

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Scuola di Ingegneria e Architettura

Corso di Laurea in INGEGNERIA INFORMATICA

Tesi di Laurea in TECNOLOGIE WEB T

State-of-the-Art Multihoming Protocols and Support for Android

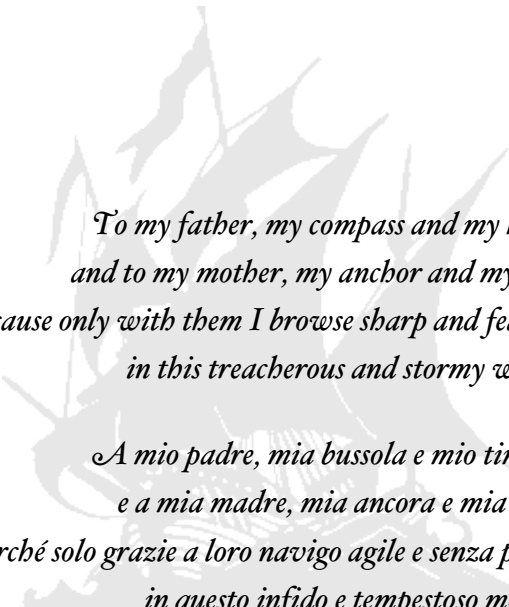
Candidato:

Luca Stornaiuolo

Relatore:

Chiar.mo Prof. Ing. Paolo Bellavista

Anno Accademico 2015/2016 - Sessione I



*To my father, my compass and my helm,
and to my mother, my anchor and my sail,
because only with them I browse sharp and fearless
in this treacherous and stormy world.*

*A mio padre, mia bussola e mio timone,
e a mia madre, mia ancora e mia vela,
perché solo grazie a loro navigo agile e senza paura
in questo infido e tempestoso mondo.*

Sommario esteso

Il traguardo più importante per la connettività wireless del futuro sarà sfruttare appieno le potenzialità offerte da tutte le interfacce di rete dei dispositivi mobili. Per questo motivo con ogni probabilità il *multihoming* sarà un requisito obbligatorio per quelle applicazioni che puntano a fornire la migliore esperienza utente nel loro utilizzo. Sinteticamente è possibile definire il multihoming come quel processo complesso per cui un end-host o un end-site ha molteplici punti di aggancio alla rete. Nella pratica, tuttavia, il multihoming si è rivelato difficile da implementare e ancor di più da ottimizzare.

Ad oggi infatti, il multihoming è lontano dall'essere considerato una feature standard nel network deployment nonostante anni di ricerche e di sviluppo nel settore, poiché il relativo supporto da parte dei protocolli è quasi sempre del tutto inadeguato. Ad esempio, gli attuali protocolli per il mobility management non sono in grado di gestire il multihoming in modo nativo e perciò devono essere combinati con altri protocolli per fornire tale supporto. Altri protocolli di rete invece, introducono dei nuovi layer nello stack IP classico per svolgere specifiche funzioni e al contempo ridurre la complessità derivante dai meccanismi implementativi del multihoming.

Naturalmente anche per Android in quanto piattaforma mobile più usata al mondo, è di fondamentale importanza supportare il multihoming per ampliare lo spettro delle funzionalità offerte ai propri utenti. Dunque alla luce di ciò, in questa tesi espongo lo stato dell'arte del supporto al multihoming in Android mettendo a confronto diversi protocolli di rete e testando la soluzione che sembra essere in assoluto la più promettente: LISP.

Il Locator/Identifier Separation Protocol definisce un'infrastruttura per il disaccoppiamento dell'identità dell'host e della sua ubicazione nel namespace degli indirizzi. Tale separazione si ottiene sostituendo gli indirizzi IP correntemente usati con due namespace separati: EID e RLOC. Separare l'identità dell'host (EID) dal suo localizzatore (RLOC) significa consentire la mobilità seamless dal momento che le applicazioni possono agganciarsi ad un indirizzo permanente, ossia l'EID dell'host. In tal modo l'ubicazione dell'host (e il suo punto d'accesso alla rete) può variare diverse volte durante la medesima connessione; ogni volta, il router LISP incapsula e instrada il traffico su un tunnel verso il nuovo RLOC, preservando la connessione dall'interruzione.

Esaminato lo stato dell'arte dei protocolli con supporto al multihoming e l'architettura software di LISPmob per Android, l'obiettivo operativo principale di questa ricerca è duplice: *a)* testare il *roaming seamless* tra le varie interfacce di rete di un dispositivo Android, il che è appunto uno degli obiettivi del multihoming, attraverso LISPmob; e *b)* effettuare un ampio numero di test al fine di ottenere attraverso dati sperimentali alcuni importanti parametri relativi alle performance di LISP per capire quanto è realistica la possibilità da parte dell'utente finale di usarlo come efficace soluzione multihoming. La versione di riferimento di LISPmob utilizzata in questa ricerca è quella per dispositivi non-rooted.

Pertanto, una prima parte più descrittiva sui protocolli che supportano il multihoming rappresenta il background teorico sul quale è costruita la seconda parte della tesi, più applicativa, su LISPmob nella sua implementazione Android.

Contents

Introduction	1
1 Multihoming and Mobility	5
1.1 Multihoming Base Concepts and Goals	6
1.1.1 Design concepts	6
1.1.2 Support strategies	8
1.1.3 Other related concepts	10
1.2 Consequences of Mobility	11
1.2.1 Networks topology	11
1.2.2 Security issues	12
1.3 Multihoming solutions	13
2 Network Protocols for Multihoming	15
2.1 Site Multihoming by IPv6 Intermediation	16
2.1.1 Overview and Goals	16
2.1.2 Protocol insights	17
2.2 Host Identity Protocol	19
2.2.1 HIP Architecture	20
2.2.2 HIP Base Exchange	21
2.2.3 End-host Mobility and Multihoming	23
2.3 Identifier/Locator Network Protocol	24
2.3.1 Architectural Overview	24
2.3.2 Node Identifier	28
2.3.3 Multihoming and Multi-Path Transport	29

2.4	Locator/Identifier Separation Protocol	29
2.4.1	Overview	29
2.4.2	LISP Mapping System	30
2.4.3	LISP-MN	31
2.4.4	LISP and legacy internetworking	34
2.5	Stream Control Transport Protocol	35
2.5.1	SCTP Overview	35
2.5.2	Protocol insights	36
2.5.3	Multihoming and Dynamic Address Reconfiguration	38
2.6	Considerations about multihoming protocols	39
3	Android and Networking	43
3.1	Android Market Share and Evolution	43
3.2	Android Architecture	46
3.2.1	Linux Kernel	46
3.2.2	Libraries	48
3.2.3	Android Runtime	48
3.2.4	Application Framework	49
3.2.5	Applications	49
3.3	Support to Connectivity	53
3.3.1	The <code>ConnectivityManager</code> class	54
3.3.2	The <code>Service</code> class	55
3.3.3	The <code>VpnService</code> and <code>VpnService.Builder</code> classes	59
3.4	State-of-the-Art Multihoming Support	61
3.4.1	HIP for Linux	64
3.4.2	LISPMob	65
4	State-of-the-Art Android LISPMob and Research Project	67
4.1	LISPMob Versions and Functionalities	67
4.1.1	Reference Version	68
4.2	High-Level Software Architecture	69
4.2.1	Package Organization	69
4.2.2	Activities	69

4.2.3	Services	74
4.2.4	Utility Libraries	75
4.2.5	Java Native Interface	76
4.3	Implementation Guidelines	77
4.3.1	The Beta Network	77
4.3.2	LISP-MN and NAT traversal	78
4.3.3	LISPMob build and install from source code	78
4.4	Research Project	80
4.4.1	Project Definition and Phases	80
4.4.2	Project Testbed	82
5	Implementation Insights on LISPMob and Experimental Results	83
5.1	Development phase	83
5.1.1	Configuration and first run	83
5.1.2	Error fixing	84
5.1.3	Building and second run	85
5.1.4	Fork, code analysis and solution implementation	86
5.2	Test phase	88
5.2.1	Seamless roaming	88
5.2.2	LISP performance	90
5.3	Analysis phase	91
5.3.1	Delay	91
5.3.2	Speed	95
5.3.3	Precision	95
	Conclusions and future work	97
A	Android Connectivity Features Evolution	99
B	The android.net package UML	111
C	LISPMob Configuration File (lispd.conf)	113
D	List of the selected active EIDs	117

Bibliography**123****Acknowledgements****129**

List of Figures

1.1	Multihomed Host	6
1.2	Multihomed Network	7
1.3	Active BGP entries (FIB)	10
2.1	SHIM6 Protocol Stack	17
2.2	SHIM6 Mapping with Changed Locators	19
2.3	HIP Architecture	21
2.4	HIP four-way handshaking	22
2.5	ILNP Architecture	25
2.6	LISP Architecture	30
2.7	Registering an EID-to-RLOC binding	32
2.8	Internetworking with non-LISP sites	35
2.9	SCTP four-way handshaking	37
3.1	History of Mobile OS Market Share (% on all devices)	44
3.2	History of Mobile OS Market Share (Millions of units)	45
3.3	Android Architecture	47
3.4	The Activity life-cycle	50
3.5	Android Connectivity Packages	54
3.6	The <code>ConnectivityManager</code> class	55
3.7	The <code>Service</code> class	56
3.8	The <code>VpnService</code> and <code>VpnService.Builder</code> classes	60
4.1	The <code>LISPMob</code> Activity	70
4.2	The <code>confActivity</code> Activity	70

4.3	The <code>logActivity</code> Activity	71
4.4	The <code>updateConfActivity</code> Activity	72
4.5	The <code>LISPmobVPNService</code> Service	73
4.6	The <code>IPC</code> class	74
4.7	The <code>ConfigTools</code> class	75
4.8	The <code>MultiSelectionSpinner</code> class	76
4.9	The <code>Notifications</code> class	76
4.10	The <code>LISPmob_JNI</code> class	77
5.1	Seamless roaming	89
5.2	Plot of \bar{x} and \bar{y}	92
5.3	Frequency Distribution of \bar{x} and \bar{y}	93
5.4	Composition of ping packets	94
B.1	The <code>android.net</code> package and relationships between its classes	112

Introduction

Future's wireless connectivity shall take advantage from all network interfaces on-board mobile devices. For this reason, multihoming will be compulsory in order to have the best user experience for all the modern Internet applications. As a concise definition, *multihoming* is the property of an end-host or an end-site of having multiple first-hop connections to the network. In practice, however, multihoming has proved difficult to implement and optimize.

