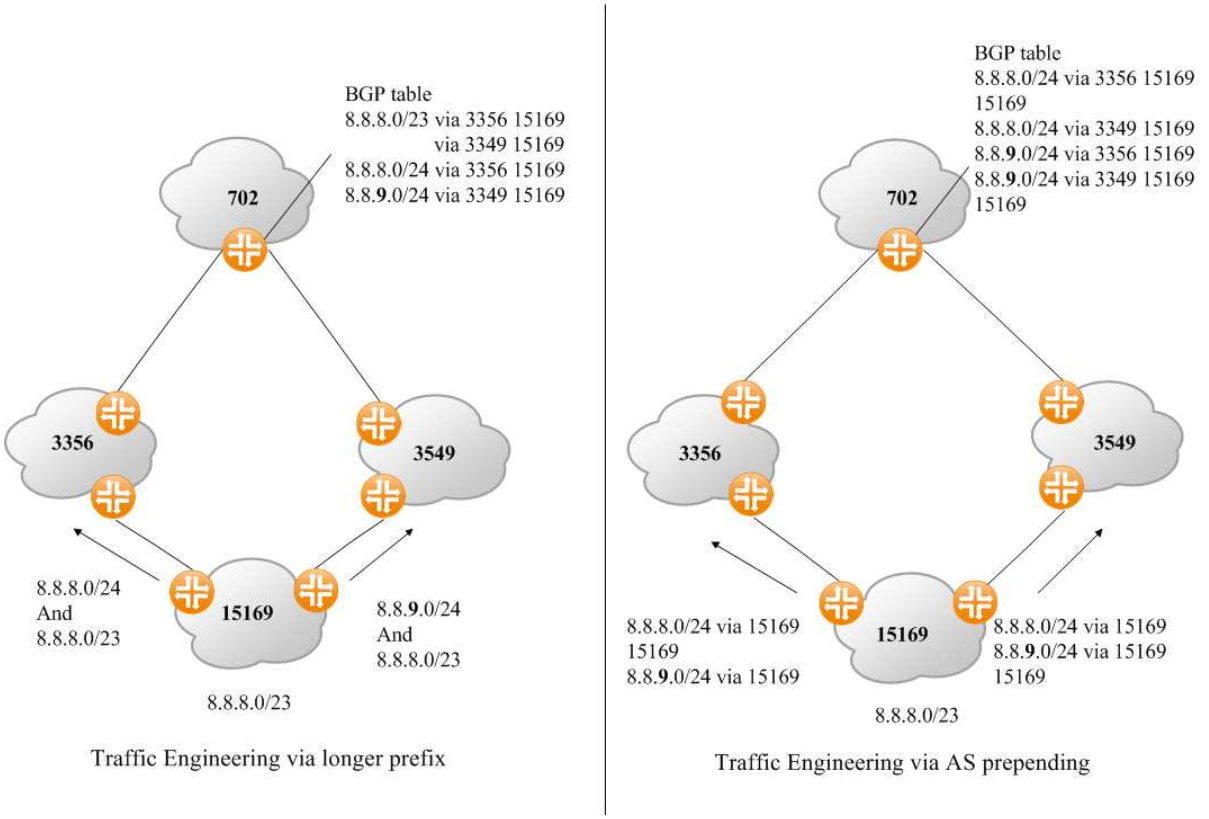

Figure 2.3: Traffic Engineering scenarios.

In the figure above, the left-hand configuration shows TE through the use of longer prefix advertisements. The longest prefix match algorithm will cause the traffic to choose the path with the longest prefix. In the left-hand configuration, 8.8.8.0/24 is advertised towards AS 3356 and 8.8.8.0/23 is advertised towards AS 3549. As a result, traffic destined for 8.8.8.0/24 will transit AS 3356 and for 8.8.9.0/24, traffic will go through AS 3549 [2].

In the right-hand configuration, the strategy of AS-path prepending is used to achieve TE. Here, consecutive repetitions of the same AS number is added to the AS-path attribute. The goal is to lengthen the AS-path attribute, making it less desirable for the BGP decision process (which prefers routes of shorter AS-path length) [8]. As seen in the right-hand configuration, 8.8.8.0/24 is advertised with an extra copy of the AS 15169 towards 3356, while 8.8.9.0/24 is advertised with an extra copy of the same AS number towards 3549. This will cause traffic from AS 702 destined for 8.8.8.0/24 to transit 3549; while traffic towards 8.8.9.0/24 (from the same source) will go through 3356 [2].

TE can be optimized using the "no-advertise" community attribute in cases where traffic is being engineered towards a single provider. By sending both the aggregated and the longer prefixes, with the "no-advertise" community value set for the longer prefixes, traffic will follow the aggregated ones to the upstream provider. Once traffic reaches the upstream provider, more-specific routes kick in and perform the desired TE function. This way, only the upstream provider deals with the increased routes while the rest of the Internet is spared and thus helping in the reduction of BGP route churn [2].

[1] Ingress traffic is composed of all the data communications and network traffic originating from external networks and destined for a node in the host network. Ingress traffic can be any form of traffic whose source lies in an external network and whose destination resides inside the host network.

[2] `http://blog.pattincon.com/internet-routing-table%E2%80%93the-good`

Address Fragmentation Address fragmentation is a phenomenon where a single entity in the network has multiple non-contiguous IP address blocks or prefixes, instead of a single prefix in the routing table. Needless to say, this inflates RT size, degrades scalability, increases IP address lookup time and deteriorates routing performance. Address fragmentation takes place because addresses are allocated to entities (i.e. customer enterprises) on an ad hoc basis. As a customer enterprise grows over time, it may file multiple separate requests for addresses. Thus, it (i.e. customer enterprise) ends up with non-contiguous address blocks. Lack of available address blocks in IPv4 causes this non-contiguousness [63]. As IPv6 (128-bit long) is being deployed gradually in the current Internet infrastructure, we need to study and design address allocation schemes that induces minimum fragmentation. Instead of using existing allocation policies, *Wang et al.* [63] came up with a new scheme called GAP or Growth based Address Partitioning. According to [63], GAP takes into account the growth rate of each customer enterprise and partitions the address space in such a way that provides maximum possible space for each customer. The authors of [63] are confident that, GAP can significantly reduce address fragmentation and improve the efficiency of address usage; because this scheme was tested on real world data.

Failure to aggregate Lack of knowledge on Classless Inter-Domain Routing (CIDR) can cause enterprises to fail in aggregation. The application of /16 and /24 to allocated aggregates ("to match the class") by well-meaning but misguided staff is a major source of de-aggregation.

Fear of Denial of Service (DoS) attacks from neighboring ISPs can also can cause an enterprise (i.e. its ISP) not to aggregate properly. In such a DoS attack, neighboring ISPs falsely announce longer prefixes. In economics terminology, this is a "beggar-my-neighbor" policy that will result in the benefit of the neighboring ISPs (i.e. through increased traffic) at the expanse of "this" ISP. This "fear factor" is a major cause for the growth of RT size. Because de-aggregating everything even to /24 will result in over 5 million extra prefixes [2].

Economical reasons It is undeniable that, there are certain benefits of de-aggregation. It provides a very effective and fine-grained method for performing TE. By increasing the granularity of the advertisements through the use of longer/shorter prefixes, the ISPs achieve better control of the distribution of traffic over the transit links. To improve security, certain networks announce more-specific prefixes than the allocated ones. This strategy is very handy in preventing prefix hijacking [37] [3].

No matter what the reason is, inflation in the size of BGP RTs create additional costs. In economics terminology, such cost is referred to as negative externality. Negative externalities occur when the consumption or production of a "good"/service by a single agent causes a harmful effect towards other agents. In such a situation, there exists no voluntary agreement between the two that would allow a negotiation for the distribution of these costs. The size inflation of RTs is considered a negative externality, because when a network de-aggregates it obtains far greater benefit than this operation actually costs and it does not consider the additional expenditure it brings to the other networks. In essence, ASs de-aggregate at the expense of all the members of the DFZ [37]. According

[3] Prefix-hijacking occurs when a malicious AS fraudulently announces to its peers that it owns a block of IP-address space, when, in reality, it does not. After a short delay, routes based on this bad announcement propagate through the Internet at large and enables this malicious AS to send and receive traffic using these addresses that it does not own. Such hijacked space can be - and has been - used to send spam, download copyrighted material, launch break-ins, or use the network to serve illegitimate purposes. `http://www.cs.uoregon.edu/Activities/Poster_Contest/2005/boothe-hijacking.pdf`

to [4], it costs "everybody else" at least $6200/year to carry one BGP route someone had announced. *Lutu et al.* [37] has used the contents of the seminal article by *Hardin* "The Tragedy of the Commons" [22] to model this problem of overuse and degradation of common (or public) resources. According to *Hardin* [22], "a tragedy of the commons occurs when a group of entities abuse a common resource and thus harming the other individuals with whom it shares this resource and themselves". The analysis of [37] shows that, ASs in the inter-domain routing engage in the detrimental behavior heavy de-aggregation because of the economic incentives; causing a tragedy of the commons. In order to avoid the heavy cost incurred by de-aggregation, tier-1 operators usually filter out any prefix that is longer than /24.

2.1 Compact Routing: A brief theoretical overview

The goal of any routing scheme is to allow any source node to route messages to any destination node, when the destination's network identifier is known [20]. In other words, it is an (distributed/centralized) algorithm that specifies the nature and systemic interaction between node-level decision-making processes or daemons, local memory that helps each daemon make routing decisions (better known as routing tables), and the transmitted information (transmitted in chunks or packets) [4]. There are basically two ways in which routing is achieved. The first approach is to store a complete routing table (i.e. shortest path) in each of the network nodes. Traditional routing algorithms take the shortest-path approach. This requires each node to store $O(n \log(n))$ bits of routing information in its local memory. However, the number of nodes in a network can be very high and thus the amount of local memory usage in each node can eventually become too expensive. In an alternative strategy, for each packet that comes out, the source can insert a complete description of the path to be routed in that packet's header. The packet header along with local routing tables enable nodes to forward the packet properly along the path between source and destination [4]. As a drawback of this approach, packet header size can grow to the size of (n) [5]. Compact routing is a research field that studies the fundamental limitations of routing scalability and designs algorithms that try to mitigate these limitations [11, 35]. In layman's terms, it analyzes how "good/bad" a particular routing algorithm is when the available resources (e.g. available memory, processing power etc.) are constrained, and provides the fundamentals for resource efficient routing protocols. Theoretically speaking, a routing protocol can be judged as compact if

1. The address and header sizes it uses grows logarithmically,
2. The size of the routing table (RT) grows sub-linearly as new nodes are added and
3. The path stretch is bounded by a constant. It remains constant independent of the network size growth [36].

Compact routing has shown that, the shortest-path approach taken by the traditional routing algorithms, cannot guarantee that, routing table (RT) sizes (on any network topology) will grow slower than linearly as functions of the network size. On the other hand, Compact Routing achieves the desired RT size reduction by increasing path stretch

[4] "What does a BGP Route cost?" by W. Herrin. `http://bill.herrin.us/network/bgpcost.html`

[5] Compact Routing: Challenges,Perspectives, and Beyond. By Dimitri Papadimitriou (Alcatel-Lucent Bell NV), Email: dimitri.papadimitriou@alcatel-lucent.be, TRILOGY Future Internet Summer School 2009. August 24-28, 2009. Louvain-la-Neuve, Belgium. `http://typo3.trilogy-roject.eu/fileadmin/publications/Other/Papadimitriou-CompactRouting.pdf`

between certain nodes. But, this path stretch is bounded. *Gavoille and Gengler* [19] has proven that, minimum value of maximum stretch for sub-linearly scaling RTs is *3*. It is therefore possible to have a routing scheme that guarantees that path stretch for a pair of nodes will never exceed three *($\leq$ 3)*, and still maintain sub-linear RTs. It is not possible to have a maximal path stretch of *2*, or *1*, with sub-linear routing tables [35, 36, 45]. It is obvious that, Compact routing schemes, do not pursue to achieve shortest-paths; instead they stretch the routing paths [35]. The term stretch is defined as the ratio between routing path's length/cost and minimum length/cost path [5]. Formally, for every pair of nodes (in all graphs in the set of graphs) the algorithm can operate on, stretch is the ratio between the length of the route taken by the algorithm and the length of the shortest path available. In both cases, length between the same pair of nodes is measured. A Stretch-3 would therefore, indicate to a path 3 times larger than the shortest possible path. For a specific routing algorithm, the maximum of this ratio among all source-destination pairs (in all the graphs in the set) can be defined as that algorithm's stretch [35]. Sub-linear scaling of a RT is needed to accommodate the needs of pervasive and ubiquitous computing. [6]

For the duration of this subsection, unless otherwise stated, n is the number of nodes in a network or graph. We are also only interested with compact routing schemes that said to be universal; i.e. they can provide routing for any type of graph (e.g. grid, tree, power-law etc.) [45]. In this regard, two compact routing algorithms are particularly important, the *Thorup-Zwick (TZ)* [61] and *Brady-Cowen (BC)* [3] schemes.

Cowen [9] was responsible for the first universal stretch-3 compact routing scheme. The routing scheme utilizes the concept of landmarks (LMs) and local neighborhoods. LMs are globally known nodes. Routing is performed by routing a packet to the nearest LM of the destination and then onwards via the local neighborhood to the destination. This idea is analogous to the postal system. The following figure shows a node, u, sending a packet to another node, v, in a different neighborhood by sending it via v's local LM, *L(v)*. The routing table of a node is constructed by maintaining next hop information for its neighbors and the global landmarks, this is all the information required as routing state. A node's neighborhood is defined as the k closest nodes to that node (where k is an arbitrary positive integer). LM set selection is done in a way so that RT sizes do not exceed $O(\sqrt[2/3]{n})$(i.e. number of LMs + a node's neighborhood size $\leq O(\sqrt[2/3]{n})$ [45]. [7]

[6] The term "stretch" should not be confused with "path inflation". One of the main reasons of "path inflation" is, intra- and inter-domain routing policies. It has got nothing to do with the stretch of the underlying routing algorithm.

[7] *Cowen*'s [9] LM set selection is performed using a greedy approximation to dominating set construction. Detailed discussion on this topic is out of the scope of the current work.

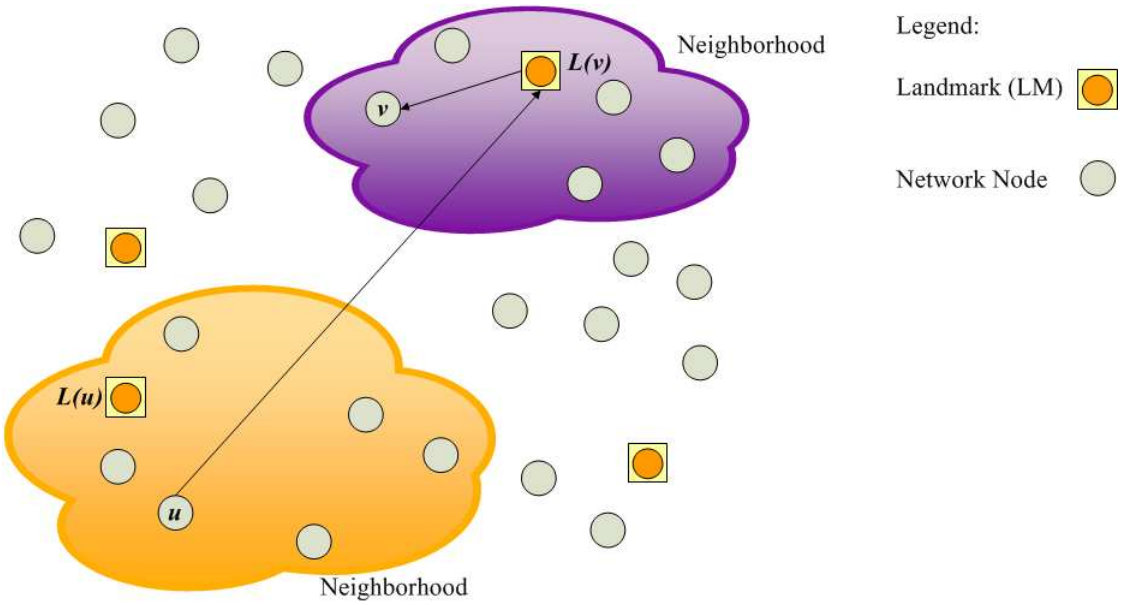

Figure 2.4: Compact routing example. [45]

TZ [61] came up with an improved universal compact routing scheme with worst-case $O(\sqrt{n})$ sized RTs and therefore will be at the focus of our attention. This reduction in RT state is achieved by changing the LM selection scheme to an iterative process. In the *TZ* [61] scheme, initially, an initial set of LMs, set A, is selected randomly with a uniform probability from all nodes. Once these LMs are selected, each node in the graph is assigned the LM closest to it. Unlike *Cowen* [9], a node's neighborhood (cluster in *TZ* terminology) is no longer its k nearest nodes, it is instead, formed with only those nodes that are closer to the node than to the closest LM [45]. Formally,

$$C(w) = \{v \in V \mid \delta(v, w) < \delta(A, w)\} \tag{2.1}$$

Equation 2.1 states that a node v is in w's cluster (i.e. *C(w)*) if w is closer to v than v is to its LM. Here, V is the set of all nodes in the graph, $\delta(v, w)$ is the distance between nodes w and v, $\delta(A, w) = min\{\delta(u, w) \mid u \in A\}$.

Should this node's cluster exceed a specified limit, then that node is then considered a candidate for becoming a LM in the next iteration of the LM selection. This process continues until all nodes have a cluster size not exceeding the limit. The threshold and the probability of a node being selected as a landmark are dependent on a value s, such that $1 \leq s \leq n$, where n is again the number of nodes in the graph. The probability that a node is selected as a landmark is $s/|potentiallandmarks|$, and the limit for a node's cluster size is $4n/s$. TZ [103] recommends a value of s such that $s = \sqrt{n/logn}$, allowing for a maximum routing table size of $6\sqrt{nlogn}$ comprising maximum cluster size of $4\sqrt{nlogn}$ and maximum landmark set size of $2\sqrt{nlogn}$ [45].

Clusters are required to ensure packets can be routed from a LM to the destination node; without this cluster information (labeled as "additional RT entry" in the following figure) packet loops would occur between landmarks and the first hop from that LM to the destination. This would result in any node not directly connected to a LM being unable to receive data. This is shown diagrammatically in the following figure, if nodes do not maintain information relating to their cluster, a packet loop may occur between nodes a and c for any packet destined for node d (follow arrows of 'step 1' and 'step 2a'). If instead cluster information is maintained at nodes c and b, packets addressed to node d will be correctly forwarded from node c, and b, to d (arrows 'step 1' and 'step 2b') [45].

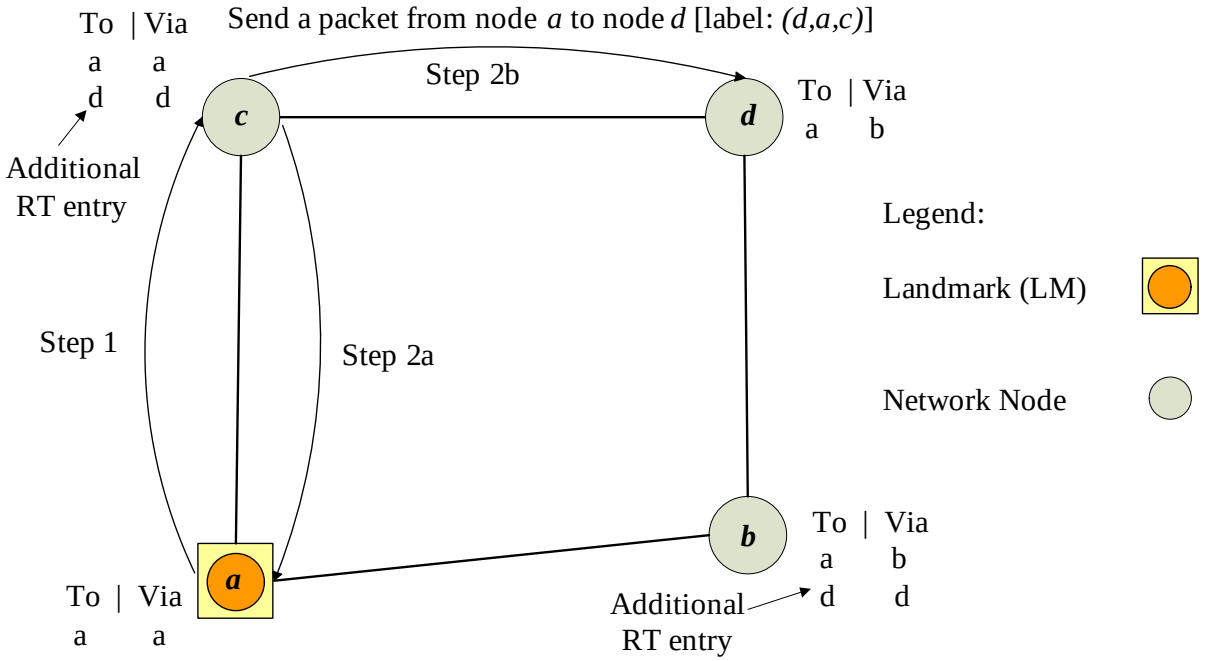

Figure 2.5: Why do we need clusters? [45]

It should be noted that, though the concept of cluster is explained here; cluster is *not* a part of our implemented prototype (i.e. CRM).

2.1.1 The TZ Routing Process

To route using the *TZ* scheme, one must know the destination's name, the destination's LM, and the port that LM uses to route traffic towards the destination (this part of the "label" signifies the node's cluster or neighborhood); these three components make up the "label"/address of a node. Routers within the network take routing decisions based entirely on these labels. For example, in figure 2.5 above, the label of node d is *(d, a, c)* where d is the destination, a is d's landmark, and the next hop for traffic to d is node c. The *TZ* scheme is a labeling scheme as the nodes in the network are renamed or labeled for the purpose of routing correctly. To route from source, a, to destination, b, the *TZ* scheme proceeds as follows: obtain the name of b's landmark node, l, route packet towards l (possible due to all nodes containing next hop information for all LMs). Upon reaching l the packet is routed towards b (possible as nodes in-between l and b will contain next hop information for b). Upon reaching b routing is complete, and the packet is successfully delivered. This routing procedure is analogous how the postal service works. Mail for the recipient is delivered to the destination's local sorting office (i.e. the LM), before being routed locally to the destination address [45].

Routing via a LM and then onward to the destination can increase the distance travelled by a packet. However, there is a limit imposed on this due to the nature of routing procedure; no *TZ* path is ever more than 3 times the shortest path between a and b (i.e. worst possible path stretch is 3). This is proved mathematically using the triangle inequality and symmetry by *TZ* [61].

Within the *TZ* [61] scheme, clusters not only allow packets to be routed from a LM to their destination, but also allows us to discover paths that are shorter than source to destination via a LM. To illustrate this, consider a packet with source *node 4*, destination *node 8 (label: (8, 7, 6))* in the following figure. The routing decision at *4* (send toward the

LM *(7)*) results in the packet being sent to *5*. *Node 5* follows the same strategy and sends to *node 6*. *Node 6* contains *node 8* within its cluster, and routes using this information, similarly for *node 9*. As you see, the packet destined for *node 8* from *node 4*, never reached the LM *node 7*. It is possible that a packet can be sent from a source to a destination without interacting with a LM [45].

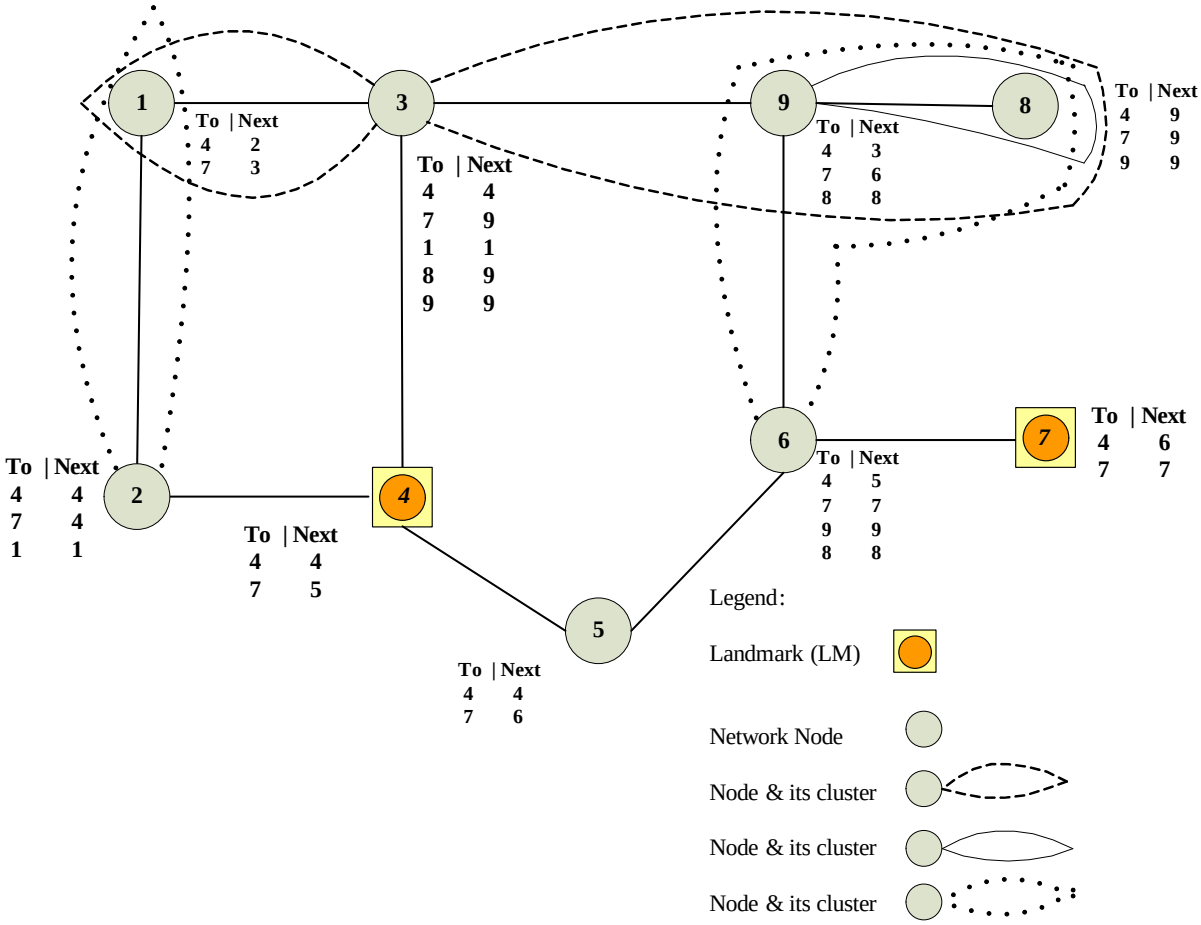

Figure 2.6: A 9 node graph showing TZ routing table information. [45]

In certain situations the path exhibited by a packet within the TZ routing scheme may match the shortest possible path between the source and the destination (i.e. Stretch-1). It is also possible for the paths exhibited between two nodes to follow different routes depending on the direction being travelled. Consider *node 2* and *node 8* of figure 2.6 *Source 2*, *destination 8* follows the path: *2->4->5->6->9->8 (length 5)*; meanwhile, *source 8*, *destination 2* follows the path: *8->9->3->4->2 (length 4)*. These paths are not identical, in fact, it is possible to have a source-destination pair such that traffic travelling in one direction exhibit stretch-1, while traffic travelling in the opposite direction exhibit stretch-3 [45].

2.1.2 Compact Routing - Limitations and Mitigations

As said earlier, in compact routing, the "label"/address of a destination *node d* is: *label(d, a, c)*; where *d* is the destinations name, *a* is the destination's LM, and *c* is the port that LM a uses to route traffic towards the *destination d* (see: figure 2.6). It should be evident from this description that, the "labeling"/addressing is dependent on the network topology and the placements of the LMs. This address structure also implies that, the LM is fixed for a node. Further complicating matters is the fact, neither IPv4 nor IPv6 can manage three separate addresses in a single unit. Even if we come up with some

clever way to fit all three parts of a "label" in an address unit; fixed number of bits would have to be allocated to each part of the "label". This means that, the maximum number of nodes in a cluster, the maximum number of nodes in the LM space gets fixed. These are massive downsides of Compact routing. It is assuming a static network topology (due to the binding between the node's address and LM) to keep the balance between the size of a LM set and clusters (through allocating fixed number of bits). Any change in the topology (e.g. if a LM goes down due to maintenance) would mean that, the network would have to be globally renumbered (i.e. re-addressing). Such assumptions are not practical if one considers the current Internet infrastructure [36].

In our proposed CRM, we are not "labeling" the network nodes; we are using the usual IPv4/IPv6 to identify the nodes. There is also no dependency between the address of the LM and the nodes that it services. Hence, if there is change in the address of either the LM or its nodes, the other (i.e. the LM or its nodes) is not affected. If the address of a LM changes then only the mappings (between that LM and the nodes that it services) need to be changed. In compact routing however, change in the address of a LM will cause global re-addressing.

2.2 Splitting location and ID

In today's Internet, two separate semantics are attached to an IP address. The first one is identity based, i.e. IP address of an end-system's interface is used in the transport and upper layer protocols as the its ID; to identify that host or sessions associated with that particular host. The second one is being a locator. An IP address is utilized in network layer protocols as a locator to find the destination host in the network topology and forward packets toward their destination. This dual meaning of IP addresses affects the way how machines connect to the Internet. If the location of a machine changes, then, by extension, it's identity also has to change (as IP address is used to serve both purposes). Hence, such "overloading" of functions (of IP address) makes it virtually impossible to build an efficient routing system without forcing unacceptable constraints (e.g. requiring renumbering upon provider changes) on end-system use of addresses. In order to circumvent these problems, a "loc/ID split" is performed.

To deploy "loc/ID split", we need to adjust addressing, headers, and protocols. Several proposals have been made for such an implementation [8]. Most of these proposals leverage one or more levels of indirection to create one or more levels of namespaces. In most cases, two namespaces are utilized, namely, Identifier (ID) and Locator (LOC). IDs define "who" (i.e. identity) the end-system's interface is. They (i.e. IDs) can only be used for routing in the edge/access network. LOCs, on the other hand, describe "how" an end-system's interface is attached to the network. What it means is, LOCs are used for routing across the global Internet. The goal of this indirection is to allow efficient aggregation in the LOC space which in turn provides persistent identity in the ID domain. Such indirection mandates the presence of a mapping system between IDs and Locators; so that packets originating from a host residing inside an edge/access network can be successfully routed through the global Internet until it reaches the destination host's edge/access network. Developing such a mapping system is the focus of this thesis.

The act of splitting locator and identity can be performed either taking the "map-and-encap" approach or by utilizing the address rewriting scheme. In our work, we concentrate

[8] There are a number of protocols that does a "Loc/ID split". For example, LISP, IVIP, HIP, ILNP, Shim6, Six/one Router, GSE etc. [46]

on the Locator ID separation protocol (or LISP) which follows the "map-and-encap" approach and is therefore discussed in detail. Discussion on address rewriting is beyond the scope of this work [43, 44] [9].

2.2.1 Map-n-Encap

The basic idea behind "map-and-encap" is to decouple the LOC space from the non-topologically assigned ID space (i.e. addresses in the ID space give no indication about the network topology and these addresses are assigned by regional registrars), which in turn should provide efficient aggregation in the LOC space.

In the "map-and-encap" scheme, when a packet is generated, both the source address (SA) and destination address (DA) are taken from the ID space (known as the "inner-header"). If the DA refers to a remote domain, then this packet at first traverses the local domain's infrastructure to a border router. It is now the border router's task to map the destination of the ID to an LOC. This LOC works as an entry point to the remote destination domain (hence the need for a mapping system). This is the "map" phase of "map-and-encap". The border router then encapsulates (i.e. appends a new header, which is referred to as the "outer-header") this packet and sets the DA as the LOC provided by the mapping system. Needless to say, this is the "encap" phase. It should be noted that, as the "encap" phase appends a new header to an existing IP packet, the "map-and-encap" scheme will work with both IPv4 and IPv6. When the packet arrives at the remote destination border router, it de-capsulates it and sends to the destination residing in that domain [43]. There is controversy as to whether or not the encapsulation overhead of "map-and-encap" scheme is problematic; arguments exist on both sides of the topic [60].

When the architecture of a mapping system for a "map-and-encap" scheme is being developed, three issues related to service scaling must be taken into consideration.

1. The rate of updates to the database,
2. The "state" required to be held by the mapping system and
3. The latency experienced during a map lookup.

By expert estimation, the size of the mapping database will be in the order of $O(10^{10})$, which implies that the update rate must be kept small to keep the latency in check. Also the architecture of the mapping system, whether it's a "push" or "pull", has an effect on latency. If a "push" strategy is taken where the entire database is pushed close to the edge (similar to BGP), latency is improved at the expense of increased state. Conversely, in a "pull" approach, a mapping request has to be sent out to find an authoritative server for that mapping (similar to DNS), which inadvertently increases latency. This analysis leads to the conclusion that, a "hybrid push-pull" approach might be most effective [43].

2.3 Locator/ ID separation protocol (LISP): A brief overview

Until now, we have explained in general the different concepts of "loc/ID split" implemented through a "map-and-encap" scheme. As mentioned in section 2.2, LISP

[9] http://blog.pattincon.com/internet-routing-table%E2%80%93the-good

employs the strategy of "map-and-encap" and from onwards we will only discuss specific details relevant to LISP.

The IDs that we have discussed thus far is referred to as Endpoint Identifiers (EIDs) in LISP terminology. This namespace is used to address end systems. The other namespace in LISP is known as Routing Locators (RLOCs) and has the same semantic as LOCs.

The edge of a LISP domain is defined by a Tunnel Router. This router is responsible for reading the EID address, checking its local RLOC to EID mapping cache and if the queried mapping is absent (i.e. a cache miss) then it interrogates the Mapping infrastructure to receive the desired RLOC (details about the LISP cache will be provided in the upcoming section 2.3.3). This is the "map" stage of "map-and-encap". In the "encap" phase, the border router encapsulates the packet with a UDP header and sets the destination address (DA) to the RLOC returned by the mapping infrastructure [44] [9].

The flow from the source towards the Internet i.e. the "map-and-encap" scheme is performed by the Ingress Tunnel Router (ITR), while the flow from the Internet towards the destination is de-capsulated by the Egress Tunnel Router (ETR) (Detailed descriptions of ITR/ETR is provided in the next section). The operation is illustrated below:

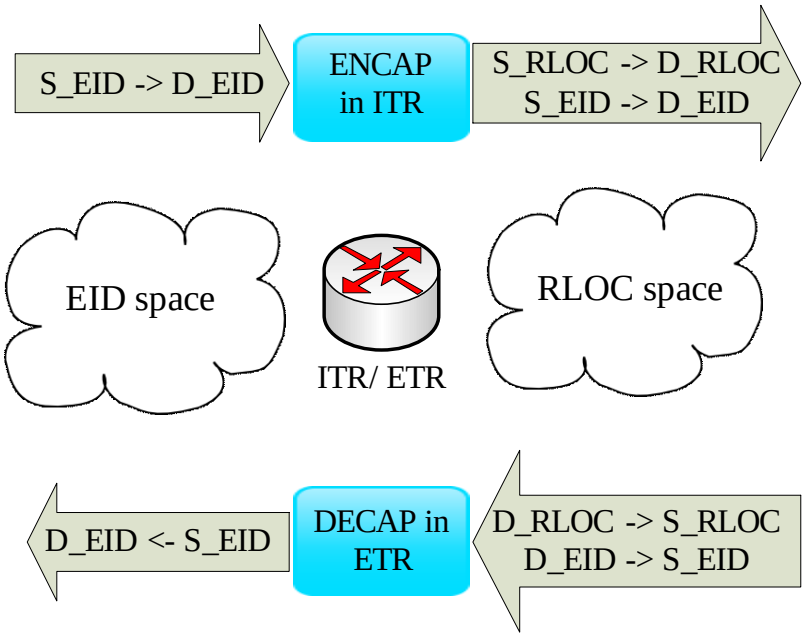

Figure 2.7: "map-and-encap" scheme. [9]

The ETR and ITR very rarely operate independently; the same device performs as an ITR while traffic is leaving the LISP site and works as an ETR for traffic destined for the site. Such an unified system is termed as an xTR [9].

In terms of LISP's architecture, this "map-and-encap" scheme is referred to as a "jack-up". Because the existing EID network layer is "jacked up" and a new network layer with RLOC addressing is inserted below it [13, 44]. The LISP "jack-up" is depicted in the next figure.

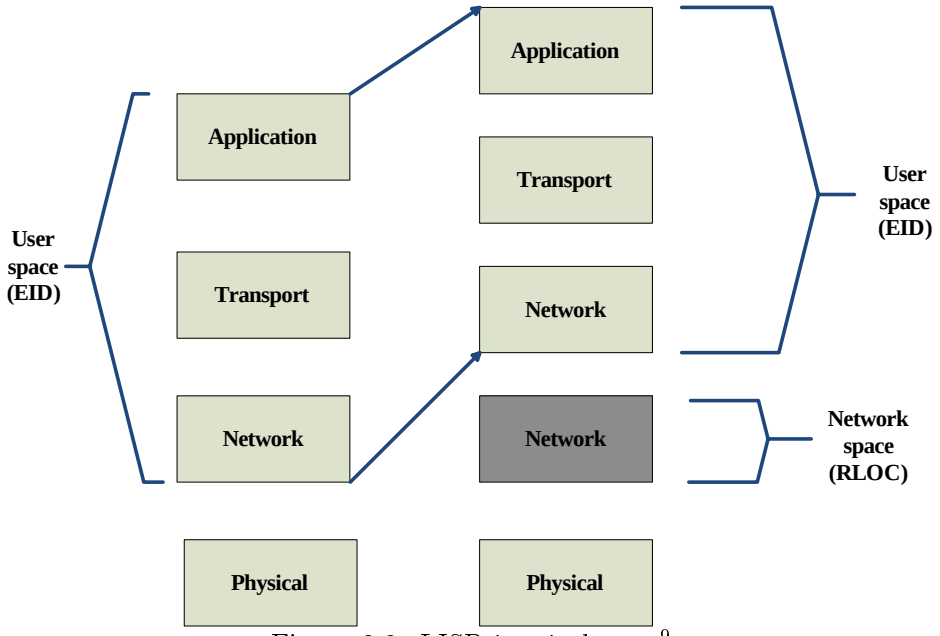

Figure 2.8: LISP is a jack-up. [9]

When it comes to the issue of EID-to-RLOC mapping system, though LISP specification defines the format of query and response messages, it makes no assumptions about the architecture of the mapping systems. As a result, several mapping systems have been proposed.

In LISP, the border router's upstream IP address is used as RLOC for the end-systems of that local domain. End-systems use EIDs for communication. While both EIDs and RLOCs are essentially IP addresses, EIDs only have a local scope (i.e. the local domain) and hence cannot be routed in the Internet. EID represents the ID of an interface (EIDs are usually looked up in the DNS by end-systems) and like MAC address; do not change even when the location of a device is changed. In other words, an EID is a PI (Provider Independent) address within the user site. When an upstream ISP changes, the EID does not change; only the mapping (EID-to-RLOC) changes.

As mentioned earlier, RLOCs are only used for routing purposes and they cannot be treated as EIDs for host-to-host communications in LISP-enabled domains. In most cases, A single RLOC is shared among many EIDs; which in turn keeps the RT size small. However, such a domain can also be multi-homed (i.e. a domain with several border routers), where one single EID can be serviced by several RLOCs [29]. In our opinion, LISP designers did a brilliant job in coming up with a solution that is network-based; i.e. no changes are required for the end-user. LISP also does not require any address changes on the user-site. It can successfully interoperate with the current Internet and could be, in future, deployed gradually. Unlike compact routing which proposes a complete new system (i.e. new Internet architecture) by theorizing a new addressing scheme for the whole network (LM based addressing), LISP takes a phased approach. In our opinion, it is this phased approach that will make LISP a success.

2.3.1 LISP Network Elements

The LISP specification [13] defines two network elements, namely, Egress Tunnel Router (ETR) and Ingress Tunnel Router (ITR). An ETR accepts an IP packet where the DA in the "outer-header" is one of its own RLOCs from the Internet. ETR then strips the "outer-header" and forwards the packet based on the next IP header (i.e. "inner-header")

found.

An ITR accepts IP packets from the local site's end systems on one side (i.e. IP packets without any LISP header) and sends out LISP-encapsulated IP packets toward the global Internet on the other side. ITR treats the "inner DA" as an EID and performs the necessary EID-to-RLOC mapping lookup. It then prepends an "outer-header" which contains one of its (i.e. ITR's) own globally routable RLOCs as the source address (SA) and the result of the mapping lookup is inserted into the DA field. Note that, the destination RLOC may turn out to be an intermediary or proxy device which has a better knowledge of the EID-to-RLOC mapping closest to the destination EID [44]. A LISP implementation also contains another infrastructure device called the LISP Map Server. It can perform two distinct functions. As a Map-Server (MS), it advertises EID prefixes into the mapping system (e.g. LISP-Alternative Topology) for ETRs that are registered to it. When acting as a Map Resolver (MR), it receives map requests from ITRs and forwards them (in case of LISP+ALT, the "forwarding" destination will be the ALT network). The MR also sends negative map replies to ITRs in response to queries for non-LISP addresses [10].