Multiaccess and multihoming are yet to become prevalent in network deployments despite years of research and development in the area. Indeed, the corresponding support is often missing from state-of-the-art protocols. For example, modern mobility management protocols are not capable of handling multihoming natively and must be combined with other protocols to enable enhanced multihoming support. Furthermore, in some proposals new layers are introduced to perform specific functionalities and aim at reducing the ensuing complexity due to multihoming mechanisms in the original protocol stack [50].

Thus, since Android is the world most used mobile platform, multihoming support is of crucial importance in order to broaden the spectrum of features available to its users. Hence, in this thesis I highlight the state-of-the-art multihoming support on Android comparing different networking protocols and testing the solution that seems to be the most promising one: LISP.

The Locator/Identifier Separation Protocol specifies an architecture for decoupling host identity from its location information in the current address scheme. This separation is achieved by replacing the addresses currently used

in the Internet with two separate name spaces: EID, and RLOC. Separating the host identity (EID) from its locator (RLOC) enables seamless endpoint mobility by allowing the applications to bind to a permanent address, the host's EID. The location of the host can change many times during an ongoing connection. Each time, the LISP tunnel routers will encapsulate the packets to the new RLOC, preserving the connection session from breaking [47].

Once examined the state-of-the-art multihoming protocols and the LISPmob's Android software architecture, the main operative goal of this research is dual: *a)* to test the *seamless roaming* from an Android device's network interface to another which may be considered as a multihoming's goal, using indeed the LISPmob implementation; and *b)* to make a wide number of tests in order to estimate from experimental results some valuable LISP performance parameters, and then to draw forth how real is the chance of using it as an effective multihoming solution for the final user. The LISPmob reference version for this research is the one for non-rooted devices.

So, the initial more descriptive part on multihoming protocols represents the theoretical background on which is built the last, more applicational part of the thesis, about LISPmob Android implementation. This is the thesis' outline:

Chapter 1 In this chapter the concept of multihoming is introduced and explained. Support strategies, other related concepts and issues relative to mobility like dynamic network topology and security, are synthetically covered as well.

Chapter 2 The latest proposed protocols with support for multihoming are here presented with references to their RFC documents and their possible realistic future usage. A systematic comparison points out pros and cons of all of them.

Chapter 3 In this chapter the Android platform is presented from the motivation underlying its choice to its networking module throughout by its architecture. Here the real state-of-the-art of multihoming support is disclosed as well as two of the most important multihoming protocol Android implementations: HIP for Linux and LISPmob.

Chapter 4 Firstly the LISPmob implementation is analyzed in detail from a high-level software engineering point of view, and then the thesis' research project is fully defined.

Chapter 5 Here the experimental results are presented. All the collected data about LISP IPv4-in-IPv4 and IPv4 are compared and put under the magnifying glass of the statistical analysis.

Chapter 1

Multihoming and Mobility

The current Internet architecture was not designed to easily accommodate mobility because Internet Protocol (IP) addresses are used both to *identify* and *locate* the hosts. Separating identity from routing location is an important design principle of inter-domain networks that was known even before the Internet was created, but unfortunately the current architecture does not implement it. Such separation would seamlessly provide mobility and multihoming, among other desirable features, to the Internet. As a result, this is still an important research topic, and many solutions centered around this idea have been proposed [47].

Multihoming and multiaccess in IP networks have been lately fostered by the exponential growth in availability of devices with multiple built-in communication technologies. Paradigms where hosts have access to various networks are not new, of course. Multihoming has long been adopted to increase resilience, dependability, and performance in high-end servers. At the other end of the network node spectrum, mobile phone manufacturers have been integrating different cellular radio access technologies into “multi-band” cell phones to realize global reachability and ease migration. Nonetheless, multiaccess network selection is currently rudimentary and automation is not implemented yet. Today, efficient multihoming and multiaccess support in heterogeneous networks is still inhibited by mechanisms that rely mainly on presets and static policies, and require user input as well.

Nodes with multiple network interfaces have the potential of connecting to different networks and capitalizing on heterogeneous network resources and, in the process, enable their users to enjoy high-performing ubiquitous communication. On the other hand, multiaccess and multihoming lead to more complex application and protocol configurations in order to meet the challenging goals of reliability, ubiquity, load sharing, and flow distribution. These communication system properties are tightly coupled with the multihoming concept [50].

1.1 Multihoming Base Concepts and Goals

1.1.1 Design concepts

End-host multihoming is defined when a host has multiple addresses configured on the links it connects to, thus having the possibility to explore several paths to reach a peer, as each address is normally advertised by different access routers.

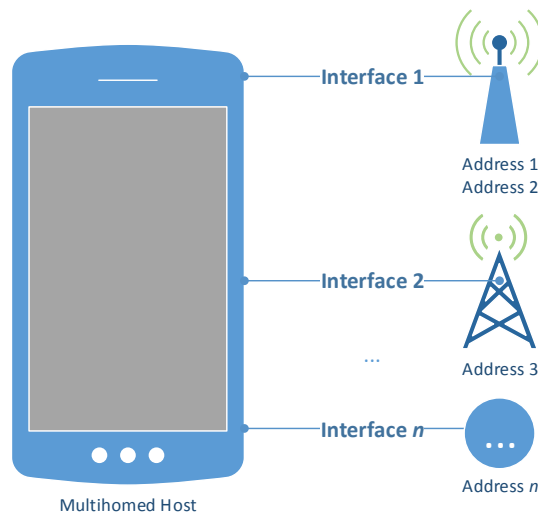


Figure 1.1: Multihomed Host

A multihomed host, on which different interfaces (logical or physical)

exist, is depicted in Figure 1.1. In addition, each interface can have different network addresses configured. For instance, *Interface 1* has been assigned two addresses, namely *Address 1* and *Address 2*. Moreover, the host can have multiple physical interfaces which have been associated with a single address, as is the case of *Interface 2* and *Interface n* with *Address 3* and *Address n*, respectively.

End-site multihoming, instead, where a site uses multiple connections to the Internet to meet objectives such as increasing network reliability or improving performance, is a common network configuration.

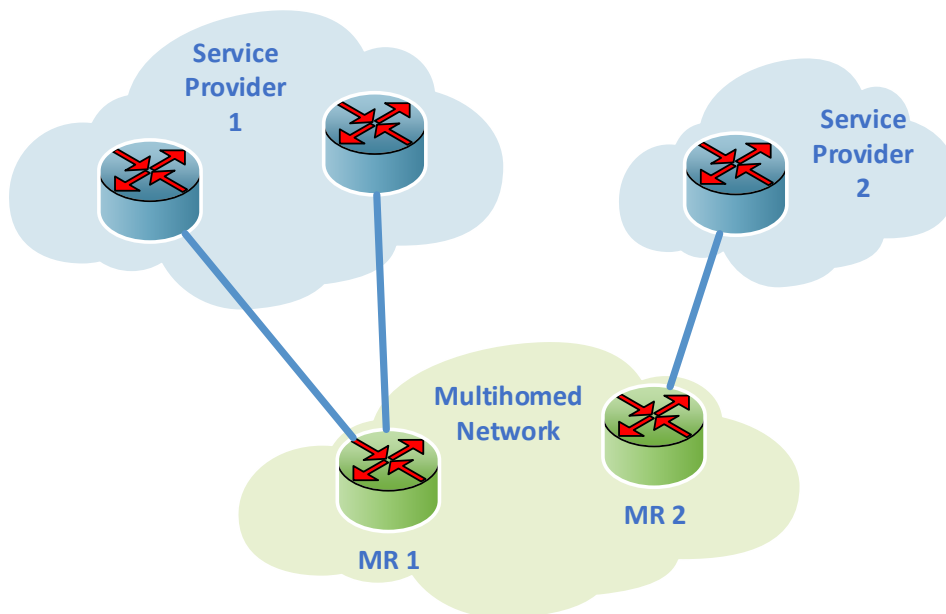


Figure 1.2: Multihomed Network

Figure 1.2 illustrates a multihomed site, which has connections to two service providers. A multihomed network can have multiple Mobile Router (MR), such as, for example, *MR 1* connecting to *Internet Service Provider 1* and *MR 2* connecting to *Internet Service Provider 2*. Moreover, a single router can have several external interfaces that connect to the same or different service providers, as the example of *MR 1*.

Multihoming has gained attention over the last few years, mainly due to the potential benefits. In particular, multihoming solutions aim to achieve the following goals: resilience, ubiquity, load balancing/sharing and flow distribution.

The diversity of multiple interfaces/paths can improve **resilience** since in case of failure of one interface/path, another can be employed to provide connectivity. For instance, a “primary-backup” model is adopted by Stream Control Transport Protocol (SCTP): if the primary path fails, the backup path can be used seamlessly without causing any application-layer service interruption. So, multiple network interfaces, in particular when used in a mobile and wireless network environment, enable **ubiquitous access** to the Internet over different media.

Load sharing goes one step further than the primary-backup model, as multiple interfaces/paths can be used simultaneously to improve throughput. For example, Iyengar *et al.* describe how one can perform concurrent multiple transfers using base SCTP [26].

Flow distribution, or flow stripping, offers an even finer granularity than load sharing. For many, flow distribution is the ultimate goal to achieve, as it implicitly means that all previous goals are also attained. Flows are stripped, perhaps even dynamically, according to policies and preferences aiming to reduce cost, optimize bandwidth use, and minimize the effect of bottlenecks to delay-sensitive applications, among others. Such policies can be defined by users or service providers.

1.1.2 Support strategies

Multihoming support could potentially be added at any layer of the protocol stack. The designer’s choice, of course, comes with certain advantages and disadvantages, and one needs to consider thoroughly the tradeoffs as well as the complexity of each solution. Deployment considerations need also to be addressed early on. In general, there are two possible approaches for introducing multihoming. On the one hand, a multihoming proposal may be completely transparent to upper layers, in such a way that there

is no disruption to ongoing sessions. On the other hand, the solution may not be transparent, but allows upper layers to participate in multihoming management and operation.

The first guideline that should be considered relates to the **locator/identifier split**. Recently, the impossibility of the current Internet's architecture to scale with the routing table size (see Figure 1.3) was deemed in an Internet Architecture Board (IAB) workshop as one of its most important problems [34], and after a detailed analysis, the overloading of IP address semantics with both *location* and *identity*, has been determined as the main source of this limitation [16]. Conventional IP architectures assume that the transport layer endpoints are the same entities as those used by the network layer. Thus, multihoming support based on a locator/identifier split requires the transport layer identity to be decoupled from the network layer locator in order to allow multiple forwarding paths to be used by a single transport session. Different approaches can be considered, either by modifying an existing protocol layer or by introducing a new layer. With the new layer approach, upper layer protocols (e.g. applications) use endpoint identifiers to uniquely identify a session while the lower layer protocols (e.g. network) employ locators. If this approach is used, a mapping between an identifier and a locator is necessary. In a multihoming context, the identifier/locator mapping must be assured by a dynamic process so that a session can include different features, such as constant endpoint identifiers throughout the session lifetime, and modification of locators to maintain end-to-end reachability [50].

Another recommendation for end-site multihoming includes the **modification of a site exit-router**. In fact, end-site multihoming can be also assured by a network element like, for instance, an exit-router which performs packet rewriting for a given locator of a correspondent node. Nevertheless, this type of approach raises security concerns, which might be difficult to overcome. Redirection attacks are such an example, which may compromise routing, since packets for a destination can be redirected to any location. Thus, the host should always be able to perform the endpoint-to-locator mapping on its own [50].

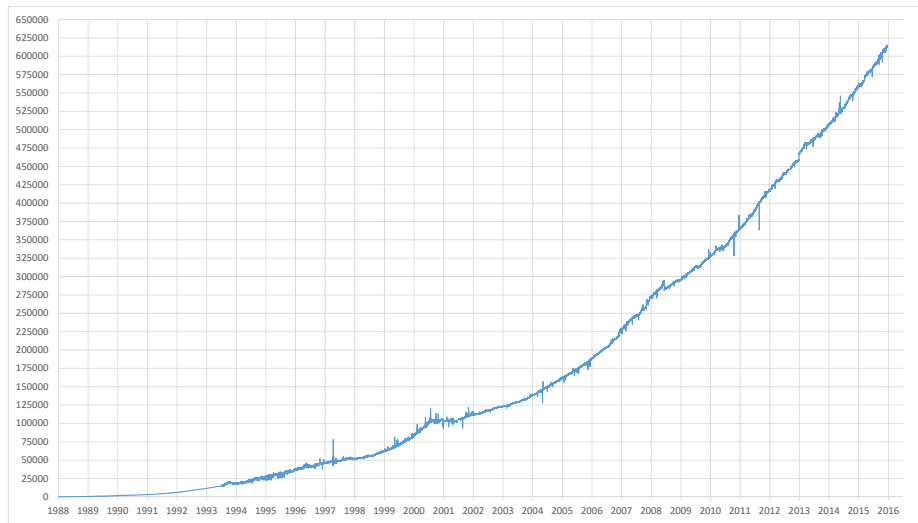


Figure 1.3: Active BGP entries (FIB)

1.1.3 Other related concepts

Multihoming is lately associated with other concepts, including multiaddressing, overlapping networks, multiple interfaces, overlay routing, scalability, security and policy.

Multiaddressing corresponds to a configuration where multiple addresses are assigned to a given host based on prefixes advertised in different connections.

Overlapping networks correspond to networks that are configured in a way that there is a common area of coverage. Typically, mobile and wireless end-nodes connecting to these (overlapping) networks must have **multiple interfaces**, each one specific to the technology sustaining the respective network.

Overlay routing is associated with inter-domain routing techniques that improve fault-tolerance and is only applied in an end-site context.

Scalability is of essence in any network architecture and multihoming is not an exception: multihoming architectures should be scalable and need to strive to minimize the impact on routers and end hosts. Basic connectivity must be always provided and if any modification is required

it should be in the form of logically separating added functions from existing ones.

Security is also paramount for future architectures. Multihoming proposals should not introduce new security threats. For instance, multihoming solutions should be resilient to redirection attacks that compromise routing, new packet injection attacks (malicious senders can inject bogus packets into the packet stream between two communicating peers) and flooding attacks, which are normally associated with DoS attacks [50].

Policy is the set of rules which define in some circumstances how it is possible to route traffic of a particular type (e.g. protocol) via particular transit providers. This can be done if the devices in the site—which source or sink that traffic—can be isolated to a set of addresses to which a special export policy can be applied [3].

1.2 Consequences of Mobility

1.2.1 Networks topology

As said, driven by the growing ubiquity of the Internet and a plethora of mobile devices with communication capabilities, distributed systems and complex applications are now the normality. The networks in which these applications must operate are inherently dynamic; typically large and completely decentralized, so that each node can have an accurate view of only its local vicinity. Such networks change over time, as nodes join, leave, and move around, and as communication links appear and disappear.

In some networks (e.g. peer-to-peer), nodes participate only for a short period of time and the topology can change at a high rate. In wireless ad hoc networks, nodes are mobile and move around unpredictably [30]. Much work has gone into developing algorithms to work in networks that eventually stabilize and stop changing, but there is not a suitable ground for reasoning about truly dynamic networks yet.

1.2.2 Security issues

The general lesson on network protocol security is that plaintext secrets (verification tags, secret sequence numbers, nonces, etc.) that can be used in more than one message are more vulnerable with multihomed or mobile endpoints than in a static setting. They may still be acceptable as security mechanisms if the rate of change is slow and connection lifetimes limited, but actually this is not always so.

An assumption made in the standard transport-layer protocol (e.g. in SCTP) is that the transport addresses belong to the association endpoints until the end of the association lifetime. The danger of this assumption is that if the endpoint loses control of an address and the address is subsequently allocated to another node, the peer will continue to send packets to the lost address. The new owner of the address will receive the packets and learn all plaintext secrets in them. If the new owner of the address is malicious, it may use this information for spoofing attacks, but the risk of attacks caused by this vulnerability is small. Multihomed endpoints usually do their best to ensure that they retain all their addresses throughout the association lifetime, for example, by using only statically configured IP addresses.

Nevertheless, in the mobility-protocol design it should be considered what happens to the old addresses of the mobile node, and the possibility that an attacker gains control of one.

In a general-purpose multihoming protocol, furthermore, there is also the possibility that the endpoints sometimes make mistakes about their address sets and even purposely misrepresent them. Also it is implicit that it is not possible to expect there to be any application-level verification or recovery mechanism that would mitigate the consequences of such false information. Thus, protocol engineers need to worry about attacks where the endpoints lie about their addresses, or more in general, in any multihoming or mobility protocol, must be considered the possibility of an endpoint making false claims about its addresses.

But not only. An address conflict in a telephony signaling protocol is probably an operator error and it is best to report the error and let the

operator resolve it. The situation changes when a general-purpose multihoming protocol is considered. There may not be any operator that could help, and thus it is essential to either avoid the conflicts or to resolve them automatically at the layer in which the conflict happens [7].

1.3 Multihoming solutions

An ideal multihoming solution which provides seamless mobility should meet the following requirements:

End-user transparency: the roaming should be completed as quickly as possible; the user should not notice any communication or service interruption, or if that is not possible, the interruptions should be reduced in duration as well as in frequency.

Network transparency: applications should use only FQDNs for communications; protocols should handle multihoming without affecting the upper layers.

Legacy compatibility: the solution should be fully compatible with the legacy infrastructure in terms of entities and protocols.

Quality of Service (QoS): node mobility should be managed in accord with the defined QoS.

Full mobility: nodes should not be aware of their mobility.

NAT-friendly: the solution should be compatible with possible Network Address Translation (NAT) policies.

As a consequence of its implementation complexity, multihoming is not supported by any widespread user application yet, nor by standard network devices. However, some working multihoming protocol implementations already are in use on enterprise network routers.

An example is Cisco IOS¹ [14] since SCTP is supported in order to specifically enhance reliability through support of multihoming at either end or both ends of the network association. It is not explicitly configured on routers by default, but it underlies several Cisco applications. Also LISP is supported

¹Cisco IOS Release 12.2(2)MB, 12.2(4)T, or later releases.

by Cisco IOS with outstanding performances [13] but the implementation is not opensource.