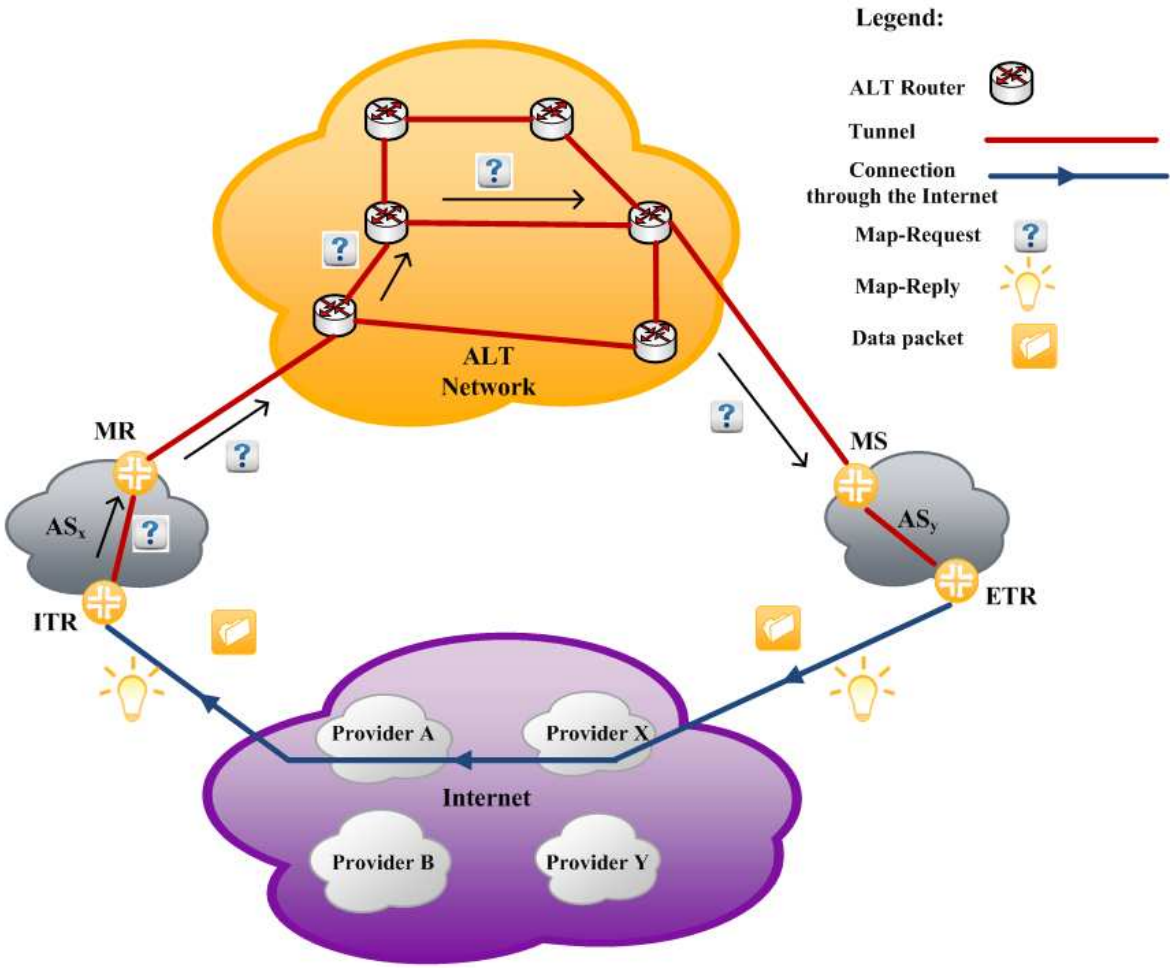

Figure 2.9: LISP architecture.

The figure 2.9 (simplified from [11]) gives a high level view of the LISP architecture. For demonstration purposes, we have assumed the mapping system to be LISP+ALT.

[10] Details of the mapping system is explained in the subsection 2.3.3.

[11] LISP: An Architectural Solution to Multi-homing, Traffic Engineering, and Internet Route Scaling. By Dave Meyer and Dino Farinacci. http://www.nanog.org/meetings/nanog45/presentations/Sunday/Dino_lisp_N45.pdf

2.3.2 LISP data plane Operation

When an end-system of a LISP-capable domain emits a packet, it puts its own EID as the SA of the packet, and puts the address of the correspondent node as the DA (Discovery of the EID of the correspondent node is done using Domain Name System [DNS]). If the destination of this packet resides in a remote domain then the packet traverses to one of the ITRs of the source domain. This ITR then checks the LISP map cache for an existing EID-to-RLOC mapping. Such caches are actually short-lived, dynamic, small and on-demand tables that are local to the ITR(s). It (i.e. LISP map cache) stores, tracks, and is responsible for timing-out and otherwise validating EID-to-RLOC mappings. If a mapping exists (in the LISP cache) then the packet is encapsulated using that map-cache policy and forwarded. If no mapping exists then it will be the LISP mapping system's responsibility to obtain it on behalf of the requesting ITR. When the map request reaches the destination ETR (through the mapping system), it (i.e. ETR) responds directly to the ITR with a map reply, which the ITR adds to its map cache. The ITR can now forward LISP packets between its EID (i.e. source) and the destination EID. When the LISP ETR receives the LISP packet, it de-capsulates it, and then forwards it to the original (EID) destination IP address. The encapsulation and subsequent de- capsulation creates a tunnel which is shown in the following figure [1].

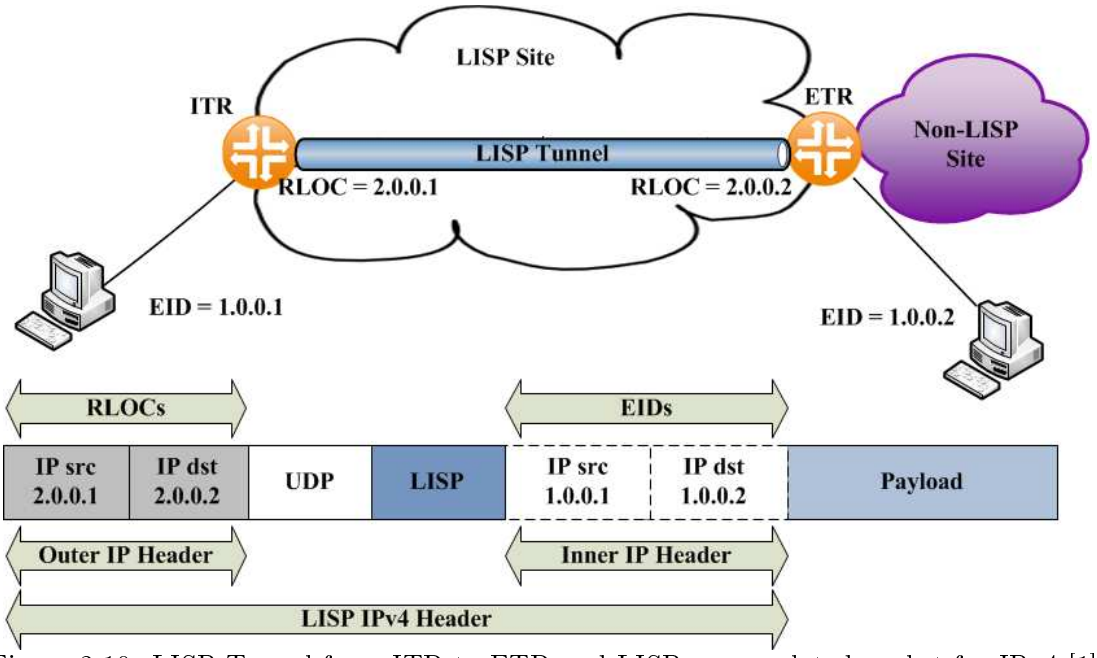

Figure 2.10: LISP Tunnel from ITR to ETR and LISP encapsulated packet for IPv4 [1].

The LISP IETF draft [13] specifies how the LISP packets should be encapsulated. What follows is the LISP header format and brief descriptions of the fields it contains, according to [13]. Note, the LISP header also comes in a IPv6 variety [12].

[12] Not shown here.

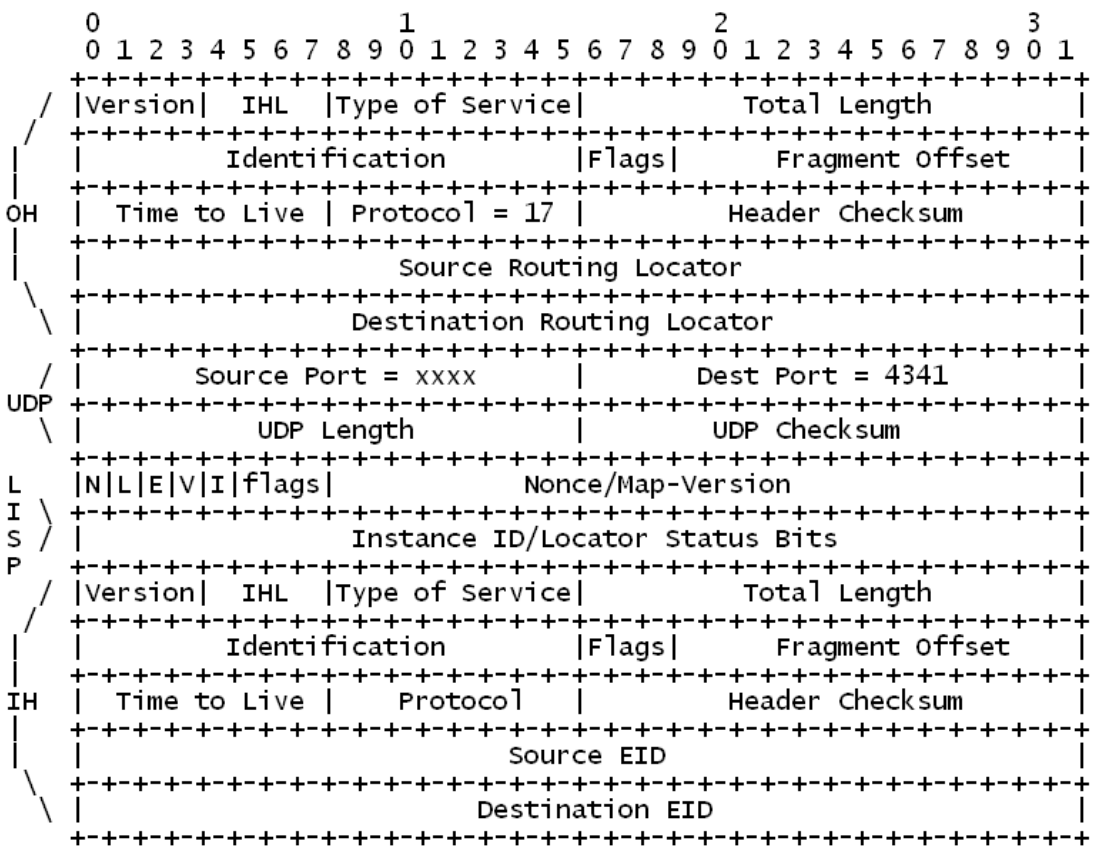

Figure 2.11: LISP IPv4-in-IPv4 Header Format [1].

Looking closely, this LISP header contains four distinct parts:

1. Outer Header (OH): OH's address fields contain RLOCs obtained from ITR's EID-to-RLOC cache.

2. UDP Header: The UDP header contains an ITR selected source port when encapsulating a packet. However, the destination port MUST be set to the well-known IANA assigned port value *4341*.

3. LISP: This part has five single bit fields. Obviously, for each of them 1 means presence and 0 means absent. The N bit is the nonce-present bit, the L bit is the Locator-Status-Bits field enabled bit, the E bit is the echo-nonce-request bit, the V bit is the Map-Version present bit, and lastly, the I bit is the Instance ID bit. The *flags* field is of 3-bit length and is reserved for future flag use. The *locator status* bits field is set by an ITR to indicate to an ETR the up/down status of the Locators in the source site. The *LISP Nonce* field is a 24-bit value that is randomly generated by an ITR when the N bit is set to 1. Detailed information regarding the configuration of these bit fields can be found in [13].

4. Inner Header (IH): IH is the header on the datagram received from the originating host. The source and destination IP addresses here are EIDs [13].

The tunneling scheme prevents EIDs from getting announced in the core Internet routing system. RLOCs are only announced to deliver packets correctly. This feature allows us to reduce the size of BGP's routing tables, which is a target of this protocol. Through figure 2.12, we take a closer look at an end-to-end packet delivery scenario.

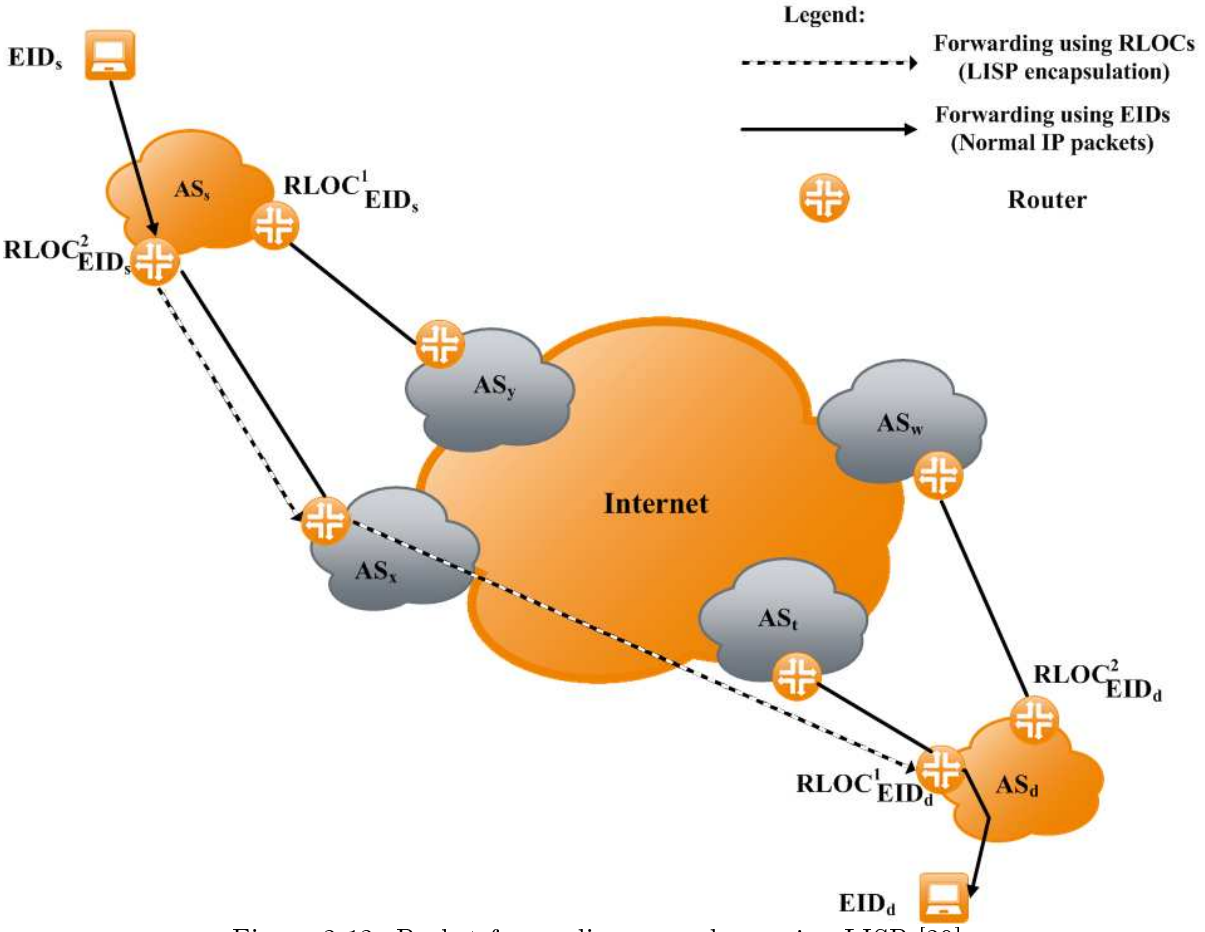

Figure 2.12: Packet forwarding procedure using LISP [29].

As seen in the figure, when EID_s starts communicating with EID_d, the initial step is to emit a first IP packet using its own EID (i.e. EID_s) as SA and the destination's EID (i.e. EID_d) as DA. Then using legacy IP routing protocols (e.g. RIP, OSPF etc.), this packet is routed inside AS_s until it is delivered to one of locators servicing EID_s. This however, does not imply source routing. The packet will reach a LISP ITR based on the destination's EID (i.e. EID_d). When the packet reaches $RLOC^2_{EID_s}$, this router will act as the ITR and encapsulate it inside a LISP packet. Now if it is determined that the best way to reach EID_d is through $RLOC^1_{EID_d}$ then a the new header prepended to the original packet containing $RLOC^2_{EID_s}$ as SA and $RLOC^1_{EID_d}$ as the DA. The packet is then routed in the Internet.

However, the question remains as to how LISP data plane figures out when to encapsulate an outgoing packet and what to put in the header [13], as well as when to de-capsulate an incoming packet. To perform these actions, two data structures are used, namely, the "local configuration" and the LISP cache. When an ETR houses the "local configuration", it becomes an authoritative ETR. The "local configuration" consists of all EID-Prefix [14] to RLOC mappings that are "owned locally" (i.e. mappings for those EID prefixes that are "behind" this router). For instance, in 2.12, the "local configurations" present in both xTRs of AS_s contain the following entry [11]:

[13] In case there are multiple ITR/ETR in the LISP site, choosing of SA is dependent on policy. The DA is however remains unaffected by any such policy.

[14] The term "EID-Prefix" refers to a power of 2 block of contiguous EIDs that can be aggregated in a single prefix.

$$\mathrm{EID_d}\text{ - Prefix : }\mathrm{RLOC}^1_{\mathrm{EID_d}}, \mathrm{RLOC}^2_{\mathrm{EID_s}}$$

It should be noted that, we have coined the term "local configuration" for two reasons. Firstly, this "local configuration" need not be a formal database. It can be generated manually as well. The second and most important reason is to avoid any ambiguity with the LISP database. The LISP database has a distributed structure and is not housed in any authoritative ETR. The mapping system as a whole forms the LISP database.

The "local configuration" is additionally utilized to determine which source RLOC (because ITRs also have "local configuration") should be used inside the outer header of an outgoing packet ($\mathrm{RLOC}^2_{\mathrm{EID_s}}$ is thus selected in 2.12).

The LISP cache is another data structure which holds the mappings for EID-Prefixes that are not "owned locally". While encapsulating a packet, this cache is used to provide the necessary information for selecting the destination RLOC. For example, in 2.12, in order to encapsulate a an outgoing packet, the ITR of ASs needs to have the following mapping in its cache:

$$\mathrm{EID_d}\text{ - Prefix : }\mathrm{RLOC}^1_{\mathrm{EID_d}}, \mathrm{RLOC}^2_{\mathrm{EID_d}}$$

The actual selection of the RLOC to use in the outer header is done based on the priority and weight associated with each RLOC (LISP does this association). The exact procedure by which this selection takes place is explained in the original LISP proposal [13] and is outside the scope of this work. It should be noted that, these mappings inside LISP cache are short-lived and are subject to timeout. When a mapping is not used for a certain period, that entry gets timed out (i.e. removed). In other words, entries in the LISP Cache is populated in an on-demand basis.

2.3.3 LISP control plane

Before going into the functionalities of the control plane, a brief introduction of the three types of packets defined by LISP specification [13] for supporting an EID-to-RLOC mapping system (which resides in the control plane) is warranted. The first type is the Map Request (de facto choice) which an ITR uses to query for a particular EID-to-RLOC mapping. Subsequently, the authoritative ETR (it contains the authoritative EID-to-RLOC mappings for a particular LISP site) responds with a Map-Reply message to the querying ITR. The second type of packet is the Data Probe packet (rarely used) which an ITR may send to the mapping system for probing. As before, the authoritative ETR responds with a Map-Reply message. To determine whether a packet is Data Probe or not, the ETR checks the "inner" and "outer" header's DA. If they are the same and is an EID then it's a Data Probe packet. The third type is the Map Reply packet and its possible uses are already explained [44].

The first packet of a new flow will trigger a cache miss (i.e. mapping entry is absent from LISP cache) and will in turn cause the data plane to ask the control plane to retrieve a mapping for that specific destination EID that had originally triggered the cache miss. In other words, the control plane is concerned with how an ITR maps a destination to an ETR. To do so, two primary tasks must be finished.

1. At first, an ETR needs to inform the network (i.e. to register into the mapping system) that it will provide "service" (i.e. an EID prefix is reachable through that ETR's RLOC) for a particular EID space.

2. Secondly, an ITR needs to look up the required ETR (called the authoritative ETR) for a destination EID.

The first task is performed using a MS. In a general deployment, the ETR registers (using a Map-Register message) its EIDs with the MS. For the second task, a mapping distribution protocol/ mapping system is used to provide a lookup infrastructure for retrieving mappings [11] from a Mapping Resolver (MR). In other words, a mapping system provides the association between an identifier and its locators. As previously mentioned in section 2.2, a mapping system for LISP can be either adopt "push" or "pull" based strategy. In a push-based systems, the ITR receives and stores all the mappings for all EID prefixes, even if it does not contact them. NERD is the only proposed push-based mapping system. In contrast, an ITR in a pull-based system, sends queries to the mapping system every time it needs to contact a remote EID and has no mapping for it [31]. LISP-CONS, LISP-DHT etc. are examples of pull-based mapping systems. The LISP + ALT (LISP Alternative Topology) [17] proposal however, is a "hybrid push-pull" scheme. It has become the prevalent LISP mapping system, because this solution is adopted by the LISP working group and is currently being deployed in the international test bed [11]. Short descriptions of the above mentioned mapping systems are given here.

2.3.3.1 LISP-ALT

As stated previously, Cisco's endorsement has made LISP-ALT the preferred mapping system for LISP. Therefore, we need to have a clear understanding of it.

As a mapping system, the LISP Alternative Topology (LISP-ALT) is distributed in a virtual overlay network. Like all other mapping system, LISP+ALT is used by an ITR or MR to find the particular ETR that contains the desired EID-to-RLOC mapping. The overlay is called the ALT network and is made up of tunnels (e.g. GRE [15] tunnel) between ALT Routers. As visualized in figure 2.13, these ALT routers are associated with EID prefixes and may be connected in a hierarchical manner with respect to these prefixes. If so, then the bottom-most ALT routers must be connected for each of its associated EID prefixes, to at least one authoritative ETR (i.e. ETR that "owns" that particular EID-prefix). ALT routers communicate with its peers via BGP and exchange aggregated EID prefixes that can be reached through them. In contrast to normal inter-domain routing, ALT routers may aggregate the prefixes that are received via BGP before forwarding them [42].

[15] Generic Routing Encapsulation or GRE is a tunneling protocol that was originally developed by Cisco for IP-in-IP tunneling.

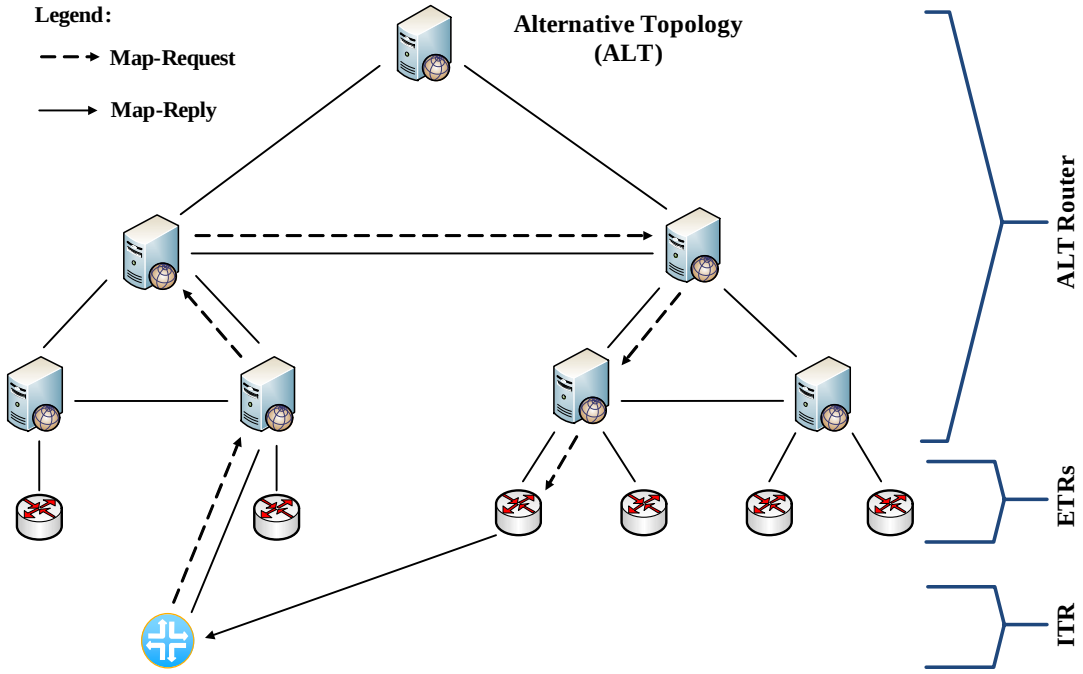

Figure 2.13: LISP ALT in a semi-hierarchical structure [42].

An ITR uses the ALT network to send an ALT datagram [16] to the authoritative ETR that "owns" the requested EID-prefix. In most cases, an ITR does not connect directly to the overlay network but instead sends the ALT datagram via a MR. Based on the destination address, an ALT router will forward the received ALT datagram to the proper ALT routers according to the prefix information provided via BGP [61]. Eventually the ALT datagram reaches the appropriate ETR, which then responds with a Map-Reply (containing the desired mapping information) that is sent directly to the ITR. This obtained mapping information is then stored in the LISP Database. This is referred to as the "pull" operation. In a similar fashion, when an ETR registers its EIDs, it utilizes the help of a MS (instead of connecting directly to the ALT network). This piece of communication with the MS is termed as a "push" (i.e. "pushing" the EID registration information towards the ALT network). It is the MS's responsibility to advertise these registered EID-prefixes on to the ALT network and also to forward the ALT datagram toward the authoritative ETR. This is why, LISP-ALT is said to have a "hybrid push/pull" architecture.

For the sake of clarity, it must be mentioned that, the ALT network does not distribute actual EID-to-RLOC mappings. It only makes a forwarding path through ALT routers using BGP (ALT routers interconnect using BGP and propagate EID-prefix updates among themselves) from the MR/ITR (i.e. MR and its associated ITR) which requires an EID-to-RLOC mapping, towards an authoritative ETR that holds this requested mapping. The MR/ITR uses this path to send the ALT Datagram towards the authoritative ETR. The Map-Reply however is sent directly to the ITR. Because originally the ALT datagram had the ITR's RLOC as the SA in its outer header. This enables the authoritative ETR to circumvent the ALT network and communicate directly with the ITR [29].

2.3.3.2 LISP-DHT

A key problem faced by LISP, is that the mapping system must be able to distribute mappings between identifiers and locators in a scalable manner. LISP-DHT fulfills this

[16] The term "ALT datagram" refers to a Map-Request that will be sent or forwarded to the ALT.

need by using a DHT lookup infrastructure which enables mappings to be retrieved in an efficient manner. This mapping system uses an overlay network that is derived from Chord. Normally, Chord DHT tends to randomize which node is responsible for a key-value pair (in this case, the key-value pair is a mapping). In LISP- DHT however, Chord is suitably modified so that, a node is associated with an EID prefix and that node's Chord identifier is chosen at bootstrap as the highest EID (in that associated EID prefix). This is done to preserve the locality of the mapping, i.e. a mapping is always stored on a node chosen by the owner of the EID prefix. When an ITR needs a mapping, it sends a Map-Request through the LISP-DHT overlay with its RLOC as SA. Each node routes the request according to its finger table (a table that associates a next hop to a portion of the space covered by the Chord ring). The Map-Reply is sent directly to the ITR(by using this ITR's RLOC) [31, 40].

2.3.3.3 LISP-CONS

The Content distribution Overlay Network Service for LISP or LISP-CONS operates through a distributed EID-to-RLOC database. This database is distributed among the authoritative Answering Content Access Resources (Answering-CAR). An Answering-CAR (aCAR) advertises "reachability" for its EID-to-RLOC mappings through a hierarchical network of Content Distribution Resources (CDRs) (but, not the mapping itself), and responds to mapping requests from the system. A CAR may also request mappings from the system (this entity is now called a Querying-CAR, or qCAR). ITRs connect to one or more qCARs to query the system for the required EID-to-RLOC mappings. This qCAR then queries the system on behalf of the ITR. These queries follow the overlay network to the authoritative aCAR, which responds with the mapping. This response may then be cached by the "local" CAR. Finally, note that neither a qCAR or aCAR need to hold the entire EID-to-RLOC database. Rather, the EID-to-RLOC mappings are separately pulled by the ITRs by querying one or more of its connected qCARs. To the best of our knowledge LISP-CONS has stopped evolving [5, 31].

2.3.3.4 NERD

NERD (Not-so-novel EID RLOC Database) is based on a monolithic database that is present on each xTR and is refreshed at regular intervals, containing all available mappings (assumed to be published by a centralized authority). This means that LISP-NERD follows a "push" distribution model, as it deliberately "pushes" all available mappings toward all existing xTRs. This also implies that, NERD does not cause any cache miss, which is a major advantage. Additionally, this strategy offers the advantage of reducing the signaling overhead; because LISP-NERD uses a HTTP-based incremental updates approach. However, LISP routers need to store all existing mappings (even the ones that are never used) and thus severely limiting its scalability. Furthermore, the bootstrap operation can become very slow, since the whole database needs to be downloaded. Additionally, any changes require a new version of the database to be downloaded by all ITRs, making it an unlikely mapping system of choice for the future global deployment of LISP [31, 40].

2.3.3.5 FIRMS

Menth et al.'s [41, 42] Future Internet Routing Mapping System (FIRMS) is a fast, scalable, reliable and secure mapping system for LISP. One of its strong suits is that,

it can relay the initial packet in case of a cache miss. Our interest in FIRMS lies in the fact that, it has conceptual similarities with our proposed CRM.

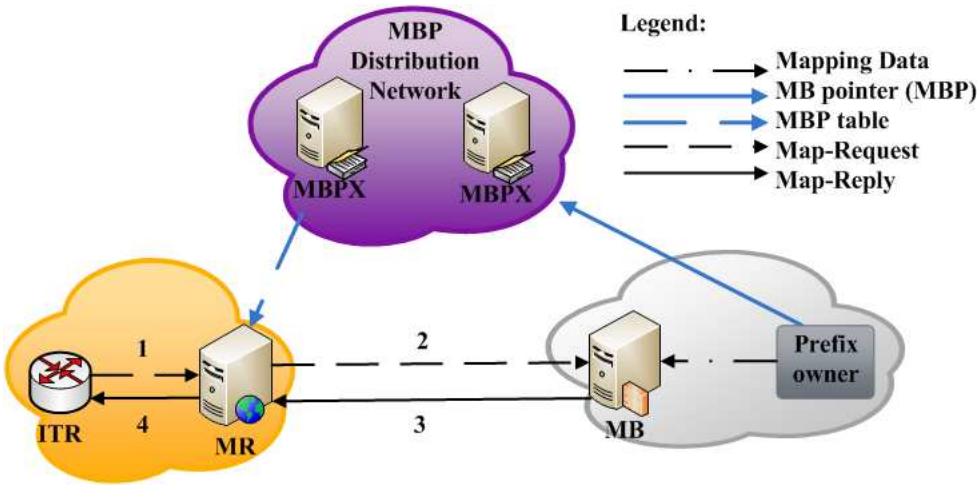

Figure 2.14: Basic operation of FIRMS [41].

The above illustrated figure 2.14 gives an idea about the basic structure and operation of FIRMS. *Menth et al.* [41, 42] assumes that, the EID prefix blocks are assigned to their "owners". Each prefix "owner" provides a Map-Base (MB) that houses the EID-to-RLOC mappings for all of its EIDs. The Map-Base Pointer (MBP) is a data structure that contains information about the MB. The prefix "owner" will register this information in the global MBP distribution network which collects all MBPs and through it constructs a global MBP table. Each ITR is also configured with a MR. The MR registers with the MBP distribution network to receive a copy of the global MBP table. When the ITR requires an EID-to-RLOC mapping for an EID, it sends a map-request to its MR. The MR looks up the address of the responsible MB in its local copy of the MBP table and forwards this Map-Request to that MB. The MB returns with a Map-Reply containing the desired EID-to-RLOC mapping to the MR which in turn is forwarded it to the requesting ITR. If a non-existing mapping is queried then a negative Map-Reply will be returned. This design requires that MRs and MBs have globally routable RLOC addresses [41, 42].

One of the ideas suggested by Menth et al. [41, 42] for the improvement of FIRMS, resemble with the design principle of our proposed CRM. According to *Menth et al.*'s [41, 42] scheme, if EIDs of a common prefix block share the same RLOC then their EID-to-RLOC mappings should be aggregated to a single EID-to-RLOC mapping. This will save storage space of caches and databases. This strategy will also make additional Map-Requests redundant when the ITR needs a mapping for a new EID that is already covered by an aggregated EID-to-RLOC mapping in its cache. Thus an aggregated EID-to-RLOC mappings minimizes the lookup delay and takes the load off the MR and the MB [41]. We will see later on in section 4.2.2.1 that, one of the first tasks that is performed by a LM of our experimental CRM is to aggregate the received EID-to-RLOC mappings if possible.

2.4 Importance of OpenLISP:

In order to successfully implement and deploy a system based on the Locator/ID Separation paradigm, it is necessary to map IDs into locators, store and distribute these mappings, and perform tunneling or address translation operations to forward packets in the core Internet. Several solutions have been proposed insofar, however, while interesting and promising, these have the drawback of taking a "clean slate revolutionary approach". In other words, they are not incrementally deployable and thus cannot be experimented on a large scale. LISP is the only evolutionary approach that does not rely on changes at the end-hosts by introducing heavy modification to the protocol stack. It is so designed

that it can be incrementally deployed while causing minimum disruption [29].

OpenLISP [28] is the only open-source implementation for FreeBSD, based on the LISP draft (version 07) by *Iannone et al.* [29] and to the best of our knowledge, it is currently the only major implementation besides the NX-OS of Cisco. OpenLISP has a "dirty slate approach" (i.e. evolutionary approach). Its primary aim is to provide an open and flexible platform for experimentation in the locator/ID separation paradigm. OpenLISP is much more than just an implementation of the LISP draft. While the LISP draft presents a detailed description of the encapsulation and de-capsulation operations, the forwarding operation and offer several options as mapping system; it does not provide any specification for an API that allows the Data plane to interact with the Control plane (i.e. mapping system). OpenLISP however, provides a new socket based solution, called Mapping Sockets, to overcome this handicap [27]. This solution is particularly important for our purposes because we are implementing and deploying a mapping system for the Control plane (i.e. CRM). Mapping sockets make OpenLISP an open and flexible solution, enabling us to pair up CRM with it. Additionally, the development and the experimentation done with OpenLISP also had an impact on the original LISP specifications, allowing corrections to some original design shortcomings and improve some engineering solutions. Therefore, it is imperative that we understand the OpenLISP architecture in a detailed manner [27].

2.5 OpenLISP: An Architectural Overview

As shown in the high level architecture of OpenLISP (See: figure: 2.15), *Iannone et al.* [29] mainly concentrated their efforts on the LISP data plane which constitutes the newly added/slightly modified routines for encapsulation, de-capsulation and an unified data structure called MapTable. MapTable is a radix tree data structure created by merging LISP cache and LISP database. Though unified, from a logical point of view, they are still separated; because the EID-Prefixes that are "locally owned" (i.e. part to the LISP database) are tagged with a "database" flag. According to *Iannone et al.* [29], they implemented the data plane directly into the kernel protocol stack; in order to figure out the how a protocol stack would look like once LISP becomes a part of it. However, for the control plane, the authors of OpenLISP deliberately did not implement any specific Mapping Distribution Protocol. Rather, they came up with "mapping sockets" that are essentially APIs (not defined in the original LISP specification) for communication between the data and control plane. In the following subsections, we will summarize the above mentioned features of OpenLISP.

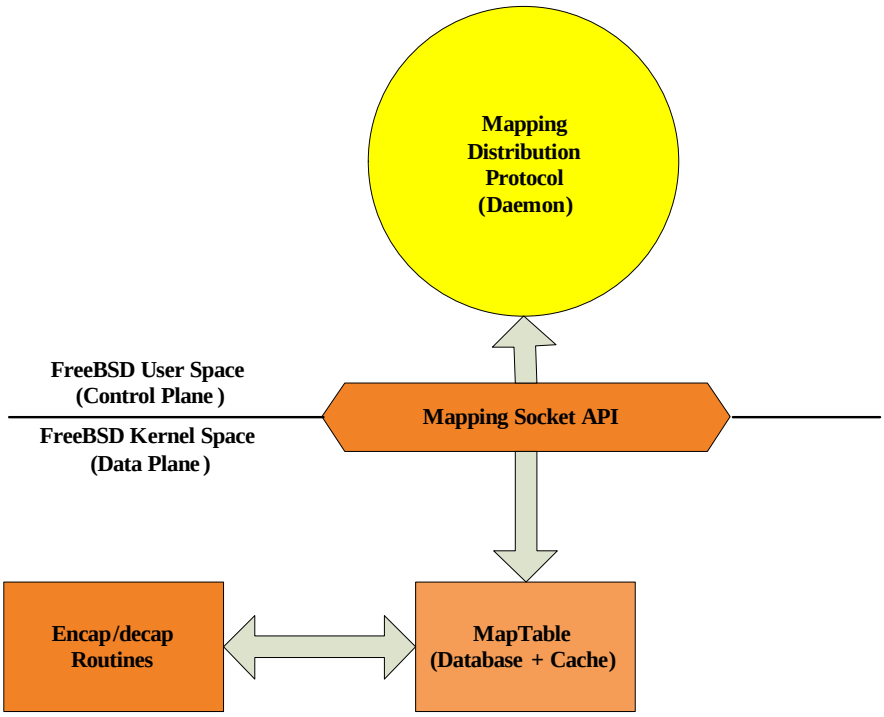

Figure 2.15: OpenLISP architecture [29].

2.5.1 OpenLISP data plane

As stated earlier, OpenLISP data plane is formed of encapsulation and de-capsulation routines and MapTable. In the following sub-subsections we describe them briefly.

2.5.1.1 Encapsulation and De-capsulation routines

Packet encapsulation and de-capsulation in OpenLISP is implemented as a patch on the IP protocol stack of FreeBSD kernel. This patch involves the addition of four new functions: namely, *lisp_ input()*, *lisp6_ input()*, *lisp_ output()*, *lisp6_ output()* and calls to these functions from the pre-existing code.

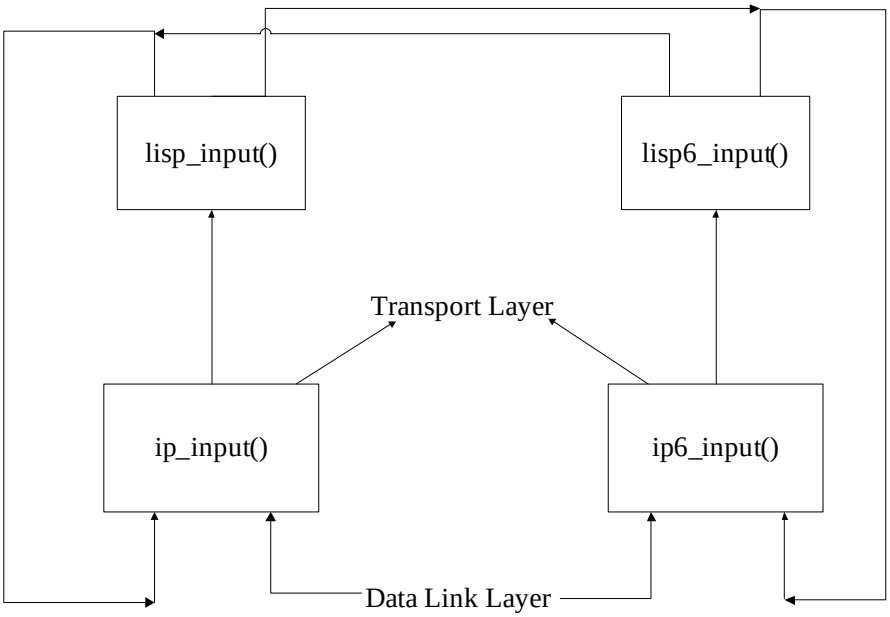

Figure 2.16: Protocol Stack Modifications for incoming packets [29].

As the names suggest, *lisp_ input()* and *lisp6_ input()* manage incoming IPv4 and IPv6 LISP packets and are positioned right above *ip_ input()* and *ip6_ input()* respectively, from which they are called. As shown in Figure. 2.16 [29], an incoming IPv4 LISP packet is at first handled by the *ip_ input()* routine, which has been suitably patched to recognize LISP packets. If it turns out to be a LISP encapsulated packet, only then *ip_ input()* will call the *lisp_ input()* function. *lisp_ input()* strips the outer header of the data packet and re-injects it in the IP layer, by putting it in the input buffer of either *ip_ input()* or *ip6_ input()*.Once the packet has been re-injected in the protocol stack, if it is determined that, the system de-capsulating the packet is not the final destination then the packet is delivered to *ip_forward()* or *ip6_forward()* function (depending on the IP version number) which sends it down to the data link layer and is subsequently transmitted toward its final destination.

While handling outgoing packets, *ip_ output()* routine checks whether the packet needs to be encapsulated with a LISP header or not. This checking is done by a twofold look up procedure. At first, using the packet's SA (i.e. source EID), *ip_ output()* looks into the LISP database for a valid mapping. If no mapping is found, then that packet is normally processed by *ip_ output()* without any encapsulation. If a mapping is found during the first round of search, then a second lookup is performed using the DA (destination EID) of that packet for a valid mapping inside the LISP Cache. If there is no valid mapping in the LISP Cache, then it means a cache miss has occurred and a message is sent through open mapping sockets to notify the control plane. If a mapping for the destination EID is present then that packet is diverted towards the *lisp_ output()* routine, which at first performs MTUs checks and then encapsulates the packet by selecting the RLOCs to be used. The above mentioned procedure is depicted in figure 2.17 [29].

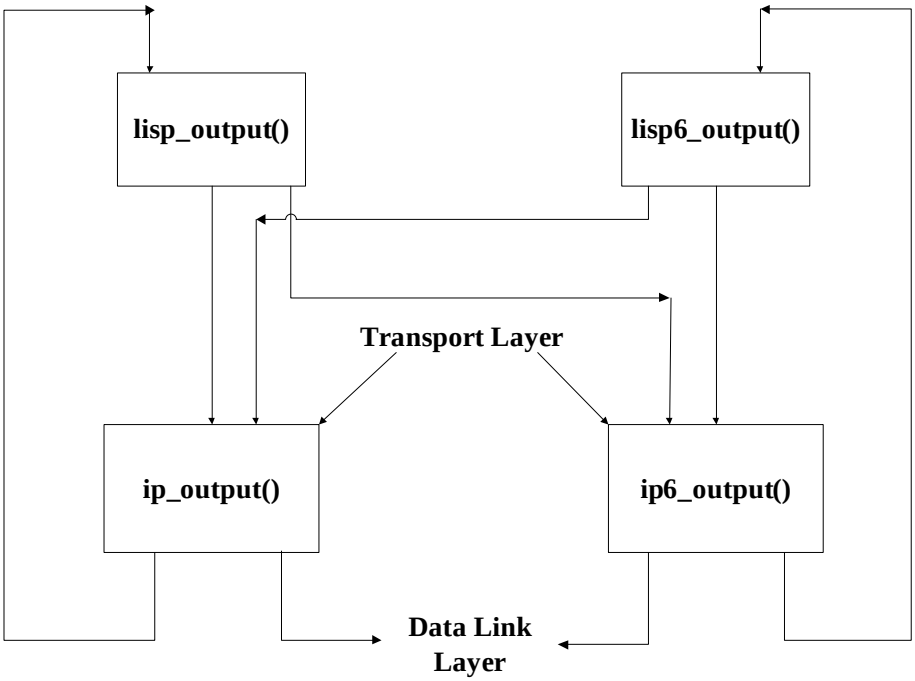

Figure 2.17: Protocol stack modifications for outgoing packets [29].

2.5.2 Map Tables

As described earlier in section 2.5, OpenLISP has a merged radix tree data structure (actually, there are two radix structures in the system, one for IPv4 EID-Prefixes and another for IPv6 EID-Prefixes) formed of LISP database and LISP Cache, called Map-Table. Additionally we have said that, the "database" flag creates a logical separation between the LISP database and LISP cache, enabling us to lookup only those entries that have their "database" flag set to a particular value.

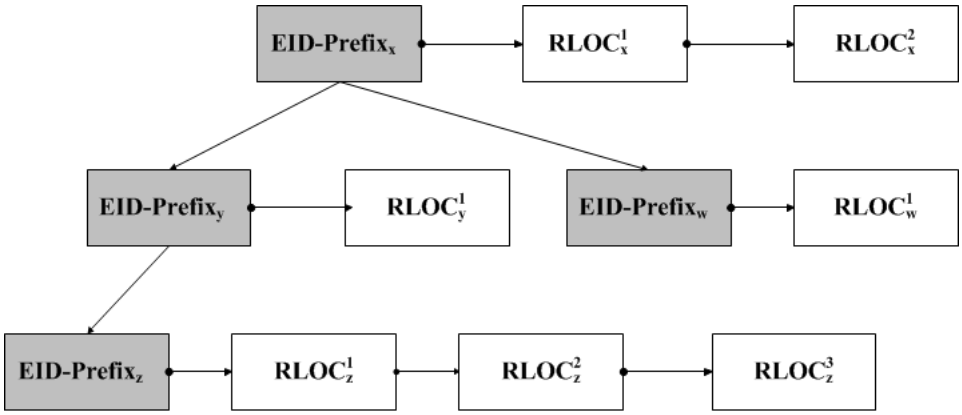

Figure 2.18: Example of MapTable data structure layout [29].

As shown in figure 2.18 [29], each entry contains a pointer to a socket address structure that holds the EID-Prefix to which the entry is related, a "flags" field (not visible in the figure), and a chained list of RLOC data structures that contain the RLOC addresses and their related metrics (each entry also contains fields necessary to build the radix tree itself whose description is outside the scope of this article). The *"flags"* field contains general flags (e.g. *"database"* flag) that apply to the whole mapping. OpenLISP deliberately uses a chained list to store RLOCs to enable a "mixed" use (i.e. IPv6 encapsulation of IPv4 packets and vice versa) of IPv4 and IPv6 RLOCs. Using the criteria described in [13],

this chained list is always kept ordered, so that the most preferable RLOCs remain at the head. This strategy enables OpenLISP to avoid scanning during encapsulation [10, 29].

2.5.2.1 MTU Management

Each RLOC in this chained list is also associated with an *MTU* field that contains the MTU setting of the interface (i.e. outgoing interface) for that RLOC. When a packet is received for LISP encapsulation, the MTU size of that RLOC's outgoing interface is calculated. If the packet size exceeds the MTU, then an ICMP "too big" message is send and the packet is dropped. Because of this implementation, a packet will never exceed the maximum interface MTU value. OpenLISP does not implement a packet fragmentation system [10, 29].

2.5.3 OpenLISP Control Plane

As explained earlier, *Iannone et al.* [29] did not implement any specific Mapping Distribution Protocol, instead, they provided two simple tools, namely map and mapstat, so that OpenLISP data plane can be accessed from a shell terminal. The *map* utility (based on the route utility) provides a command-line interface to add, remove, and view mappings from MapTables. It is utilized to specify any arbitrary request which is delivered via the mapping sockets API described in the next section. On the other hand, the *mapstat* command (based on the *netstat* utility) allows users to retrieve and display various LISP statistics (e.g. dump of the content of the MapTables, show the statistics concerning encapsulation and de-capsulation operations etc.) [10, 29].

2.5.3.1 Mapping Sockets API:

In order to communicate between LISP data plane and LISP control plane, *Iannone et al.* [29] defined "mapping sockets" that are based on UNIX style raw sockets in the newly defined AF_MAP domain. These new type of sockets allow mapping distribution protocols running in the user space to send/receive messages to/from the kernel space while performing operations such as, modification of the kernel's data structure (e.g. MapTables). Mapping sockets also offer signaling functionality, i.e. to allow the kernel to notify user space daemons when specific events related to LISP (e.g. cache miss) occurs. Like routing sockets, mapping sockets broadcast, i.e. when messages are sent from the kernel to the user space, they are delivered to all open sockets. This enables all the processes dealing with LISP to be notified after specific events occur or when changes are made by one of these processes.

If a process has opened a mapping socket then it can utilize the following operations on the MapTables:

- MAPM_ADD: Used to add mappings to the table and read result from the kernel.
- MAPM_DELETE: Used to delete mappings from the table.
- MAPM_GET: is used to retrieve a mapping. A specific EID should be given as the query.

Mapping sockets also allows the OpenLISP kernel patch to trigger messages, e.g., MAPM_MISS when there is no mapping available.

2.6 Dissemination of End Point reachability

At the heart of our experimental mapping system is the advertisement, which consists of EID-prefix and the corresponding RLOC (through which the EID-prefix is made reachable). Obviously, BGP is the chosen for the inter-domain advertisements and Quagga is selected as the routing software that implements BGP.

The following sub-sections are organized in a way where we at first provide a brief description on how BGP works. Then we move on to Quagga's architecture and lastly, we explain our reasoning for the chooses we had made regarding advertisement in our work.

2.6.1 Routing protocols: terminologies explained

Routing protocols are typically categorized as either Interior Gateway Protocols (IGPs), used for exchanging information inside an AS (e.g. OSPF, IS-IS, RIP and EIGRP) or an Exterior Gateway Protocol (EGP) used for exchanging information between ASes (e.g., BGP). Both IGPs and EGPs share the common purpose of exchanging routing information between routers, but differ in their features and performance. Each router can implement multiple protocols simultaneously. One single router can even house multiple instances of the same protocol. To maintain boundaries on how routing information is shared, each routing protocol is treated as a separate process in the router.

It should be noted that, by default, no information is exchanged between the aforementioned routing processes. Each routing process is associated with one or more interfaces on the router. For two routing processes running on different routers to directly exchange routing information, those processes must be "adjacent". The definition of "adjacent" depends on the routing process. Two BGP processes are treated as "adjacent" if those processes are explicitly configured to speak to each other and if it is possible to open a TCP connection between those two routers. For OSPF, IS-IS, RIP, or EIGRP processes to be "adjacent", the conditions are:

1. These processes must be of the same type,
2. There must be a link between the routers on which these processes run and
3. Each process must be configured to cover the interface at its end of the link [38].

2.6.2 BGP: the basics

Now in its fourth version, the Border Gateway Protocol (BGP) is an inter-AS (i.e. inter-domain) routing protocol. It is classified as a path-vector routing protocol. Such a protocol maintains the path information that gets updated dynamically. Therefore, updates that have looped through the network and returned to the same node are easily detected and discarded. As a path vector protocol, BGP exchanges entire paths to any given destination between neighbor ASs. BGP includes mechanisms for policy based routing, and defaults to shortest path routing if routing policies are not enforced [16].

Here, each BGP speaker [17] calculates the preferred route for a target IP addressed network and passes that route to its neighboring BGP speakers. With this exchanged information, AS connectivity is constructed which in turn prunes routing loops and make policy decisions that will be enforced at the AS level. BGP runs over TCP. As TCP is a

[17] Any node/router running BGP as its routing protocol is referred to as a BGP speaker or BGP system.

reliable transport protocol for establishing and maintaining connections; it eliminates the need for retransmission, acknowledgement and sequencing of packets [51, 57].

An AS uses Interior Gateway Protocol (IGP) along with some common metrics to route packets within the boundaries of that AS. In order to route packets to other ASes, it uses BGP. BGP speakers connected within an AS constitute "internal peers"; while the BGP speakers that are connected across ASes form "external peers" [51].

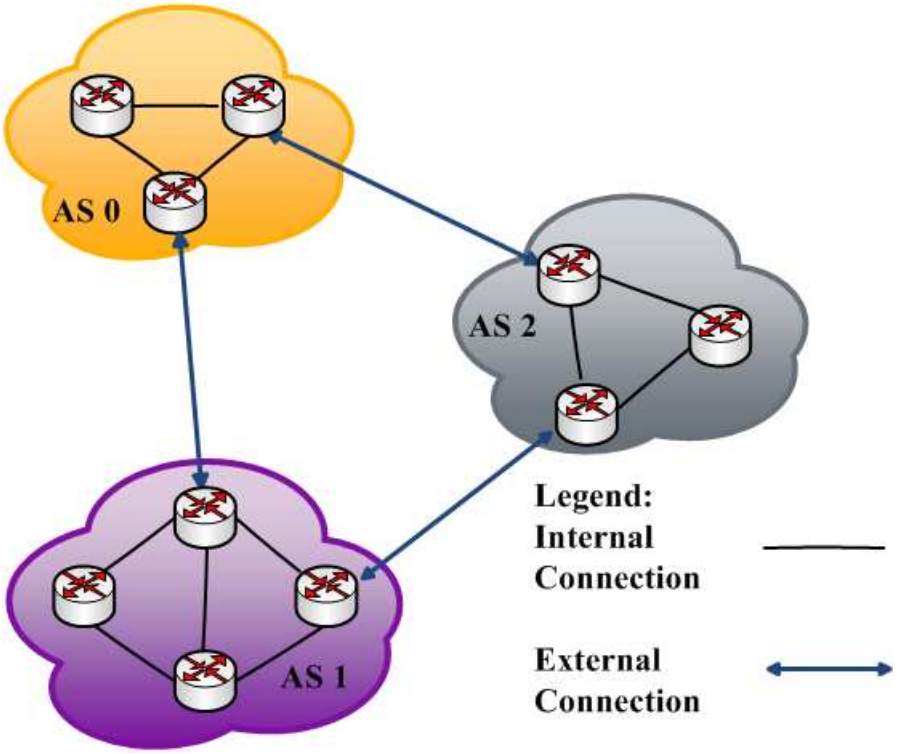

Figure 2.19: BGP terminologies [51].

Figure 2.19 shows the terminologies associated with BGP. ASs 0, 1, and 2, each contain a set of BGP Speakers. As depicted in the diagram, any router maintaining a external connection (to its AS) is termed as a Border Router. Border Routers of ASs 0, 1, and 2 have external connections and thus form BGP External peers for this topology. BGP Internal peers may maintain a meshed topology when the network is small enough, as indicated by the figure [51] . In a larger network, mesh topology might cause scalability issues. Therefore, route reflectors (or to a lesser extent confederations) are used to avoid the mesh topology.

BGP4 is also a classless routing protocol. This means that, it uses Classless Inter-Domain Routing (CIDR) address notation regardless of whether the AS is running a class-full or classless IGP. BGP4 takes both class-full and classless route information from IGP and advertises the addresses using CIDR address notation. This allows BGP4 to advertise any AS address and also aggregate multiple consecutive AS addresses together, which leads to the potential minimization of routing table entries. Hence, if two downstream ASs are advertising 173.29.1.0/24 and 173.29.2.0/24 respectively, then if policy allows then BGP4 can aggregate it into 173.29.0.0/22 and then advertise it upstream [18].

[18] Routing With BGP4, White paper. http://www.alcatel-lucentbusinessportal.com/private/active_docs/OAWAN600_BGP4_wp8.23.pdf

2.6.3 BGP capabilities negotiation

The capability negotiation with BGP-4 is achieved through a relatively new optional parameter called Capabilities. Its purpose is to provide BGP4 with the capability to negotiate without requiring the termination of BGP peering. If a BGP speaker supports capabilities negotiation then, when it sends an OPEN message to a BGP peer, the message may include an optional Capabilities parameter. The parameter lists the capabilities supported by the speaker. A BGP speaker examines the information inside the Capabilities parameter of an OPEN message to determine which capabilities the peer supports. If a BGP speaker determines that a peer supports a given capability, then the speaker can use that capability with its peer.

A BGP speaker determines that a peer doesn't support capabilities negotiation (i.e. does not implement the Capabilities parameter) if in response to an OPEN message that carries this optional parameter, the speaker receives a NOTIFICATION message that contains an error sub-code set to "Unsupported Optional Parameter". If this occurs, the BGP speaker should attempt to re-establish the connection without sending the Capabilities optional parameter to that peer.

If a BGP speaker that supports a certain capability determines that its peer does not support that particular capability then the speaker may send a NOTIFICATION message to the peer and terminate peering. This message contains the capability (or capabilities) that causes the speaker to send this message. The decision to send this message and terminate peering is local to the BGP speaker [21].

2.6.4 BGP messages

BGP uses TCP connections to create a reliable environment for exchanging routing information. However, there is a subtle difference between BGP's TCP connection and TCP connection made for other application programs. Once established, BGP's TCP connection can last for a long time (unless something unusual/ error occurs). For this reason, BGP's TCP connection is sometimes termed to as a semi-permanent connection [16].

To create neighborhood relationship, a BGP speaker opens a TCP connection with a neighbor and sends an OPEN message. If the neighboring BGP speaker accepts this relationship then it responds with a KEEPALIVE message, which means that a relationship is established between the two routers. KEEPALIVE messages are also sent periodically to ensure the aliveness of the connection. Subsequently, if there are changes in the topology that results in changes to RTs, incremental UPDATE messages are exchanged between the BGP speakers. UPDATE messages are used by a BGP speaker to withdraw previously advertised routes, announce a new route to a destination or do both. Notification messages are sent in response to error conditions or when a BGP speaker wants to close the connection [16, 51, 57].

2.6.5 Route selection and BGP Routing Information Bases (RIBs)

Before describing the specific details of BGP RIBs, we will explain the relationship between routing process RIBs, route creation, route redistribution, and the router RIB that stores routes used to forward packets.

A route can be modeled as an IP subnet address (e.g. 11.0.0.0/8) along with some

additional attributes (e.g. weights, AS path etc.) that the router may use to calculate a next-hop to reach that subnet. A router can learn about a route in several possible ways. Routes to all the directly connected subnets are always available to the router or routes can also be configured manually (e.g. static routes). However, through the use of routing protocols, routes can be learned dynamically. Different protocols exchange different types of routing information to convey routes between routing processes, e.g. OSPF and IS-IS use link-state advertisements; while BGP uses path-vector records. At the end, all the processes learn the routes [38].

The basic design of a router can be abstracted by the model depicted in figure 2.20, where each routing process maintains its own RIB (where its associated routing state is stored).

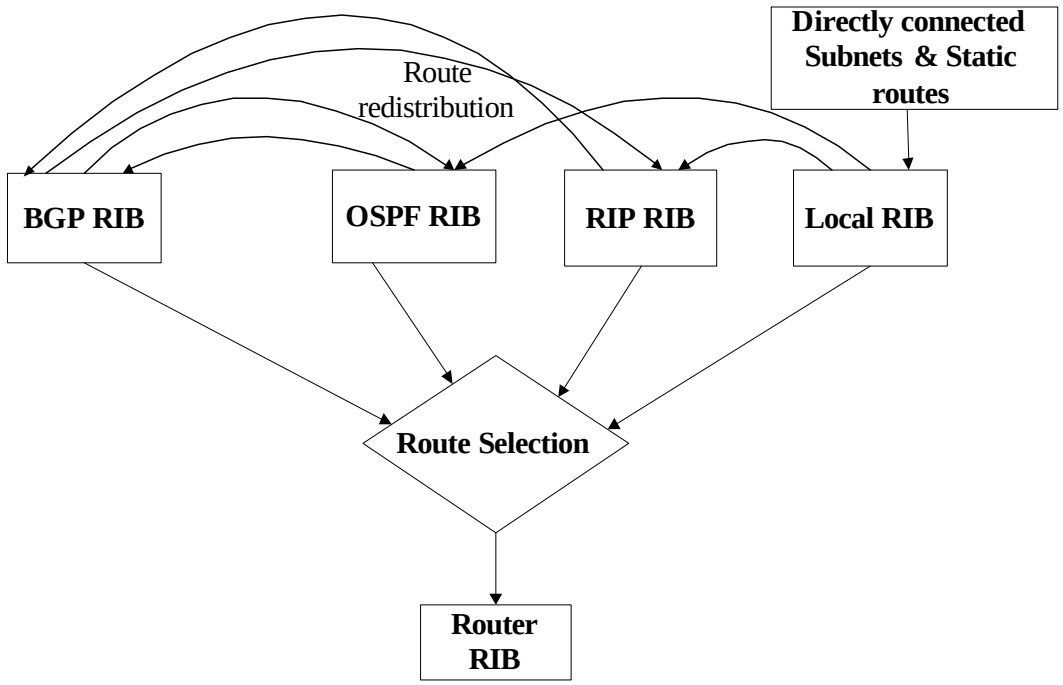

Figure 2.20: The abstract relation between routing process RIBs, route creation, route redistribution, and the router RIB [38].

Inside a single router, a mechanism called "route redistribution" is used to transfer routes between routing processes. This is illustrated by the arrowed lines in figure 2.20. To model the handling of routes for directly connected subnets and static routes, there is a local RIB that holds these routes. This makes the handling of static routes parallel to that of dynamically learned routes, as route redistribution mechanism will then be used to redistribute routes from the local RIB into the other routing processes. Routing policies are the mechanisms that decide the exchange of routes between different routers and between routing processes residing on the same router. Redistribution and routing policy determines the path a packet will take from its source to its destination. There is also another kind of policy control known as packet filtering. Packet filtering enables a router to classify the incoming or outgoing packet stream based on the properties associated with individual packets or packet streams. Packets that match the filtering policy are either forwarded (allowed) or dropped (denied). It should be noted that, packet filters are interface specific, they are configured statically, they cannot be shared across routers and can only be altered manually [38].

As shown in the figure 2.20, the routes that a router uses for forwarding packets are centrally stored in the router RIB. Route selection logic is employed to select which routes from the routing process RIBs should be stored into the router RIB. However,

BGP's route selection process works on only those routes that are present in BGP's own RIB. A second route selection process determines which of those routes should go into the router RIB [38].

BGP's RIB consists of three separate parts (logically these are separate; but they may reside in one file); namely, the Adj-RIBs-In, the Loc-RIB and the Adj-RIBs-Out. The whole BGP update process is illustrated in the following figure [19].

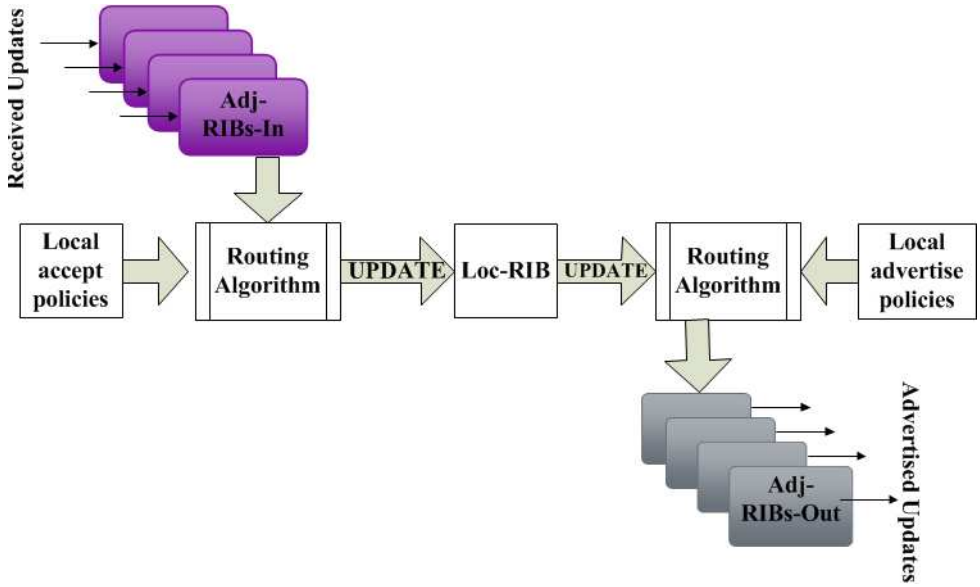

Figure 2.21: BGP update process.

- The Adj-RIBs-In stores the routing information that comes from neighboring/adjacent BGP speakers (through inbound UPDATE messages). Using its decision process, a BGP speaker chooses the best routes from the routing information available in Adj-RIBs-In.
- A BGP speaker generates the contents of Loc-RIB by applying its local policies to the routing information available from its Adj-RIBs-In. Loc-RIB is the main database of routes that the local BGP speaker will be using.
- The Adj-RIBs-In stores the routing information that a BGP router advertises to other BGP speakers. Information from the Loc-RIB is placed into the Adj-RIBs-Out and then sent out to other peers using UPDATE messages.

In essence, route storage is the function of RIB in each BGP speaker [20].

2.6.6 BGP Path attributes

Path attributes are the mechanism through which BGP stores detailed information about routes (the term "route" is used interchangeably with "path" in BGP) and also describe those details to BGP peers.

There are several path attributes, each of which describes a particular characteristics of a route. Attributes are divided into categories based on their level of importance

[19] Securing Inter-Domain Routing (March 2005). By Geoff Huston. `http://cidr-report.org/papers/isoc/2005-03/route-sec-2-ispcol.html`

[20] BGP Routing Information Base (RIB). `http://www.networkers-online.com/blog/2010/03/bgp-routing-information-base-rib/`

and specific rules are designed to manage their propagation. Most important path attributes are called well-known mandatory attributes. These must be present in every route description in UPDATE messages and they must be processed by every BGP speaker. *ORIGIN*, *AS_PATH*, *NEXT_HOP* are well-known mandatory attributes. Well-known discretionary path attributes must be recognized by a BGP speaker when received; but may or may not be included in the UPDATE message. In other words, these attributes are optional information for a sender, but mandatory for a receiver (to process). *LOCAL_PREF* and *ATOMIC_AGGREGATE* are well-known discretionary path attribute. Path attributes are further differentiated into optional transitive and optional nontransitive, depending on how they are processed by a receiving BGP speaker that does not recognize them. Optional transitive attributes (e.g. *aggregator*) must be passed to the next BGP peer by the BGP speaker that has not implemented this attribute (in an UPDATE message). The community attribute is optional transitive and is an important part of the novelty that this work introduces. An optional nontransitive attribute (e.g. *MULTI_EXIT_DISC*) is one that must be discarded if the receiving BGP speaker does not implement it [34].

A BGP speaker mainly uses the following attributes to qualify the routes that will be used for routing as well as those that will be advertised to its neighbors:

ORIGIN: This attribute specifies the origin of the path information. It indicates whether the path originally came from an interior routing protocol, the depricated exterior gateway protocol (EGP) or from some other source.

AS_PATH: It's a list of AS numbers that specifies the sequence of ASs though which this UPDATE message has passed. It is used to calculate routes and detect the presence of routing loops.

NEXT_HOP: It defines the IP address of the border router that should be used as the next hop to the destinations listed in an UPDATE message.

MULTI_EXIT_DISC (MED): If a path includes multiple exit or entry points to/from an AS then this attribute may be used as metric to choose one exit or entry point over the others.

LOCAL_PREF: This attribute is used in communication between BGP speakers residing in the same AS to indicate the level of preference towards a particular route.

ATOMIC_AGGREGATE: A BGP speaker may be presented with a set of overlapping routes (routes with a common prefix) from one of its peers. For example, consider a route to the network 34.15.67.0/24 and a route to another network 34.15.67.0/26. The latter network is a subset of the former, making it more specific. The BGP speaker may select the less specific route (i.e. avoiding the more specific one) and then it will attach the *ATOMIC_AGGREGATE* attribute to this route when propagating it to other BGP speakers. Less specific routes are always preferred unless policies state otherwise.

AGGREGATOR: This attribute is included in UPDATE messages and it contains the AS number and BGP ID of the router that had performed route aggregation. Mainly this attribute is used for troubleshooting. For our purposes, this attribute is used to identify the LM; as we will see in chapter - 4.

COMMUNITY attribute: In BGP, a "community" refers to a group of prefixes that share some common property and can be configured using the BGP COMMUNITY

attribute. It's a way for a route advertiser to "signal" to a route receiver some additional information about the BGP route. It is intended to alter the way the receiver makes decision about forwarding and also to effect the further propagation of the BGP route. The COMMUNITY attribute is of variable length, consisting of four octet (i.e. it's 32 bit long) values. The first two octets contain an AS number; while the semantics of the last two octets are defined by the AS. A community attribute can have values ranging from $0x0000000$ to $0x0000FFFF$; while values from $0xFFFF0000$ through $0xFFFFFFFF$ are reserved. A community value can also be presented in a colon-separated or ASN: nn format; where the ASN value defines the AS that needs to be affected, and is contained in the first 16 bits. The remaining 16 bits then contain a Local Preference value. In CRM, we have used the colon-separated format of the community value. Additionally, three specific community values have global significance and their operations must be implemented by any community-attribute-aware BGP speaker. These values are:

- NO_EXPORT ($0xFFFFFF01$): All routes received carrying a NO_EXPORT communities value must not be advertised to any external BGP peers. This route must be kept within the local AS.
- NO_ADVERTISE ($0xFFFFFF02$): All routes received carrying this community value must not be advertised to any peer, internal or external.
- NO_EXPORT_SUBCONFED ($0xFFFFFF03$): Routes with this community value must not be advertised to external BGP peers residing outside the sub-AS.

One of the important novelties of this work lies in the fact that, the COMMUNITY attribute is used here not to effect the routing or propagation decision, rather to carry end-point reachability information (explained in section 4.2.2.5.2) [16, 34, 51, 52, 57] [20].

2.6.7 BGP decision process

The BGP decision process is used for installing (in the local router/node) and selecting routes by applying policies to the routing information stored in Adj-RIB-In for subsequent advertisement. The output of this process is the set of routes to be advertised to all BGP peers is stored in Adj-RIB-Out and the set of routes that will be used by the local BGP speaker to forward packets is stored in Loc-RIB. This decision process consists of three phases:

In the first phase, the degree of preference for each route received from the local AS is determined based on $LOCAL\_PREF$ or preconfigured policy information. If the route belongs to a non-local AS then the degree of preference is calculated using only the preconfigured policy information [51].

The second phase involves selecting routes that are suitable for addition to the Loc-RIB. Here, route selection is based either on the highest degree of preference or on a single route to a destination. If there is a tie between different routes to the same destination then the following criteria are applied iteratively for breaking that tie:

- Select the route with the lowest $MULTI\_EXIT\_DISC$ attribute,
- Select the route with the lowest cost (interior distance).
- If there are several routes with the same cost then select the route that is advertised by a BGP speaker in a neighboring AS, with the lowest ID (i.e. AS number). Otherwise, select the route advertised by a BGP speaker with the lowest ID.

The third phase involves route dissemination, which takes place whenever one of the following events occurs:

- When Loc-RIB has changed,
- When locally generated routes learnt by means that are outside of BGP has changed,
- When a new BGP speaker-BGP speaker connection is established [51].

2.6.8 Packet traversal in a BGP domain

BGP provides a networked view of ASs where different ASs send/receive traffic to other ASes. The following figure provides an insight into how packet traversal takes place across ASs in a BGP enabled network.

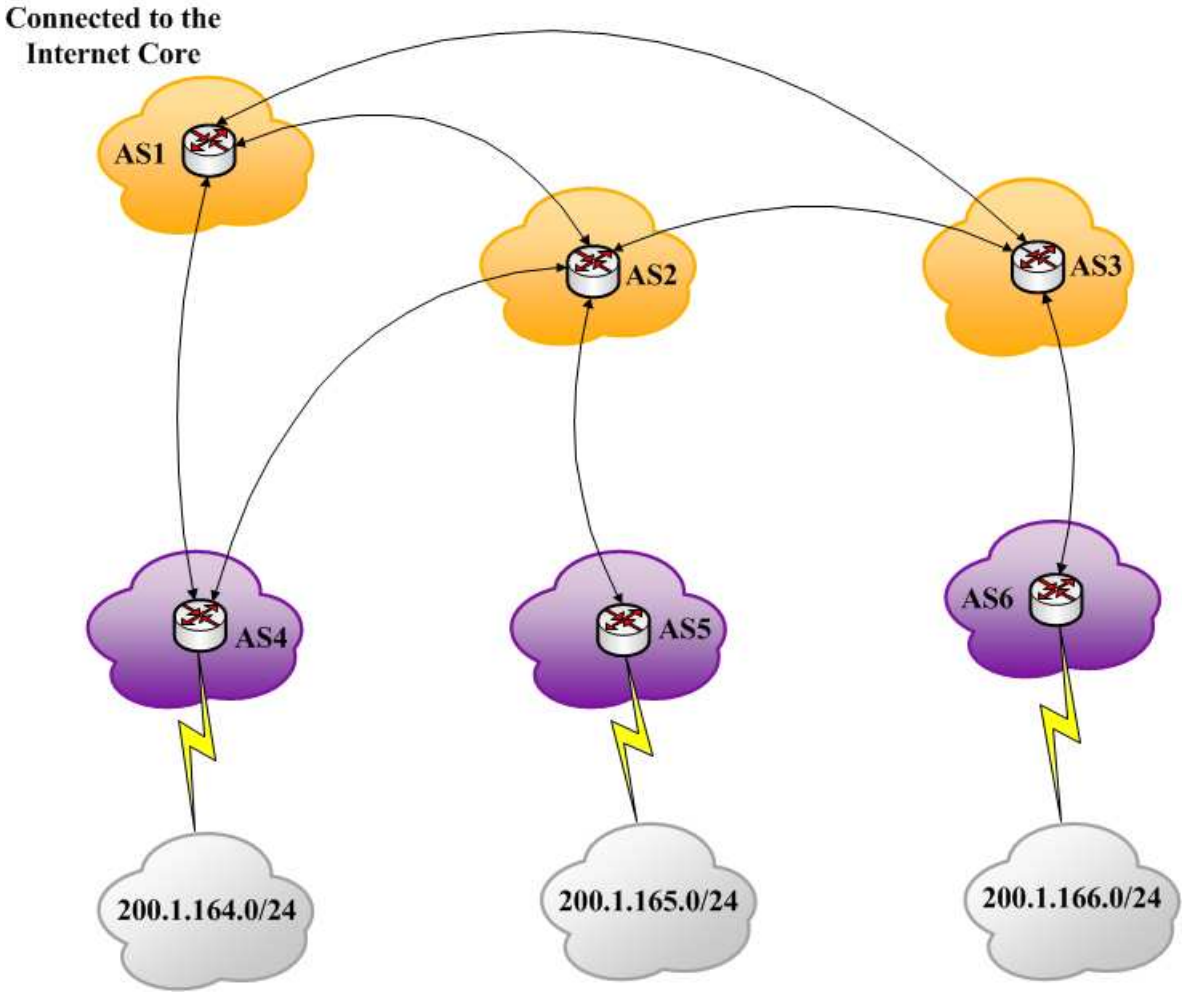

Figure 2.22: A sample BGP network topology with different ASs [51].

For the sake of simplicity we have assume that, each of the ASes house only a single router. In this configuration, AS1 is connected directly to the Internet core, while AS2 and AS3 provide transit service to other ASs which need to get connected to the Internet and are thus referred to as Transit AS. ASs 4, 5 and 6 provide connections to networks 200.1.164.0/24, 201.1.165.0/24 and 202.1.166.0/24 respectively.

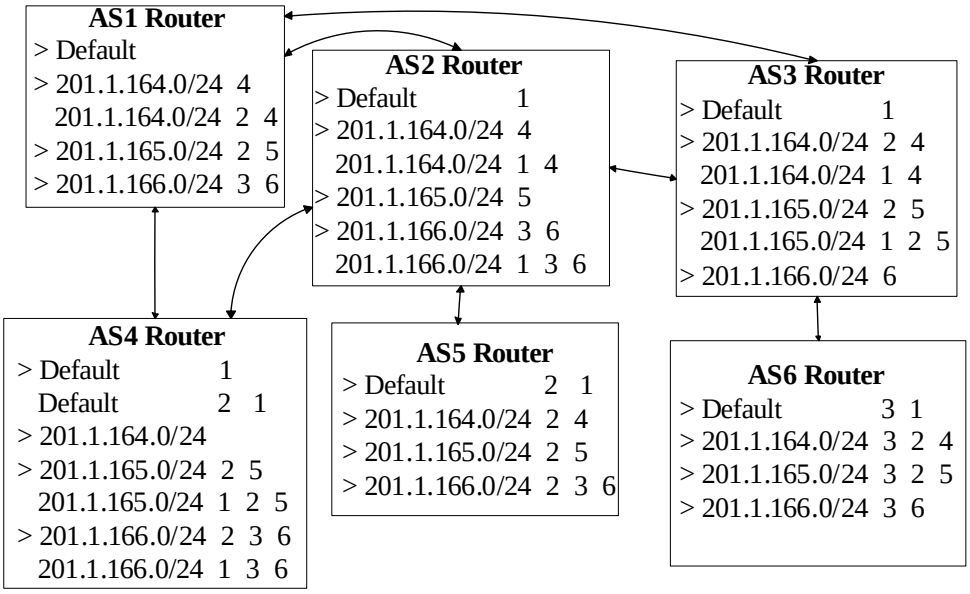

Figure 2.23: Routing tables [51].

The process of RT creation for reaching the network 200.1.164.0/24 is described below:

1. AS4's BGP router advertises to its neighbors that, the network 200.1.0/24 is attached to it.

2. ASes 1's and 2's BGP routers will install this routing information in their routing tables and advertise it to their respective neighbors. However, ASs 1's and 2's BGP routers will not advertise it to AS 4 (to avoid loop formation). Because AS 4 is the originator of this routing information.

3. The advertisement from AS1 and AS2 BGP routers reach AS3 and AS5. Subsequently, AS3 will advertise this information to AS6 [21]. In this way all the BGP routers in all ASs receive the routing information.

4. When a BGP router receives more than one route to a particular destination network then the BGP decision process selects the appropriate route to be installed based on its implemented policy. This happens at AS3's BGP router which receives separate advertisements from both AS1 and AS2 regarding the path for 200.1.164.0/24 [51].

2.7 Address Aggregation

Aggregation of IP addresses is a crucial part of this thesis work. But, we have chosen to describe it in chapter - 4; because the concept of address aggregation is tightly intertwined with our implementation work. Therefore, in our opinion, discussion on this topic will best serve the reader if it is placed in chapter - 4; where we have explained our software implementation in details.

[21] Both ASs 5 and 6 are a stub ASes. Because they are connected to only one other AS.

Chapter 3

External Tools

Before going into the nitty-gritty of architecture and implementation details, we would like to discuss the external tools used for this work. We will also provide justification regarding the choice of these tools. We urge the readers not to be impatient; because we will explain the detailed configuration and functionalities of these tools in the upcoming chapters. For the time being, bear with us.

Our work employs an existing open-source routing software, namely, Quagga for the routing functionalities. For storing the network state information we have used MySQL. Additionally, for controlling the access to this information by concurrently running processes, SQL's transaction facilities are utilized through MySQL. An MD5 message-digest algorithm is applied to generate hash values. Detecting changes in the generated hash values signifies a "change" in the network residing "behind" the LM. In the following subsections, we discuss the aforementioned tools briefly.

3.1 Quagga routing suite

At first, we would like to explain the rational for choosing Quagga and then move onto describe its architecture. For a suitable open-source routing software, we had a couple of options; such as, GateD, Zebra, OpenBGPD/OpenOSPFD, BIRD, XORP and obviously, Quagga. Out of these options, GateD is effectively "dead". Since its rights were acquired from Merit by NextHop Technologies, updates have become rare. Another option is Zebra, whose updates have stopped since its primary developer Kunihiro Ishiguro cofounded a commercial entity named IP Infusion Inc. IP Infusion Inc was more or less the end result of the Zebra project. The last release of Zebra was more than half a decade old and the last "recent update" was done about 10 years ago [1] . OpenBGPD/OpenOSPFD has only undergone slow development and is not expected to become a serious alternative any time soon. BIRD on the other hand, is going through active development and has a serious following in the research community. This was one of two serious alternatives that we had while we were looking for a suitable open-source routing software. We did not choose it, because BIRD's interface resembles that of Junos (the network operating system used in Juniper Networks hardware systems), which was unfamiliar to us. XORP was always more of a research package, but we wanted to be as close as possible to operational and production level environment.

Quagga is based on the Zebra router. In other words, it has been built based on the code of Zebra. It has some features that Zebra lacks and vice versa; but overall they are very

[1] `http://www.zebra.org/recent.html`

much similar. Quagga supports the most number of routing protocols (for UNIX based platforms) maintained by any open-source routing software that we are aware of, and most importantly it has a very active mailing list that enables users to look/ask for help whenever needed, which in turn prompted us to choose it as the routing protocol for this work [24].

Quagga version 0.99.4 supports implementations of RIPv1, RIPv2, RIPng, OSPFv2, OSPFv3, BGP-4, and BGP-4+. Quagga also supports special BGP Route Reflector and Route Server behavior. In addition to traditional IPv4 routing protocols, Quagga also has implementation for IPv6 routing protocols [30, 53]. It should be noted that, Quagga works independently from the OS over which it is installed. This is not the case for some other open source routers (e.g. Vyatta) or commercial routers, where the OS and the routing engine are built together. With routers like Vyatta, users can access the OS; but for commercial routers like Cisco or Juniper, users only have access to the router's interface [2].

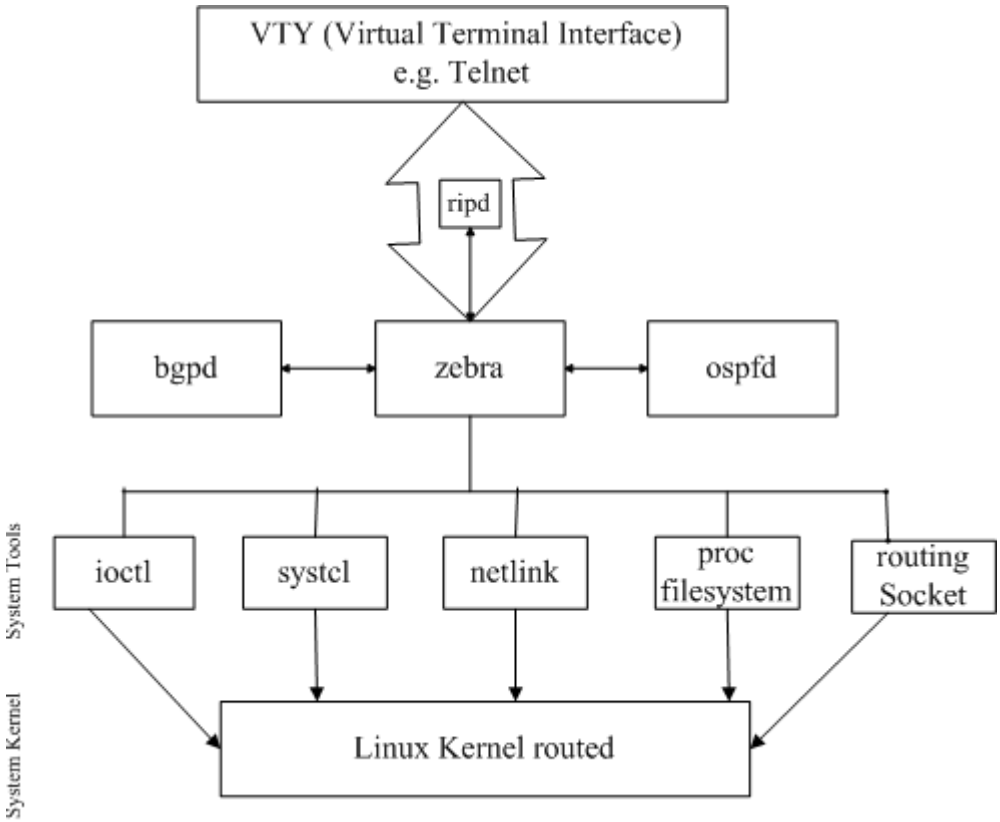

Figure 3.1: Quagga Architecture [51]

Fig. 3.1 [51] illustrates the architecture of Quagga, along with the associated daemons of BGP, OSPF, RIP, RIPv2, RIPng and OSPF6. As shown in the figure, the protocol daemons interact with the kernel routing table via Zebra daemon. Zebra defines its own TCP-based protocol (referred to as Zebra-protocol) to handle the inter-process communication between itself and the other protocol daemons. Each protocol daemon houses its own routing table and sends its own selected routes to the Zebra daemon, which is responsible for interacting and managing the routes to be installed in the forwarding table. Zebra daemon is also responsible for the allocation and distribution of services and resources to the aforementioned protocol daemons [51].

Quagga was originally designed to use multi-threaded mechanism when it runs on a kernel that supports multiple threads. But, the current thread library for gnu/Linux or FreeBSD

[2] `http://openmaniak.com/quagga.php`

has some problems running reliable services such as routing software, making it impossible for Quagga to use threads. Instead, the *select()* system call is used to achieve multiplexing of events among the sockets. With this function, a routing process tells the kernel that it will remain inactive until some specific event occurs [30, 55].

Quagga provides an interface that accepts Telnet connections and can be used for remote configuration of the router. As shown in Fig. 3.1, this interface is known as VTY (Virtual TeleType). The configuration commands that are used during the Telnet session are very much similar to that of Cisco's Internet Operating System (IOS). Some issued commands (via the VTY) effect the routing process immediately; while others take effect only after the running configuration is written to the memory and after the respective process has been restarted. Typically, commands that deal with adding or removing interfaces require restarting of process. Interfaces that are up and running, their parameters can be altered "on the fly" [24].

Like the command-line interfaces of commercial routers, VTY is also organized around the idea of modes. One can move in and out of two user modes while configuring Quagga, and the mode determines which commands can be used. First one is the normal mode, while the other is enable mode. Normal mode users can only view system status. An Enable mode user can change system configuration. It is worth mentioning that, Quagga's system administration (i.e. mode) is independent from UNIX's account feature [30].