Praveen *et al.* have developed a Linux-based multihoming solution that consists of an outgoing load balancer and an incoming load balancer, particularly suitable for small and medium enterprises [44]. The outgoing load balancer does policy-based routing to choose the best link for each type of traffic. It calculates the characteristics of the paths via each of the ISPs to the most frequently accessed destinations. The best link is chosen based on the path characteristic that is relevant to the application protocol of the packet, as determined by the user-defined policy. For infrequently visited destinations, a global policy that considers the first hop connectivity to the ISPs is used. The incoming load balancer distributes the load on the servers within the network among the links to the ISPs in the ratio of available bandwidths on these links by modifying the DNS entries appropriately. Finally, both the incoming and outgoing load balancers do dead gateway detection to minimize downtime.

Other solutions have been developed and analyzed but with controversial outcomes. For example, Borsari presented a work about VoIP multihoming support on Symbian [10] where the reusability-oriented code modifications were in unavoidable conflict with the performance and the energy efficiency aspects, which are gravely predominant on mobile devices.

Chapter 2

Network Protocols for Multihoming

When a mobile Internet host changes its location and its point of access to the Internet changes, its IP address typically changes. The aim of mobility protocols is to solve the following two problems: to enable continuous communication over address changes, and to provide a reachability mechanism whenever the mobile is connected to the Internet. Mobility solutions exist for all major protocol layers. Link-layer mobility protocols avoid IP address changes. Network-layer protocols (e.g. Mobile IP (MIP)) hide them from the layers above. Transport-layer mobility protocols maintain a continuous connection between two endpoints over address changes. Higher, session and application-layer solutions re-establish transport-layer connections after an address change.

A multihomed Internet host usually has multiple IP addresses. While the goal of mobility protocols is to enable communication for moving hosts, the aim of multihoming is typically to increase reliability in a static setting: when one address fails, communication is switched to another one. However, despite their different goals, mobility and multihoming can be seen as two flavors of the same phenomenon, *dynamic multi-addressing*, that is the property of a multihomed or mobile endpoint of having a set of IP addresses that changes dynamically [7].

In the following sections, the most important multihoming protocols are synthetically introduced.

2.1 Site Multihoming by IPv6 Intermediation

2.1.1 Overview and Goals

The Site Multihoming by IPv6 Intermediation (SHIM6) protocol is a site-multihoming solution in the sense that it allows existing communication to continue when a site that has multiple connections to the Internet experiences an outage on a subset of these connections or further upstream. However, SHIM6 processing is performed in individual hosts rather than through site-wide mechanisms.

The goals of SHIM6 are:

- to preserve established communications in the presence of certain classes of failures, for example, Transmission Control Protocol (TCP) connections and User Datagram Protocol (UDP) streams;
- to have minimal impact on upper-layer protocols in general and on transport protocols and applications in particular;
- to address the security threats in IP version 6 (IPv6) through a combination of the Hash-Based Address (HBA)/Cryptographic Generated Address (CGA) approach;
- to not require an extra roundtrip up front to set up shim-specific state. Instead, allow the upper-layer traffic (e.g. TCP) to flow as normal and defer the set up of the shim state until some number of packets have been exchanged;
- to take advantage of multiple locators/addresses for load spreading so that different sets of communication to a host (e.g. different connections) might use different locators of the host.¹

The problem to solve is the end-site multihoming, with the ability to have the set of site addresses change over time due to site renumbering.

¹Notice that this might cause load to be spread unevenly; thus, the term “load spreading” is preferred instead of “load balancing”.

Further, it is assumed that such changes to the set of locator addresses can be relatively slow and managed: slow enough to allow updates to the Domain Name System (DNS) to propagate.

The SHIM6 proposal does not fully separate the identifier and locator functions that have traditionally been overloaded in the IP address. However, the term “identifier” or, more specifically, Upper-Layer Identifier (ULID), refers to the identifying function of an IPv6 address. “Locator” refers to the network-layer routing and forwarding properties of an IPv6 address.

2.1.2 Protocol insights

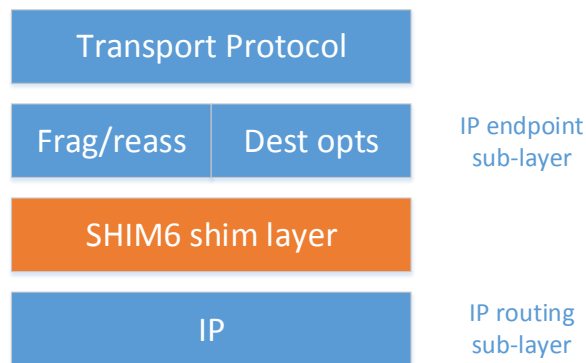


Figure 2.1: SHIM6 Protocol Stack

The proposal uses a multihoming shim layer within the IP layer, as shown in Figure 2.1, in order to provide Upper-Layer Protocol (ULP) independence. The multihoming shim layer behaves as if it is associated with an extension header, which would be placed after any routing-related headers in the packet (such as any hop-by-hop options). However, when the locator pair is the ULID pair, there is no data that needs to be carried in an extension header; thus, none is needed in that case.

Layering the fragmentation header above the multihoming shim makes reassembly robust in the case that there is broken multi-path routing that results in using different paths, hence potentially different source locators, for different fragments. Thus, the multihoming shim layer is placed between

the IP endpoint sub-layer, which handles fragmentation and reassembly, and the IP routing sub-layer, which selects the next hop and interface to use for sending out packets.

Applications and upper-layer protocols use ULIDs that the SHIM6 layer maps to/from different locators. The SHIM6 layer maintains state, called “ULID-pair context”, per ULID pair in order to perform this mapping. The mapping is performed consistently at the sender and the receiver so that ULPs see packets that appear to be sent using ULIDs from end to end. This property is maintained even though the packets travel through the network containing locators in the IP address fields, and even though those locators may be changed by the transmitting SHIM6 layer.

The context state is maintained per remote ULID (approximately per peer host) and not at any finer granularity. In particular, the context state is independent of the ULPs and any ULP connections. However, the forking capability enables SHIM6-aware ULPs to use more than one locator pair at a time for a single ULID pair.

The result of this consistent mapping is that there is no impact on the ULPs, and in particular, there is no impact on pseudo-header checksums and connection identification.

Conceptually, one could view this approach as if both ULIDs and locators are present in every packet, and as if a header-compression mechanism is applied that removes the need for the ULIDs to be carried in the packets once the compression state has been established. In order for the receiver to re-create a packet with the correct ULIDs, there is a need to include some “compression tag” in the data packets. This serves to indicate the correct context to use for decompression when the locator pair in the packet is insufficient to uniquely identify the context.

There are different types of interactions between the SHIM6 layer and other protocols. Those interactions are influenced by the usage of the addresses in these other protocols and the impact of the SHIM6 mapping on these usages. A detailed analysis of the interactions of different protocols, including SCTP, MIP, and Host Identity Protocol (HIP), can be found in [2]. Moreover, some applications may need to have a richer interaction with the SHIM6 sublayer.

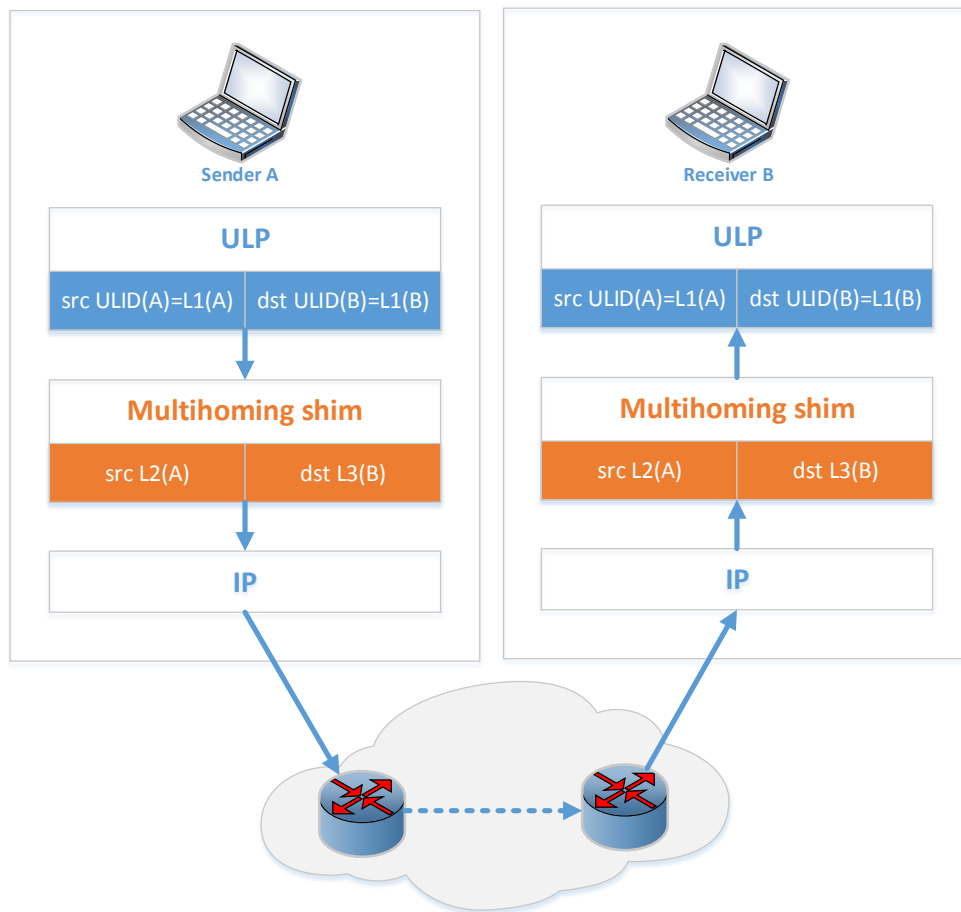


Figure 2.2: SHIM6 Mapping with Changed Locators

In order to enable that, an Application Programming Interface (API) has been defined to enable greater control and information exchange for those applications that need it [40].

2.2 Host Identity Protocol

The Host Identity Protocol (HIP) allows consenting hosts to securely establish and maintain shared IP-layer state, allowing separation of the identifier and locator roles of IP addresses, thereby enabling continuity of communications across IP address changes. HIP is based on a Sigma-compliant Diffie-

Hellman key exchange, using public key identifiers from a new Host Identity namespace for mutual peer authentication. The protocol is designed to be resistant to Denial-of-Service (DoS) and Man-in-the-Middle (MitM) attacks. When used together with another suitable security protocol, such as the Encapsulated Security Payload (ESP), it provides integrity protection and optional encryption for upper-layer protocols, such as TCP and UDP [37].

2.2.1 HIP Architecture

The Internet has two important global namespaces: IP addresses and DNS names. These two namespaces have a set of features and abstractions that have powered the Internet to what it is today. They also have a number of weaknesses. Moreover, semantic overloading and functionality extensions have greatly complicated these namespaces.

The Host Identity namespace fills an important gap between the IP and DNS namespaces: it consists of Host Identifiers. A Host Identifier (HI) is cryptographic in its nature—it is the public key of an asymmetric key-pair. Each host will have at least one Host Identity, but it will typically have more than one. Each Host Identity uniquely identifies a single host (e.g. two hosts cannot have the same Host Identity). The Host Identity, and the corresponding Host Identifier, can be either public (e.g. published in the DNS) or unpublished. Client systems will tend to have both public and unpublished Identities.

There is a subtle but important difference between Host Identities and Host Identifiers. An Identity refers to the abstract entity that is identified. An Identifier, on the other hand, refers to the concrete bit pattern that is used in the identification process.

When HIP is used, the actual payload traffic between two HIP hosts is typically, but not necessarily, protected with IP Security (IPSec). The Host Identities are used to create the needed IPSec Security Associations and to authenticate the hosts. When IPSec is used, the actual payload IP packets do not differ in any way from standard IPSec-protected IP packets [36].

There are two main representations of the Host Identity, the full HI and

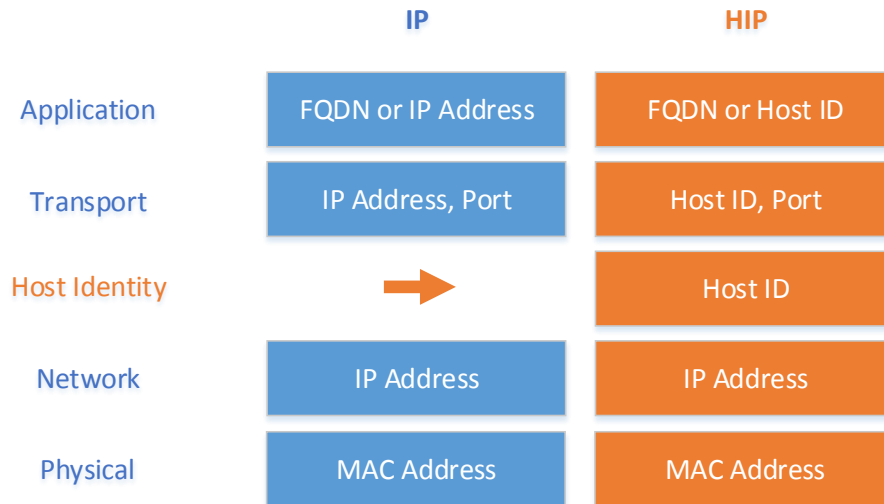


Figure 2.3: HIP Architecture

the Host Identity Tag (HIT). As said, the HI is a public key and directly represents the Identity. Since there are different public key algorithms that can be used with different key lengths, the HI is not good for use as a packet identifier, or as an index into the various operational tables needed to support HIP. Consequently, a hash of the HI, the HIT, becomes the operational representation. It is 128b long and is used in the HIP payloads and to index the corresponding state in the end hosts. The HIT has an important security property in that it is self-certifying [37].

2.2.2 HIP Base Exchange

The HIP base exchange is a two-party cryptographic protocol used to establish communications context between hosts. The base exchange is a Sigma-compliant four-packet exchange. The first party is called the Initiator and the second party the Responder. The four-packet design helps to make HIP DoS resilient. The protocol exchanges Diffie-Hellman keys in the 2nd and 3rd packets, and authenticates the parties in the 3rd and 4th packets. Additionally, the Responder starts a puzzle exchange in the 2nd packet, with

the Initiator completing it in the 3rd packet before the Responder stores any state from the exchange.

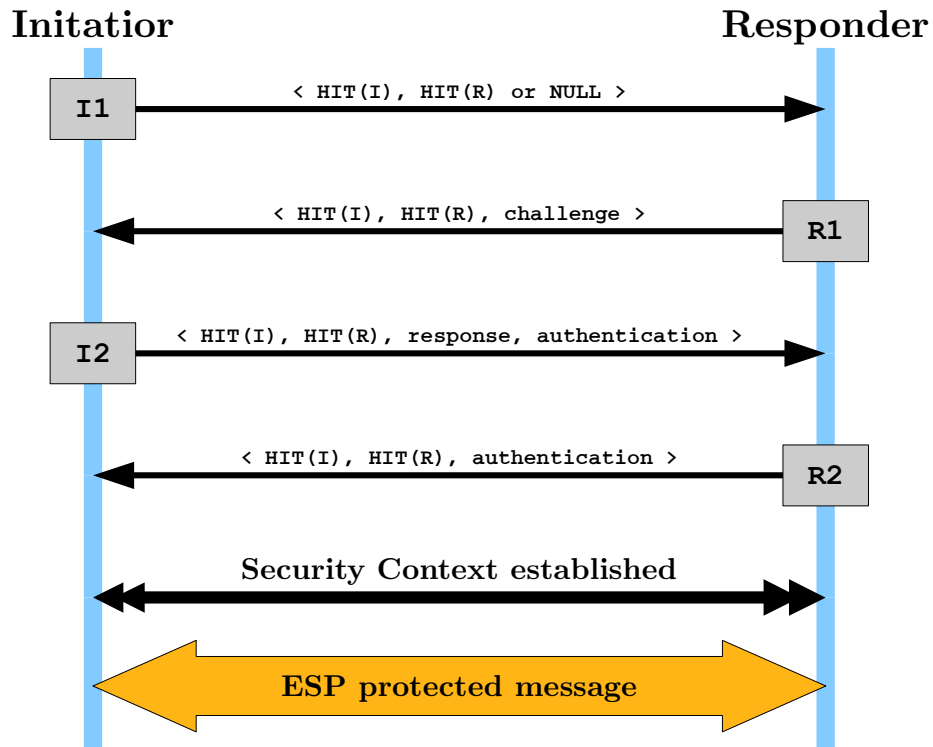


Figure 2.4: HIP four-way handshaking

The exchange can use the Diffie-Hellman output to encrypt the Host Identity of the Initiator in the 3rd packet or the Host Identity may instead be sent unencrypted. The Responder's Host Identity is not protected. It should be noted, however, that both the Initiator's and the Responder's HITs are transported as such (in cleartext) in the packets, allowing an eavesdropper with a priori knowledge about the parties to verify their identities. Data packets start to flow after the 4th packet. The 3rd and 4th HIP packets may carry a data payload in the future. However, the details of this are to be defined later as more implementation experience is gained.

Finally, HIP is designed as an end-to-end authentication and key establishment protocol, to be used with ESP and other end-to-end security protocols,

but the base protocol does not cover all the fine-grained policy control found in Internet Key Exchange (IKE) (that, for example, allows IKE to support complex gateway policies). Thus, HIP is not a replacement for IKE [37].

2.2.3 End-host Mobility and Multihoming

Architecturally, HIP provides for a different binding of transport-layer protocols. That is, the transport-layer associations (e.g. TCP connections and UDP associations) are no longer bound to IP addresses but to Host Identities.

It is possible that a single physical computer hosts several logical end-points. With HIP, each of these end-points would have a distinct Host Identity. Furthermore, since the transport associations are bound to Host Identities, HIP provides for process migration and clustered servers. That is, if a Host Identity is moved from one physical computer to another, it is also possible to simultaneously move all the transport associations without breaking them. Similarly, if it is possible to distribute the processing of a single Host Identity over several physical computers, HIP provides for cluster-based services without any changes at the client end-point.

As said, HIP decouples the transport from the internetworking layer, and binds the transport associations to the Host Identities. Consequently, HIP can provide for a degree of internetworking mobility and multihoming at a low infrastructure cost. HIP mobility includes IP address changes (via any method) to either party. Thus, a system is considered mobile if its IP address can change dynamically for any reason like Point-to-Point Protocol (PPP), Dynamic Host Configuration Protocol (DHCP), IPv6 prefix reassignments, or a NAT device remapping its translation. Likewise, a system is considered multihomed if it has more than one globally routable IP address at the same time. HIP links IP addresses together, when multiple IP addresses correspond to the same Host Identity, and if one address becomes unusable, or a more preferred address becomes available, existing transport associations can easily be moved to another address.