The different routing daemons of Quagga run on preconfigured ports. Though these port numbers can be changed; though we have utilized the default ports for our work [51]. The following is a table of default ports on which routing daemons run:

Table 3.1: Quagga's default ports for Routing Daemons

Routing Daemon	Port number
zebra	2601/tcp
ripd	2602/tcp
ripngd	2603/tcp
ospfd	2604/tcp
bgpd	2605/tcp
ospf6d	2606/tcp

In our work, we are only concerned with the bgpd daemon, which is suitably configured and is used to advertised the community value.

For clarity, it should be noted that, Quagga possesses only routing capabilities and the functionalities associated with it (e.g. access lists, route maps etc.). The OS system kernel takes care of the actual packet forwarding based on the routes that Quagga has calculated. Additionally, Quagga does not provide any "non-routing" functionalities such as DHCP server, NTP server/client or SSH access [24] [3].

3.1.1 Advertise End point reachability with Quagga

At the heart of our work is the dissemination of end point reachability information. Our approach involves LISP, which requires us to advertise the mappings between EID-prefix and RLOC. For this, our initial choice was to use Multiprotocol Label Switching (MPLS) with BGP.

MPLS uses a forwarding mechanism called "label switching" where packets are forwarded based on labels. However, end systems are unaware about labeled packets and it's the

[3] `http://openmaniak.com/quagga.php`

routers that will add a label when entering "MPLS area" and remove that label after leaving it.

In label switching, only the first router (known as ingress Label Edge Router or iLERs) performs a routing lookup. But instead of finding a next-hop, it finds the final destination router and also figures out a pre-determined path from "here" to that final router. The first router then applies a "label" (or "shim") based on this information. The label is used as an index into a table which specifies the next hop and a new label. The old label is swapped with the new label and the packet is forwarded to its next hop. All future routers (known as Label Switching Routers or LSRs) use this label to route the traffic without needing to perform any additional routing lookups. As indicated in by the name, LSRs swap the MPLS labels. When this packet reaches the final destination router (referred to as egress Label Edge Routes or eLERs), the label is removed and the packet is delivered via normal IP routing. In short, in an MPLS network, packet-forwarding decisions are made solely on the contents of this label, without needing to examine the packet itself. The MPLS architecture does not authorize any single method of signaling for label distribution. BGP has been enhanced to piggyback the label information [4]. The following is a simplified figure [5] that gives an idea about the whole MPLS process.

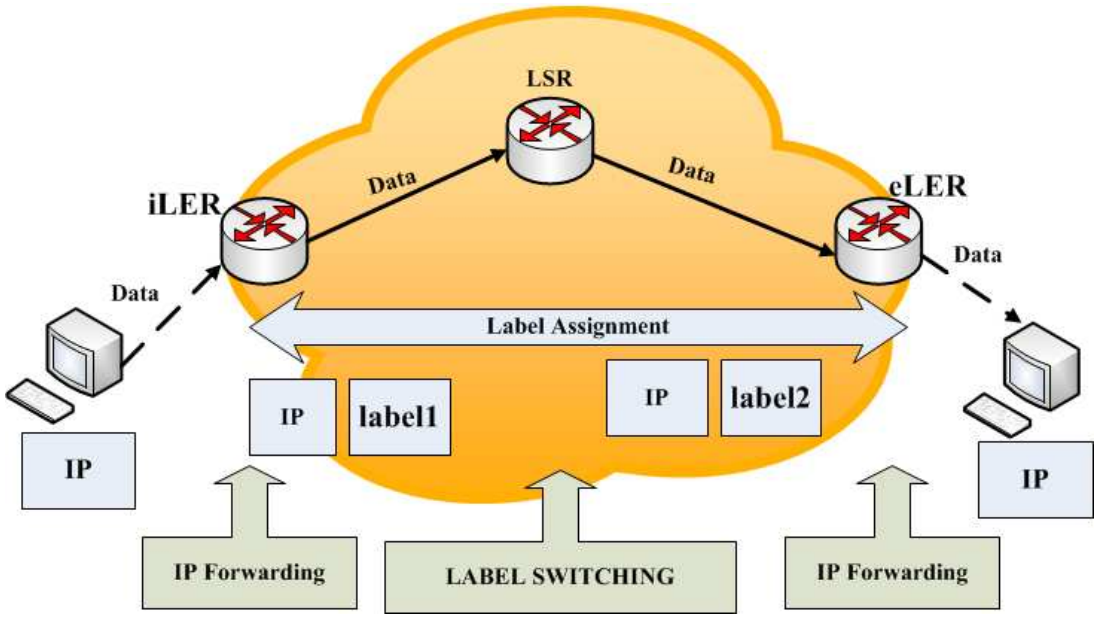

Figure 3.2: MPLS label switching operation

The main reason for our initial inclination towards MPLS is that, the aforementioned label and IP comes from two different namespaces. It is similar to the semantics of EID and RLOC. However, when we tried to configure MPLS in Quagga, we discovered that, Quagga only supports statically assigned labels, making MPLS unsuitable for our deployment. This disadvantage of Quagga has lead us to choose a new and innovative approach where minimum BGP functionality is assumed. To achieve this, we had to concatenate all the routes, pass it through a MD5 hash function, convert the hash value to a number and finally use BGP's community attribute to advertise it for the dissemination of end point reachability information. How BGP's community attribute was configured in Quagga for advertising end point reachability is explained in the next chapter. Changes in the advertised community value indicate a change in the network "behind" the authoritative ETR.

[4] Basic MPLS Tutorial. `http://www.iptut.com/mpls-knowledge/basic-mpls-tutorial`
[5] Multiprotocol Label Switching (MPLS). `http://blinky-lights.org/networking/mpls.pdf`

3.2 MySQL

In this section, we continue with the trend; i.e. to justify the reasons for choosing MySQL and afterwards move onto its description. MySQL offered us with a ready made DBMS to store network state information. It provided us the flexibility to add new entries and an easy way to collect data for future measurements and analysis. Another reason was to use transactions, so that access to a common resource by multiple processes can be controlled.

The MySQL database system uses a client-server architecture that centers around the server known as, mysqld. This server program does the actual manipulation to the databases. Client programs cannot do it directly. Instead, they communicate user's intent to the server by means of SQL (Structured Query Language). Client programs are installed locally, but the server can be installed anywhere, as long as the clients are able to connect to it. MySQL is an inherently networked database system, so clients can communicate with a server that is running locally or one that is running somewhere on the other side of the globe. After getting connected, clients interact with the server by sending SQL statements to it, so that database operations can be performed, and receive the statement results from it [12]. For our purposes, we have used *MySQL ver. 14.14 distrib 5.1.49* and both the clients and the server interacted locally.

One such client is the *mysql* program, included in MySQL distributions. It is the principal, and very powerful, MySQL command-line tool. Almost every administrative or user-level task can be performed with it in one way or another. When used interactively, *mysql* prompts the user for a statement, sends it to the MySQL server for execution, and then displays the results. This capability makes *mysql* useful in its own right, but it can also be used non-interactively; e.g. to read statements from a file or from other programs. This enables the user to use *mysql* from within scripts or *cron* jobs or in conjunction with other applications [12, 47] . For our work, we have utilized *mysql* both interactively and non-interactively.

3.2.1 Using Transactions through MySQL

Besides using SQL statements through MySQL for select, insert, update and delete operations; one of our main goal was to use transactions. Doing so enables us to hand over the tasks of concurrency and integrity issues to MySQL.

In SQL, a transaction is a group of SQL statements taken together as a logical unit. In many occasions, the process of entering data requires adding data into several different tables and perhaps modifying data in a couple of tables. If any of those operations fail, then the entire set of operations must be undone or rolled back. In order to ensure that one has performed all of the operations or none of the operations, we use transactions. In order for a group of statements to be considered a transaction, it must pass the *ACID* test. *ACID* is an acronym for four properties:

- **Atomic:** This property refers to the all-or-none behavior of the group of statements. If any of the statements fail, then the entire group of statements must be discarded. Only when all of the statements execute without error are the results of the entire group of statements saved into the database.
- **Consistent:** The database must be in a consistent state at the end of the transaction. The SQL statements must be applied without error, and all database structures must be correct and saved.

- **Isolated:** Data that transactions change must not be visible to other users/processes before or during the changes are being applied. Other database users/processes must see the data as it would exist at the end of the transaction so that they do not make decisions or errors based on inconsistent data.
- **Durable:** At the end of the transaction, the database must save the data correctly. Power outages, equipment failures, or other problems should not cause partial saves or incomplete data changes [47].

3.2.2 Choosing a Transactional Storage Engine

MySQL supports several storage engines, but all of them do not support transactions. To use transactions, one must use a transaction-safe storage engine. To view which storage engines do support transactions, the MySQL syntax *SHOW ENGINES* can be used. The default MySQL storage engine *MyISAM* neither provides foreign key constraints nor supports transaction facility. Currently, transactional engines include *InnoDB*, *NDB*, and *BDB*. For our purposes, we have used the *InnoDB* storage engine. *InnoDB* tables in MySQL are *ACID* compliant. It uses a fine-grained, row-level locking mechanism. This means that different transactions can run on the same table at the same time as long as they are all only reading or do not use the same rows if they are writing.

Using transaction syntax provides atomicity. For our current work, we have used the syntaxes *START TRANSACTION*, *COMMIT* and *ROLLBACK* (in case of error or failure to execute query). Transactions and foreign key constraints give consistency. One can choose the level of isolation that transactions have from one another. The binary log and repair tools provide durability [64, 65].

3.3 MD5 message-digest algorithm

Hash algorithms, also known as message digest algorithms that generate a unique message digest for an arbitrary message. This digest can be considered as a fingerprint of the message and it must contain the following properties:

- The hash must be easy to compute.
- It must be very hard to compute the message from the digest and
- It must be hard to find another message which has the same message digest as the first message.

Hash algorithms are used widely in cryptographic protocols and Internet communication in general. Several widely used hash algorithms exist. One of the most famous is the MD5 message digest algorithm developed by Ronald Rivest. This algorithm divides the input message into chunks of 512-bit blocks. It takes in a variable-length message and from it generates 128-bit fixed-length hash value as output by iteratively applying a compression function consisting of 64 steps [15]. For our purposes, we have used an open-source off-the-shelf software developed by RSA Data Security Inc. In other words, we have treated the whole process of generating a hash value by the MD5 message digest algorithm as a "black box". Therefore, we are only concerned with this off-the-shelf software's interface; i.e. how to pass the input message and subsequently receive the hash value output. In exactly what way this software generates hash values is out of the scope of this work.

There are a lot of open-source off-the-shelf implementation for the MD5 message digest algorithm to choose from. The reasons for choosing this particular software are:

- It is implemented in C (CRM is developed with C) and
- Its interface was relatively easy to understand.

Chapter 4

Software Implementation

This chapter describes the details of the LM implementation from a software point of view. The main goal of software implementation is to build a LM from scratch. This thesis aims to implement a software that performs the tasks of a LM; so that we could test the validity of our idea and gain first-hand experience regarding the pitfalls and complicacies that come with the actual implementation. Through this implementation we intend to validate a working system that is feasible with minimal dependency for the kernel/system specific data structures. This would prove portability and modularity. We also aim to minimizing the dependency on BGP, so that we could substitute BGP with any other protocol that can offer the desired services; i.e. to provide topology discovery and reachability announcements for EID-prefixes. In this implementation we are using BGP to carry only an indicator that reflects a change in network state. BGP is not used for distributing EID-prefixes. This could have been done by using multi-protocol support of BGP and by defining an address family for EID. But Quagga's inability to sustain MPLS refrained us from taking this path.

4.1 Architectural Design

In order to build the functionalities of a LM, we were required us to construct and process the Map-Register and Map-Notify messages [1] according to the LISP specification [13]. We created the data structures needed for this purpose by mimicking the coding style and standards of OpenLISP [28, 29]. This was done, while keeping in mind that, in near future, CRM would be paired up with OpenLISP. The implementation work for this thesis is done entirely in C programming language and the chosen OS is Ubuntu 10.10 (Linux).

The following is a "high level" abstract view of CRM. Our intention is not to overwhelm the readers just yet with too much detailed information about the implementation. The reader will be introduced with a detailed (and rather complicated) schema in the forthcoming section 4.2.

[1] To be exact, the functionalities of a Map-Notify message is extended in our work.

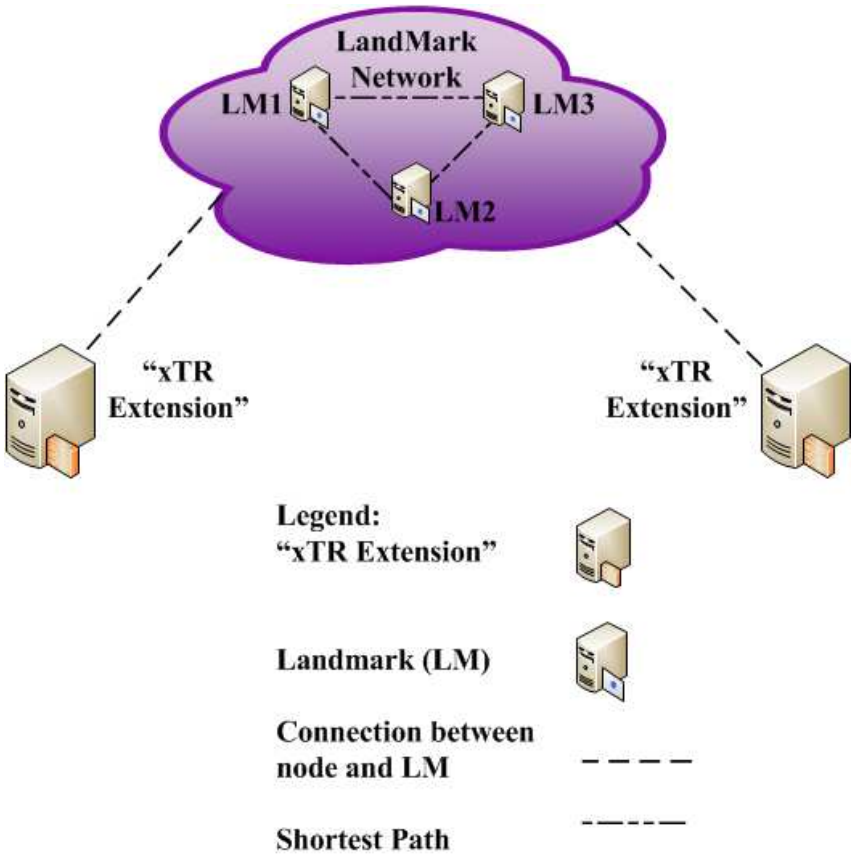

Figure 4.1: "High Level" abstract view of CRM.

In addition to the CRM which lies in the Control Plane, we have developed "xTR extension" as shown in the above depicted figure. According to the LISP specification [13], a "typical" xTR lies in the Data Plane and is only responsible for sending and receiving Map-Register and Map-Notify messages respectively. The "typical" Map-Notify message is returned in response to sending a Map-Register message. The Map-Notify message sent by a Map-Server is used as an acknowledgement after the receipt of a Map-Register message. In our work, we expanded the usability of a Map-Notify message so that it would indicate to the xTR whether the mapping sent through the Map-Register message is delegated or not. Based on this information the "xTR extension" would then contact the relevant LM where delegation should take place. This extra responsibility of a xTR is not defined in the LISP specification [13].

An alternative approach would have been to let the "current" LM (which has discovered another "better" suited LM) resend the delegated Map-Register messages directly to the "better" suited LM (or "delegated" LM). The term "better" means that, the "delegated" LM is able to perform a more aggressive aggregation as defined by the LM selection algorithm [14]. This approach would have meant that, the "xTR extension" would have no information on whether the Map-Register messages it had previously sent were delegated or not. But, if a "xTR extension" gets delegation information, it can decide for itself whether to send future Map-Register messages to the "current" LM or to the "better" suited LM.

Another important motivation factor was to mitigate potential DOS attacks. When a "xTR extension" is involved with relaying of Map register messages to the delegated LM, it knows which LMs are "legitimate" for sending Map-Register messages. If the "xTR extension" did not have this information then any LM (maybe not "legitimate") could sent Map-Register messages and in evidently launch a DOS attack. The only way to prevent

this would require an authentication token that is known by both the original/delegating LM and the "xTR extension". The token would prove the authenticity of the LM (i.e. this LM is really delegated by the original LM and "xTR extension") that was sending Map-Register messages. But this adds complexity, as we should also have the capability of revoking such tokens.

The aforementioned factors have motivated us to give the task of decision making to the "xTR extension". Though this approach has increased the work load of the "xTR extension", it has eliminated the delay concerning the sending of Map-Register messages to a LM that can only perform suboptimal aggregation (i.e. "current" LM instead of "delegated" LM and vice versa).

From this point on, the terms "xTR extension" and "xTR" will be used interchangeably when it comes to CRM.

4.1.1 Requirements:

Detailed description regarding the functionalities of a "theoretical LM" is defined in [11, 20, 35]. However, people who conceived the idea of Compact Routing [2] saw it as a "clean slate" approach. As a consequence of this presumption, their addressing scheme of a node contained three elements, which in turn, can never be realized neither by IPv4 nor IPv6. This first supposition ultimately leads to a static network topology. We were required to get around these "impracticable assumptions" and we did so by our novel approach; where we used BGP's Community attribute through Quagga to disseminate the information regarding the changing state of the network. All the designs and ideas that have lead to a "practical LM" are our own. It is a "dirty slate" approach that uses the everyday UDP and TCP client/server components to realize the functionalities of a "practical LM".

4.1.2 Software Development Environment:

This section describes the environment in which this software prototype has been developed.

4.1.2.1 Platform, Standard and License:

Apart from the proprietary LM selection algorithm developed by Flinck et al. [14], the application is under the GPL licensing scheme. The project is developed in Ubuntu 10.10 (Maverick Meerkat), which is a free and open source OS based on the Debian Linux distribution. Multiple instances of Ubuntu 10.10 is executed inside VirtualBox virtual machines. Availability of documentation and local expertise (i.e. NSN) have lead us to this choice. This application mostly uses the standard C libraries of GCC. The implemented code adheres to the standards of Portable Operating System Interface (POSIX). POSIX is a standard that is being jointly developed by the IEEE and The Open Group. It defines a standard operating system interface and environment, including a command interpreter (or "shell"), and common utility programs to support applications portability at the source code level. The current revision of POSIX is The Open Group Base Specifications Issue 6 and also the IEEE Std 1003.1-2001. Adhering to POSIX standards facilitate code portability between systems. There are more than ten parts to the POSIX standard. But

[2] Note: the notion of a LM is part of Compact Routing.

the relevant one for us is POSIX.1 that defines C programming interfaces (i.e. a library of system calls) for files, processes, and terminal I/O [32].

As OpenLISP runs on FreeBSD; maintaining a POSIX standard in the code would mean that this software can also run in FreeBSD without any hitch. We have utilized only one external library: mysql.h ; as it covers the basics of MySQL programming with the C API.

4.1.2.2 Software Editor:

The chosen software editor is Code::Blocks (Release 10.05 rev 0) [3] as it is oriented towards C/C++. It provides syntax highlighting and code folding through the use of the Scintilla editor component and all of the open files are organized into tabs, which can be closed and opened at the user's will with the navigation pane or with the close button on the tabs. Compared to Eclipse, it is a light weight editor. It should be noted that, we only used Code::Blocks for editing purposes. We did not use any of its in-built compiling or debugging facilities.

4.1.2.3 Make Utility:

For managing and maintaining the software (i.e. compiling, building), GNU's make utility is used. Make is most helpful when the program consists of many component files, as in our case. By creating a descriptor file containing dependency rules, macros and suffix rules, the user can instruct make to automatically rebuild the program whenever one of the program's component files is modified. Make is smart enough to only recompile the files that were affected by changes; thus saving compile time [4]. The chosen compiler for our prototype development is GCC (version 4.4.5).

4.1.2.4 Debugging:

> *"The most effective debugging tool is still careful thought, coupled with judiciously placed print statements."* – Brian Kernighan, "Unix for Beginners" (1979).

We concur with the above statement and we have also used *printf()* debugging extensively while developing CRM. It consists of ad hoc addition of lots of *printf()* statements to track the control flow and data values in the execution of a piece of code. Code is temporarily added, to be removed as soon as the bug at hand is solved; for the next bug, similar code is added [5].

However, in occasions we have also used gdb (the GNU Debugger) for adding conditional breakpoints [6] and stepping through the code [7]. Additionally, the development of CRM involved processing a large number of dynamically allocated (i.e. allocated in runtime) char arrays. In C, we used *malloc()/calloc()* functions for this purpose. In many instances,

[3] `http://www.codeblocks.org/`

[4] An Introduction to the UNIX Make Utility. `http://frank.mtsu.edu/~csdept/FacilitiesAndResources/make.htm`

[5] What is the proper name for doing debugging by adding 'print' statements? `http://stackoverflow.com/questions/189562/what-is-the-proper-name-for-doing-debugging-by-adding-print-statements`

[6] A breakpoint is a line in the source code where the debugger should break execution.

[7] After a conditional breakpoint is added we can go through the code one line at a time and locate the source of the error. This is accomplished using the step command

dynamic allocation has caused segmentation fault. A segmentation fault (often shortened to segfault) is a particular error condition that occurs when a program attempts to access a memory location that it is not allowed to access, or attempts to access a memory location in a way that is not allowed. This is managed by the OS's memory management layer. The strategy that we followed to debug segfault errors was to load the core file into gdb, do a back-trace, move into the scope of the code, and lastly, list the lines of code that caused the segfault [8].

Using a lot of dynamically allocated char arrays have also lead to memory leaks in the prototype. A memory leak occurs when a program no longer uses some chunk of dynamically allocated memory, but the memory was not de-allocated (through *free()* function). When the program started to "grow", there were a lot of instances where chunks of memory were constantly allocated (e.g. in a loop) and were not freed afterwards. We used Valgrind to detect such cases of memory leaks and point them out. Valgrind tracks each memory allocation and de-allocation. When it sees that no pointer exists in the program for the allocated memory chunk (or its content), the program has no way of de-allocating it, which in turn means that a memory leak has occurred [9].

4.2 Implementation of Major components

In terms of functionality, the implementation can be divided into four major components:

1. UDP client (or "xTR Extension"),
2. UDP server,
3. TCP client and
4. TCP server.

It should be noted that, UDP client (that implements "xTR Extension") is not a part of the LM (though it is a part of the CRM concept). The LM consists of TCP client, TCP server and UDP server. Previously, through figure 4.1, we have shown the readers a very "high level" abstract view of CRM. Now it is time to provide a "complete" functional diagram of the whole system.

[8] Debugging Segmentation Faults and Pointer Problems. `http://www.cprogramming.com/debugging/segfaults.html`

[9] Using Valgrind to debug memory leaks. `http://www.linuxprogrammingblog.com/using-valgrind-to-debug-memory-leaks`

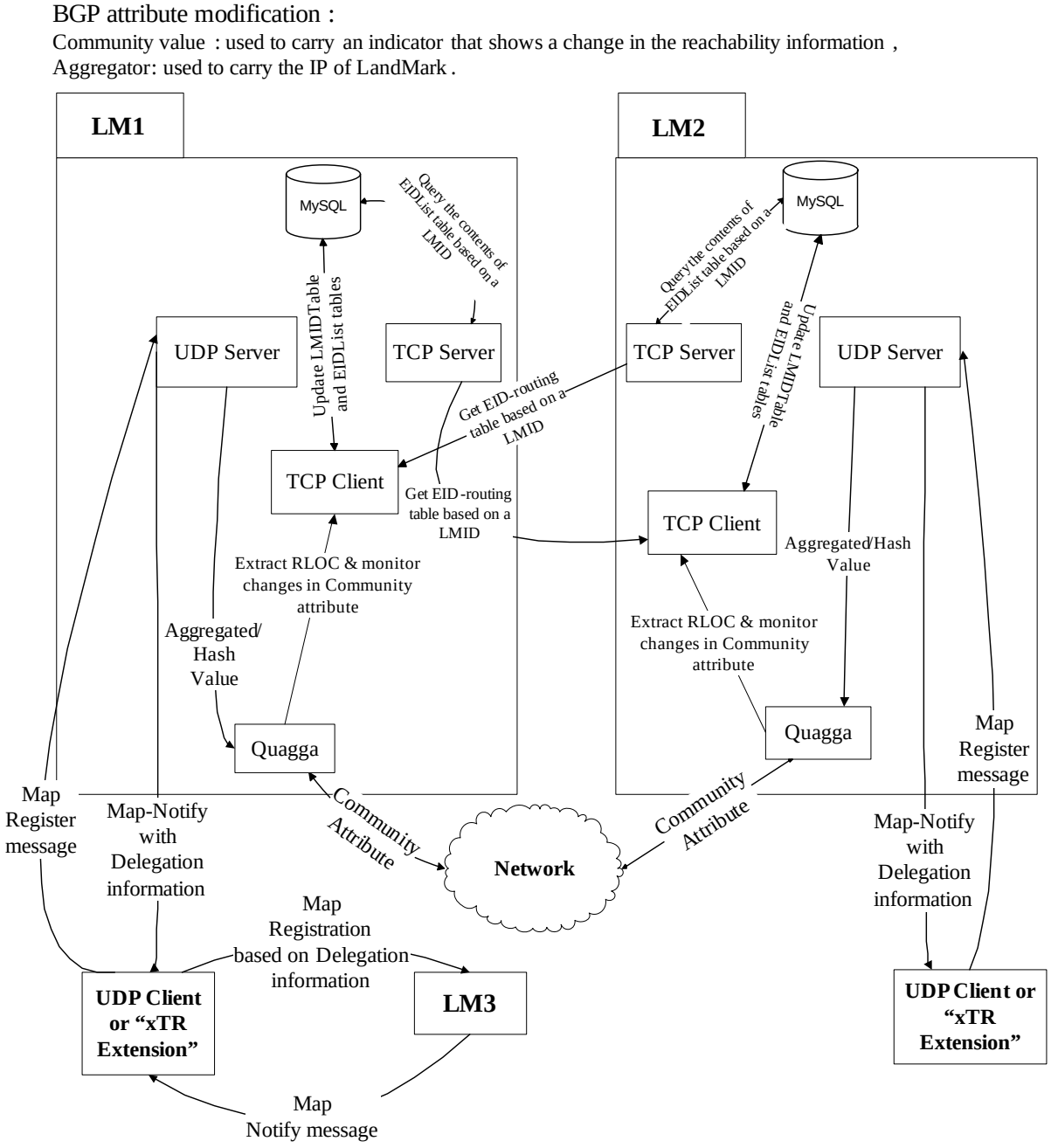

Figure 4.2: Major Components and functionality of CRM.

Figure 4.2 provides a detailed view of how the whole system works. Though the functional diagram looks quite complicated and intricate, we assure the audience, once we have explained the individual components in the following subsections, the whole system will start to make sense.

4.2.1 UDP Client (or "xTR Extension")

Like any other mapping system for LISP, in CRM also, an xTR (when functioning as an authoritative ETR) sends a Map-Register message to a LM to declare the presence of an EID-prefix that it owns as well as the RLOCs that should be used for exchanging subsequent Map-Request and Map-Reply messages. In other words, these registration requests contain all the EID-to-RLOC mappings owned by that xTR; i.e., all the EID-numbered networks that are connected to that particular xTR's site.

Our UDP Client's (or "xTR Extension's") processing involves 5 steps:

1. Read from the input file that holds the mappings between EID-prefix and RLOC into a linked-list,
2. Process the input so that it can be put inside Map-Register messages,
3. Send the Map-Register messages to the server using *sendto()* function,
4. Read back the server's response i.e. Map-notify packet using *recvfrom()*, and lastly,
5. Determine whether Map-Register messages are needed to be resent to a delegated LM.

Our client program does not ask the kernel to assign an ephemeral port to its socket. When it comes to an UDP socket, if that socket has not yet had a local port bound to it, then the first time this process calls *sendto()* an ephemeral port is chosen by the kernel for this socket. We have also assigned NULL pointers to the fifth and sixth arguments of the *recvfrom()* function; because we do not want know about the server through this function [59]. An element of the Map-notify message [10] provides us with this information.

We have constructed the Map-Register and the Map-Notify message using a set of structures. The following structures construct a Map-Register message.

[10] Discussed later in the current section.

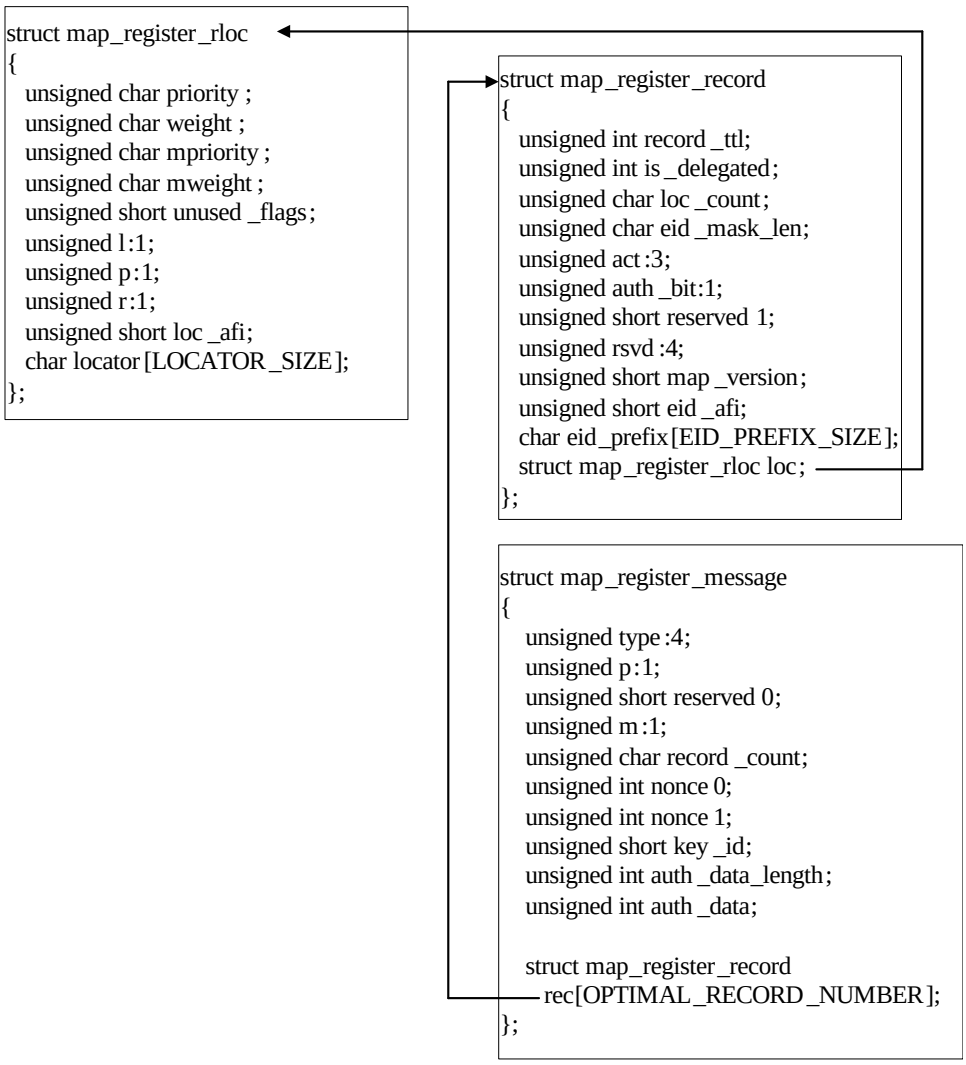

Figure 4.3: Formation of a Map-Register message.

It should be noted that, we have constructed Map-Register messages, each of which consists of multiple records. How many records are to be included within a message depends on the MTU (maximum transmission unit) of the network. For our test network, we determined the MTU size to be 1500 bytes. On average, each record is of the size 100 bytes. This means that, a maximum of 15 records can fit inside 1 Map-Register message.

As shown in figure 4.3, the structure *map_ register_ message* defines a Map-Register message. It has an element called *rec*; which in turn is an array of type structure *map_ register_ record*. It is this structure that actually holds all the information of the records. A single record primarily holds the mapping between an EID-prefix and its RLOC. The char array *eid_ prefix [EID_ PREFIX_ SIZE]* holds the address of the EID-prefix. For the purposes of aggregation and delegation each Map-Register record also has an element called *eid_ mask_ len* that holds the mask length of the EID-prefix. The variable unsigned int *is_ delegated* is defined by us to determine whether the EID-prefix present in that message can be delegated or not. In other words, it is a Boolean variable with a default value of zero (i.e. "no delegation"). If the subsequent Map-Notify message indicates that the EID-prefix residing in the Map-Register message can be delegated, only then, this variable is set to one.

Each structure of type *map_ register_ record* has another structure type element called *map_ register_ rloc* that holds the all the information regarding RLOC. The RLOC address is stored in the char array *locator[LOCATOR_ SIZE]*. It should be noted that, except

for the *is_ delegated* variable, all the other elements are defined according to the LISP specification [13] and OpenLISP [28] standard.

Next, we move on to Map-Notify message. The Map-Notify message works as an acknowledgement for the Map-Register message. Constructing a Map-Notify message is much simpler than a Map-Register message. According to LISP specification [13], a Map-Notify message is constructed by copying certain values from the Map-Register message for which acknowledgement is made. Figure 4.4 shows how a Map-Notify message is constructed.

Figure 4.4: Formation of a Map-Notify message.

A Map-Notify message contains two custom defined elements. First one is char array *server_ ip_ address[EID_ PREFIX_ SIZE]* which is a part of the structure *map_ notify_ message*. It contains the IP address of the TCP server part of an LM. It is used to inform the TCP client, the IP address of the TCP server with which it is suppose to connect. This IP address is stored in the DB. Second one is unsigned int *is_ delegated*. Its purpose is the same as in a Map-Register message. Interestingly, the LISP specification [13] requires us to copy certain variables from a Map-Register message into a Map-Notify message before sending it to the UDP client. Because of this specification, the UDP client gets to know which of its Map-Register record can be delegated (actually the EID-prefix inside a Map-Register record is delegated). Based on this information (through Map-Notify message's unsigned int *is_ delegated* element), the UDP client then can send those Map-Register messages again to the delegated LM.

All the preprocessor commands, function prototype declarations and definitions of the custom data types (i.e. structures) were provided through a custom header file known as *map_ registration.h.*

So, just to recap what we have said so far, the primary tasks of a UDP client include the construction of Map-Register messages and sending them to the UDP server. Based on the delegation information inside a Map-Notify message that comes as an acknowledgement, the UDP client then resends only those Map-Register messages whose EID-prefixes can be delegated.

4.2.2 UDP Server (for Aggregation)

The UDP server is the most important part of the LM. Because, it provides natural aggregation, delegation and if needed assigns virtual prefix. As we all know, UDP is a connectionless, unreliable, datagram protocol, quite unlike the connection-oriented, reliable byte stream provided by TCP. In an UDP client/server scenario, the client does not establish a connection with the server. Instead, it just sends a datagram to the server using the *sendto()* function. In our case, the client is sending Map-Register messages to the server. The server also does not accept a connection from a client. Instead, it just calls the *recvfrom()* function, which waits until data arrives from some client. The *recvfrom()* function returns the protocol address of the client, along with the datagram, enabling the server to send a response to the correct client. In our system, a Map-Notify message is the response sent by the UDP server. In accordance with the LISP specification [13], we have used the port 4342 for our UDP server.

Construction of an "usual" UDP server is simpler compared to a TCP server. In most cases, an "usual" UDP server is iterative. There is no call to fork, so a single server process handles any and all clients. Within a simple infinite loop the "usual" UDP server reads the next datagram arriving at its port using *recvfrom()* and sends datagram back using *sendto()*. As the server is iterative, queuing takes place in the UDP layer for the respective socket. Each UDP socket is equipped with a receive buffer and each datagram that arrives for this socket is placed in that socket's receive buffer. When this server process calls *recvfrom()*, the next datagram from the buffer is returned to this process in a FIFO order. This way, if multiple datagrams arrive for this socket before the process can read what is already queued for the socket, the arriving datagrams are just added to the socket receive buffer [59].

The inquiring reader might wonder why we have used the term "usual". It is for the purpose of clarity. Though the above mentioned description of a typical UDP server is applicable to our server, a LM's UDP server additionally performs natural aggregation, delegation, virtual prefix generation (if needed) and lastly advertise BGP community attribute through Quagga. The figure 4.5 is a functional diagram of CRM's UDP client and server.

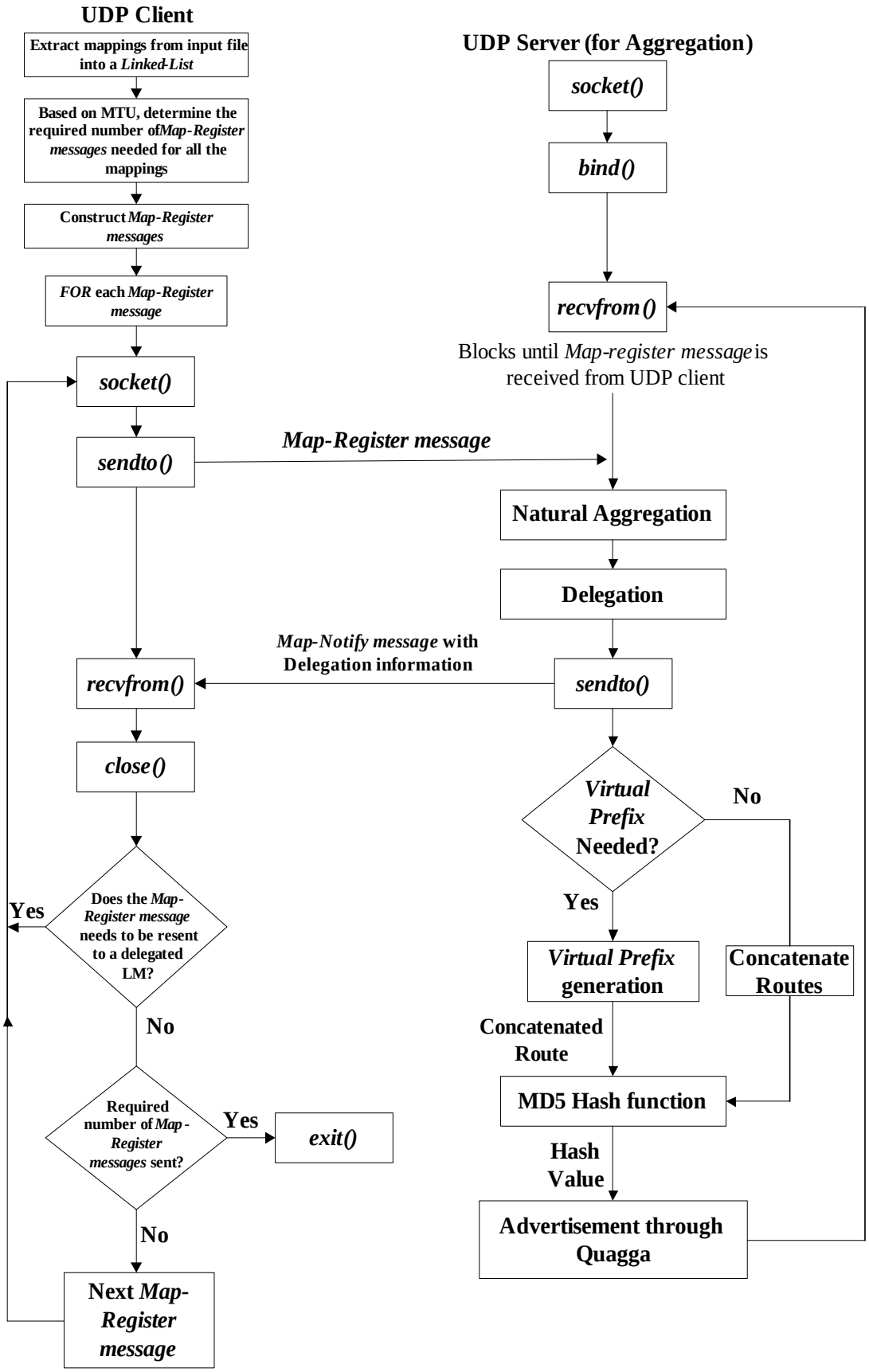

Figure 4.5: Functional diagram of CRM's UDP client and server.

Now we move onto discussing the major parts of the UDP server, namely,

Natural Aggregation,

Delegation,

Virtual Prefix generation and lastly

Advertise BGP's community attribute through Quagga.

Note, the MD5 hash function is not native. As explained in section 3.3, we have utilized the services of a third party external tool for our purposes.

4.2.2.1 Natural Aggregation

In the interest of avoiding confusions, we would like to provide the readers a little background information on route aggregation; before jumping into the implementation details.

Route aggregation [7] is a technique of organizing network layer IP addresses in a hierarchical way so that addresses become "topologically significant". Route aggregation summarizes routes so that, there are fewer routes to advertise across the Internet. Usually, a service provider is allocated a contiguous block of IP addresses, which it then divides (into smaller allocated blocks) and leases to its downstream subscribers (e.g. smaller ISPs). Because these addresses are contiguous, the ISP can advertise one route on the global Internet. UDP server's natural aggregation actually refers to route aggregation in the form of CIDR (Classless Inter-Domain Routing).

CIDR (or Supernetting) replaced the old process of assigning Class A, B and C addresses with a generalized network "prefix". Instead of being limited to prefix lengths of 8, 16 or 24 bits, CIDR uses prefixes anywhere from 13 to 27 bits. Thus, blocks of addresses can be assigned to networks as small as 32 hosts or to those with over 500,000 hosts. This allows for address assignments that much more closely fit an organization's specific needs. As stated earlier in section 1.1, a CIDR address contains the standard 32-bit IP address, along with information on how many bits are used for the network prefix. For example, in the CIDR address 206.13.01.48/25, the "/25" indicates the first 25 bits are used to identify the unique network leaving the remaining bits to identify the specific host. It is this prefix length that enables CIDR to perform "route aggregation" in which a single high-level route entry represents many lower-level routes in the global routing tables. For example, if an ISP owns the network 206.13.0.0/16, then this ISP can offer 206.13.1.0/24, 206.13.2.0/24, and so on to its customers. While advertising to other providers, the ISP only needs to advertise 206.13.0.0/16.