When a node moves while communication is already ongoing, address changes are rather straightforward. The peer of the mobile node can just

accept a HIP or an integrity protected IPSec packet from any address and ignore the source address. However, a mobile node must send a HIP readdress packet to inform the peer of the new address(es), and the peer must verify that the mobile node is reachable through these addresses [36].

More lately, Pierrel *et al.* introduced a policy system for simultaneous multiaccess based on HIP called HIP SIMultaneous Multi-Access (SIMA) [43]. The proposal extends HIP by allowing flows to use different paths independently of each other, since HIP does not support load sharing. To enable flow distribution, flows are identified by source and destination ports and by the HIT. The RendezVous Server is also extended to be able to store flow policies. Whilst these policies define the usage rules of the available interfaces, the proposal does not detail the policy specification (e.g. rules actions, interface priority, and cost) [50].

2.3 Identifier/Locator Network Protocol

Since Identifier/Locator Network Protocol (ILNP) has been recommended by Internet Engineering Task Force (IETF) in [31] for future routing architecture, the protocol is presented here. Basically, IETF has found ILNP to be a clean solution since it separates location from identity in a clear, straightforward way that is consistent with the remainder of the Internet architecture: unlike the many map-and-encap proposals, there are no complications due to tunneling, indirection, or semantics that shift over the lifetime of a packet's delivery.

2.3.1 Architectural Overview

ILNP takes a different approach to naming of communication objects within the network stack. Two new data types are introduced which subsume the role of the IP Address at the network and transport layers in the current IP architecture. ILNP explicitly replaces the use of IP Addresses with two distinct name spaces, each having distinct and different semantics:

Identifier: a non-topological name for uniquely identifying a node.

Locator: a topologically bound name for an IP subnetwork.

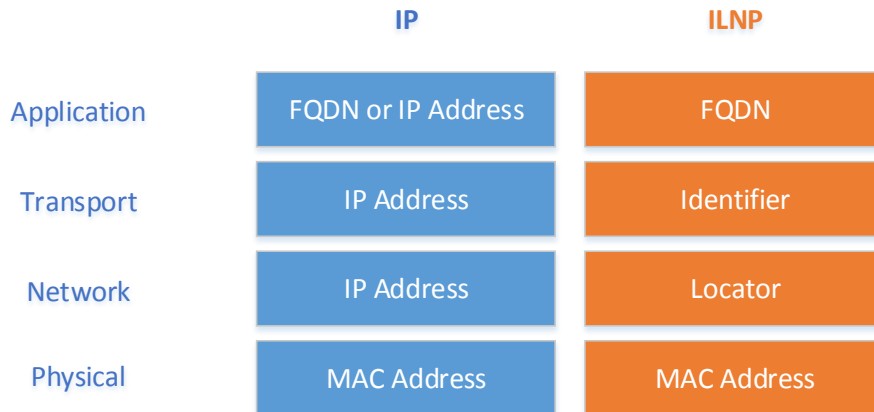


Figure 2.5: ILNP Architecture

The use of these two new namespaces in comparison to IP is shown in Figure 2.5. If an application uses a Fully Qualified Domain Name (FQDN) at the application-layer, rather than an IP Address or other lower-layer identifier, then the application perceives no architectural difference between IP and ILNP. In fact, ILNP does not require applications to be rewritten to use a new Networking API. So existing “well-behaved” IP-based applications should be able to work over ILNP as is.

In ILNP, transport-layer protocols use only an end-to-end, non-topological node Identifier in any transport-layer session state. It is important to note that the node Identifier names the node, not a specific interface of the node. In this way, it has different semantics and properties than either the IP Address or the IP interface identifier. Anyway, the use of the ILNP Identifier value within application-layer protocols is not recommended. Instead, the use of either a FQDN or some different topology-independent namespace is recommended.

At the network-layer, Locator values, which have topological significance, are used for routing and forwarding of ILNP packets, but Locators are not used in upper-layer protocols.

As well as the new namespaces, another significant difference in ILNP is that there is no binding of a routable name to an interface, or Sub-Network Point of Attachment (SNPA), as there is in IP. The existence of such a binding in IP effectively binds transport protocol flows to a specific, single interface on a node. Also, applications that include IP Addresses in their application-layer session state effectively bind to a specific, single interface on a node. Dynamic bindings exist between Identifier values and associated Locator values, as well as between `{Identifier, Locator}` pairs and (physical or logical) interfaces on the node.

This change enhances the Internet Architecture by adding crisp and clear semantics for the Identifier and for the Locator, removing the overloaded semantics of the IP Address, by updating end-system protocols, but without requiring any router or backbone changes, excepted DNS. In ILNP, the closest approximation to an IP Address is an Identifier-Locator Vector (I-LV),² which is a given binding between an Identifier and Locator pair.

Where, today, IP packets have:

- Source IP Address, Destination IP Address

Instead, ILNP packets have:

- Source I-LV, Destination I-LV

Hence, with these naming enhancements, the Internet Architecture is improved by adding explicit harmonised support for many functions, such as multihoming, mobility, and IPSec.

Network-layer

Today, network-layer IP sessions have 2 components:

- Source IP Address (A_S)
- Destination IP Address (A_D)

Instead, network-layer ILNP sessions have 4 components:

- Source Locator(s) (L_S)
- Source Identifier(s) (I_S)
- Destination Locator(s) (L_D)

²However, it must be emphasised that the I-LV and the IP Address are *not* equivalent.

- Destination Identifier(s) (L_S)

Incidentally the phrase “ILNP session” refers to an ILNP-based network-layer session, having the 4 components in the definition above. For engineering efficiency, multiple transport-layer sessions between a pair of ILNP correspondents normally share a single ILNP session.

Transport-layer

Today, transport-layer sessions using IP include these 5 components:

- Source IP Address (A_S)
- Destination IP Address (A_D)
- Transport-layer protocol (e.g. UDP, TCP, SCTP)
- Source transport-layer port number (P_S)
- Destination transport-layer port number (P_D)

Instead, transport-layer sessions using ILNP include these 5 components:

- Source Identifier (I_S)
- Destination Identifier (I_D)
- Transport-layer protocol (e.g. UDP, TCP, SCTP)
- Source transport-layer port number (P_S)
- Destination transport-layer port number (P_D)

IP Address and I-LV

Historically, an IP Address has been considered to be an atomic datum, even though it is recognised that an IP Address has an internal structure: the network prefix plus either the host ID (IP version 4 (IPv4)) or the interface identifier (IPv6). However, this internal structure has not been used in end-system protocols; instead, all the bits of the IP Address are used.³

While it is possible to say that an I-LV is an approximation to an IP Address of today, it should be understood that an I-LV is not an atomic

³Additionally, in IPv4, the IPv4 subnet mask uses bits from the host IDentity (ID), a further confusion of the structure, even though it is an extremely useful engineering mechanism.

datum, being a pairing of two data types, an Identifier and a Locator; and that it has different semantics and properties to an IP Address.

2.3.2 Node Identifier

Identifiers, also called Node Identifier (NID), are non-topological values that identify an ILNP node. A node might be a physical node or a virtual node. For example, a single physical device might contain multiple independent virtual nodes. Alternately, a single virtual device might be composed from multiple physical devices.

A node may have multiple Identifier values associated with it, which may be used concurrently. In normal operation, when a node is responding to a received ILNP packet that creates a new network-layer session, the correct NID value to use for that network-layer session with that correspondent node will be learned from the received ILNP packet. In normal operation, when a node is initiating communication with a correspondent node, the correct I value to use for that session with that correspondent node will be learned either through the application-layer naming, through DNS name resolution, or through some alternative name resolution system.

Once a NID value has been used to establish a transport-layer session, that Node Identifier value forms part of the end-to-end (invariant) transport-layer session state and so must remain fixed for the duration of that session. And this means, for example, that throughout the duration of a given TCP session, the Source NID and Destination NID values will not change.

In normal operation, a node will not change its set of valid Identifier values frequently. However, a node may change this set over time, for example, in an effort to provide identity obfuscation. When a node has more than one NID value concurrently, the node might have multiple concurrent ILNP sessions with some correspondent node, in which case NID values may differ between the different concurrent ILNP sessions.

2.3.3 Multihoming and Multi-Path Transport

For multihoming, there are two main cases and one sub-case to consider:

Host-Multihoming (H-MH): a single host is, individually, connected to multiple upstream links, via separate routing paths, and those multiple paths are used by that host as it wishes. That is, use of multiple upstream links is managed by the single host itself. For example, the host might have multiple valid Locator values on a single interface, with each Locator value being associated with a different upstream link (provider).

Multi-Path Transport (MPT): multiple paths are used to transfer data for an end-to-end session [33]. This can be considered a special case of H-MH.

Site-Multihoming (S-MH): a site network is connected to multiple upstream links via separate routing paths, and hosts on the site are not necessarily aware of the multiple upstream paths. That is, the multiple upstream paths are managed, typically, through a site border router, or via the providers.

Essentially, for ILNP, multihoming is implemented by enabling multiple Locator values to be used simultaneously by a node, and dynamic, simultaneous binding between one (or more) Identifier value(s) and multiple Locator values. Other details can be found in [6].