In our prototype, the IP addresses that need to be aggregated are EID-prefixes extracted from Map-register messages. The goal is to only advertise their (i.e. EID-prefixes) SUPERNET out to the world. Deducing the SUPERNET actually means to find the best possible parent address from the extracted EID-prefixes. To do so, we have taken the following steps:

1. Insert the extracted EID-prefixes into the nodes of a Linked-List. Assign an ANCESTOR_FLAG to each of the nodes and UNSET it by default.
2. For every EID-prefix, traverse the Linked-List and try to detect possible parent. If a parent is found then the ANCESTOR_FLAG is SET in the EID-prefix for which the parent is located.

3. At the end, the EID-prefix that is the best possible parent will have its ANCESTOR_FLAG as UNSET. Additionally, "orphan" EID-prefixes will also have their ANCESTOR_FLAGs as UNSET.

It is evident that, of the three steps mentioned, the second step is the crucial one; where we traverse Linked-List for each node (that is why we need nested while loops) and compare between two IP addresses (i.e. EID-prefixes) to determine if one is the parent of the other.

As mentioned, for the purpose of processing, we have used a Linked-List to store the EID-prefixes. A linked-list data structure typically maintains a pointer to the first element in the list. This pointer provides an entry point into the Linked-List. All basic operations start with this pointer, which we will denote as *"header"*. The pointer *"nextNode"* will reference the immediate next element, while the last item will have a reference to NULL. The pointer *"tempNode"* is used so that the inner while loop can be traversed. Here, the *"="* refers to assignment and *"->"* is the arrow operator.

```
nextNode = header
while nextNode != null do

    tempNode = header
    while tempNode != null do

        call COMPARE (tempNode IP, tempNode IPLen, nextNode IP, nextNode IPLen)
        tempNode=( tempNode nextElement )
    end while

nextNode=( nextNode nextElement )
end while
```

Figure 4.6: Schema to traverse the whole Linked-List for each element of the Linked-List.

The above schema calls a function *COMPARE()* and passes the IP addresses to be compared and their respective prefix lengths. The reader should know that, all the IP addresses are in binary format (as char array). Hence the leftmost bit becomes the Most Significant Bit (MSB). We convert IP addresses in binary format so that they can be compared efficiently. Within *COMPARE*'s definition, *IP1* and *IP2* are the two arguments where two IP addresses are passed. We want to determine whether *IP2* is a parent of *IP1* or not through this function. We also know, prefix lengths enables CIDR to perform "route aggregation". Therefore, *IP1Len* and *IP2Len* are also passed as prefix lengths of *IP1* and *IP2* respectively. Additionally, *STR1* and *STR2* are two char arrays that hold the extracted substrings temporarily.

The following pseudo code shows how the parent-child relationship between two IP addresses is determined.

```
function COMPARE (IP1,IP1Len,IP2,IP2Len)
  if IP2Len < IP1Len then

    STR1 = starting from MSB, extract substring from IP 1 of length IP2Len.
    STR2 = starting from MSB, extract substring from IP 2 of length IP2Len.

    if STR1 == STR2 then
      SET ANCESTOR_FLAG.
    endif

  endif

end COMPARE
```

Figure 4.7: Schema to determine the parent-child relationship between two IP addresses.

According to figure 4.7, two prerequisites must be meet before *IP2* can judged as the parent of *IP1*. These two conditions are:

1. *IP2*'s prefix length must be less than IP1's prefix length.
2. If and only if the first condition is fulfilled then extract substrings from both *IP1* and *IP2* (the extraction starts from MSB) of length *IP2Len*. If the extracted substrings are the same only then we can say that, *IP2* is a parent of *IP1*.

During this first round of aggregation, the "current" LM does not need to know what other LMs are advertising.

4.2.2.2 Delegation

In our context, the term "delegation" means that, there is another LM that is advertising a "more aggregated" EID-prefix. The concept of delegation is entirely our novel idea. The of task delegation essentially refers to a second round of aggregation. Exactly how a LM should learn about the delegation information is explained in the upcoming section 5.1. Based on the delegation information available to the current LM, aggregation is performed on the output of the first round of natural aggregation. The delegation information states, which EID-prefixes are advertised by "other" LMs. We then try to determine whether any of the EID-prefixes advertised by "other" LMs are parents of the EID-prefixes obtained as a result of the first round of aggregation (in the "current" LM). If so (i.e. a parent-child relationship exists) then we can say that, the "other" LM is better suited to handle the aggregation of the Map-Register message that has caused the output of the first round of aggregation.

Even as the author, the above sounds puzzling to me! An example should clear up all the confusion. Let's say, after the first round of aggregation, we have data that says, a LM located at RLOC 15.10.30.1 is advertising an EID-prefix 110.0.1.0/24. Now, this current LM gets to know that, another LM located at RLOC 100.200.30.40 is advertising an EID-prefix 110.0.0.0/8. After a second round of aggregation it becomes apparent that, 110.0.0.0/8 is a parent of 110.0.1.0/24. This means that, the Map-Register message that contained EID-prefix 110.0.1.0/24 should be sent to the LM located at RLOC

100.200.30.40 instead of 15.10.30.1. In other words, the task of aggregation for the Map-Register message that contained EID-prefix 110.0.1.0/24 should be delegated to the LM located at RLOC 100.200.30.40. Because, the LM at RLOC 100.200.30.40 is advertising a more aggregated EID-prefix (110.0.0.0/8 is certainly more aggregated than 110.0.1.0/24).

After the delegation phase, the UDP server sends a Map-Notify message with delegation information back to the UDP Client (or "xTR Extension").

4.2.2.3 Virtual Prefix generation: the decision process

When a new EID-prefix registers into CRM, we need to decide whether an existing LM can aggregate the new EID-prefix, or if some other LM is advertising a more aggregated prefix, or if we need to enforce EID-prefix aggregation through instantiating a virtual EID-prefix. According to [14], the choice depends on the "compactness" of the system that can be calculated from the number of announced unique EID-prefixes and LMs. If the number of LMs is less than square root of the number of announced identifiers then the system is considered compact and can grow. But when the number of LMs approaches the maximum allowed in the TZ scheme [33], a virtual EID-prefix needs to be instantiated.

We have successfully implemented and integrated this algorithm with CRM. Though the algorithm itself is not complicated, we will refrain ourselves from divulging its particulars. Because, it is a proprietary algorithm developed at Nokia Siemens Networks (NSN) and is still awaiting publication.

4.2.2.4 Virtual Prefix Generation:

For the sake of "better" aggregation, the address space can be partitioned into large prefixes; i.e. larger than any aggregatable prefix in use today. These prefixes are called virtual prefixes (VP). In other words, a VP is a prefix used to aggregate its contained regular prefixes. While generating VPs, an ISP divides the global address space into a set of virtual prefixes. For instance, an ISP could divide the IPv4 address space into 128 parts with a /7 representing each part (0.0.0.0/7 to 254.0.0.0/7) and uses it for a core/edge type of configuration (commonly seen today). That is, the core routers would maintain more specific routes, and edge routers could maintain default routes to the core routers, and suppress as much as they wish. Each ISP can independently select the size of its VPs [33, 49, 50].

Allocating such large aggregatable prefixes will yield an uneven distribution of real prefixes across the VPs. However, the VPs need not be of the same length; i.e. they can be a mix of /6, /7, /8 (for IPv4), and so on. As long as the VPs cover the complete address space, the ISP can choose them in such a way that they contain a comparable number of real prefixes. It should be evident by now that, the VPs are not topologically valid aggregates, i.e. there is not a single point in the Internet topology that can hierarchically aggregate the encompassed prefixes [33, 49, 50].

In CRM, after performing natural aggregation and delegation (which is actually a second round of aggregation based on delegation information), we end up with aggregated and orphan EID-prefixes (i.e. un-aggregatable addresses). If *Flinck et. al*'s [14] algorithm opts for it, only then, CRM will create VP. ISPs generate multiple VPs to cover the whole IP address space. Whereas in CRM, we will generate a single VP based on the aggregated and orphan EID-prefixes acquired after performing natural aggregation and delegation. For our purposes, the VP length that we have chosen is /16 and labeled it as

MAXGROUP.

In our prototype, we have generated this VP by observing the following steps:

1. Convert each of the aggregated and orphan EID-prefixes into binary.
2. Starting from MSB, extract substring from each of these EID-prefix of length MAXGROUP.
3. For every extracted substring (from EID-prefix), compare it with all the other substrings. If both substrings are deemed equals then DO NOTHING and move on to the next substring for further comparison.
4. If substrings are NOT equal then QUIT. Because unequal substrings mean that, we have failed to generate a suitable VP.

The critical factor that determines the possible success or failure of finding a VP is, its chosen length. The possibility of success is quite high if we choose a low VP length (e.g. /7,/8). In CRM, choosing a VP of length /16 proved to be sufficient for our purposes; as the number of EID-prefixes that we have worked on was limited. Opting for a low VP length also has its downsides. It would mean that, the LM has to "service" a large number of real prefixes; which might diminish its functionalities.

4.2.2.5 Hash value generation and Advertisement through Quagga

From the functional diagram shown in figure 4.5, it should be clear to the reader that, the last two tasks performed by the UDP server are: passing the concatenated routes to the MD5 Hash function and lastly, advertise the aggregator and community attributes. As stated in prior section 3.3, we have used an "off-the-shelf" software to generate the hash value and therefore its internal details is out of the scope of this work.

After the necessary aggregations are performed, we concatenate the routes (which will be the input of the MD5 Hash function) with only those EID prefixes that does not have any ancestor. This way, only when the aggregation changes; the input to the MD5 Hash function changes and subsequently, the change in the network's state is advertised through BGP's community attribute. To put it simply, a change in network's state means there was a change in the aggregation.

For performing successful BGP advertisement, the BGP daemon must be configured in two stages:

1. Static Configuration and
2. Dynamic Configuration.

At the initial stage, we make necessary changes manually (i.e. not through executing UDP server) to the *bgpd.conf* file, which in turn, is referred as static configuration. Changes in this file will only take effect after the BGP daemon is restarted.

In terms of dynamic configuration, Quagga comes with an additional tool known as *"vtysh"*, which acts as a single cohesive front-end to all its daemons (e.g. bgpd, ospfd etc.). It is actually an integrated user interface shell that connects to each daemon with UNIX domain socket and then works as a proxy for user input. CRM's UDP server uses *"vtysh"*; so that, BGP configuration change can take effect "on the run" [30].

Both the static and dynamic configurations are necessary for successful advertisement. The two following subsections will provide detailed information regarding the aforementioned two stages.

4.2.2.5.1 Static Configuration (BGP Daemon) The *bgpd.conf* configuration file can be divided into five main sections; namely, macros, global configuration, routing domain configuration, neighbors and groups and filter. Out of these, we are only concerned with the global configuration (contains the global settings for *bgpd*) and neighbors and groups (neighbor definitions and properties are set in this section) sections [11]. These are inputted manually for successful BGP advertisements. Every machine that is running our experimental mapping system, along with Quagga will have the following commands [12] in its *bgpd.conf* file:

1. *router bgp 100*
2. *bgp router-id 192.168.56.101*
3. *network 192.168.57.0/24*
4. *neighbor 10.144.17.27 remote-as 7675*
5. *neighbor 10.144.17.27 advertisement-interval 10*
6. *neighbor 10.144.17.27 send-community*
7. *neighbor 10.144.17.27 route-map OUTBOUND out*
8.
9. *route-map OUTBOUND permit 10*

In line 1, the *"router bgp 100"* command sets up a local BGP router belonging to *AS 100*. The next command in line 2 (i.e. *"bgp router-id 192.168.56.101"*) sets the router ID to the given IP address. This given IP must be local to the machine. The *"network 192.168.57.0/24"* command in line 3 announces the specified network as belonging to our AS (i.e. *AS 100*). All the three aforementioned commands belong to the global configuration section [51].

Next, we come to the neighbors and groups section, where each neighbor of the local BGP speaker is specified (we did not define any group in our configuration). In line 4, the command *"neighbor 10.144.17.27 remote-as 7675"* specifies an adjacent neighbor for the BGP router. The IP address and AS number belongs to the BGP peer [51].

In line 5, we set the time for the *minimum route advertisement interval (MRAI)*. The *MRAI* timer defines the minimum time interval between successive advertised updates of a prefix to a "peer", where by "peer", we indicate to every BGP speaker connected to the sending BGP speaker. The ip-address refers to the neighbor and the time is specified with an integer in seconds. Hence, the *MRAI timer* here is set to 10 seconds for the neighbor located at 10.144.17.27. The RFC for BGP-4 [52], along with the protocol definitions, also suggests the use of an *MRAI timer*. It should be noted that, the current Quagga implementation deploys a "burst" *MRAI timer* where all updates to a given BGP peer

[11] `http://resin.csoft.net/cgi-bin/man.cgi?section=5&topic=bgpd.conf`

[12] Only the relevant commands are explained here. Also for the sake of clarity, line numbers are pre-pended with each routing command.

are held until the next MRAI timer interval expires, at which time all queued updates are announced. Such a solution was chosen because of its implementation simplicity. Though, the RFC for BGP-4 [52] explicitly states that the timer should affect updates as well as withdrawals (we did not perform any route withdrawal in our configuration), Quagga only applies the *MRAI timer* on updates and not on withdrawals [53].

In line 6, we have used the *"neighbor 10.144.17.27 send-community"* command to include the community attribute in route updates sent to the specified BGP neighbor located at 10.144.17.27. This is an integral part of our work. Because instead of using it for routing/propagation decisions, we have altered its used for the dissemination of end point reachability information.

In line 7, we have used a route map on a per-neighbor basis to filter updates. In general, route maps are used to control and modify routing information that is exchanged between routing domains. A route map can be applied to either in bound or outbound updates (we have only used outbound updates in our configuration) and it enables the user to control the redistribution of routes between two BGP peers. In our configuration, the neighbor route-map out command applied to the route map OUTBOUND for outgoing routes to neighbor 10.144.17.27.

In line 9, we define the aforementioned route-map by the command, *"route-map OUTBOUND 10"*. *10* is just a sequence number here and for our purposes does not mean anything [13].

4.2.2.5.2 Dynamic Configuration (BGP Daemon) This part of the configuration is performed by UDP server as it is executing. Values for two BGP attributes are generated by the UDP server "on the fly". These two attributes are: aggregator and community value.

In order to execute the necessary commands, we have utilized Quagga's integrated user interface shell or *"vtysh"* with the *-c* option. This option is useful for gathering info from Quagga or reconfiguring daemons from inside shell scripts. The command format for using the *-c* option looks like: *vtysh -c "command"*. For example, if we want to observe the routing table for BGP from BASH, then the command would be: *vtysh -c "sh ip bgp"*. After the command is executed, *"vtysh"* exits [30].

The relevant command for setting the *aggregator* is: *set aggregator as [as-number] [ip-addr]* where *[as-number]* is AS number of the *aggregator* (an integer between 1 to 65535) and *[ip-addr]* is the IP address of the aggregator. On the other hand, the command for setting the community value is: *set community [community-number]* where *[community-number]* specifies a valid community number between 1 and 4294967200. We have used the colon-separated format of expressing community values [14]. As indicated previously, one of CRM's novelties lies in the fact that, the community attribute here does not affect the routing or propagation decision; rather, it carries end-point reachability information. Therefore, instead of inserting an ASN in the first 16 bits, the output from the MD5 hash function is placed here. The remaining 16 bits are always set to 0, as we have no use for local preference value in CRM.

Once the necessary *"vtysh"* commands are constructed with appropriate *aggregator* and *community values*, we execute them with *system()* function that takes valid Linux Shell command as a char array in its argument [59]. For example, the command for setting the

[13] Route-Maps for IP Routing Protocol Redistribution Configuration. `http://www.cisco.com/en/US/tech/tk365/technologies_tech_note09186a008047915d.shtml`

[14] Refer to previous subsection 2.6.6.

community value would be executed as:

system("vtysh -c \"set community 2369:0\""). Note that, we have used an escape sequence where the double quotation mark preceded by a backslash (\"). We have done this so that, it specifies a literal double quotation mark (as the double quotation mark has special meaning in C).

4.2.3 TCP Client (for EID Topology Discovery)

Like an "usual" TCP client, our system does not start off by creating a connection with the TCP server. Rather, every 5 seconds the client machine checks the BGP announcements received through Quagga. From these announcements, the TCP client extracts the *community* and *aggregator* attributes. These acquired *community* and *aggregator* attributes might be "new" to the DB. Or, there might be a change in the community value (from the immediate previous instance) announced by a particular *aggregator*. Only due to the two above mentioned reasons will our TCP client connect with the TCP server. Change in *community* value is an indicator of a change in the network state. The *aggregator* on the other hand, is an optional transitive attribute which may be included in BGP updates. A BGP speaker that performs route aggregation may add this attribute which would contain its own AS number and BGP Identifier [52]. In our use case, we have mandated that, this attribute needs to be used to identify the particular LM that is advertising the BGP announcements.

For learning from BGP's announcements, our system uses *vtysh*, which is an integrated shell for Quagga. Again, the commands are given in *#vtysh -c "command"* format, so that we may run them from the Linux shell. We have used the *popen()* function to execute these Quagga commands. In other words, we have used process pipes. The *popen()* function allows a program to invoke another program as a new process and either pass data to it or receive data from it. The command string is the name of the program to run, together with any parameters. In our case, we are running Quagga commands with the parameter *"-c"*. The *popen()* function's prototype also has an *open_ mode* that must be either *"r"* or *"w"*. In our case, the *open_ mode* is set to *"r"*, because output from the invoked program (i.e. *#vtysh -c "command"*) is made available to the invoking program (i.e. TCP client) in order to read from the file stream *FILE** that is returned by *popen()*. The file stream is read using the usual *stdio* library function *fread()* [47].

The TCP client program does not know beforehand which network is advertised by the LM. To obtain this information, we execute the *#vtysh -c "show ip bgp"* command to get all the entries in the BGP routing table. The output looks something like:

```
xTR# sh ip bgp
BGP table version is 0, local router ID is 10.144.17.65
Status codes: s suppressed, d damped, h history, * valid, > best, i - internal,
              r RIB-failure, S Stale, R Removed
Origin codes: i - IGP, e - EGP, ? – incomplete

   Network          Next Hop            Metric LocPrf Weight Path
*> 192.168.87.0     192.168.56.103           0             0 100 i
*> 192.168.45.0     192.168.56.102                         0 100 200 i
*> 192.168.57.0     192.168.56.103           0             0 100 i
*> 192.168.67.0     0.0.0.0                  0         32768 i

Total number of prefixes 4
```

Figure 4.8: Command output for *#vtysh -c "show ip bgp"*.

This output is acquired as a char array by the TCP client. Then we use a regular expression [15] to extract all the IP addresses from this output. Our regular expression looks like:

$$[[: \textit{digit} :]]\{1,3\}\backslash\backslash.[[: \textit{digit} :]]\{1,3\}\backslash\backslash.[[: \textit{digit} :]]\{1,3\}\backslash\backslash.[[: \textit{digit} :]]\{1,3\}$$

As the whole prototype adheres to POSIX standard, we have used POSIX character set [94] for our regular expression. Here, *[[:digit:]]* refers to any number between 0 and 9. Using curly braces (i.e. *{1,3}*) we have stipulated how many times the preceding digit is allowed to be repeated. So, *[[:digit:]]{1,3}* matches characters that are digits and can be repeated at least 1 time but not more than 3 times. In regular expression, the "dot" (.) is a metacharacter [16] . The backslashes are used to turns off the special meaning of this metacharacter. Thus, a period is matched by a "\.". However, in addition to any valid IP address, the aforementioned regular expression will also match 999.999.999.999 as if it were a valid IP address. But the output that we acquire from *#vtysh -c "show ip bgp"* command will only have valid IP addresses. Thus the aforementioned regular expression will suffice in detecting proper IP addresses. For executing the regular expression, POSIX regex functions: *regcomp()* and *regexec()* are used (defined in regex.h).

Afterwards, for each extracted IP address we execute, *#vtysh -c "show ip bgp IP_address"* command. Only a network that is advertised by a LM would have an *aggregator* and *community* value in the output , like the following:

[15] A regular expression is a pattern describing a certain amount of text. `http://www.grymoire.com/Unix/Regular.html`

[16] A metacharacter is a character that has a special meaning; instead of a literal meaning to a regular expression engine.

vtysh -c “sh ip bgp 192.168.87.0”
BGP routing table entry for 192.168.87.0/32
Paths: (1 available, best #1, table Default-IP-Routing-Table)
Not advertised to any peer
100, (**aggregated by 7675 15.10.30.1**)
192.168.56.103 from 192.168.56.103 (10.144.13.56)
Origin IGP, metric 0, localpref 100, valid, external, best
Community: 2310:0
Last update: Fri Jan 2 17:04:10 1970

Figure 4.9: Command output for *#vtysh -c "show ip bgp 192.168.87.0"*.

We had to be innovative because there is no "direct" command (for Quagga) to extract the *aggregator* attribute for a particular network. Any network that is not advertised by the LM will not have these attributes. As figure 4.9 shows, the command *#vtysh -c "show ip bgp 192.168.87.0"* has produced 15.10.30.1 as *aggregator* and *2310:0* as *community* value. This means that, 15.10.30.1 is the LM's ID.

After the *aggregator* (or the ID of the LM) and community attributes are extracted from the output char array through pattern matching, two things can happen. For one, the acquired LM ID and its *community* value might be absent from the DB. If this is the case, then we *INSERT* this new information. Secondly, there might be a pre-existing *community* value for that LM. In this case, we just compare the newly acquired *community* value with its immediate previous instance for that particular LM. In both scenarios, the TCP server will be connected; so that the client can get an updated view of the network state.

The TCP client gets to know the server's address by inquiring (through SQL's SELECT statement) the ServerAddress table. This task is performed by using the usual methods described in the upcoming section 4.3.6.

Afterwards, a connection is established and the client starts to receive information by taking the following steps:

1. For establishing a connection, the client creates the basic construct of a socket by utilizing *socket()* system call.
2. Information such as IP address of the server and its port number is bundled up in a structure and a call to *connect()* system call is made which tries to connect this socket with the server. Note that, we have not bounded (i.e. bind) our client's socket to any particular port. This is because the client does not require its socket to be attached to any well-known port and so, it generally uses a port assigned by the kernel (i.e. it is an ephemeral port).
3. Once the connection is made, the server starts to send data through the client's socket descriptor. The client reads this data through the *read()* system call on the its socket descriptor.

After the data is read in, we either perform only INSERT operation or carry out DELETE and INSERT operation on the EIDList table for a particular LM (which is equivalent to UPDATE) by using the well-known functions described in the upcoming section 4.3.6. By doing so, the EIDList table on the client side will now contain an updated view of the network state.

4.2.4 TCP Server

For our purposes, a "classic" TCP server is sufficient and we have build it by carrying out the following steps:

1. Create a socket with a call to *socket()*,
2. Fill this socket's address structure with *INADDR_ANY* (this allows the server to accept a client connection on any interface; i.e. it is a wildcard address) and port number (we have used 9877). We have chosen the port number in a manner so that it is greater than 1023 (because we do not need a reserved port), greater than 5000 (to avoid conflict with the ephemeral ports) and finally, less than 49152 (to avoid conflict with the "correct" range of ephemeral ports). Additionally, the chosen port number should not conflict with any registered port.
3. Bind the server's port (i.e. 9877) to the socket by calling *bind()*. It should be noted that, the concept of using a TCP server is novel to CRM. None of the other previously defined mapping systems (e.g. LISP-ALT) have any such entity. Therefore, we are not bounded by LISP [13] or any specification of any other mapping system to use a particular port. We are free to choose any port number as long as it meets the requirements mentioned in step 2.
4. By calling *listen()*, the socket is then converted into a listening socket, on which incoming connections from clients will be accepted by the kernel.

The execution of *socket()*, *bind()*, and *listen()*, are the normal steps for any TCP server to prepare what is call the listening descriptor (listenfd in our case).

Now, it is our intention to design and implement a multi-client TCP server. In a simplistic a single-user system, we usually run in a *"busy wait"* loop, repeatedly scan the input for data and read it if it arrives. But, this behavior is very expensive in terms of CPU time. For a TCP server which supports multiple client simultaneously, we must have the capability to tell the kernel that we want to be notified if one or more I/O conditions are ready (i.e. input is ready to be read, or the descriptor is capable of taking more output). This capability is called I/O multiplexing and is provided by the *select()* and *poll()* functions. In our case, I/O multiplexing is used so that, the TCP server can use both the listening socket and its clients' connection sockets at the same time. In other words, the TCP server can deal with multiple clients by waiting for a request on many open sockets at the same time. For this, we have used *select()* in our TCP server. This system call in our TCP server looks like:

```
result = select(nfds, &readfds, (fd_set *)NULL,
(fd_set *)NULL, NULL);
```

The first parameter *nfds* refers to the highest-numbered file descriptor in any of the three sets, plus 1. Three independent sets of file descriptors are watched by *select()*. Those listed in *readfds* will be watched to see if characters become available for reading. As our TCP server is only interested in reading data from multiple clients, the other two descriptor sets is set to *NULL*. The last parameter in *select()* is *timeout* and is set to a *NULL* pointer, which in turn means that if there is no activity on the sockets, the call will block forever.

Four macros are used to manipulate these descriptor sets. *FD_ZERO()* clears a set. *FD_SET()* and *FD_CLR()* respectively adds and removes a given file descriptor from a set. Lastly, *FD_ISSET()* tests to see if a file descriptor is part of the set or not.

After select returns, the descriptor sets will have been modified to indicate which descriptors are ready for reading.

We now continue the steps for building an I/O multiplexed TCP server:

5. Wait for clients and requests inside an infinite loop (or *"busy wait"* loop). Through *select()* we instruct the kernel to notify us if a file descriptor is ready for reading. We have used *FD_ISSET* to determine which descriptor(s) is showing activity or needs attention.

6. If the activity is on a listening socket then it must be a request for a new connection from the client; which we allow by calling *accept()*. This system call creates a new socket to communicate with the client and returns its descriptor. The newly created socket will have the same type as the server's listen socket. We add this newly created file descriptor to *read_fd_set* by calling the macro *FD_SET*.

7. If the activity is not on a listening socket then it must be due to a client socket. If close is received (i.e. the client has gone away) then we terminate the socket connection by calling *close()* system call. In our server, we have done this when *read()* returns zero. We also remove the client socket descriptor from *read_fd_set* by calling the macro *FD_CLR*.

8. If none of the aforementioned two conditions apply then it means that, we need to "serve" the client (and *read()* does not return zero or -1). To do so, we carry out *SELECT* operation on the EIDList table for a particular LM by using the well-known functions described in the upcoming section 4.3.6. Then we process the extracted contents so that it can be sent to the requesting client as a char array. The *write()* system call is used to write this newly created char array in the client's socket.

9. Continue the loop and wait for new clients and requests.

The following figure gives a detailed functional diagram of the interaction between TCP client and server. The dashed line in the middle separates client from the server.

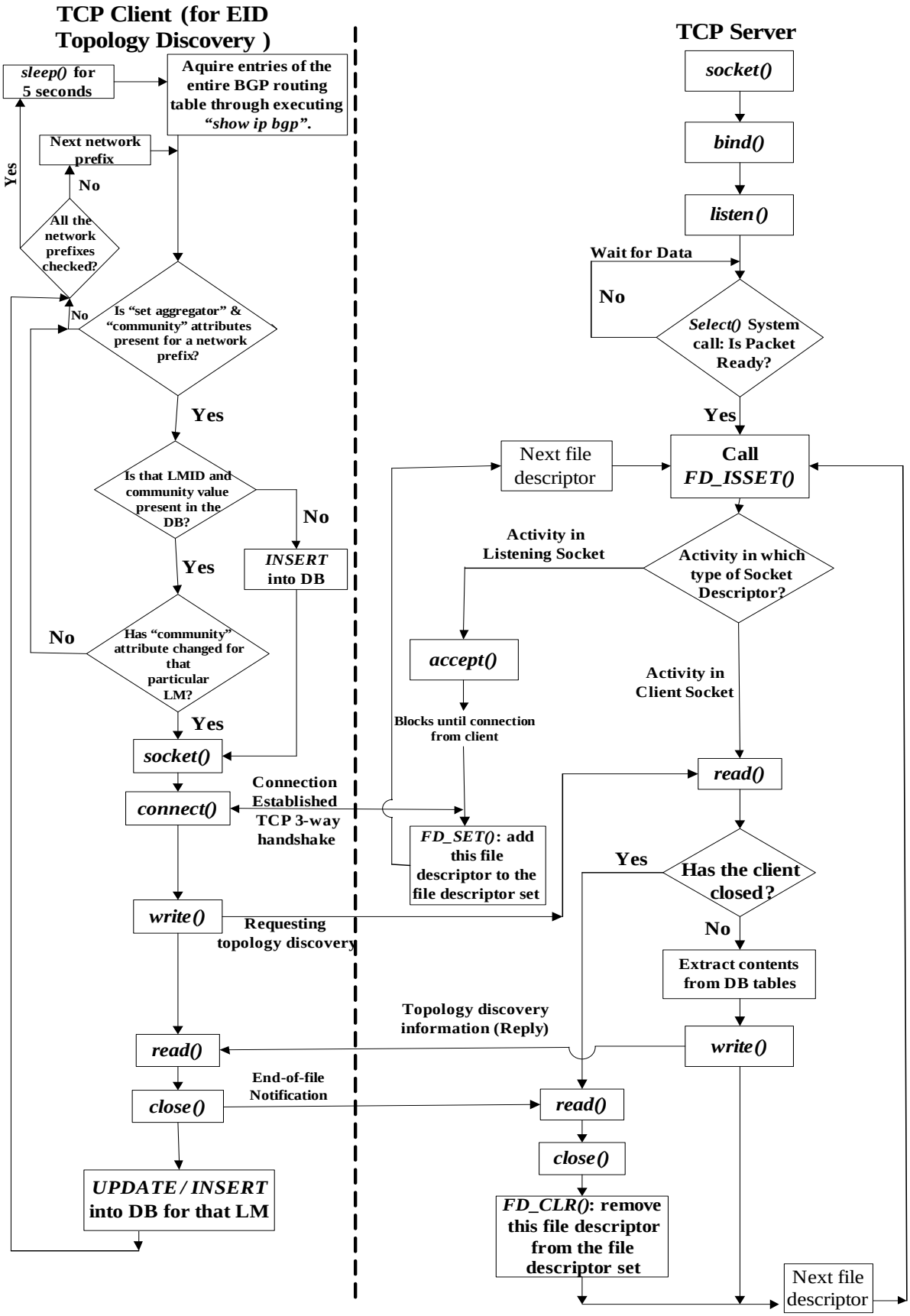

Figure 4.10: Functional diagram of CRM's TCP client and server.

4.3 Database Design

From the descriptions provided thus far on CRM, it should be clear to the reader that database is an integral part of this system. A database (DB) is not merely the place where data are stored; it also contains information regarding the relationships between those data. The user phrases the desired data manipulation requests in terms of data relationships. Afterwards, a piece of software known as a database management system (DBMS) translates between the user's request for data and the physical data storage. We have used MySQL as the DBMS for its Open-Source qualities. Within MySQL, we have created a DB called *LMIDstructure* for CRM.

The formal way in which the user express data relationships to a DBMS is known as a data model. Most DBMSs support only one data model. MySQL supports relational data model and therefore is at the center of our discussion. Before going into the design of *LMIDstructure* to understand its relational data model, we need to identify the data relationships. The DBMS-independent technique for documenting these data relationships is known as the entity-relationship diagram (ER diagram). Prior to describing CRM's ER diagram we will provide a short discussion on entities, their attributes, entity identifiers and referential integrity; so that the reader does not get confused with these terminologies. At the end of this subsection, we will also present a short description of the actual implementation [23, 56].

4.3.1 Entities and their Attributes

An entity is something about which we store data. A *LMIDTable* is an entity, as is a *EID-routing* that holds the mapping between a LM and EID-prefix. Though entities can be abstract, in CRM, all the entities are tangible. The *LMIDstructure* database that we had formed for CRM creates a directory that contains files corresponding to the entities of this database.

Entities have attributes that describe them. For example, an *EID-routing* entity is usually described by a *LMID*, *EIDPrefix*, *EIDPrefixLength*, *Is_Delegated* and *Delegated_RLOC*. A *LMIDTable* entity on the other hand, has attributes such as, *LMID* and *HASH_VALUE*. Each attribute has a domain, i.e. permissible values for that attribute. Usually, before assigning domains to the attributes we (i.e. DB designers) looked at the DBMS (i.e. MySQL) to see which data types are supported. For our purposes, MySQL supported data types *CHAR*, *VARCHAR(maxlength)* and *INT* were the relevant ones. In CRM, a lot of data regarding IPv4 addresses are handled. IPv4 addresses are stored with the domain *VARCHAR(maxlength)*; which contains variable length strings of a maximum length, 16 (i.e. maxlength). Because an IPv4 address expressed as a string can have a maximum length of 16. The DBMS (i.e. MySQL) enforces a domain through a domain constraint. Whenever a value is stored in the database, the DBMS verifies that it comes from that attribute's specified domain.

When we represent these entities in a database, we actually store only the attributes. Each group of attributes that describes a single real world occurrence of an entity acts to represent an instance of that entity. For example, in figure 4.11, you can see four instances of *EID-routing* entity stored in our database. If we had 1000 mappings, then there would be 1000 collections of *EID-routing* attributes. We should keep in mind that the figure does not make any statements about how the instances are physically stored. The following figure 4.11 is purely a conceptual representation of instances/records [23, 56].

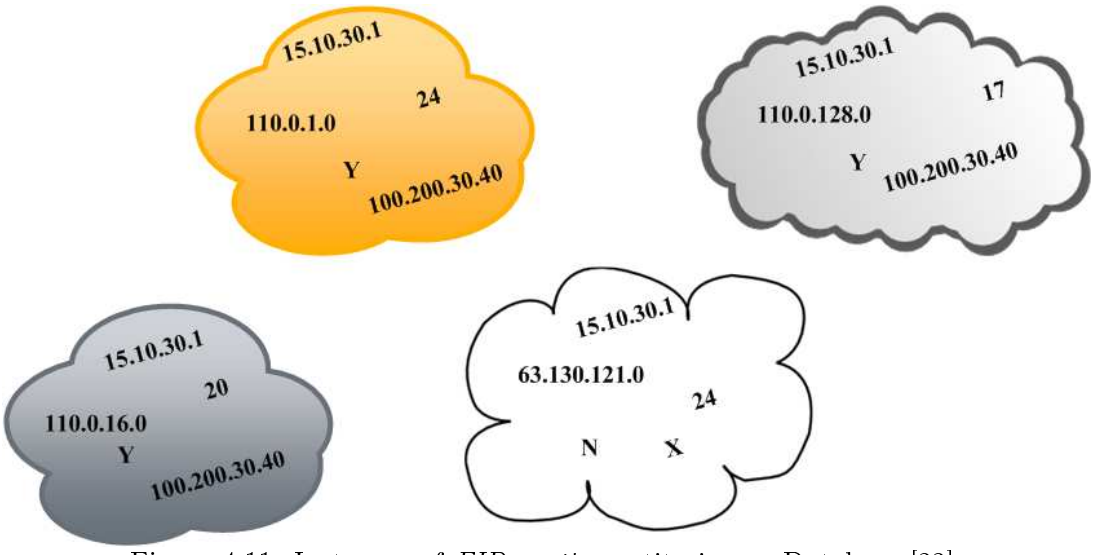

Figure 4.11: Instances of *EID-routing* entity in our Database [23].

4.3.2 Entity Identifiers

The only purpose for putting the data that describes an entity into a database is to retrieve that data at some later time. This means that we must have some distinguishing marker that separates one entity instance from another; so that we can always be certain that we are retrieving the precise entity instance we want. This is called an entity identifier. It is also known as a Primary Key.

In CRM, for example, the *LMIDTable* entity has *LMID*, each of which is unique in the whole Internet; making it an entity identifier that distinguishes one LM from the other. Another solution would be to join the *LMID* and *EIDPrefix* and *EIDPrefixLength* attributes of the *EID-routing* entity. This combination of columns i.e. a concatenated identifier, would uniquely identify each mapping. The *EID-routing* entity requires a concatenated identifier; because there can be multiple mappings for each LM. Also each EID can only be uniquely identified with a combination of *EIDPrefix* and *EIDPrefixLength*. Therefore, each mapping is distinctively recognized with a concatenated identifier consisting of *LMID* and *EIDPrefix* and *EIDPrefixLength*.

When we store an instance of an entity in a database, we want the DBMS to ensure that the new instance has an unique identifier. The above is an example of a constraint on a database, a rule to which data must adhere. The enforcement of such a constraint helps us to maintain data consistency and accuracy [23, 56].

4.3.3 Foreign Key and Referential Integrity

When an entity contains an attribute that is the same as the primary key of another entity, then that attribute is called a foreign key. The matching of foreign keys to primary keys represents data relationships in a relational database.

Foreign keys may be a part of a concatenated primary key or they may not be part of the entity's primary key at all. In CRM, for example, the *LMIDTable* and *EID-routing* entities:

LMIDTable (LMID, HASH_VALUE) EID-routing (LMID, EIDPrefix, EIDPrefixLength, Is_Delegated, Delegated_RLOC)

The *LMID* attribute in the *EID-routing* entity is a foreign key that matches the primary key of the *LMIDTable*. Here, the *LMID* attribute is a part of the primary key of *EID-routing* entity. Unless the foreign key is a part of a concatenated primary key (which is in our case); they can be null.

With foreign key, comes a constraint called referential integrity enforced by the relational data model. It states that, every non-null foreign key value must match an existing primary key value. Of all the constraints on a relational database, this is probably the most important; because it ensures the consistency of the cross-references among instances of entities. Referential integrity constraints are stored in the database and enforced automatically by the DBMS (i.e. MySQL). As with all other constraints, each time a user/application enters or modifies data, the DBMS checks the constraints and verifies that they are met. If the constraints are violated, the data modification will not be allowed [23, 56].

Referential integrity also includes the techniques known as cascading update and cascading delete. Let's consider CRM's situation concerning entities: *LMIDTable* and *EID-routing*. Referential integrity enforces the following three rules:

1. We may not add an instance/record to the *EID-routing* entity unless the *LMID* attribute points to a valid record in the *LMIDTable* entity.
2. If the primary key for a record in the *LMIDTable* entity changes, all corresponding records in the *EID-routing* entity must be modified using a cascading update.
3. If a record in the *LMIDTable* entity is deleted, all corresponding records in the *EID-routing* entity must be deleted using a cascading delete.

Before we move on to the ER diagram, it should be mentioned that, most of the entities are identical in both client and server of CRM. This is because, in CRM, after information is disseminated, both client and server should have matching information. *LMIDTable* and *EID-routing* are the identical entities that reside in both client and server to hold the disseminated information. There are however, couple of client/server specific isolated entities. Such entities do not possess any relationship with other entities and therefore, will be mentioned briefly in the next section. [17]

4.3.4 ER Diagram

ER diagrams provide a way to document the entities in a database along with the attributes that describe them. There are actually several styles of ER diagrams. The two most commonly used styles are Chen and Information Engineering (IE). The IE model tends to produce a less cluttered diagram and therefore will be used for all the diagrams in this thesis. In the IE model, rectangles are used to represent entities. Each entity's name along with its attributes appear inside the rectangle.

The relationships that are stored in a database are between instances of entities. There are three basic types of relationships that we encounter: one-to-one, one-to-many and many-to-many. The relationship between *LMIDTable* and *EID-routing* is one-to-many and is therefore discussed in detail [23, 56].

[17] Interested readers are advised to see Appendix B where documentation regarding the definition of all the entities in terms of SQL is provided.

As a formal definition, if we have instances of two entities (*LMIDTable* and *EID-routing*), then a one-to-many relationship exists between two instances ($LMIDTable_i$ and $EID\text{-}routing_i$) if $LMIDTable_i$ is related to zero, one, or more instances of entity *EID-routing* and $EID\text{-}routing_i$ is related to at most one instance of entity *LMIDTable*. The relationship between instances of *LMIDTable* and *EID-routing* is one-to-many, because each LM can have multiple mappings (i.e. instances of *EID-routing*) associated with it. Hence, the ER diagram looks like:

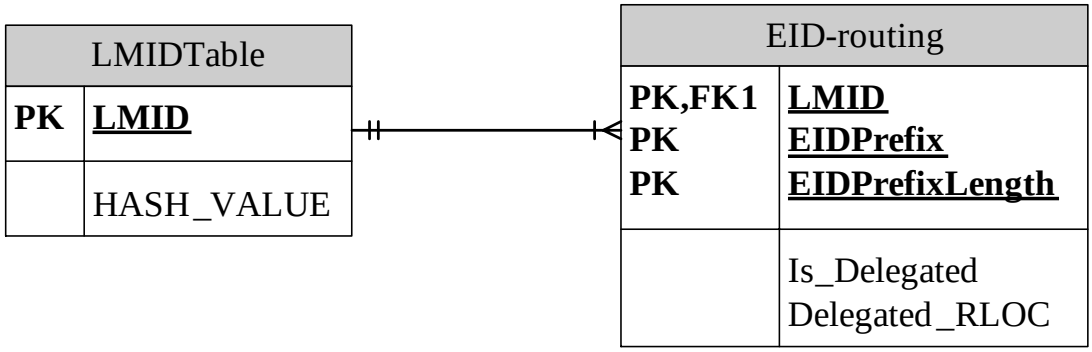

Figure 4.12: One-to-many relationship between *LMIDTable* and *EID-routing*.

The double line on the right side of *LMIDTable* entity means that each mapping (i.e. a single instance of *EID-routing* entity) is related to one and only one instance of *LMIDTable*. As zero is not an option, the relationship is mandatory. The *three-legged teepee* connected to the *EID-routing* entity means that, an instance of *LMIDTable* will have one, or more mappings (i.e. instances of *EID-routing*) associated with it. This is because, TCP client will contact the designated TCP server to update itself with the state of the network that is "behind" the LM. In other words, every LM will have one or more mappings associated with it. Therefore, the "many" side of the relationship must also be mandatory.

Now that we described the relationship between two of the most important entities of CRM, we will now provide a complete view of the entire database residing in CRM. In figure 4.13, on the left, we see the client side and on the right, we have the server side.