2.4 Locator/Identifier Separation Protocol

2.4.1 Overview

The Locator/Identifier Separation Protocol specifies an architecture for decoupling host identity from its location information in the current address scheme. This separation is achieved by replacing the addresses currently used in the Internet with two separate name spaces: Endpoint Identifier (EID), and Routing LOCator (RLOC). Host applications bind to host's EID, which is used as the address for transport connections; while RLOCs are IPv4 or

IPv6 addresses used for routing through transit networks. In order to reach a host, identified by its EID, one must first find the current location of the host. Separating the host identity (EID) from its locator (RLOC) enables seamless endpoint mobility by allowing the applications to bind to a permanent address, the host's EID. The location of the host can change many times during an ongoing connection. Each time, the Locator/Identifier Separation Protocol (LISP) tunnel routers will encapsulate the packets to the new RLOC, preserving the connection session from breaking [47].

2.4.2 LISP Mapping System

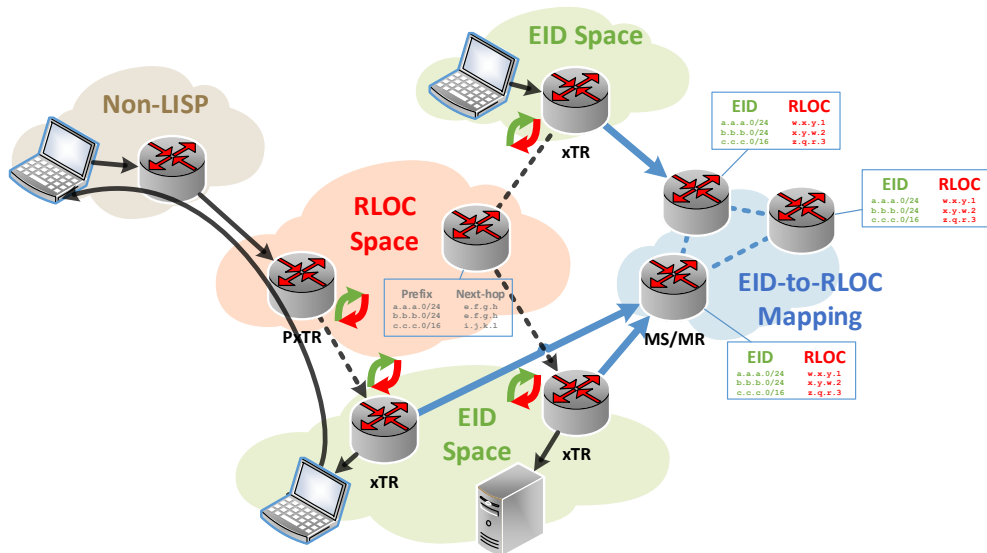


Figure 2.6: LISP Architecture

As shown in Figure 2.6, the LISP Mapping System is a central aspect of the LISP architecture. It is a publicly accessible service that publishes location information associated with EIDs (EID-to-RLOC mappings). Main elements of LISP Mapping System are Map Servers and Map Resolvers. EID-to-RLOC mappings are stored in Map Servers. Each Map Server is associated with a partition of the EID name space, and stores the location information

for those EID prefixes. Therefore, each LISP mobile node is associated with a specific Map Server where it registers its EID-to-RLOC mapping, and updates it according to its movement. In this context, Map Servers have assigned a set of EIDs and delegate them to either LISP tunnel routers or mobile nodes.

Map Resolvers are used as an interface to the mapping system for looking up EID location information; they have a similar functionality as DNS resolvers have in today's Internet: LISP mobile nodes send EID lookup requests (**Map Request**) to the mapping system through Map Resolvers [47].

2.4.3 LISP-MN

The lightweight tunnel router is the implementation of LISP Mobile Node (LISP-MN) on the endpoint or mobile node. Mobile node tunnel routers are used to encapsulate outgoing packets in a LISP header based on RLOCs before leaving the mobile node, and to remove the LISP header from incoming packets before sending them to upper layers ultimately reaching the destination application. The LISP-MN protocol is then best understood as the concatenation of three different phases [47]:

- Registering EID and obtaining an RLOC;
- Signaling EID-to-RLOC bindings and transmitting data-packets;
- Handover.

These phases are now explained in detail.

Registering EID and obtaining an RLOC

Each LISP-MN is configured with at least one EID. As said before, an EID is either a standard (/32) IPv4 or (/128) IPv6 address, identifies the node uniquely and remains static independently of its location. If the node has also a DNS entry, this entry returns the EID which is typically assigned by the Map Server provider.⁴

In order to connect to the Internet, the LISP-MN also needs at least an RLOC. RLOCs are obtained by traditional mechanisms such as DHCP or con-

⁴In LISPmob, analyzed in section 3.4.2, it is configured in a static file.

figured manually and are location dependent. This means that as the mobile node roams across providers, it will obtain a different RLOC in each location. For each new RLOC obtained by the LISP-MN, the node has to inform about the new EID-to-RLOC binding to its Map Server. In order to do so LISP defines the Map Register signaling message that includes the EID and the RLOC. The node may include multiple RLOCs if the node is multihomed and LISP supports any combination of IPv4 and IPv6 for EIDs and RLOCs. The LISP-MN and the Map Server share a pre-configured key⁵ which is used to sign the Map Register to ensure authentication. Once the Map Server receives a valid Map Register containing an EID-to-RLOC mapping it will make it accessible throughout the Mapping System [47].

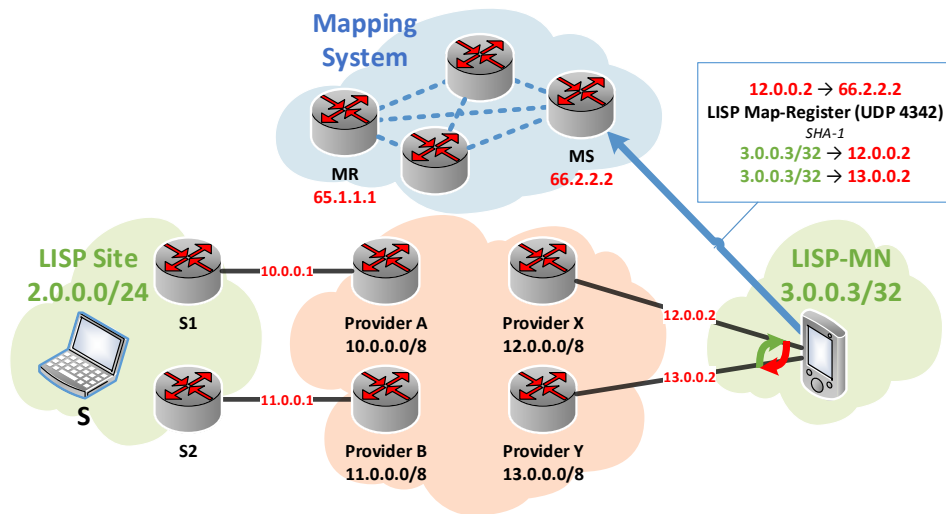


Figure 2.7: Registering an EID-to-RLOC binding

A LISP-MN is configured with the EID 3.0.0.3/32 and two RLOCs from two different providers X 12.0.0.2 and Y 13.0.0.2. The MN registers these two bindings 3.0.0.3/32 → 12.0.0.2 and 3.0.0.3/32 → 13.0.0.2 into its Map Server (identified with the address 66.2.2.2). The MN and a MS have a pre-configured key and this Map Register message is signed and hence, authenticated.

⁵Again, in LISPmob it is configured in a static file.

Signaling EID-to-RLOC bindings and transmitting data-packets

The static node first retrieves the EID of the mobile node by querying the DNS and then transmits a packet addressed to this EID just as in the plain Internet. The packet is routed until it reaches the LISP tunnel router. Upon reception, the tunnel router checks whether it knows the EID-to-RLOC binding or not. For this purpose each LISP node includes a data-structure called Map-cache which stores such information.

If the tunnel router's Map-cache does not contain the binding for the destination EID, it will trigger a **Map-request** message. This message is used to query the Mapping System for a particular binding: it is sent to the Map Resolver, which is typically co-located with the Map Server, and in turn, the Map Resolver forwards the **Map-request** message through the Mapping System that routes it, according to the destination EID, until it reaches the Map Server that provides mapping services to the Mobile Node (MN).

The Map Server then constructs a reply for the **Map-request** using another LISP defined message, the **Map-reply**. This message mainly contains the EID of the MN, the set of RLOCs that provide connectivity to the MN, and the priorities and weights of each locator which are used for ingress load-balancing. Finally the **Map-reply** also contains a Time-To-Live (TTL) that defines the amount of time for which this particular mapping is valid, and a nonce to avoid unsolicited replies. The **Map-reply** is sent directly to the tunnel router that will install this binding in its Map-cache and will use it to encapsulate packets towards the MN until the TTL expires. At this point it will request a fresh binding.

Typically, as in TCP, the node will reply with another data-packet addressed to the EID of the static node. Since the Map-cache of the MN does not have the binding for such EID, it will trigger a **Map-request** querying for the RLOCs of the LISP site. This **Map-request**, as in the previous case, will be routed through the Mapping System towards the Map Server servicing the site that in turn replies with a **Map-reply**. The mobile node, upon reception of this message, will install this binding in its Map-cache and will start to encapsulate data packets directly to the locators of the LISP site according

to whatever policies have been defined (*weight* and *priorities*). Such data-packets are decapsulated at the tunnel router and forwarded, as in the plain Internet, to the final destination [47].

Handover

When the MN changes its point of attachment, it will regain connectivity in a new subnetwork, possibly serviced by a new provider. In this case it will first obtain a new RLOC and, as described before, will send the new EID-to-RLOC binding on its Map Server.

In order to resume existing connections, the MN has to update all the EID-to-RLOC bindings stored in the Map-cache of the routers with which it is communicating. To do so the MN will send a special signaling message called Solicit-Map-Request (SMR) to all these routers/mobile nodes. Upon reception of such message, the peering router/mobile node will trigger a `Map-request` addressed towards the EID of the soliciting MN. This message will be in turn forwarded by the Mapping System until it reaches the Map Server servicing the MN that will reply with a `Map-reply` containing the updated RLOCs.