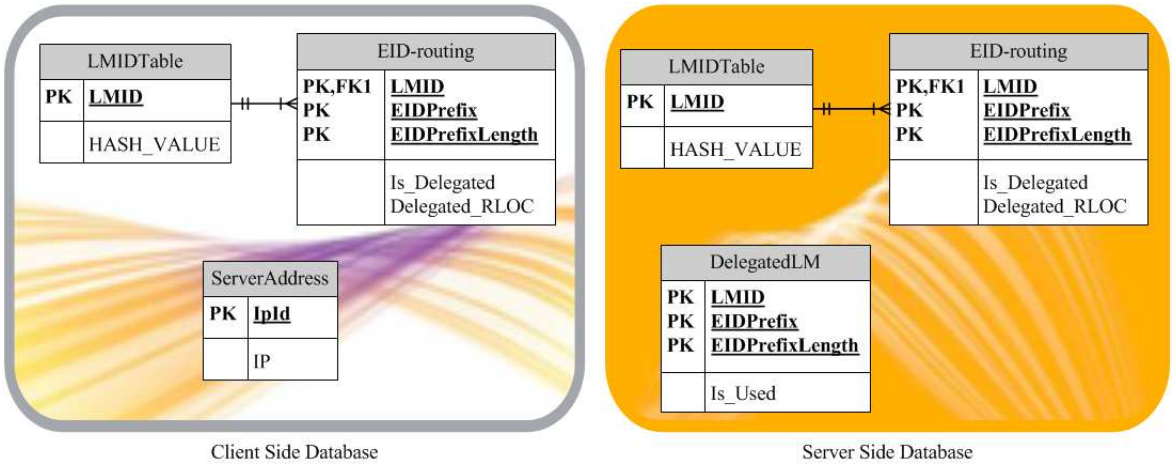

Figure 4.13: complete view of the entire database residing in CRM.

As expected, *LMIDTable* and *EID-routing* are present in both client and server side; as they have to hold matching information. On the client side, there is an "isolated" table with the definition:

ServerAddress(IpId, IP);

When the Map-Notify message returns to the UDP client, the server's IP address (found within this Map-Notify message) is stored within this table. So that, the TCP client knows which server to connect.

On the server side, we see another "isolated" table with the definition:

DelegatedLM (LMID, EIDPrefix, EIDPrefixLength, Is_Used);

Once the "current" LM obtains the delegation information (which is actually the mappings announced by "other" LMs) [18], it stores that information in the *Delegated_LM* table. Based on this information, the UDP server performs the task delegation (i.e. a second round of aggregation).

4.3.5 The Relational Data Model

Once the ER diagram is completed, we can translate it into the formal relational data model required by our DBMS (i.e. MySQL). We call databases that adhere to this model as relational databases. A relational database is a database whose logical structure is made up of nothing but a collection of relations (or entities). In mathematical set theory, a relation is the definition of a table with columns (attributes) and rows (tuples). (The word "table" is used synonymously with "relation"). The definition specifies what will be contained in each column of the table, but does not include data. When rows of data are included, we get an instance of a relation.

The diligent reader should notice that, translating from ER diagram to relational data model changes the terminologies; while the concept underneath remains the same. From onwards, *LMIDTable* and *EID-routing* entities will be termed as *LMIDTable* and *EID-routing* tables, attributes will be known as columns. A column in a relation must have two properties: for one, the column's name must be unique within the table. And secondly, every column in a table must be subjected to a domain constraint (same as, attribute's domain constraint). A row in a relation, must also possess two properties: firstly, only one value is allowed at the intersection of a column and row; i.e. a relation cannot allow multi-valued attributes. Secondly, there are no duplicate rows in a relation. To enforce unique row constraint the relation (or table) must define a primary key. In terms of relational data model, a primary key is a column or combination of columns that uniquely identifies each row [23, 56].

4.3.6 Implementation

The MySQL version that we have used for our system is:

mysql Ver 14.14 Distrib 5.1.49, for debian-linux-gnu (i686) using readline 6.1

In order to access MySQL from CRM, we had to use mysql.h, which is the most important header file for MySQL function calls. All the functions mentioned in the current subsection is defined in mysql.h. In our opinion, the APIs used to access/manipulate MySQL data are not common knowledge. Because they differ for every DBMS. That is why, we will provide short descriptions about these APIs and how they are utilized.

There are two steps involved in connecting to a MySQL database from C (CRM is coded in C) and they are:

[18] How the "current" LM gets the delegation information is explained in the upcoming section 5.1

1. Initialize a connection handle structure and
2. Physically make the connection.

For performing the first step 1:

```
MYSQL* conn_ptr_db;

conn_ptr_db = mysql_init(NULL);
```

We have pass NULL to the *mysql_init()* routine, and a pointer to a newly allocated connection handle structure is returned (in *conn_ptr_db*).

For step 2, we offer the parameters for a connection using the mysql_real_connect() routine:

```
mysql_real_connect(conn_ptr_db,"localhost",
"tunnelRouterCli","xTR","LMIDstructure",0,NULL,0);
```

where *conn_ptr_db* is the connection handle structure, *"localhost"* is the current system's hostname, *"tunnelRouterCli"* is the user name for MySQL, *"xTR"* is the MySQL password and "*LMIDstructure*" is the database that we have created. The rest of the parameters can be set to *0* or *NULL* for our purposes.

After the program (i.e. CRM) gets connected with MySQL, we performed four distinct operation on the data: *INSERT*, *DELETE*, *UPDATE* and *SELECT*. The first three are fairly simple to perform as they do not return any data. We simply execute valid SQL as a text string using the API:

```
int mysql_query(MYSQL *connection,
const char *query);
```

where the first parameter is connection handle structure (i.e. *conn_ptr_db*) and second one is a SQL command in a NULL terminated string.

As the SQL *SELECT* statement returns some data, it poses a little more challenge. At first, like *INSERT/DELETE/UPDATE* statements, we use *mysql_query()* to execute the SELECT statement. After its successful completion, we will start to retrieve the data using *mysql_use_result(MYSQL *connection)*. Data is acquired in a row by row manner. The *mysql_use_result()* routine returns a pointer to a new structure called a result set structure (of type *MYSQL_RES **), or *NULL* if it fails. Let's say, result is such a pointer. Now that we have the data (inside the result set structure), we can process it using *mysql_fetch_row(result)*.The *mysql_fetch_row(result)* function pulls a single row out of the result structure and puts it in a row structure (of type *MYSQL_ROW*). A variable of type *MYSQL_ROW* is a pointer to an array of strings representing the values of each column in a single row. The *mysql_fetch_row(result)* function is run in a loop so that all the rows in a table are returned. It returns NULL when the data runs out or if an error occurs. Finally, when we are finished with the result set structure (i.e. *MYSQL_RES* result*), we use *mysql_free_result(result)*, so that MySQL library can clean up the objects it has allocated.

As stated in sub-section 3.2.1, in CRM, SQL statements are suitably grouped into transactions. The syntaxes that we have used for this purpose are *START TRANSACTION*, *COMMIT* and *ROLLBACK*. All of these are executed using *mysql_query()* function.

Chapter 5

Analysis

5.1 Results

We begin analyzing this work by explaining the rational for utilizing compact routing; why we need a novel idea for solving the routing scalability problem. The Routing Research Group (RRG) is responsible for research and recommendation of a new routing architecture for the Internet. The survey conducted in [36], analyses many of the proposals [1] that were put in front of the RRG group. This survey also includes additional investigation regarding some of the concerns with specific proposals and how some of those concerns should be mitigated [36]. For practical purposes, our comparison is limited between LISP-ALT and CRM. Because, all the other aforementioned systems have not progressed any further from a proposal; they neither have any existing implementation nor have plans of any future development; they do not even have any simulation and most importantly any vendor backup. LISP-ALT on the other hand, has gained the backing of Cisco and thus immerged as the de-facto standard for a mapping system of LISP. Therefore, any mapping system created for LISP would have to be scrutinized against the perceived benefits of LISP-ALT. Moreover, we will explain the motivations that has convinced us to ground critical functions that affect the scalability of our routing system (i.e. CRM) into Compact Routing.

In LISP-ALT, when a cache miss occurs, the ITR will create a Map-Request for the destination EID (the idea of encapsulating the original packet as a data probe is deprecated) which is then sent to an ALT Router. A Map-Request message might have to traverse an undefined number of ALT routers before reaching the desired authoritative ETR for the destination EID-prefix. This authoritative ETR will afterwards respond to the ITR with a Map-Reply message. However, the initial packet that had caused the cache miss is dropped [17].

The aforementioned operations cause "initial packet delays", which in turn degrade performance and thus create major hurdles in the voluntary adoption of LISP-ALT on a wide enough basis to solve the routing scalability problem [36]. Conversely, in a system where CRM provides mapping facility, if an ITR receives a packet originated by an end system within its site (i.e. a host for which the ITR is on exit path out of the site) and the destination EID for that packet is not known in the ITR's mapping cache, then a cache-miss event occurs. This will prompt the ITR to generate a Map-Request message

[1] For example: LISP-ALT, Routing Architecture for the Next Generation Internet (RANGI), Internet Vastly Improved Plumbing (Ivip), Hierarchical IPv4 Framework (hIPv4), Compact routing in locator identifier mapping system, Global Locator, Local Locator, and Identifier Split (GLI-Split), Tunneled Inter-domain Routing (TIDR) etc.

for the destination EID. The packet causing this cache-miss is piggy backed on to the Map-Request message and is relayed to the LM that is servicing the source ITR. This LM will do either one of three things:

1. If this LM knows the required mapping through its own *"EID-forwarding table"* [2] then it forwards Map-Request (along with the piggy backed packet causing the cache miss) to the correct ETR.
2. This LM might forward the message to another LM that is announcing a more aggregated prefix.
3. The last option for this LM is to drop the packet if it cannot perform either of the two actions mentioned earlier.

It should be mentioned that, in a LM-space, all the LMs know about each others' announcements. Hence, it might be so that, the LM servicing the source ITR might not announce matching EID-prefix; however, it will certainly know which other LM is announcing matching EID through its own EID forwarding table. This 2-level hierarchy (i.e. a Map-Request only has to traverse MS and then LM to know the mapping) is a key characteristic of Compact Routing that restricts the stretch of the path under a given constraint ($\leq$3) which in turn, guarantees an upper bound to the delay caused by the initial "cache miss packet" (in the control plane). In LISP-ALT, the Map-Request message may traverse an undefined number of ALT routers before it reaches the destination ETR. In other words, LISP-ALT cannot ensure any upper bound to the control plane delay.

An ALT network's delays are compounded by its inherent "aggressive aggregation", without considering the geographic location of the routers. Tunnels between ALT routers may span intercontinental distances and traverse many Internet routers [36]. On the other hand, a CRM will have strict policies while distributing aggregation capabilities, that takes into account the geographical location of MSs and LMs.

Compact Routing guarantees that, RT size will grow sub-linearly. This can be ensured because, in Compact Routing, an end-system must know the routes to all the LMs in the LM-space [3]. This gives the upper size limit of the RT inside an end-system [4]. If LISP-ALT is deployed as the mapping system, there will be no such restrictions regarding the growth of RTs. In our CRM, an end-system is made aware of the routes to its LM through the information passed via the Map-Notify message.

Most mapping systems designed to operate between two distinct namespaces (e.g. LISP-ALT, LISP-DHT, CRM etc.), will create an overlay routing system [5]. In such a scheme, there are two routing systems "sitting" on top of each other. One deals with EIDs, while the other with RLOCs. Here, the EID based routing is termed as the "overlay" and it involves the use of two data structures, namely, *"EID-routing table"* and *"EID-forwarding table"*. At the bottom, we have Quagga, which advertises RLOC through BGP. The *"RLOC-RIB"* and *"RLOC-FIB"* (shown in figure 5.1) refers to the regular RIB and FIB found in routing softwares (e.g. Cisco's IOS, Quagga etc.).

[2] Explained later in the current section.

[3] To be precise, an end-system also needs to be aware of all the shortest path routes within its locale.

[4] Compact Routing: Challenges,Perspectives, and Beyond. TRILOGY Future Internet Summer School, August 24-28, 2009. Louvain-la-Neuve, Belgium. By: Dimitri Papadimitriou (Alcatel-Lucent Bell NV). Email:dimitri.papadimitriou@alcatel-lucent.be. `http://typo3.trilogy-roject.eu/fileadmin/publications/Other/Papadimitriou-CompactRouting.pdf`

[5] The alternative is to come up with an algorithmic mapping between the addresses, which would negate the reasoning for an overlay network.

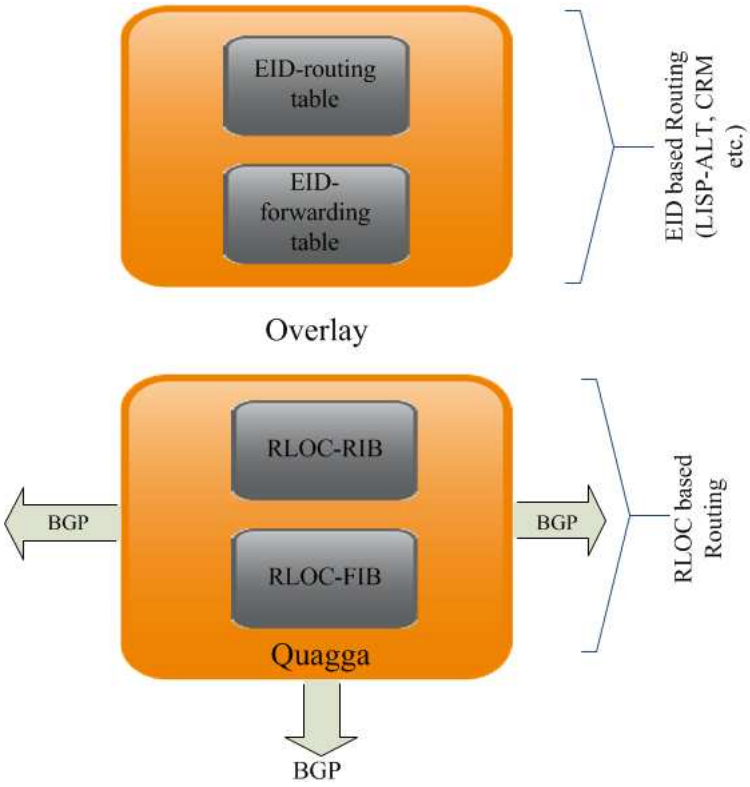

Figure 5.1: Overlay Routing Scheme.

Figure 5.1 gives a conceptual visualization of such an overlay routing system. CRM is designed to provide a remedy for the routing scalability problem. As a consequence, we have utilized Quagga only for RLOC routing; so that its routing facilities are used in the least possible way.

Our implementation of the CRM system is only concerned with the processing of Map-Register messages and its suitable aggregation, delegation and ultimately, the dissemination of this information to all LMs in the LM-space. Map-Request and subsequent Map-Reply are not a part of this work (because both are user plane messages).

Before we go further into analyzing our system and compare it with LISP-ALT, we would like to elaborate on the topology and the input data used for testing. At first, we examine the "test" topology as shown in the following figure.

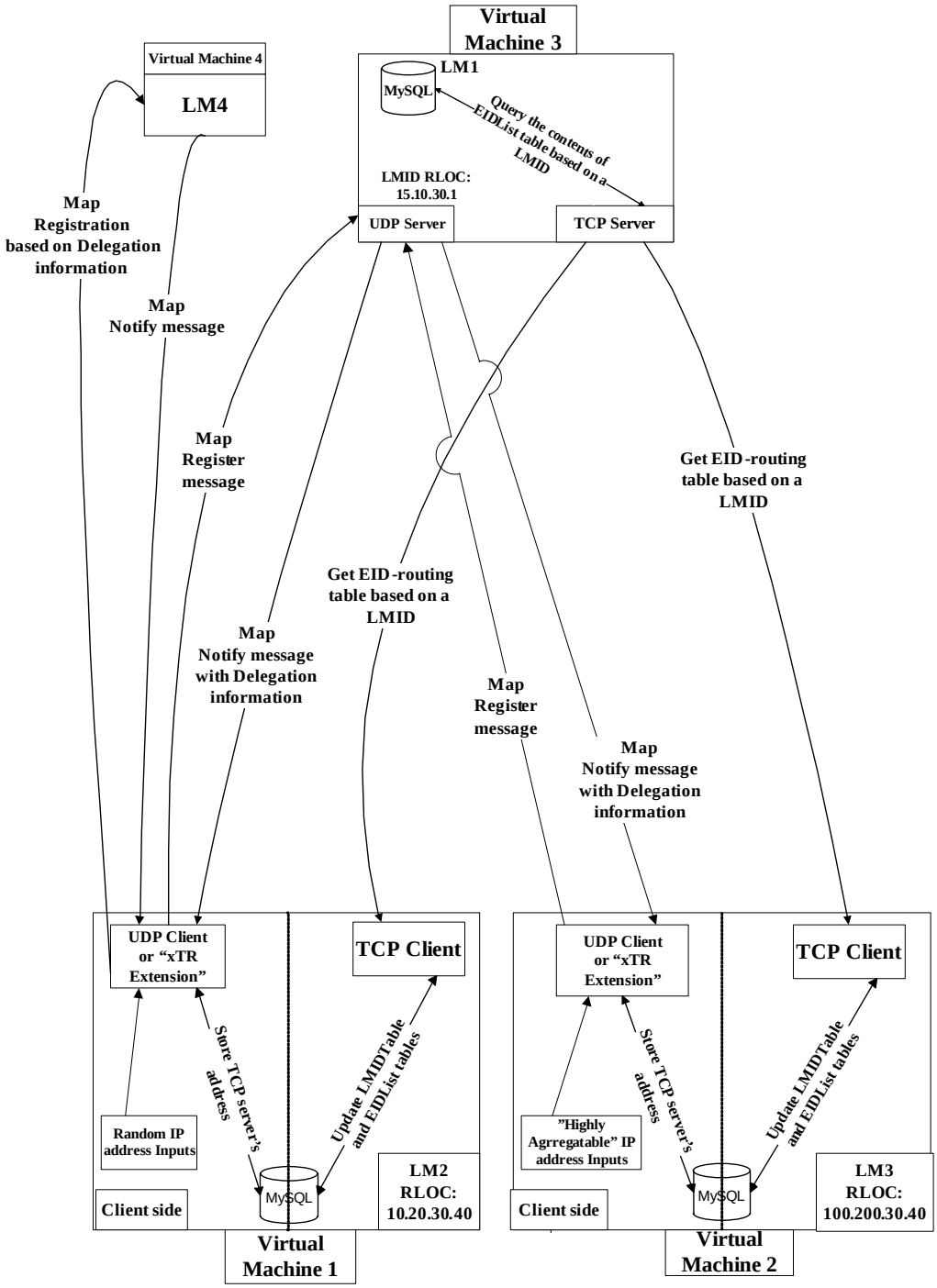

Figure 5.2: Experimental topology for testing the capabilities of CRM.

Before the reader starts to get confused, we need to stress that, the previous figure is an instantiation of the architectural figure 4.2 showed in prior section 4.2. In other words, the figure above shows how the architecture is implemented and deployed. The whole architecture is deployed with the help of three Virtual Machines (VMs). Both VMs 1 and 2 houses UDP client (i.e. "xTR Extension"), TCP client and a MySQL database. The dashed diving lines in the middle of VM 1 and 2 signifies the fact that, these VMs house elements of both client side (i.e. UDP client) and LM (i.e. TCP client). However, the MySQL DB is common to both client and LM (that's why it is in the middle). For our testing purposes, VM 1 and 2 are sending Map-Register messages to the LM1. The third system or VM3 contains LM1. The rationalization behind this type of setup is that, we wanted to test the abilities of the servers (both TCP and UDP) to handle multiple

clients. The details about the messages exchanged between these elements are explained in the previous section 4.2. VM 4 houses LM4 which is accessed by the UDP client (to send Map-Register message) in case of Delegation. The inner workings of LM1 and LM4 are same.

At this point, we need to explain how a LM gets to know what other LMs are advertising. From the beginning, our aim was to develop a mapping system that makes minimal use of BGP advertisements. That is why, Quagga only advertises community attributes, not the actual mappings. We know, during delegation, the "current" LM will try to decide whether there is any other LM that is better suited or not. In order to make that decision, the "current" LM needs to know what other LMs are advertising. Whenever there is a change in the network residing "behind" a LM, its advertised community attribute changes. This prompts "current" LM's TCP client to fetch the contents of EID-routing table belonging to the LM whose network state has changed. This is how, one LM knows what other LMs are advertising (and thus overlay routing). It should also be noted that, the TCP client never fetches *EID-forwarding table*.

Next, we move on to the input data which is used for both LISP-ALT and CRM systems. A part of the input is from [68] and this sample was taken on 30/06/2010. This section of the source originates from randomized distribution of prefix lengths to mimic the real world. The 1st octet of the prefix is fixed by us; while the 2nd, 3rd and 4th octets are completely random. Though the sample is more than 1 year old, this trend of the randomization is still valid in the more recent samples. We have taken 100 such prefixes. As shown in figure 5.2 such inputs are originated from VM1. The other part of the input is "highly aggregatable" and is generated from VM2 (see figure 5.2). We have used 21 such prefixes. Though the size of the input is small, we believe that, it is sufficient for our prototype to provide a proof of concept.

The "EID-routing" table refers to the MySQL table (or any other database table) used for internal "book-keeping" purposes. It contains all the EID-prefixes that the current LM was able to aggregate and delegate. Additionally, this table contains the orphan EID prefixes that the current LM will announce. A snapshot of a CRM's *"EID-routing table"* looks like the following:

LMID	EIDPrefix	EIDPrefixLength	Is_Delegated	Delegated_RLOC
15.10.30.1	110.0.1.0	24	Y	100.200.30.40
15.10.30.1	110.0.128.0	17	Y	100.200.30.40
15.10.30.1	110.0.16.0	20	Y	100.200.30.40
15.10.30.1	110.0.2.0	23	Y	100.200.30.40
15.10.30.1	110.0.32.0	19	Y	100.200.30.40
15.10.30.1	110.0.4.0	22	Y	100.200.30.40
15.10.30.1	110.0.64.0	18	Y	100.200.30.40
15.10.30.1	110.0.8.0	21	Y	100.200.30.40
15.10.30.1	110.1.0.0	16	Y	100.200.30.40
15.10.30.1	110.16.0.0	12	Y	100.200.30.40
15.10.30.1	110.2.0.0	16	Y	100.200.30.40
15.10.30.1	120.14.64.0	18	Y	10.20.30.40
15.10.30.1	63.130.121.0	24	N	X
15.10.30.1	81.130.29.0	24	N	X

Figure 5.3: "EID-routing" table for CRM.

The *"LMID"* column holds the RLOC address of the "current" LM. The *"EIDPrefix"* and *"EIDPrefixLength"* column refers to the aggregated EID-prefix and its length

respectively. *"Is_Delegated"* is a Boolean column (i.e. Y/N) that shows whether there is a "better" aggregation point or not. The *"Delegated_RLOC"* column holds the RLOC address of a LM that can provide a "better" aggregation for a particular EID-prefix. In other words, these LMs (of the *"Delegated_RLOC"* column) are advertising "better" EID-prefixes. The term "better" in our case means that, the LM bearing the *"Delegated_RLOC"* is announcing a parent address. The IP addresses present in the "LMID" and "Delegated_RLOC" columns are also visible in figure 5.2.

It should be noted that, there are no existing open source implementations of LISP-ALT for the academia to explore. Therefore, we made a minimal implementation of LISP-ALT (to be accurate, we have realized the functionality of an ALT router) by observing the methods described in [17]; so that a meaningful comparison can be provided for the audience. At the heart of LISP-ALT is natural aggregation; which is the same as the first stage aggregation done in CRM. So from a development point of view, the initial aggregation stage is common for both CRM and LISP-ALT. After that, CRM takes a completely different path. To put it simply, after the first stage natural aggregation, CRM presses on with delegation and finally if needed, generates virtual prefix. For that reason, the *"EID-routing"* table for LISP-ALT for the same set of inputs (i.e. same as CRM) will provide us with a different output snapshot:

LMID	EIDPrefix	EIDPrefixLength	Is_Delegated	Delegated_RLOC
15.10.30.1	110.0.1.0	24	N	X
15.10.30.1	110.0.128.0	17	N	X
15.10.30.1	110.0.16.0	20	N	X
15.10.30.1	110.0.2.0	23	N	X
15.10.30.1	110.0.32.0	19	N	X
15.10.30.1	110.0.4.0	22	N	X
15.10.30.1	110.0.64.0	18	N	X
15.10.30.1	110.0.8.0	21	N	X
15.10.30.1	110.1.0.0	16	N	X
15.10.30.1	110.16.0.0	12	N	X
15.10.30.1	110.2.0.0	16	N	X
15.10.30.1	120.14.64.0	18	N	X
15.10.30.1	63.130.121.0	24	N	X
15.10.30.1	81.130.29.0	24	N	X

Figure 5.4: "EID-routing" table for LISP-ALT.

It is evident from the tabular output that, LISP-ALT has no delegation capabilities. It can only perform natural aggregation and thus if the input is random (i.e. addresses that are not highly aggregatable) then LISP-ALT is forced to announce all the EID-prefixes through BGP. Therefore, the actual size of the RT becomes heavily dependent on the input.

CRM on the other hand, only advertises the current LM's RLOC and a hash value (through BGP's *community attribute*). Whenever there is a change in the current LM's aggregation and delegation capabilities, a new hash value is advertised through BGP. The presence of a new hash value prompts all the neighbors of the current LM to establish a TCP connection with it (i.e. with the current LM). With these TCP connections, current LM's altered aggregation and delegation capabilities gets disseminated to its neighbors. Hence, compared to LISP-ALT's "large" amount of advertisements (and in the process creating a "large" RT), CRM utilizes BGP in a minimalistic way. CRM's nominal utilization of BGP advertisements can also be viewed from the screenshots of

Wireshark's "packet details" output:

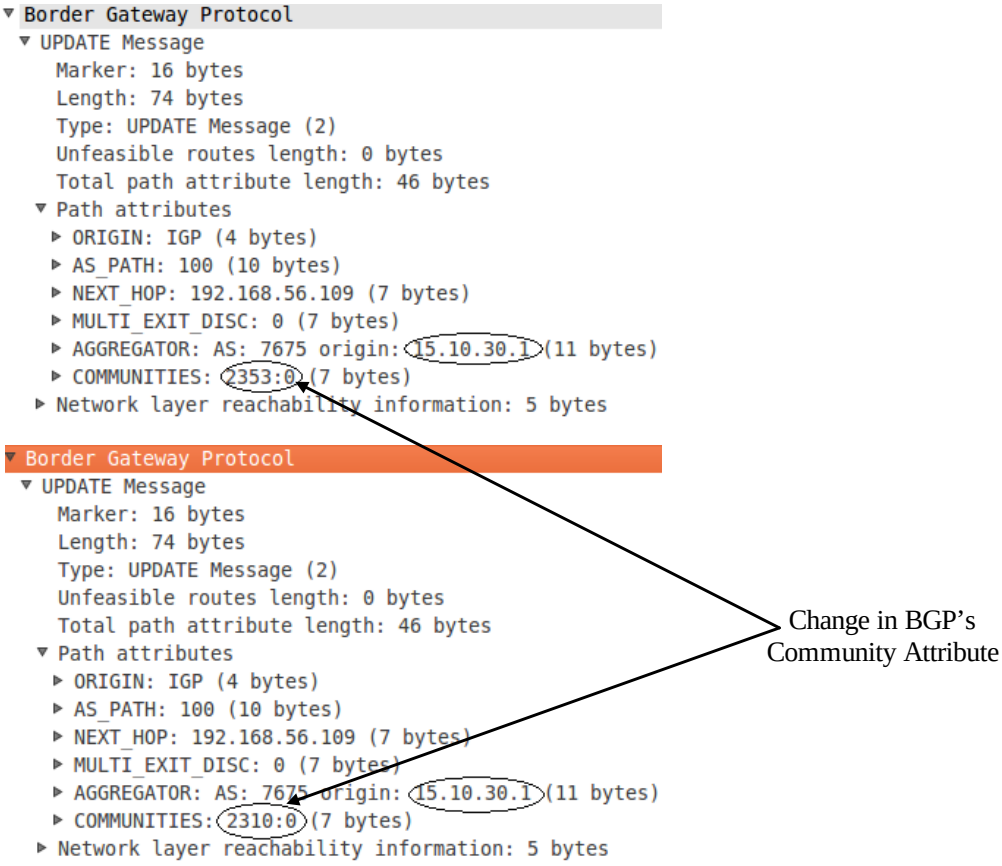

Figure 5.5: Wireshark's packet details output of two BGP UPDATE messages.

The above depicted figure shows two consecutive BGP UPDATE messages (top to bottom) where the hash value has changed. As stated before, with BGP's *AGGREGATOR* attribute, we are advertising the LM's RLOC (in this sample, it is 15.10.30.1); with *COMMUNITIES* attribute we are advertising hash value (in the current trial, the hash value has changed from "2353:0" to "2310:0").

Though we have grounded the critical routing scalability issues with Compact Routing, our CRM allows us to deal with dynamic topologies. This capability is stemming from the fact that, we are reusing LISP's registration messages. An xTR is registering to that LM which is deemed provide a more optimal (or "better") aggregation. Changed hash value works as a trigger for other LMs to check for an updated "EID-routing table".

The most important data structure for a mapping system dealing with distinct namespaces is, the *"EID-forwarding table"* [6]. This table is derived from *"EID-routing table"* and delegation information available in the current LM. It houses all the locally owned registered and announced EID-prefixes. It also contains all the aggregates that are announced by the other LMs (i.e. delegation data when applicable). A sample snapshot of CRM's *"EID-forwarding table"* looks like the following:

[6] In our implementation, this is also a MySQL table (or it can be any other DB table of choice).

LMID	EIDPrefix	EIDPrefixLength
15.10.30.1	63.130.121.0	24
15.10.30.1	81.130.29.0	24
10.20.30.40	120.0.0.0	8
100.200.30.40	110.0.0.0	8

Figure 5.6: EID-forwarding table of CRM.

The significance of the aforementioned tabular data is that, the LM bearing the RLOC 15.10.30.1 is announcing the EID-prefixes 63.130.121.0/24 and 81.130.29.0/24. This table also signifies that, the LM bearing the RLOC 15.10.30.1 has delegated the EID-prefixes 120.0.0.0/8 and 110.0.0.0/8 to LMs 10.20.30.40 and 100.200.30.40 respectively. When a Map-Request message from the user plane arrives at a LM, the "*EID-forwarding table*" is consulted. If the EID-prefix for which the Map-Request is generated (to be precise, a Map-Request is created due to an absent mapping) is either found or is a child of any of the EID-prefix entries present in the table, then we have a match. When there is a match, the current LM can either handle the message by itself (i.e. if the matched EID-prefix is announced by the current LM) or it can forward it (i.e. the Map-Request) towards the LM where the matched EID-prefix has been delegated.

As stated earlier, an ALT router does not perform any delegation and for this, its "EID-routing table" and "*EID-forwarding table*" will house the same set of LMIDs and EID-prefixes. Without delegation, LISP-ALT's "*EID-forwarding table*" will be considerably larger than that of a CRM's. This will in turn, cause greater delay when the "*EID-forwarding table*" is searched for an exact match or child EID-prefix.

5.2 Cost Profile, possible Optimizations and Comparisons

In this section, we hope to objectively evaluate our experimental mapping system with the help of a profiler, suggest possible optimizations for future implementations and most importantly compare the costs of LISP-ALT and CRM. Traditionally, the cost of an algorithm [7] is measured by the time spent (i.e. time cost) and space used (i.e. space cost) in the key operations for a specific size of input. However, both LISP-ALT and CRM are not centered around any specific piece of algorithm; making it difficult to pinpoint the "key operations". Also, the "time spent" by a program is subjective to the hardware (i.e. CPU, memory etc.) it is running on. Additionally in most cases, the space used by an algorithm is simply a multiple of the data size; which is very much true for both LISP-ALT and CRM and thus, is not a good indication of the "cost". Therefore, we have taken an alternative route for deducing the costs of LISP-ALT and CRM; we have defined and located the functions that perform "key operations" by using a profiler. The cost of those "key operations" are then compared.

To locate the "key operations", we have utilized the services of a profiling tool called Kcachegrind [8]. Kcachegrind's call graph view not only shows the frequentness of functions; but also their respective "inclusive costs" in percentages. The "inclusive cost" of a function is measured by the total number of CPU operations that occur between entering and

[7] The terms "cost" and "complexity" are loosely interchangeable in this context.

[8] `http://kcachegrind.sourceforge.net/html/Home.html`

exiting that function. Though the absolute number of CPU cycles might vary according to the underlying hardware, the percentages remain the same; making "inclusive cost" impervious to hardware change. The "inclusive cost" of all functions that are called (from the current function) are summed. That is why, for a generic C program, the function main() has a cost of *~100%*. However, "inclusive cost" is not the only criteria for defining whether a function performs any "key operation" or not. If a function executes some important logical procedures then it is also judged to have "key operations". Once we have located the functions performing the "key operations", we will use that information to suggest possible improvements and most significantly utilize the "inclusive costs" to provide us with a meaningful comparison between LISP-ALT and CRM.

We will start off this section with UDP Client side, because the whole program is launched from this point (for both LISP-ALT and CRM). Our intension is to analyze the different parts; so that in future, if optimization is required we will know which functions to look at.

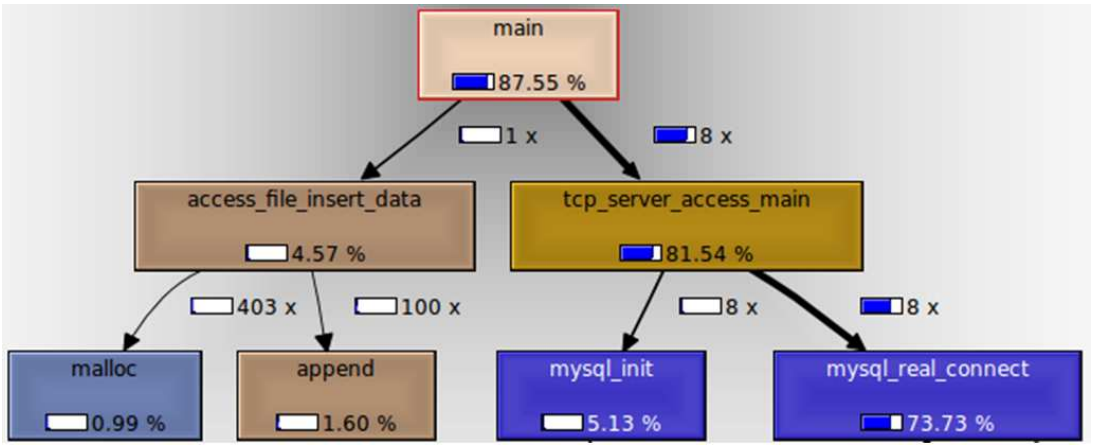

Figure 5.7: Call graph view of UDP Client (formatted and summarized).

From the call graph view, it should be clear that, more than 2/3 of the "cost" for the UDP client is caused by the function "*tcp_server_access_main()*". This function's name might be a little misleading for the readers. The "*tcp_server_access_main()*" function does not access the TCP server. It only inserts/updates the remote TCP server's address obtained from Map-Notify message in the local DB. It should be noted that, the task of "*tcp_server_access_main()*" is only limited to CRM (i.e. the right sub-tree is exclusive to CRM). Most of its (i.e. "*tcp_server_access_main()*") cost is endured while calling a library (i.e. mysql.h) function, namely, "*mysql_real_connect()*". This function must complete successfully before one can execute any other API functions that require a valid MySQL connection handle structure.

Now, we have to decide which are the functions that houses the "key operations". For CRM's UDP client, both "*access_file_insert_data()*" and "*tcp_server_access_main()*" house "key operations". It is because "*access_file_insert_data()*", in spite of its low "inclusive cost" performs the essential task of collecting information from the input file and allocates memory for it. "*tcp_server_access_main()*" on the other hand, has a high "inclusive cost" and it provides the TCP client with the TCP server's address. Upon inspection, it should also be apparent that, the choice of DBMS (i.e. MySQL) heavily effects the cost of the UDP client; because *mysql_real_connect()* is responsible for *~75%* of the total cost. Therefore, in future, it might be worthwhile to look at other Open Source DBMS options (e.g. Firebird, PostgreSQL, Berkeley DB etc.). C (GCC) APIs for these DBMSs might have been cheaper. Using SQL's transaction capabilities through

MySQL for controlling the access to a common resource by multiple processes is also driving up the cost (through waiting time). Using semaphore for the same purpose might be another alternative that we should explore. However, for LISP-ALT's UDP client, "*access_file_insert_data()*" is the only function with "key operations". As this function never accesses the DB, its "inclusive cost" is quite low.

Though there are clear distinctions between the functionalities of CRM's and LISP-ALT's UDP client, we have refrained ourselves from using this part of the program for comparison. It is because, in the whole of things, UDP client's functionality bears miniscule cost compared to the UDP server side. Therefore, only the UDP server side will be used for the cost comparison between LISP-ALT and CRM.

The curious readers should note that, call graph views shown in this subsection are all formatted and summarized (through grouping and filtering) for our purposes. A detailed view of the call graphs are available in appendix- A. Originally, the tree in figure 5.7 does not begin with main() function [9]. Because the compiler/OS does not load the program image and jumps to *main()* directly. To be precise, *main()* is the starting point of a C program from a programmer's perspective. Before calling *main()*, a process has already executed a bulk of code for initialization and has "cleaned up the code for execution". The functions that hold these surrounding codes are provided by the compiler/OS and can be seen in a detailed call graph view [10]. Some functions that contribute less than *5%* of the total cost are also visible here. Additionally, this detailed call graph view shows functions that are not displayed by name, instead through hexadecimal numbers. These correspond to places where debugging information or stack information are not available. If the detailed call graph view (Refer to appendix- A) is examined carefully, it should be clear that, these "hexadecimal numbered functions" reside typically inside or between libraries. They are absolute virtual addresses. One cannot find these addresses in the program (e.g. using the "nm" command) because either they are in dynamically loaded libraries with no debugging information, or they were relocated, or maybe they were in parts of the program that were not compiled using debug symbols (e.g. code from static libraries). A full description of these aforementioned topics is out of the scope of this work and therefore we will cease to discuss them any further. From now on, only formatted and summarized call graph views will be described in this section. The inquisitive readers are advised to look into appendix- A for detailed call graph views.

LISP-ALT does not have any TCP client-server part. It is because, an ALT router learns about the "network state" of other ALT routers through BGP advertisements; it does not need a TCP connection to update its own "network state" information. Hence, the TCP client/server sections are exclusive to CRM. As previously stated in sections 4.2.3 and 4.2.4, this portion of the program is responsible for extracting the RLOC and community attribute, for putting the community attribute into MySQL table and if the community attribute changes then the TCP client connects with the TCP server to get an updated view of the DB.

[9] Not shown in this thesis.

[10] `http://www.lisha.ufsc.br/teaching/os/exercise/hello.html`

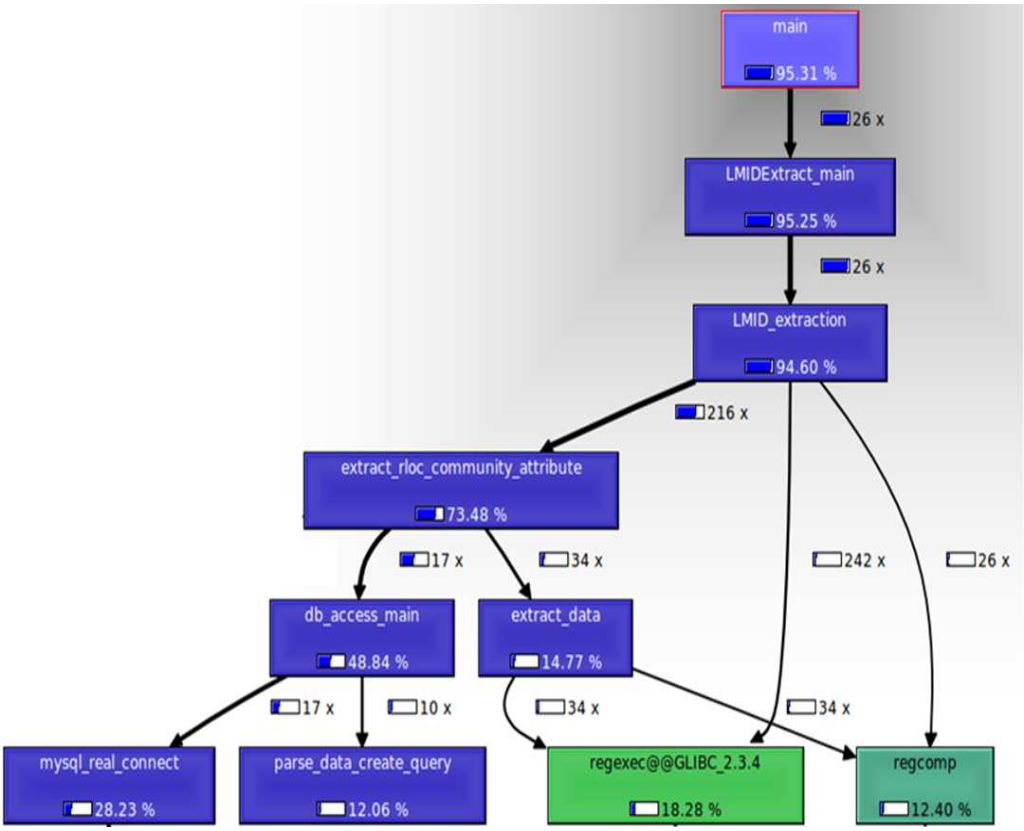

Figure 5.8: Call graph view of TCP Client (formatted and summarized).

For the second time, we see that (5.8), functions that are involved in accessing the MySQL tables account for *~50%* of the total cost (i.e. because of *db_ access_ main()* function). Interestingly, *~20%* of *LMID_ extraction()*'s cost (i.e. *94.60% - 73.48% = 21.12%*) involve executing regular expressions. The extract_data() function (costs *~15%*) is entirely engaged in the same task. This has happened because, there is no direct way to extract the RLOC and community attribute from the BGP advertisements. We had to use regular expressions in two stages, at first, to extract IP address and then use those IP addresses to determine (again using regular expressions) whether BGP's *set aggregator* and *community* attributes are present or not. Because of the aforementioned reasons, *LMID_ extraction()*, *db_ access_ main()* and *extract_ data()* are the functions deemed to have "key operations" for TCP client. As before, it might be a good idea to try out some other Open Source DBMS. But we have to remember that, using an alternate Open Source DBMS might reduce the CPU cycles; but the percentage figures might not change. Because the TCP client is heavily dependent on the information exchange with the underlying Database.

Regretfully however, there is no way around the problem of using regular expressions. It is because in reality there is no alternative for Quagga when it comes to developing small prototypes [11]. Because it is the most widely deployed, documented and researched Open Source Routing software and with Quagga, we have no other way of extracting the desired information.

Now, we look at the TCP server part which is also unique to CRM. Also mentioned earlier in section 4.2.4, its primary task is to provide the TCP client with an updated view of the DB.

[11] XORP could be an option. However, XORP uses quite heavy C++, making it infeasible for systems with limited CPU and RAM. BIRD on the other hand, shows its usefulness when number of peers » 200. So, for our purposes, Quagga is the best choice.

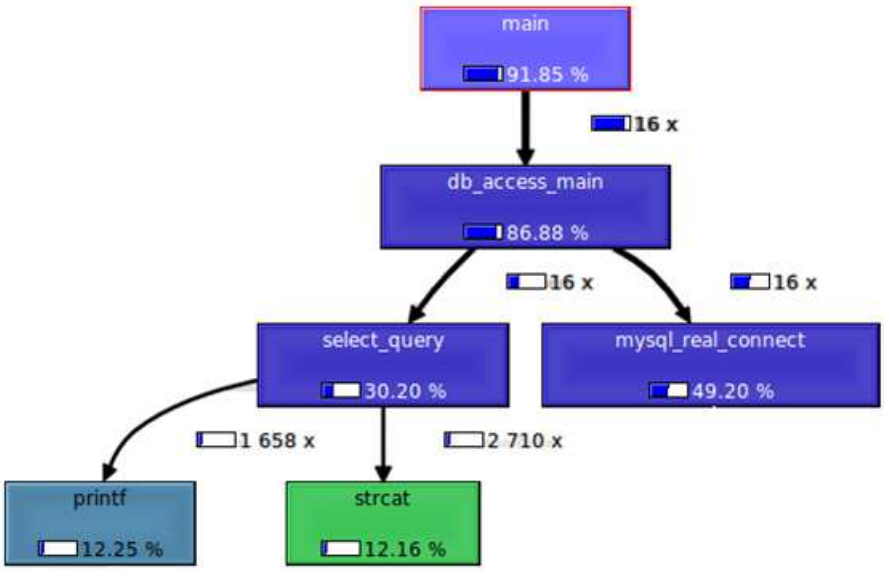

Figure 5.9: Call graph view of TCP Server (formatted and summarized).

As expected, *mysql_ real_ connect()* accounts for *~50%* of the total cost. The *select_ query()* is the function that extracts the current view of the EIDList table based on the provided LMID and it also costs *~30%*. Evidently, these two functions carry out "key operations". As before, a different Open Source DBMS might provide cost optimization. Like TCP client, the TCP server is heavily dependent on the information exchange with the used Database; which in turn means that, an alternate Open Source DBMS might reduce CPU cycles, while keeping the percentage values intact. One important thing to notice here, are the *printf()* and *strcat()* library functions. Though they are called thousands of times, they cannot be deemed as performing "key operation". Because their costs are low and they do not perform any logically significant operations. Therefore, the observation being, frequentness cannot be treated as characteristic in determining whether a function houses "key operations" or not.

Lastly, we inspect the UDP server. From the detailed descriptions found in section 4.2.2, we know that, the UDP server is the part of the program where we can observer the clear difference between LISP-ALT and CRM. At first, we examine LISP-ALT.

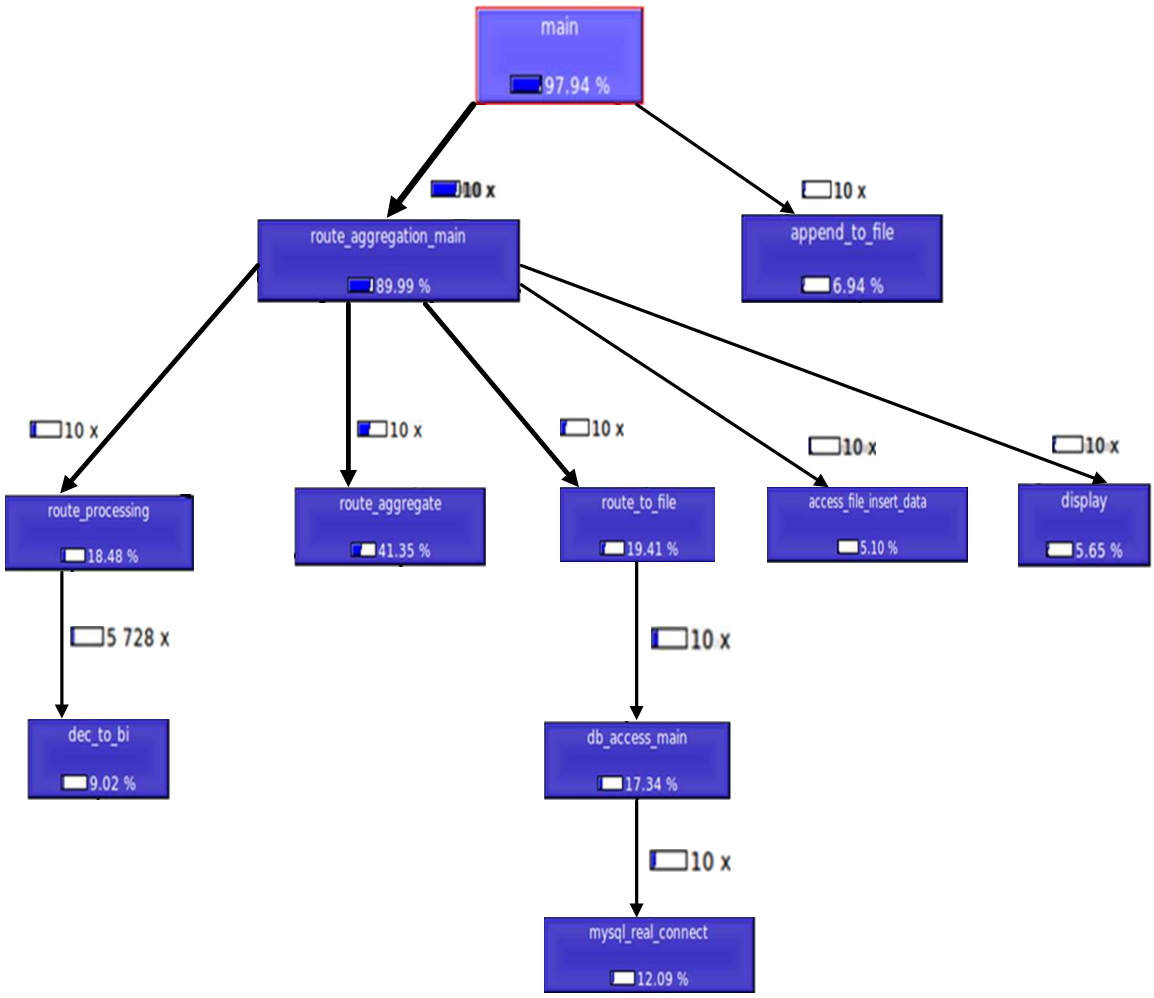

Figure 5.10: Call graph view of LISP-ALT (formatted and summarized).

As expected, the *route_aggregate()* function, responsible for the natural aggregation, consumes ~42% of the total cost. Clearly, this function performs "key operations". The *route_processing()* function, in spite of its low cost (*~19%*), fills data structure holding the routes with necessary information required for natural aggregation. Hence, it is judged to have "key operations". The *route_to_file()* is also a function with "key operations"; because it seeks out the aggregated addresses and then puts it into the DB. Notably, the UDP server is not entirely dependent on the interactions with the DB (actually, this part of the program is computation centric) and thus we see a relatively low cost (*~18%*) of the *db_access_main()* function. Additionally we observe that, though the *dec_to_bi()* function is called more than 5,700 times, it is not a function with "key operations". In addition to its low cost, the *dec_to_bi()* function performs the trivial task of converting decimal numbers into binary. Therefore, just based on its frequency we cannot judge the *dec_to_bi()* function to have "key operations".

In future optimizations, we should use a Radix tree [12] as the data structure that holds the addresses while aggregation is taking place (for both LISP-ALT and CRM), instead of a linked list [13]. A Radix, or Patricia tree is a compressed tree that stores strings. Unlike usual trees, radix tree edges may be labeled with multiple characters that provides an efficient data structure for storing strings that share common prefixes. These can be strings of characters, bit strings such as integers or IP addresses, or generally arbitrary sequences of objects in lexicographical order. Radix trees support lookup, insert, delete

[12] Radix Tree is successfully used in OpenLISP. Refer to 2.5.
[13] Usage of linked list is explained in section 4.2.2.

and find predecessor operations in *O(b)* time where b is the maximum length of all strings in the set. According to *Sklower et al.* [58], by using a binary alphabet, strings of $b = 32$ *bits* and next hops as values, radix trees can support IP routing table longest match lookup [2]. At the moment, we are employing brute force methodology in a linear data structure like linked list to determine the ancestry of nodes. This means that, for n addresses we have to perform $O(n^2)$ comparisons. For our purposes, this is sufficient. Because our primary aim was to provide a proof of concept and we have succeeded in doing so. But if this prototype were to progress into production phase then we must implement an optimized data structure, such as, Radix tree.

Now, we move to the final part of this subsection where we describe CRM and compare it with LISP-ALT. From prior discussion, we know that, in addition to the natural aggregation, CRM performs delegation and finally if needed generates Virtual Prefix. Our scheme for Virtual Prefix generation follows the algorithm of [14] which dictates that, if the size of the system does not grow quickly then Virtual Prefix is not created. As our system is quite small the Virtual Prefix generation part never gets executed and is thus absent from the call graph view.

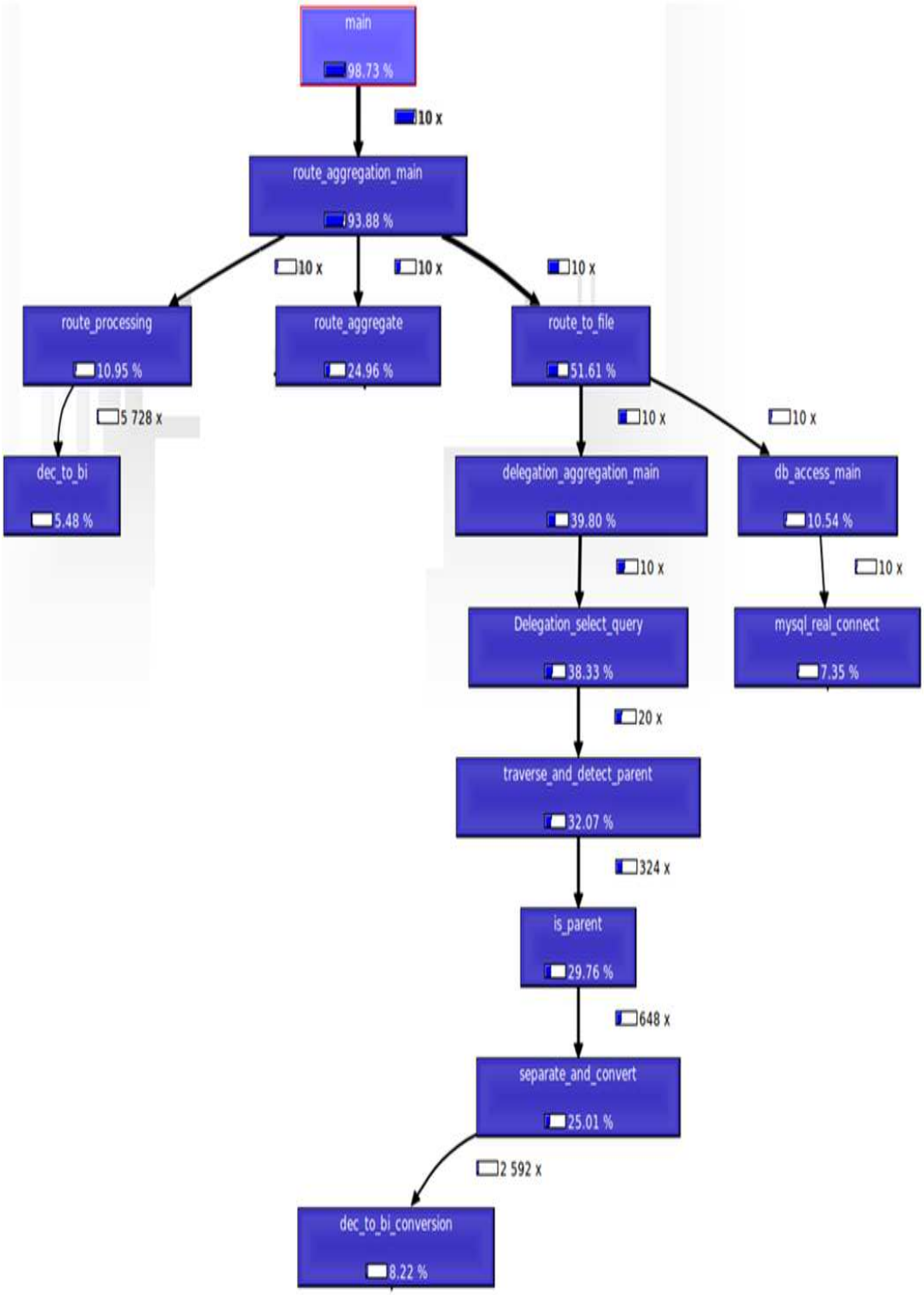

Figure 5.11: Call graph view of CRM (formatted and summarized).

Unsurprisingly, the *route_to_file()* function (or node) in addition to its previously mentioned tasks (i.e. right branch), now has a new left branch that does the delegation. From section 4.2.2, we know that, in delegation, we perform a second round of aggregation if there exists a LM which advertises "better" EID prefixes. Definitely, in this new branch, the function *is_parent()* holds the "key operations". It has a cost of ~30% and as the name suggests, detects the ancestry between IP addresses based on delegation information.

At the first glance (of figure 5.10 and 5.11), the reader might think that, LISP-ALT is

better than CRM; as it has lesser number of nodes in its tree (and thus less cost). But such a judgment would be premature. Because the call graph views of LISP-ALT and CRM only shows the cost of one router. As CRM is based on Compact Routing, it is guaranteed that, a control plane message will only have to climb a two level hierarchy before reaching the destination xTR. This implies that, the cost of CRM is strictly bounded. Stated repeatedly, LISP-ALT comes with no such assurances. A control plane message might have to travel indefinite number of ALT-routers before reaching the destination xTR. This means that, the cost for LISP-ALT might grow in an unpredictable rate. Furthermore, the larger size of the EID forwarding table (because for an ALT router the size of its "*EID-routing table*" and "*EID-forwarding table*" are the same) and its handling (e.g. searching) will cause additional complexities.

As mentioned previously, the first natural aggregation is common to both LISP-ALT and CRM. That is why, if we examine figures 5.10 and 5.11, we will find couple of common functions. These are: *route_aggregate()*, *route_processing()*, *route_to_file()*, *dec_to_bi()*, *db_access_main()* and *mysql_real_connect()*. These functions are essential for the 1st level aggregation. The comparison has to be made from the call graph of CRM (i.e. figure 5.11); because it houses all the functions (of both LISP-ALT and CRM). After close inspection of fig. 5.11, we calculate that, the functions required for 1st level aggregation costs *~50%*. With this, we can infer that, for one router, LISP-ALT costs about half of CRM. For the sake simplicity, we have changed the unit from percentage to cycles; i.e. we have assumed a cost of *~100%* is equal to 100 cycles. Therefore, when a CRM system of one router costs 100 cycles, a LISP-ALT system of one ALT router will cost 50 cycles. The following bar graph of figure 5.12 shows how the cost increases when the number of routers increase.

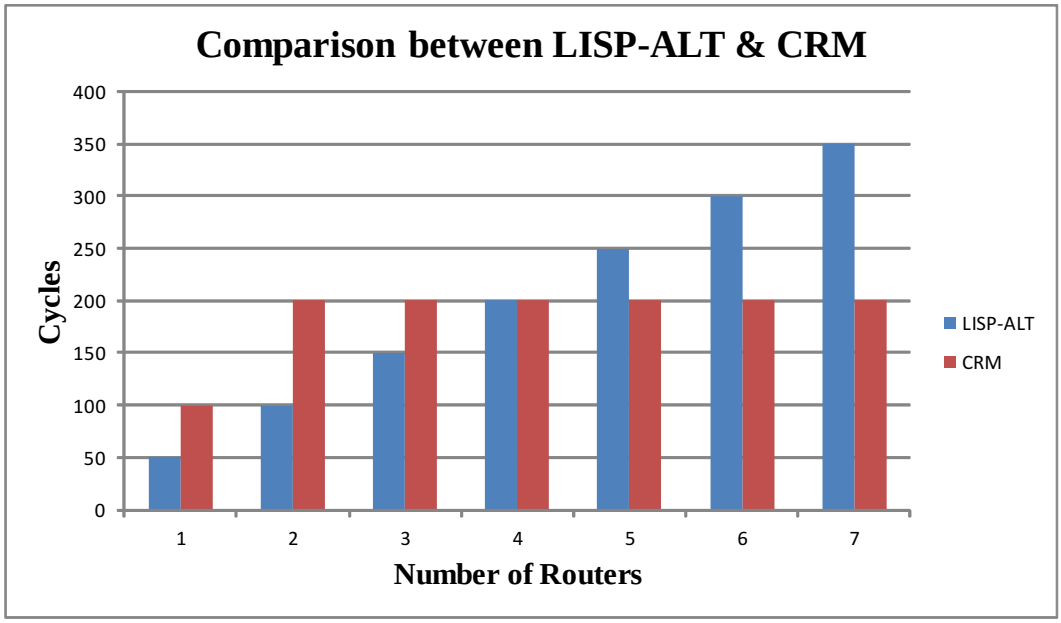

Figure 5.12: Performance comparison between LISP-ALT and CRM.

With two routers, CRM costs twice of LISP-ALT. However, the cost of LISP-ALT grows rapidly and becomes equal to that of CRM when there are 4 routers. After this threshold, LISP-ALT's cost overtakes that of CRM. This phenomenon is due to the fact that, CRM only has a two level (i.e. two routers) hierarchy, making its cost firmly bounded. LISP-ALT on the other hand, has no such boundaries which in turn results in rapid cost rise. The conclusion that we can draw is that, even for a "Mickey mouse" system, CRM will cost far less than LISP-ALT.

5.3 Implementation Analysis

> *"Even the best planning is not so omniscient as to get it right the first time."* – Fred Brooks (Turing award winning software engineer and computer scientist, author of "The Mythical Man-Month")

From the onset of this implementation, we had envisioned a system where we would utilize Quagga's MPLS capabilities to disseminate the information regarding the dynamic changes of the network. However, our presumption proved to be wrong when we discovered that, Quagga only supports statically assigned MPLS labels, making it unsuitable for propagating information concerning the dynamicity of the network. This setback forced us to come up with a novel solution; where we used TCP connections to update the database tables that stored the current status of the network. Our approach is similar to that of NERD [14] where a monolithic database is present on each xTR and is refreshed at regular intervals, containing all available mappings. In CRM however, the database is present in the LMs; not in the xTRs and we update this database only if there is a change in the network topology. Though the TCP connection strategy is adequate for our prototype, we are concerned with the cost. We predict, in a live network, our CRM will establish numerous TCP connections and thus may overwhelm the network. If Quagga had supported dynamic MPLS labeling then we would not have any need to update the whole database. Using BGP's UPDATE message we could have disseminated the information (concerning the network's "state") incrementally. To the best of our knowledge, patches are being developed as we speak so that Quagga will start supporting dynamic MPLS labeling in the near future.

After the code profiling, it became obvious that, choosing MySQL for storing the network's "state" information was a costly choice (not to be confused with a bad/inappropriate choice). The APIs (e.g. *mysql_ init()*, *mysql_ real_ connect()* etc.) that handled the communication between CRM and database has proven to be overwhelmingly expensive. From hindsight, we should have refrained from using any database (e.g. MySQL, Firebird, PostgreSQL, Berkeley DB etc.) and utilize something simpler e.g. CSV (Comma-Separated Values) files for storing the data. The data used to store the network's current state is not complicated enough to require the versatile functionalities of MySQL (or any other accomplished DBMS). Most of its functionalities were wasted. On the other hand, MySQL gave us the capability to Normalize data; we were able to maintain uniqueness and consistency by using primary and foreign keys respectively. With MySQL, we have the programmable flexibility to add new entries, split/join tables as required and most importantly, insure the atomicity of the operations on tables. Though CSV files might have been cheaper; it would have increased the development time by many folds. Because C (GCC) does not come with any APIs that can be used to insert/delete/update entries in a CSV file. We would have to implement all that functionality ourselves; otherwise stated, "reinvent the wheel". Furthermore, if we had chosen CSV files then we would have been forced to use semaphores for controlling access to a common resource by multiple processes. Semaphore operations are inherently slow: Solaris on a 1993 Sparc takes 3 microseconds for a lock-unlock pair and Windows NT on a 1995 Pentium takes 20 microseconds for a lock-unlock pair [15]. Even though these figures look outdated, they are very much relevant in the current context. From a developer's point of view, a semaphore has its own set of shortcomings. For example, semaphores are too low level, making them prone to mistakes by programmers (e.g. P(s) followed by P(s)). A programmer must

[14] Refer to section 2.3.3.4.

[15] Mutexes, rwlocks, etc. The Good, the Bad and the Ugly. By: Tim Cook. Performance and Applications Engineering,Sun Microsystems. `http://forge.mysql.com/w/images/1/18/TimC.pdf`

also keep track of all calls to wait and to signal the semaphore. If this is not done in the correct order, the error might cause a deadlock. Furthermore, semaphores are usually used for both condition synchronization and mutual exclusion. But these are distinct and different events, making it difficult to know which meaning any given semaphore may have. Also, implementing semaphores can dramatically increase the development time [16]. Nevertheless, in future, semaphores might be implemented and tested through CRM so that questions surrounding its (i.e. semaphore's) cost in the current context can be answered definitely.

At the moment, we are using SQL's transaction capabilities (accessed by MySQL) through C(GCC)'s APIs as an alternative for semaphores. Hence, our decision to use MySQL aided us in developing a prototype swiftly and efficiently.

[16] Disadvantages of Semaphores. http://www.cs.colostate.edu/~cs551/CourseNotes/ConcurrentConstructs/DisAdvSems.html

Chapter 6

Conclusion and Future Work:

This thesis presented the prototype of a novel mapping system: CRM, and compared its cost with LISP-ALT. CRM is based on LISP & Compact Routing and was developed to assess the feasibility of the concept that originated in Nokia Siemens Networks (NSN).

The primary objective of this thesis work was to provide proof of concept, which we did by implementing our novel mapping system, CRM. We looked closely at the existing routing scalability issues and in order to mitigate them we designed/implemented such a mapping system where the critical functions that affect the scalability of the routing system are grounded to the theory of Compact Routing. While constructing, we were also able to use existing routing facilities in the way of Quagga. Our preliminary plan was to use multiprotocol BGP to advertise the EID-prefixes hosted by each LM . However, we learnt that, Quagga only supports statically assigned labels that did not meet our needs for the dynamic binding of EIDs and LMs and their RLOCs. Such a major hindrance lead us towards a new and innovative approach, where we used BGP's community attribute to disseminate the changes in "network state". This tactic allowed us to fulfill our aim of using BGP in a minimalistic way with the benefit of ultimately allowing us to retire BGP and replace it with a suitable simpler topology discovery protocol.

While developing CRM, we made a conscious effort to design and construct a system that can be "loosely" integrated. In other words, our mapping system did not require any modifications to the network stack and it is completely modular from the underlying system (e.g. OS kernel). This in turn means that, CRM has minimal dependencies; it is an independent mapping system that can be made compatible with any system (i.e. routing software, OS, topology discovery protocol etc.) by making minimal changes to its interface.

The results and subsequent analysis shows that, CRM is feasible for deployment in the current Internet. Furthermore, our comparison has proven CRM to be far cheaper than LISP-ALT in terms of complexity.

At the moment, CRM is running on a handful of VMs. Therefore, in the future, it is pertinent that we test CRM in a more realistic network topology that simulates the genuine behavior of the Internet. Furthermore, we are now exchanging the mappings between LMs by establishing TCP connections. We predicted that, this can weigh down the network. At the time when CRM was being developed, Quagga did not have support for multiprotocol BGP that is used for MPLS; which would have facilitated incremental distribution of the routing table changes. CRM would have been benefited from such an approach. Ultimately, CRM would be integrated with OpenLISP; so that OpenLISP could run in the data plane whereas CRM would operate in the control plane.

Bibliography

[1] Marc Ferrer Almirall. Wireless lisp: Prototyping and evaluation a lisp implementation in the bowl testbed at deutsche telekom labs. Master's thesis, Universitat Politècnica de Catalunya., May 2011.

[2] Robert Beverly and Karen Sollins. An internet protocol address clustering algorithm. In *Proceedings of the Third conference on Tackling computer systems problems with machine learning techniques, 2008, CA, USA.* USENIX Association Berkeley., 2008. `http://www.usenix.org/event/sysml08/tech/full_papers/beverly/beverly.pdf`.

[3] A. Brady and L. J. Cowen. Compact routing on power law graphs with additive stretch. In *Proceedings of the 8th workshop on Algorithm engineering and experiments*, pages 119–128, Jan 2006.

[4] Arthur R. Brady. A compact routing scheme for power-law networks using empirical discoveries in power-law graph topology. Master's thesis, Department of Computer Science, Tufts University., 2005.

[5] S. Brim, N. Chiappa, D. Farinacci, V. Fuller, D. Lewis, and D. Meyer. Lisp-cons: A content distribution overlay network service for lisp. `http://tools.ietf.org/html/draft-meyer-lisp-cons-04`, October 2008.

[6] Tian Bu, Lixin Gao, and Don Towsley. On characterizing bgp routing table growth. *Computer Networks: The International Journal of Computer and Telecommunications Networking - Special issue on The global Internet archive*, 45, May 2004.

[7] E. Chen and J. Stewart. A framework for inter-domain route aggregation. `http://www.ietf.org/rfc/rfc2519.txt`, February 1999.

[8] Luca Cittadini, Wolfgang Muhlbauer, Steve Uhlig, Randy Bush, Pierre Francois, and Olaf Maennel. Evolution of internet address space deaggregation - myths and reality. *IEEE Journal on Selected Areas in Communications Special issue title on scaling the internet routing system: an interim report*, 28:1238–1249, October 2010.

[9] L. J. Cowen. Compact routing with minimum stretch. *Journal of Algorithms*, 38:170 – 183, 2001.

[10] Attilla de Groot. Implementing openlisp with lisp+alt. Master's thesis, University of Amsterdam., April 2009.

[11] Michael Dom. Compact routing. *Algorithms for Sensor and Ad Hoc Networks*, 4621:182 – 202, 2007. `http://www.mdom.de/fsujena/publications/Dom07_s.pdf`.

[12] Paul DuBois. *MySQL Cookbook*. O'Reilly Media, 2006.

[13] D. Farinacci, V. Fuller, D. Meyer, and D. Lewis. Locator/id separation protocol (lisp). `http://tools.ietf.org/pdf/draft-ietf-lisp-15.pdf`, July 2011.

[14] Hannu Flinck, Petteri Poyhonen, and Johanna Heinonen. Analysis of landmark selection method. 2011.

[15] Behrouz A. Forouzan. *Cryptography and Network Security.* McGraw-Hill, 2007.

[16] Behrouz A. Forouzan. *TCP/IP Protocol Suite.* Mcgraw-Hill, 2009.

[17] V. Fuller, D. Farinacci, D. Meyer, and D. Lewis. Lisp alternative topology (lisp+alt). http://tools.ietf.org/html/draft-ietf-lisp-alt-07, June 2011.

[18] V. Fuller and T. Li. Classless inter-domain routing (cidr): The internet address assignment and aggregation plan. http://www.ietf.org/rfc/rfc2460.txt, aug 2006.

[19] C. Gavoille and M. Gengler. Space-efficiency for routing schemes of stretch factor three. *Journal of Parallel distributed computing*, 61:679 – 687, 2001.

[20] Cyril Gavoille. An overview on compact routing. In *Proceedings of the 2nd Research Workshop on Flexible Network Design.* University of Bologna Residential Center - Bertinoro, Italy, Oct 2011. `http://dept-info.labri.fr/~gavoille/article/iGav06a.pdf`.

[21] Sam Halabi and Danny McPherson. *Internet Routing Architectures, Second Edition.* Cisco Press, 2000.

[22] G. Hardin. The tragedy of the commons. *Science*, 162:1243 – 1248, Dec 1968.

[23] Jan L. Harrington. *Relational database design clearly explained.* AP Professional., 1998.

[24] Visa Holopainen. Interior gateway protocol (igp) metric based traffic engineering. Master's thesis, Helsinki University of Technology, 2006.

[25] G. Huston. 2005 - A BGP Year in Review. In *Proceedings of APNIC 21*, March 2006.

[26] G. Huston. As6447 - bgp table analysis report. `http://bgp.potaroo.net/as6447/`, October 2011. Updated on 5.8.2011, Cited on 5.8.2011.

[27] L. Iannone, D. Saucez, and O. Bonaventure. Openlisp: An open source implementation of the locator/id separation protocol. In *Proceedings of The ACM SIGCOMM 2009 Demo Session, Barcelona, Spain.*, August 2009. This work has been partially supported by Cisco and the European Commission within the the INFSO-ICT-216372 TRILOGY and ECODE projects.

[28] L. Iannone, D. Saucez, and O. Bonaventure. Openlisp implementation report. `http://tools.ietf.org/html/draft-iannone-openlisp-implementation-01`, January 2009.

[29] Luigi Iannone, Damien Saucez, and Olivier Bonaventure. Implementing the locator/id separation protocol: Design and experience. *Computer Networks*, 55:948–958, March 2011.

[30] Kunihiro Ishiguro. Quagga a routing software package for tcp/ip networks. `http://pf.itd.nrl.navy.mil/ospf-manet/quagga.pdf`, Jan 2012.

[31] Lorand Jakab, Albert Cabellos-Aparicio, Florin Coras, Damien Saucez, and Olivier Bonaventure. Lisp-tree: A dns hierarchy to support the lisp mapping system. *IEEE Journal on Selected Areas in Communications*, 28:1332 – 1343, October 2010.

[32] Al Kelley and Ira Pohl. *A Book on C.* Addison-Wesley Inc., 1998.

[33] Varun Khare, Dan Jen, Xin Zhao, Yaoqing Liu, Dan Massey, Lan Wang, Beichuan Zhang, and Lixia Zhang. Evolution towards global routing scalability. *IEEE Journal on Selected Areas in Communications*, 28:1363 – 1375, October 2010. Institute of Computer Science,University of Würzburg, Germany.

[34] Charles M. Kozierok. *TCP/IP Guide: A Comprehensive, Illustrated Internet Protocols Reference.* No Starch Press, 2005.

[35] Dmitri Krioukov, Kc Claffy, Kevin Fall, and Arthur Brady. On compact routing for the internet. *Newsletter ACM SIGCOMM Computer Communication Review*, 37, July 2007.

[36] T. Li. Recommendation for a routing architecture. `http://tools.ietf.org/html/draft-irtf-rrg-recommendation-16`, June 2011. "Compact routing in locator identifier mapping system" is proposed by Hannu Flinck, Nokia Siemens Networks, Espoo, Finland.

[37] Andra Lutu and Marcelo Bagnulo. The tragedy of the internet routing commons. In *Proceedings of 2011 IEEE International Conference on Communications (ICC)*, pages 1 – 5, 2011.

[38] David A. Maltz, Geoffrey Xie, Jibin Zhan, Hui Zhang, and Gísli Hjálmtýsson. Routing design in operational networks: A look from the inside. In *Proceedings of the 2004 conference on Applications, technologies, architectures, and protocols for computer communications*, 2004.

[39] Daniel Massey, Lan Wang, Beichuan Zhang, and Lixia Zhang. A scalable routing system design for future internet. `http://www.cs.ucla.edu/~lixia/papers/07SIG_IP6WS.pdf`.

[40] Laurent Mathy and Luigi Iannone. Lisp-dht: Towards a dht to map identifiers onto locators. In *Proceedings of the 2008 ACM CoNEXT Conference, New York, USA.*, 2008.

[41] Michael Menth, Matthias Hartmann, and Michael Höfling. Firms: A mapping system for future internet routing. *IEEE Journal on Selected Areas in Communications*, 28:1326 – 1331, October 2010. Institute of Computer Science,University of Würzburg, Germany.

[42] Michael Menth, Matthias Hartmann, and Michael Höfling. Mapping systems for loc/id split internet routing. Technical Report 472, Institute of Computer Science, University of Würzburg., May 2010.

[43] D. Meyer, L. Zhang, and K. Fall. Report from the iab workshop on routing and addressing. `http://www.ietf.org/rfc/rfc4984.txt`, Sep 2007.

[44] David Meyer. The locator identifier separation protocol (lisp). `http://www.cisco.com/web/about/ac123/ac147/archived_issues/ipj_11-1/111_lisp.html`, March 2008.

[45] Graham Mooney. Evaluating compact routing algorithms on real-world networks. Master's thesis, Department of Computer Science, The University of Glasgow., 2010.

[46] Avinash Mungur and Christopher Edwards. Scalability of the locator identity split mapping infrastructure to support end-host mobility. In *Proceedings of the IEEE 35th Conference on Local Computer Networks*, pages 352–355, October 2010.

[47] Richard Stones Neil Matthew. *Beginning Linux Programming.* Wiley Publishing, Inc., 2007.

[48] R. Oliveira, R. Izhak-Ratzin, B. Zhang, and L. Zhang. Measurement of Highly Active Prefixes in BGP. In *Proceedings of the IEEE GLOBECOM, 2005*, 2005.

[49] Tapio Partti. Improving internet inter-domain routing scalability. Master's thesis, Faculty of Electronics, Communications and Automation, Aalto University., 2011.

[50] Cristel Pelsser, Akeo Masuda, and Kohei Shiomoto. Scalable support of interdomain routes in a single as. In *Proceedings of the IEEE Global Telecommunications Conference, 2009, Honolulu.*, pages 1 – 8. NTT Network Service Systems Laboratories, NTT Corporation, Japan, November 2009.

[51] Avinash Ramanath. A study of the interaction of bgp/ospf in zebra/zebos/quagga. `http://www.quagga.net/docs/BGP_OSPF_Interaction_Report.pdf`.

[52] Y. Rekhter, T. Li, and S. Hares. A border gateway protocol 4 (bgp-4), January 2006.

[53] Mattia Rossi. Implementing path-exploration damping in the quagga software routing suite version 0.99.13 - patch set version 0.3. Technical Report 090730A, Centre for Advanced Internet Architectures, Swinburne University of Technology., Melbourne, Australia, July 2009.

[54] Miguel Á. Ruiz-Sánchez, Ernst W. Biersack, and Walid Dabbous. Survey and taxonomy of ip address lookup algorithms. *Network, IEEE*, 15:8 – 23, Mar/Apr 2001.

[55] Oscar Santolalla. Implementation of is-is extensions for routed end-to-end ethernet. Master's thesis, Helsinki University of Technology, 2009.

[56] Abraham Silberschatz, Henry Korth, and S. Sudarshan. *Database System Concepts.* Addison-Wesley Inc., 2005.

[57] Paul Simoneau. *Hands-On TCP/IP.* Osborne/McGraw-Hill, 1997.

[58] K. Sklower. A tree-based routing table for berkeley unix. In *Proceedings of the USENIX Conference.*, pages 93 – 99, 1991.

[59] W. Richard Stevens, Bill Fenner, and Andrew M. Rudoff. *UNIX Network Programming Volume 1, Third Edition: The Sockets Networking API.* Addison-Wesley Inc., 2003.

[60] F. Templin. The subnetwork encapsulation and adaptation layer (seal). `http://tools.ietf.org/html/rfc5320`, February 2010.

[61] M. Thorup and U. Zwick. Compact routing schemes. In *Proceedings of the 13th annual ACM symposium on Parallel algorithms and architectures*, pages 1 –10, July 2001.

[62] P. Verkaik, A. Broido, Kc Claffy, R. Gao, Y. Hyun, and R. van der Pol. Beyond cidr aggregation. Technical report, CAIDA, 2004.

[63] Mei Wang and Larry Dunn. Reduce ip address fragmentation through allocation. In *Proceedings of 16th International Conference on Computer Communications and Networks, 2007.*, pages 371 – 376, 2007.

[64] Luke Welling and Laura Thomson. *MySQL Tutorial.* MySQL Press, 2004.

[65] Paul Wilton and John W. Colby. *Beginning SQL.* Wiley Publishing, Inc, 2007.

[66] Xipeng Xiao, A. Hannan, B. Bailey, and L.M. Ni. Traffic engineering with mpls in the internet. *Network, IEEE*, 14:28 – 33, Mar/Apr 2000.

Appendix A

Detail Call Graph Views of CRM

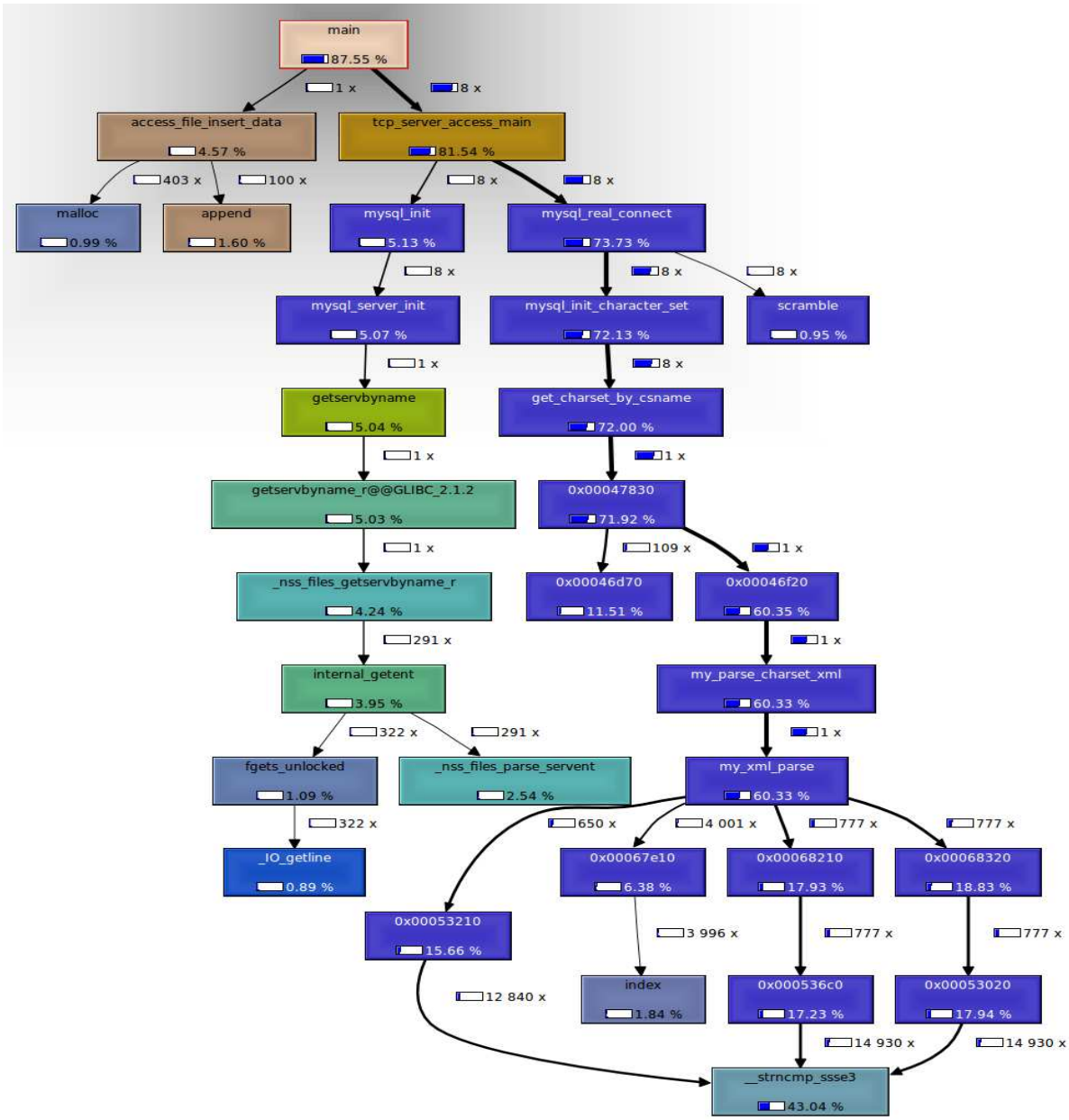

Figure A.1: Call graph view of UDP Client.

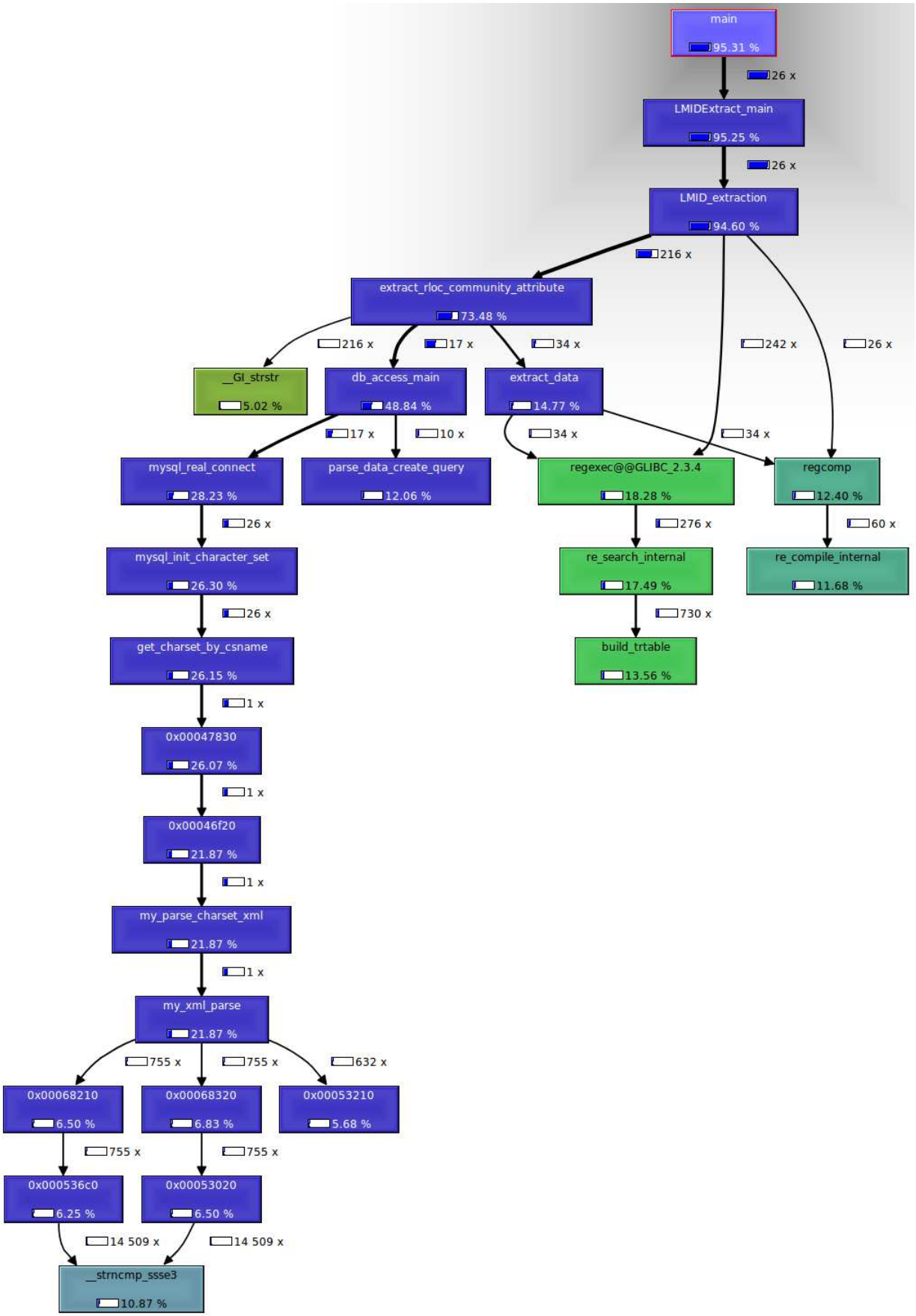

Figure A.2: Call graph view of TCP Client.

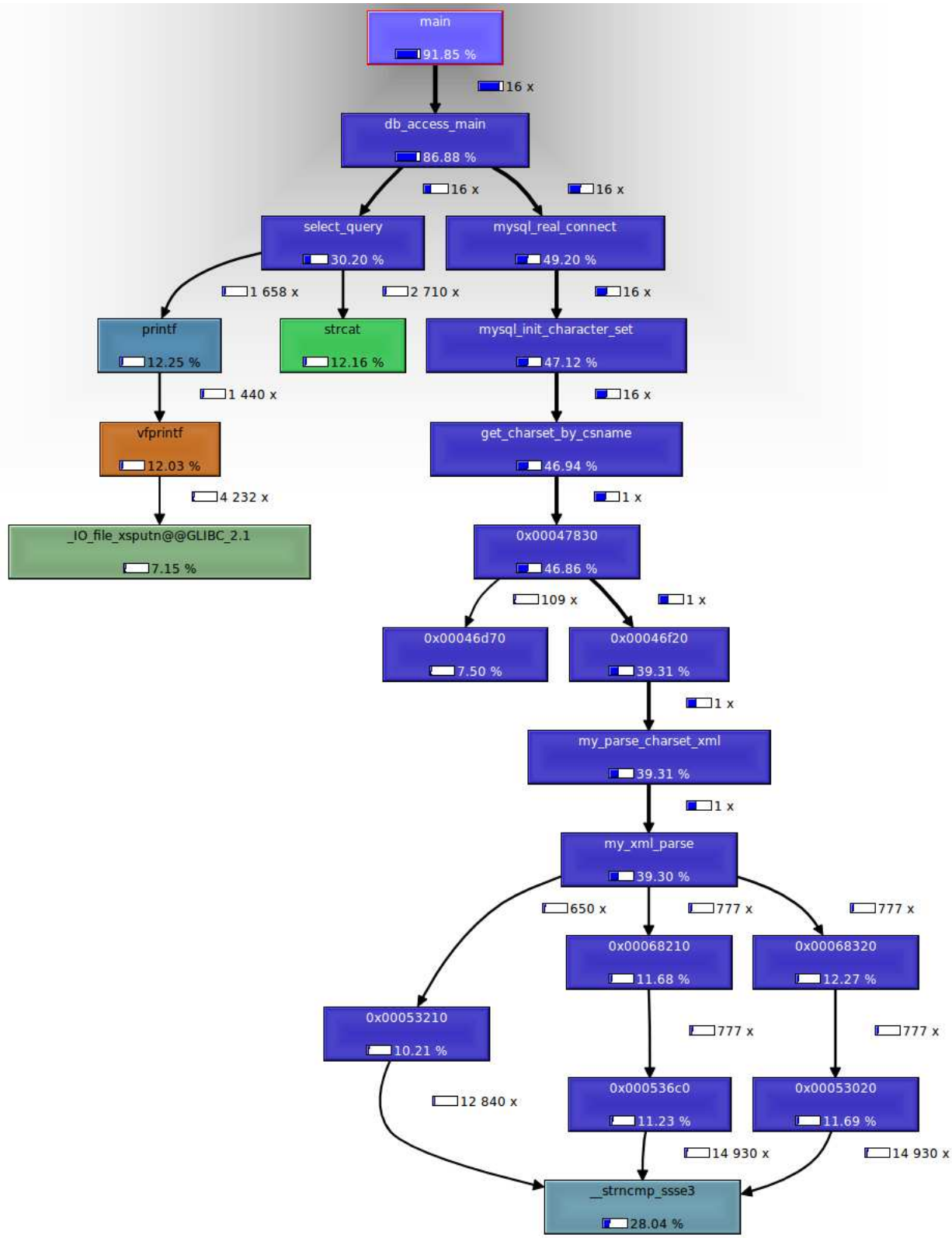

Figure A.3: Call graph view of TCP Server.

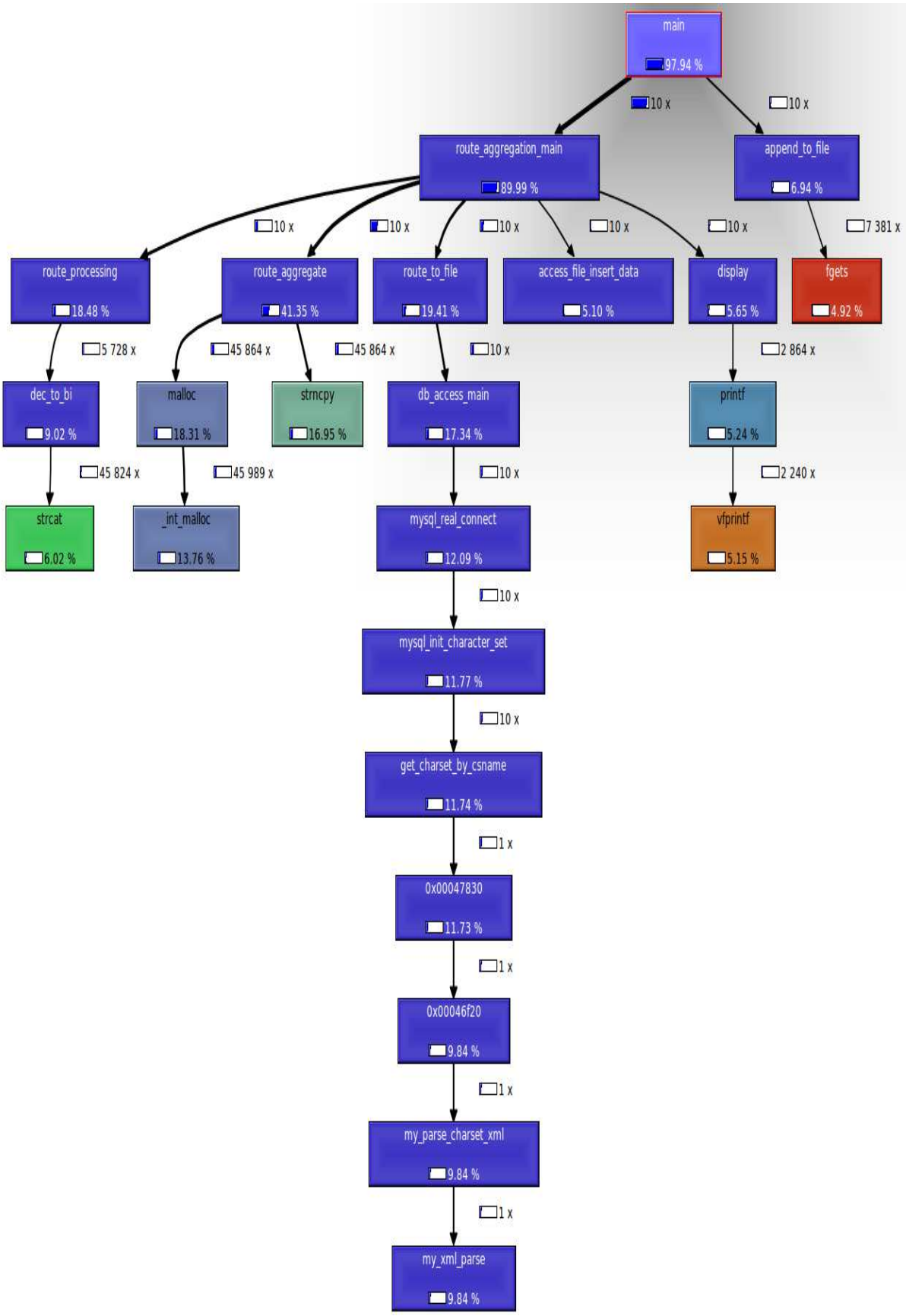

Figure A.4: Call graph view of LISP-ALT.

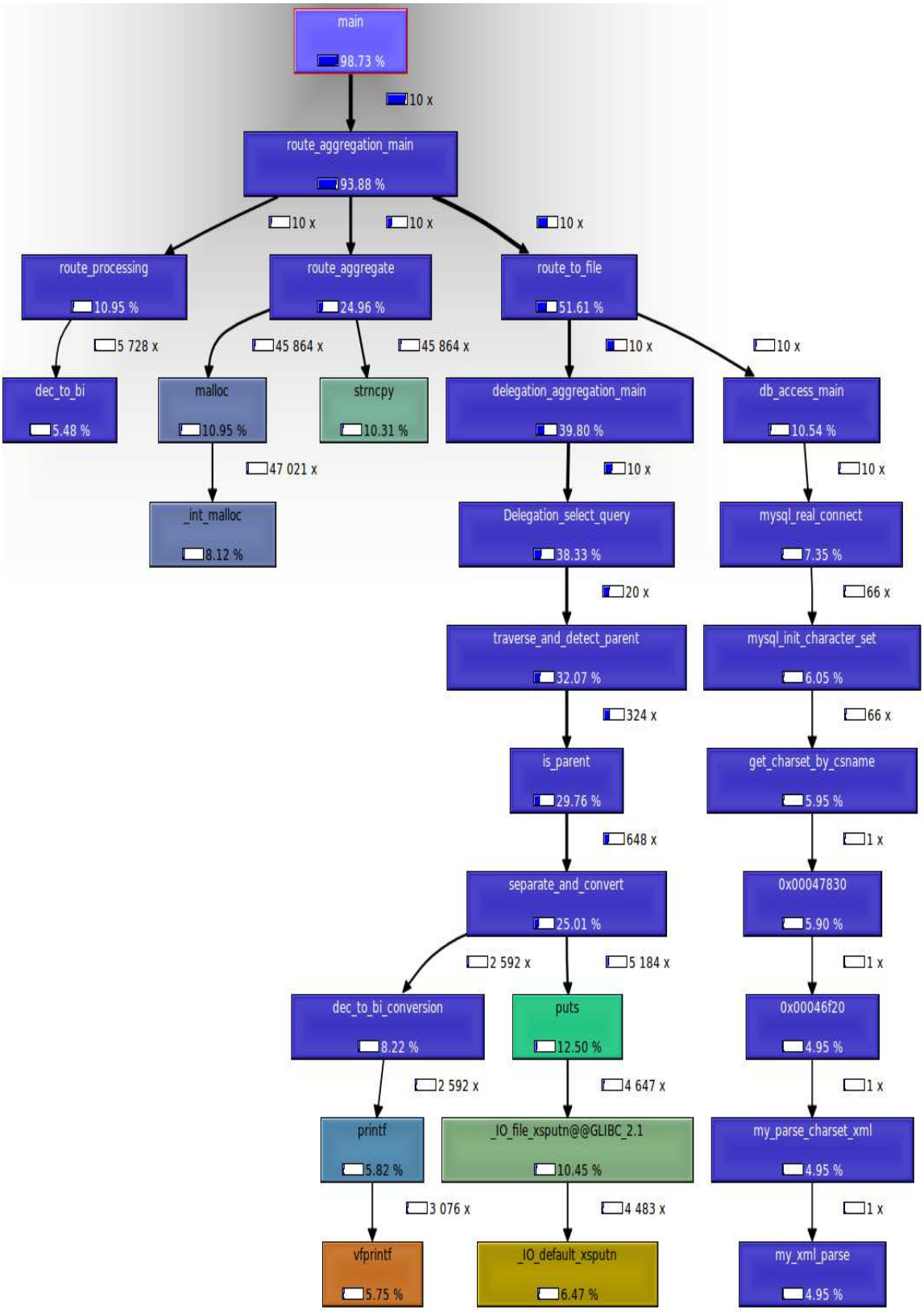

Figure A.5: Call graph view of CRM.

Appendix B

SQL Queries executed through MySQL

This appendix shows the SQL queries utilized through MySQL to create the required tables for CRM.

B.1 SQL Queries

```
/*
In order to run this file, use the following at
the "mysql prompt":
mysql> SOURCE /home/ahuq/TCPClient/sql_query_v3.sql;
*/
USE LMIDstructure;
START TRANSACTION;
drop table if exists EIDList;
/*The table with the foreign key must be deleted first*/
drop table if exists LMIDTable;

CREATE TABLE LMIDTable (LMID VARCHAR(19) NOT NULL UNIQUE,
HASH_VALUE INT NOT NULL DEFAULT 0, PRIMARY KEY(LMID))
ENGINE=InnoDB;

CREATE TABLE EIDList(LMID VARCHAR(19) NOT NULL,
EIDPrefix VARCHAR(40) NOT NULL, EIDPrefixLength
VARCHAR(3) NOT NULL DEFAULT '0',Is_Delegated CHAR NOT NULL
DEFAULT 'N', Delegated_RLOC VARCHAR(19) NOT NULL DEFAULT 'X',
UNIQUE(LMID,EIDPrefix,EIDPrefixLength),FOREIGN KEY(LMID)
REFERENCES LMIDTable(LMID)) ENGINE=InnoDB;

drop table if exists ServerAddress;

CREATE TABLE ServerAddress(IpId INT NOT NULL AUTO_INCREMENT,
IP VARCHAR(19) NOT NULL UNIQUE, PRIMARY KEY(IpId)) ENGINE=InnoDB;

COMMIT;

START TRANSACTION;
drop table if exists DelegatedLM;
CREATE TABLE DelegatedLM(LMID VARCHAR(19) NOT NULL,EIDPrefix
VARCHAR(40) NOT NULL, EIDPrefixLength VARCHAR(3) NOT NULL DEFAULT '0',
Is_Used CHAR NOT NULL DEFAULT 'N', UNIQUE(LMID,EIDPrefix)) ENGINE=InnoDB;

COMMIT;
```

Appendix C

Implementation Code of CRM

The experimental prototype described in this thesis is implemented using C(GCC). This appendix presents C code that implements some main sub-tasks of CRM. However, the reader must realize that, the actual prototype is comprised of many more complicated functions that are not shown here for the sake of simplicity. This appendix is intended to provide the reader a "high-level" view.

C.1 UDP client

C.1.1 Send map register message and receive map notify packet.

```
/**
*
* @brief  This function will send map registration message to the
* server in chunks (the size of chunk depends on the choosen MTU)
* and it will receive map notify packet for each of the map
* register packets.
* @author A. M. Anisul Huq
* @param
* @retval 0
*
*/

int main(int argc, char* argv[])
{
    //!a global variable declared in the header file has to be
    //!initialized inside a function.
    udp_cli_program_cycle = 0;
    //!udp_cli_program_cycle initialized

    int sockfd;
    struct sockaddr_in server_addr;
    struct map_register_pkt* map_register_packet;
    struct map_notify_pkt* recved_map_notify_packet;
    struct node* starting_point;
    double number_of_loops;
    int i;
    num_of_mapping = 0;//global variable initialization.

    sockfd = socket(AF_INET,SOCK_DGRAM,0);
    if(sockfd < 0)
    {
```

```
32          perror("Socket error: ");
            exit(1);
34      }

36      xx1 = 0;//!for test purposes.
        //initialize the addresses
38      bzero(&server_addr,sizeof(server_addr));
        //assign values to the sockaddr_in type structure
40      server_addr.sin_family = AF_INET;
        server_addr.sin_port = htons(SERVER_PORT);
42      inet_pton(AF_INET,SERVER_IP,&server_addr.sin_addr);

44      /**Read the file and aquire the data dynamically.*/
        starting_point = access_file_insert_data();
46      //printf("\npossible error site.3");
        position = starting_point;
48
        /** Depending on the MTU (i.e. OPTIMAL_RECORD_NUMBER)
50      we calculate how many packets (i.e. map_register_packet)
        need to be transmitted to the server.
52      */
        number_of_loops =
54      ceil( (double)(num_of_mapping)/OPTIMAL_RECORD_NUMBER );
        //!the dividend must be in double format to get a correct
56      //!output.
        /** The number of packets needed to be transmitted is
58      "number_of_loops".*/

60    for(i = 0; i < (int)number_of_loops; i++)
        {
62          int current_turn_recs;
            if ((num_of_mapping - (i * OPTIMAL_RECORD_NUMBER))
64                                  >= OPTIMAL_RECORD_NUMBER)
               current_turn_recs =  OPTIMAL_RECORD_NUMBER;
66          else
               current_turn_recs = num_of_mapping -
68                                  (i * OPTIMAL_RECORD_NUMBER);
            /**
70          In the last iteration, there might not be
            "OPTIMAL_RECORD_NUMBER" number of records left.
72          'current_turn_recs' determines how many records can be
            included in a map_register_packet.
74          */
            udp_cli_program_cycle++;
76          //!counting the number of cycles for complexity analysis.
            map_register_packet =
78          map_register_packet_initialization(map_register_packet
                                    ,position,current_turn_recs);
80          /**
            The "map_register_packet" is a structure that has an
82          element called "rec"; which in turn is an array of
            type "structure map_register_record". The "structure
84          map_register_record" has an element "char
            eid_prefix[EID_PREFIX_SIZE]" which holds the eid prefix.
86          It also has an element called "struct map_register_rloc
            loc" which holds the RLOC value.
88          */
            //send to the server
90          if( sendto(sockfd,map_register_packet,sizeof(struct
            map_register_pkt),0,(struct sockaddr*)&server_addr,
```

```
92          sizeof(server_addr)) == -1)
            {
94              perror("sendto() error: ");
                exit(1);
96          }
            //!now we receive notification packet.
98          recved_map_notify_packet = map_notify_packet_initialization
                                            (recved_map_notify_packet);
100
            recved_map_notify_packet = get_map_notify_packet(sockfd,
102                                         recved_map_notify_packet);

104         printf("\nnumber of records in the packet:  %d",
                              recved_map_notify_packet->record_count);
106
            printf("\n TCP server address:  %s",
108                 recved_map_notify_packet->server_ip_address);//!new

110         tcp_server_access_main(recved_map_notify_packet->
                                                server_ip_address);
112     }
        printf("\nTotal number of program cycles in UDP client: %ld: \n",
114                                                 udp_cli_program_cycle);
        return 0;
116 }
```

C.2 TCP client

C.2.1 Extraction of BGP's Aggregator and Community Attribute.

```
1   /**
    *
3   * @brief   Extract Aggregator and Community attribute
    *          from Quagga's advertisement.
5   * @author A. M. Anisul Huq
    * @param   vtysh_output     Quagga's advertisement in
7   *                           the form of a char array.
    * @param
9   * @retval none
    *
11  */

13  void LMID_extraction(char* vtysh_output)
    {
15      int rc;
        regex_t* myregex = calloc(1, sizeof(regex_t));
17
        //! Compile the regular expression
19      rc = regcomp( myregex,
        "[[:digit:]]{1,3}\\.[[:digit:]]{1,3}\\.[[:digit:]]{1,3}\\.
21      [[:digit:]]{1,3}", REG_EXTENDED );

23      regmatch_t pm;
        int j;
25      char* extracted_ip = (char*)malloc(STANDARD_IP_LENGTH + 1);
        memset(extracted_ip,'\0',STANDARD_IP_LENGTH + 1);
27      char *temp_storage = (char*)malloc( STATIC_ARRAY_SIZE + 1 );
        //! "+1" is for NULL character.
29      memset(temp_storage,'\0',STATIC_ARRAY_SIZE + 1);
```

```
    char *to = (char*)malloc(STATIC_ARRAY_SIZE + 1 );
31  memset(to,'\0',STATIC_ARRAY_SIZE + 1);
    strcpy(temp_storage,vtysh_output);
33
    j = regexec(myregex,temp_storage,1,&pm,0);
35  //!make the first match!

37  strncpy(extracted_ip, (temp_storage+pm.rm_so),
                        (pm.rm_eo - pm.rm_so ) );
39  //is_aggregator(extracted_ip);
    //!Test if aggregator attribute is there or not.
41
    extract_rloc_community_attribute(extracted_ip);
43  //!function call that does pattern matching and
    //!extracts RLOC and Community value.
45
    //!printf(" he he :%s\n",extracted_ip);
47  memset(extracted_ip,'\0',STANDARD_IP_LENGTH);
    memset(temp_storage,'\0',STATIC_ARRAY_SIZE + 1);
49
    strncpy(to, (vtysh_output+pm.rm_eo),
51              (strlen(vtysh_output) - pm.rm_eo ) );

53  while(1)
    {
55      j = regexec(myregex,to,1,&pm,REG_NOTBOL);

57      if(j!=0)
        {
59          tcp_cli_program_cycle++;
            //!counting the number of cycles
61          //!for complexity analysis.
            break;
63      }

65      strcpy(temp_storage,to);
        strncpy(extracted_ip, (temp_storage+pm.rm_so),
67                          (pm.rm_eo - pm.rm_so ) );
        extract_rloc_community_attribute(extracted_ip);
69      memset(extracted_ip,'\0',STANDARD_IP_LENGTH + 1);
        memset(temp_storage,'\0',STATIC_ARRAY_SIZE + 1);
71
        //!extract string and then compare(start)
73      strncpy(temp_storage, (to+pm.rm_eo), (strlen(to)
                                        - pm.rm_eo ) );
75      //!put the substring extracted from "to"
        //!into "temp_storage".
77
        //!This is done for temporary storage!
79      memset(to,'\0',STATIC_ARRAY_SIZE + 1);
        //!"to" is now filled with NULL.
81
        strcpy(to,temp_storage);
83      //!Now "to" has the extracted string.

85      memset(temp_storage,'\0',STATIC_ARRAY_SIZE + 1);
        //!Now we fill "temp_storage" with NULL so that
87     //!it can be used in the next iteration.
     }
89
```

```
    //!Garbage collection (start)
91  free(extracted_ip);
    free(temp_storage);
93  free(to);
    //!Garbage collection (end)
95  }
```

C.3 TCP Server

C.3.1 I/O Multiplexed TCP Server to handle multiple clients simultaneously.

```
1   int main()
    {
3       signal(SIGINT,(void*)sighandler);
        //!to handle CTRL + C and we will exit normally.
5
        int listenfd, connfd, pid, client_len;
7       char buff[BUFSIZ], cli_data[BUFSIZ],send_data[BUFSIZ];
        int i,fd,n;
9
        fd_set active_fd_set, read_fd_set;
11      struct sockaddr_in tcp_cliaddr, tcp_servaddr;

13      listenfd = socket(AF_INET, SOCK_STREAM, 0);

15      if(listenfd < 0)
           printf("Error in Socket creation! \n");
17
        bzero(&tcp_servaddr, sizeof(tcp_servaddr));
19      bzero(&tcp_cliaddr, sizeof(tcp_cliaddr));

21      tcp_servaddr.sin_family = AF_INET;
        tcp_servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
23      tcp_servaddr.sin_port = htons(TCP_SERVER_PORT);

25      if(  bind(listenfd, (struct sockaddr*)
                 &tcp_servaddr, sizeof(tcp_servaddr))==-1)
27      {
            printf("\nError in binding!!!!\n");
29          exit(0);
        }
31      listen(listenfd,LISTENQ); //MUST
        printf("Waiting for client on %d \n",TCP_SERVER_PORT);
33
        //initialize the set of active sockets.
35      FD_ZERO(&active_fd_set);
        FD_SET(listenfd,&active_fd_set);
37
        while(1)
39      {//1
            read_fd_set = active_fd_set;
41
            if( select(FD_SETSIZE,&read_fd_set,NULL, NULL, NULL)<0)
43          {
                perror("select error");
45              exit(EXIT_FAILURE);
            }
47
            for(fd = 0; fd < FD_SETSIZE; fd++)
```

```
49          {//2
               if(FD_ISSET(fd, &read_fd_set))
51             {//3
                   if(fd == listenfd)//!new client wants to connect
53                 {//4
                      client_len = sizeof(tcp_cliaddr);
55                    connfd = accept(listenfd,(struct sockaddr*)
                                          &tcp_cliaddr,&client_len);
57
                      FD_SET(connfd, &read_fd_set);
59                    //!Add the fd to the fd_set

61                    printf(" Connected from %s,  port  %d \n",
                      inet_ntop(AF_INET,&tcp_cliaddr.sin_addr,
63                    buff,sizeof(buff)),ntohs(tcp_cliaddr.sin_port));
                   }//4
65                 else
                   {//4
67                     //!read from the client
                       bzero(cli_data,BUFSIZ);
69                     n = read(fd,cli_data,BUFSIZ-1);

71                     if( n < 1 )//!means the client has closed.
                       {//5
73                        close(fd);
                          FD_CLR(fd,&read_fd_set);
75
                          printf("Removing client #%d from list.",fd);
77                     }//5
                       else //!means the client has data.
79                     {//5
                           printf("zzzz %s\n",cli_data);
81                         //!Here we have to put code to access the db.

83                         memset(send_data,'\0',BUFSIZ);

85                         strcpy(send_data,db_access_main(cli_data));
                           //!access then MySQL database through
87                         //!TCPServerDB.c program.

89                         printf("\n Data sent to the client:
                                                  %s \n",send_data);
91                         if((write(connfd,send_data,strlen(send_data))<0))
                               error("ERROR writing to socket");
93                     }//5
                   }//4
95
               }//3
97         }//2
       }//1
99
       return 0;
101 }
```

C.4 UDP Server

C.4.1 Listens for map register packets, sends the received data for aggregation and afterwards returns map notify message to the client.

```
1
/**
3 *
 * @brief  functions as the UDP server.
5 *
 * At the heart of this function in an infinite while()
7 * loop, that continuously listens for map registration packets.
 * Once it receives a map registration packet, it extracts and
9 * puts the data in a file called "mappingDB.txt". Then it sends
 * map notification to the client. After that, received data is
11 * aggregated!
 *
13 * @author A. M. Anisul Huq
 * @param  none
15 * @retval none
 *
17 */

19 int main(int argc, char* argv[])
{
21    signal(SIGINT,(void*)sighandler);
      //!to handle CTRL + C and we will exit normally.
23
      /**
25    When the UDP server starts for the first time,
      the mappingDB.txt needs to be empty. Otherwise
27    data will be corrupted!
      */
29
      FILE* fp;
31    fp = fopen("mappingDB.txt","w");
      fclose(fp);
33    auto int sockfd,client_len;
      struct map_register_pkt* recved_map_register_packet;
35    struct map_notify_pkt* map_notify_packet;
      struct sockaddr_in server_address, client_address;
37    sockfd = socket(AF_INET,SOCK_DGRAM,0);

39    if(sockfd < 0)
      {
41       printf("cannot open socket \n");
         exit(1);
43    }

45    //allocate memory to recved_map_register_packet
      recved_map_register_packet
47                      = recved_map_register_packet_initialization
                                (recved_map_register_packet);
49
      //initialize the addresses
51    bzero(&server_address,sizeof(server_address));
      bzero(&client_address,sizeof(client_address));
53
      server_address.sin_family = AF_INET;
```

```
55      server_address.sin_addr.s_addr = htonl(INADDR_ANY);
        server_address.sin_port = htons(SERVER_PORT);
57
        if( bind(sockfd,(struct sockaddr*)&server_address,
59                                sizeof(server_address)) == -1)
           perror("Server side bind error");
61
        printf("\nwaiting for data on port UDP %d\n",SERVER_PORT);
63
        while(1)
65      {
           client_len = sizeof(client_address);
67         int n = recvfrom(sockfd, recved_map_register_packet,
           sizeof(struct map_register_pkt), 0,
69              (struct sockaddr*)&client_address, &client_len);

71         if(n > 0)
           {
73            printf("\nUDP payload size: %d \n",n);
              /**we have to put the data in the file even
75            if no notification is sent.*/
              append_to_file(recved_map_register_packet);
77
              /** Checks to see whether map notify bit is SET or not.*/
79            if(recved_map_register_packet->m == 1)
              {
81               printf("we need to");
                 printf(" send reply.\n");
83
                 map_notify_packet
85                    = map_notify_packet_initialization
                           (map_notify_packet,recved_map_register_packet);
87
                 /**send client notification.*/
89               if( (sendto(sockfd,map_notify_packet,
                 sizeof(struct map_notify_pkt),0,(struct sockaddr*)
91               &client_address, sizeof(client_address))) == -1)

93                  perror("sendto() error");

95               //!Garbage collection
                 free(map_notify_packet);
97               //!We have already sent notification back to the client.
                 //!Hence we can free.
99            }

101           /**
              Everytime after the routes have been appended, routing
103           aggregation must take place and aggregated route
              is added to a new file.
105           */
              route_aggregation_main();
107           //!call for route aggregation to start.
           }
109
           if(n == -1)
111        {
              perror("error......");
113           exit(1);
           }
```

```
115     }
        close(sockfd);
117
        //!Garbage collection (start)
119     free(recved_map_register_packet);
        //!When the program ends, we can get rid off
121     //!the "recved_map_register_packet".
        //!Garbage collection (end)
123     return 0;
    }
```

C.4.2 Prefix length calculation for address aggregation.

```
2   /**
    *
4   * @brief  This function goes through the linked list,
    * parses it and calculates the prefix length.
6   *
    * The "struct node" has elements like, ip_address_bi
8   * and mask_ip_bi (both r strings). As the names suggest,
    * "ip_address_bi" contains the binary version of an ip
10  * address and mask_ip_bi (or prefix IP) contains the
    * binary version of the prefix. Until now, these two fields
12  * remained vacant. This function fills them by first
    * extracting all the octets (in two steps) and then by
14  * converting them to binary with the help of the function
    * "dec_to_bi()". This function also calculates the mask
16  * length.
    *
18  * @author A. M. Anisul Huq
    * @param  snode           pointer to the starting node of
20  *                         the linked list.
    * @retval
22  *
    */
24
    void route_processing(struct node* snode)
26                  //!snode is starting node.
    {
28      auto char delims[] = "/";
        auto char mdelims[] = ".";//micro delimeter
30
        auto char* token = NULL;
32      auto char* mtoken = NULL;//micro token
        auto char* last;
34
        char* separated_token[2];
36      char* mseparated_token[4];

38      char* original_eid_prefix;

40      auto struct node* cnode = snode;
        auto int i,j;
42
        /**
44      we parse the eid prefix; first on the basis of "/"
        (the outer while loop)
46      and then on the basis of "." (the inner while loop).
        */
```

```
48      while( cnode != NULL )
        {
50          bzero(temp_ip_address,ADDRESS_LENGTH);
            bzero(temp_mask_ip,ADDRESS_LENGTH);
52          /**temp_ip_address and temp_mask_ip contains the
            binary version of the ip address and mask ip.*/
54
            //store cnode->temp_eid_prefix in original_eid_prefix.
56          original_eid_prefix = malloc(strlen(cnode->temp_eid_prefix) +1);
            strcpy(original_eid_prefix,cnode->temp_eid_prefix);
58
            i = 0;
60          token = strtok( original_eid_prefix, delims );

62          while( token != NULL )
            {
64              /**the outer while loop separates on the basis of "/".*/
                separated_token[i] = malloc(strlen(token) + 1);
66              strcpy(separated_token[i],token);

68              /**the inner while loop separates on the basis of "."*/
                j = 0;
70              mtoken = strtok_r( separated_token[i], mdelims, &last );

72              while( mtoken != NULL)
                {
74                  mseparated_token[j] = malloc(strlen(mtoken) + 1);
                    strcpy(mseparated_token[j],mtoken);
76                  //printf("%s  ", mseparated_token[j] );

78                  //!convert to binary. and also decide the
                    //!length of the mask (start)
80                  if(i == 0)
                        dec_to_bi( atoi(mseparated_token[j]), 0);
82
                    if(i == 1)
84                  //!we count the mask_len only in case of mask IP.
                        dec_to_bi( atoi(mseparated_token[j]), 1);
86                  //!convert to binary. and also decide the
                    //!length of the mask (end)
88
                    j++;
90                  mtoken = strtok_r( NULL, mdelims, &last );
                }
92
                if(i == 1)
94              {
                    cnode->mask_len = number_of_ones;
96                  /**"number_of_ones" is a global variable which is
                    used to count the mask length.*/
98                  number_of_ones = 0;
                }
100
                i++;
102             token = strtok( NULL, delims );
            }
104
            strcpy(cnode->ip_address_bi,temp_ip_address);
106         strcpy(cnode->mask_ip_bi,temp_mask_ip);
```

```
108         cnode->ancestor_flag = 0;
            //!initially we presume that, there are no ancestor
110         //!for this node.

112         cnode = cnode->next;
        }
114     //!Garbage collection (start)
        free(original_eid_prefix);//!Lookout.
116     auto int k;

118     for(k = 0;k < 2;k++)
        {
120         free(separated_token[k]);
        }
122
        for(k = 0;k < 4;k++)
124     {
            free(mseparated_token[k]);
126     }
        //!Garbage collection (end)
128 }
```

C.4.3 IP Address aggregation.

```
    /**
2   *
    * @brief  This function determines whether a node has an
4   *         ancestor or not.
    * @author A. M. Anisul Huq
6   * @param  snode           pointer to the starting node of
    *                         the linked list.
8   * @retval
    *
10  */

12  void route_aggregate(struct node* snode)
    {
14      //display(snode);
        auto int i;
16      auto int match_flag = 1;

18      char* cnode_substring;
        char* tnode_substring;
20
        struct node* cnode = snode;
22      //!cnode means current node
        struct node* tnode;
24      //!tnode means traversing node

26      while(cnode!=NULL)
        {//loop for cnode (start)
28          //tnode = cnode->next;
            /**
30          For every cnode, we traverse the whole linked list
            to find a possible ancestor. Thats why, the "tnode" or
32          "traversing node" is always initialized to the starting
            location.
34          */
            tnode = snode;
36          while( tnode != NULL)
```

```
        {//!loop for tnode (start)
38          if( tnode->mask_len < cnode->mask_len)
            /**tnode maybe an ancestor of cnode. To be a parent,
40          tnode's mask_len must be less than cnode's mask_len.*/
            {
42              cnode_substring = (char*)malloc(tnode->mask_len + 1);
                strncpy(cnode_substring, cnode->ip_address_bi,
44                                                  tnode->mask_len);
                cnode_substring[tnode->mask_len]='\0';
46
                tnode_substring = (char*)malloc(tnode->mask_len + 1);
48              strncpy(tnode_substring, tnode->ip_address_bi,
                                                  tnode->mask_len);
50              tnode_substring[tnode->mask_len]='\0';

52              if( strcmp(tnode_substring,cnode_substring) == 0 )
                {
54                  cnode->ancestor_flag = 1;
                    /**we have determined that our current node or
56                  cnode has an ancestor!
                    */
58              }
            }
60          tnode = tnode->next;
        }//loop for tnode (end)
62

         cnode = cnode->next;
64    }//loop for cnode (end)

66    //!Garbage collection (start)
      free(cnode_substring);//!Lookout.
68    free(tnode_substring);
      //!Garbage collection (end)
70  }
```

C.4.4 Virtual Prefix Generation.

```
1

    /**
3   *
    * @brief  This function finds a "common" address based
5   *         on a pre-defined prefix length.
    * @author A. M. Anisul Huq
7   * @param  none
    * @retval returns a dummy value.
9   *
    */
11
    char* find_common()
13  {
        auto char* last;
15      auto char temp_input[STATIC_ARRAY_SIZE];
        memset(temp_input,'\0',STATIC_ARRAY_SIZE + 1);
17      strcpy(temp_input,virtual_prefix_input);
        int is_virtual = 0;
19
        auto char* ip_address_bi
21              = (char*)calloc( ADDRESS_LENGTH + 1,sizeof(char));
        auto char* prev_address_bi
23              = (char*)calloc( ADDRESS_LENGTH + 1,sizeof(char));
```

```
    auto char* curr_address_bi
25              = (char*)calloc( ADDRESS_LENGTH + 1,sizeof(char));

27  auto int j,i = 0;
    auto char* token = NULL;
29  auto char* mtoken = NULL;
    auto char delims[] = ",";
31  auto char mdelims[] = ".";
    token = strtok( temp_input, delims );
33
    while( token != NULL )
35  {
        //!the outer while loop separates on the basis of "/".
37
        auto char* temp_ip
39        = (char*)calloc( STANDARD_IP_LENGTH + 1,sizeof(char));
        strncpy(temp_ip,token,(strpbrk(token, "/") - token));
41      //! to get rid of the prefix length.
        printf("\n temp_ip: %s  ", temp_ip);
43
        //!the inner while loop separates on the basis of "."
45      j = 0;
        mtoken = strtok_r( temp_ip, mdelims, &last );
47      while( mtoken != NULL)
        {
49          auto char* temp_part = (char*)calloc( 5,sizeof(char) );
            strcpy(temp_part,mtoken);
51          printf("\n temp_part: %s  ", temp_part);
            //!convert to binary. and also decide the length of
53          //!the mask (start)
            if(j == 0)
55              strcpy( ip_address_bi,
                        ddec_to_bi_conversion( atoi(temp_part)) );
57          else
                strcat(ip_address_bi,
59                      ddec_to_bi_conversion( atoi(temp_part)) );
            j++;
61          mtoken = strtok_r( NULL, mdelims, &last );
        }
63
        printf("\nBinary address: %s \n",ip_address_bi );
65
        if(i == 0)
67      {
            strcpy(prev_address_bi,ip_address_bi);
69          strcpy(curr_address_bi,ip_address_bi);
            printf("\nYES!\n");
71          is_virtual = 1;
        }
73      else
        {
75          strcpy(prev_address_bi,curr_address_bi);
            //!Binary IP from previous iteration goes into
77          //!"prev_address_bi".

79          strcpy(curr_address_bi,ip_address_bi);
            //!Now "curr_address_bi" has the current IP's binary
81
            if(strcmp(strndup(prev_address_bi,MAXGROUP),
83                      strndup(curr_address_bi,MAXGROUP))==0)
```

```
                {
                    printf("\nYES!\n");
                    is_virtual = 1;
                }
                else
                {
                    is_virtual = 0;
                    break;
                }
            }
            i++;
            token = strtok( NULL, delims );
        }

        if(is_virtual == 1)
        {
            printf("\nthis chunk is aggregatable with
                                a MAXGROUP of 16.\n");
            //!Now we calculate the VIRTUAL prefix.
            bin_to_dec(curr_address_bi);
            //!This is where everything starts!
            t_display_and_update(v_start,global_vir_ip);
        }
        else
        {
            printf("\nthis chunk is NOT aggregatable.\n");
        }

    return "a";
}
```

C.4.5 Advertisement through Quagga.

```
/**
*
* @brief  This function at first removes any network that
*         has the same address as our RLOC and then
*         the it advertises the RLOC and community value.
*
* In order to reduce the delay between advertisements the
* minimum route advertisement interval (MRAI) between
* consecutive BGP routing updates is set to 10 seconds. We
* did NOT do this through this function. It was done by hard
* coding in the /etc/quagga/bgpd.conf file. That's
* why, we have a sleep of 15 seconds.
*
* @author A. M. Anisul Huq
* @param  t_rloc            The RLOC IP that will be
*                           advertised through Quagga.
* @param  community_value   Coomunity value to be advertised.
* @retval none
*
*/

void vtysh_input(char* t_rloc,int community_value)
{
    char remove_input[BUFSIZ];
    char modify_input[BUFSIZ];
    char temp_num[5];
    memset(temp_num,'\0',5);
```

```
28      memset(remove_input,'\0',BUFSIZ);
        memset(modify_input,'\0',BUFSIZ);
30
        strcpy(remove_input,"vtysh -c \"configure terminal\"
32          -c \"router bgp 100\" -c \"no network ");
        strcat(remove_input,t_rloc);
34      strcat(remove_input,"/32\" -c \"no ip prefix-list PLIST1
        permit 0.0.0.0/0 le 32\"");
36      printf("write buffer:   %s\n",remove_input);
        system(remove_input);
38
        strcpy(modify_input,"vtysh -c \"configure terminal\"
40          -c \"router bgp 100\" -c \"network ");

42      strcat(modify_input,t_rloc);
        //!the network mentioned in "t_rloc" will be advertised.
44      strcat(modify_input,"/32\"
            -c \"neighbor 10.144.13.65 remote-as 7675\"");
46      //!Note I have hard coded the neighbor info.
        //!This does not cause any problem!
48
        strcat(modify_input," -c \"ip prefix-list PLIST1
50                                  permit 0.0.0.0/0 le 32\"");

52      strcat(modify_input," -c \"route-map OUTBOUND permit 10\"
        -c \"match ip address prefix-list PLIST1\" -c \"set community ");
54
        sprintf(temp_num,"%d",community_value);
56      strcat(modify_input,temp_num);
        strcat(modify_input,":0 \"");
58      strcat(modify_input," -c \"set aggregator as 7675 ");

60      //!this remote-as number 7675 is also present in
        //!this machine's static configuration. Fix it for
62      //!the other machines.

64      strcat(modify_input,t_rloc);
        strcat(modify_input,"\"");
66
        printf("write buffer:   %s\n",modify_input);
68      system(modify_input);

70      //we have to introduce a delay of 15 secs here.
        sleep(15);
72      //if we are advertising a new network ......
        /**To withdarw the previous community value.
74      So that a new community value can be advertised.*/
        system("vtysh -c \"configure terminal\"
76          -c \"route-map OUTBOUND permit 10\"
                -c \"match ip address prefix-list PLIST1\"
78                  -c \"set community none\"");
    }
```