2.4.4 LISP and legacy internetworking

Communications between LISP nodes and legacy non-LISP enabled nodes are fasten by ad hoc LISP internetworking components called Proxy Tunnel Router (PxTR). A PxTR attracts traffic by announcing, by means of Border Gateway Protocol (BGP), EID addresses. Then it queries the Mapping System to obtain the corresponding RLOC bindings and encapsulate the data-packets towards them. Additionally, PxTRs also decapsulate packets sent by LISP-enabled sites and nodes towards the non-LISP Internet. This is done to avoid ingress filtering issues [47].

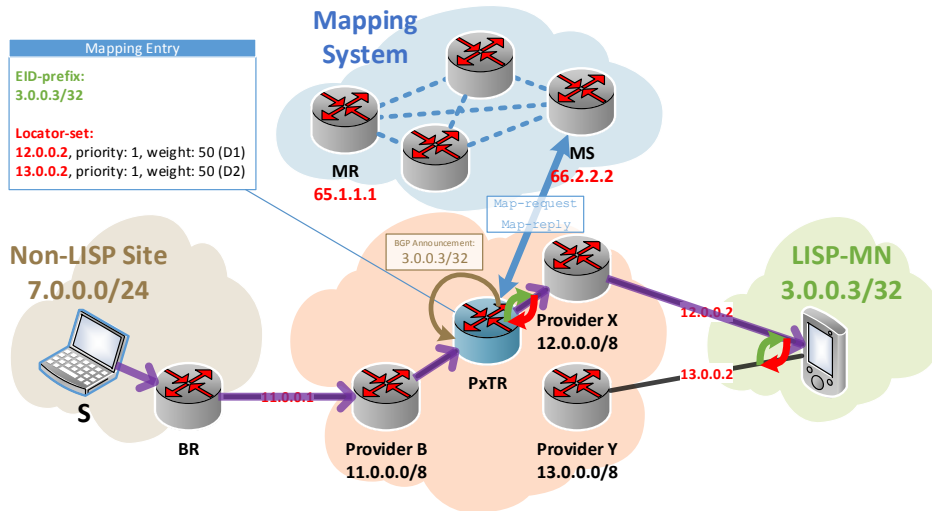


Figure 2.8: Internetworking with non-LISP sites

The PxTR is announcing at the BGP DFZ an aggregated EID prefix that covers the one configured at the MN. By means of this BGP announcement, PxTR attracts traffic addressed to the EID of the MN (3.0.0.3/32) and, upon reception of a data-packet, it queries the Mapping System to obtain the locator set (12.0.0.2 and 13.0.0.2) and proceeds to encapsulate packets. In turn, the PxTR is also used to decapsulate data-packets addressed to non-LISP sites.

2.5 Stream Control Transport Protocol

The Stream Control Transport Protocol (SCTP) is a reliable message-based transport protocol developed by the IETF that could replace TCP in some applications. SCTP allows endpoints to have multiple IP addresses for the purposes of fault tolerance and there is on-going work to extend the SCTP multihoming functions to support dynamic addressing and endpoint mobility [7].

2.5.1 SCTP Overview

The SCTP is a standard transport-layer protocol for the IPv4 and IPv6 Internet. SCTP was originally intended for the transport of Public Switched

Telephone Network (PSTN) telephony signaling messages over IP but it is now specified as a general-purpose alternative to TCP and UDP.

An SCTP association is a relationship between two SCTP endpoints. An endpoint is a set of transport addresses and a transport address consists of a network-layer address and a port number. In SCTP, all transport addresses of an endpoint must share the same port number. Thus, in practice, an SCTP endpoint is identified with a non-empty set of IP addresses and a single port number. A pair of transport addresses is called *path*. Each transport address can belong to only one endpoint at a time, so, this means that no special endpoint identifiers are needed. The receiver of an SCTP packet identifies the source and destination endpoints and the association to which the packet belongs based on the source and destination IP addresses and port numbers.

An SCTP packet comprises a common header and zero or more *chunks*. The chunks may carry either SCTP signaling information or user data (DATA chunk). Multiple chunks, such as user data and acknowledgements, may be bundled into one packet. Also, SCTP provides ordered and reliable multi-stream transport [7].

2.5.2 Protocol insights

SCTP Handshake

First, endpoints exchange random 32b nonces.⁶ The header of all but the first packet from Endpoint A to B must include B's nonce. Correspondingly, B must include A's nonce in the header of all packet that it sends to A. The SCTP specification calls the nonces "verification tags" (denoted by `Tag_A` and `Tag_B` in Figure 2.9). These verification tags serve the same security purpose as the randomly initialized sequence numbers in TCP, that is, they provide a level of security against packet spoofing.

During the SCTP handshake, each of the two endpoints may send to the other a list of IP addresses (in the `INIT` and `INIT ACK` chunks). Each endpoint

⁶Here and after I use the Internet Engineering Task Force (IEEE) 1541-2002 standard to denote bit as b and bytes as B.

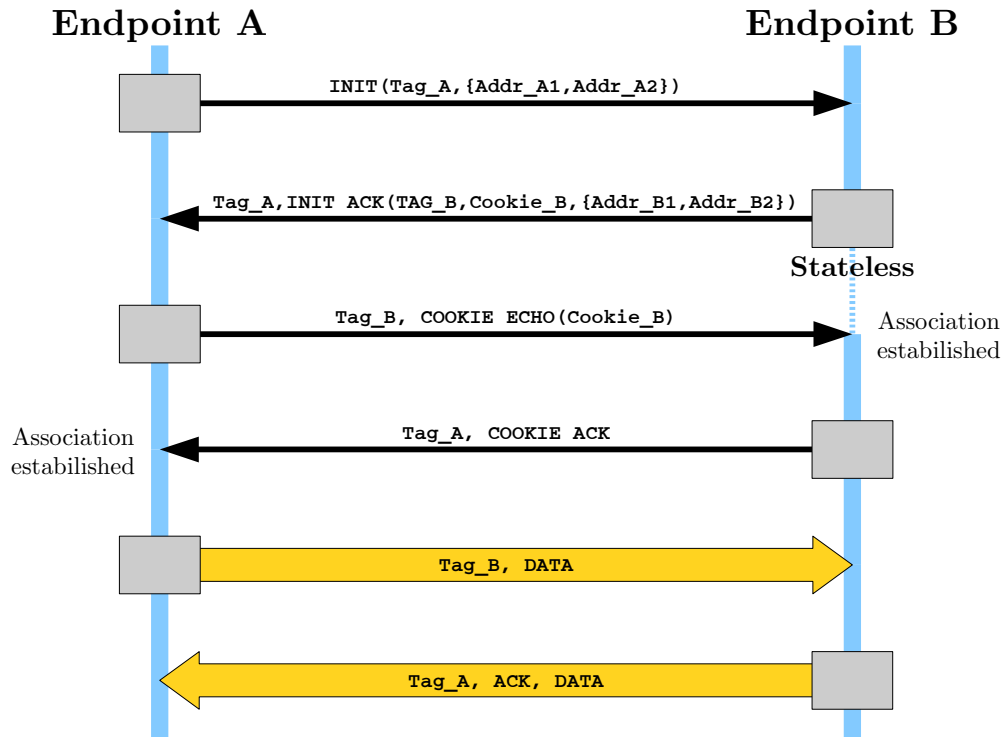


Figure 2.9: Sctp four-way handshaking

selects one of the peer's addresses as the primary destination address, and one of its own addresses as the best source address for routing packets to the destination. If the choice is not mandated by the upper-layer protocol, the algorithm for choosing the destination address is implementation dependent. The typical choice is either the source address of the first received packet or the first address in the peer's list.

An important feature of the Sctp handshake is that the respondent (Endpoint B) remains stateless between sending the 2nd and receiving the 3rd message. The respondent encodes the protocol state, including the contents of the INIT, into a state cookie, which it sends to the initiator (Endpoint A) in the INIT ACK. The initiator returns the cookie to the respondent in the COOKIE ECHO. This prevents state-exhaustion attacks similar to the TCP SYN flooding [7].

Acknowledgements and Abort messages

The acknowledgements contain a cumulative sequence number for the received data in which no gaps remain. Like TCP, an SCTP endpoint maintains congestion windows that limit the amount of unacknowledged data that may be in flight at a time. There is a separate window for each transport address of the peer endpoint. The window size is calculated with a TCP-like algorithm that includes slow-start and congestion-avoidance phases, and is limited by the receiver's advertised buffer space.

The SCTP specification defines the **ABORT** chunk for closing an association in an error situation. For example, an endpoint sends an **ABORT** when it receives an out-of-the-blue packet (e.g. one that does not match any existing association). This causes the receiver of the **ABORT** to delete its association state. The abort mechanism is, of course, used only by nodes that support the SCTP protocol. When a non-SCTP node receives an out-of-the-blue SCTP packet, it either sends an Internet Control Message Protocol (ICMP) error message or it silently discards the packet. The possible ICMP messages are “**Destination unreachable**” and “**Unknown next header type**” (the latter in IPv6) [7].

2.5.3 Multihoming and Dynamic Address Reconfiguration

Standard SCTP supports multihoming with a static set of addresses. Each endpoint sends all packets from the chosen source address to the primary destination address. The other addresses are used only if the primary path fails (e.g. if the primary destination address becomes unreachable). The policy for selecting the new address pair in failover is implementation dependent.

Each endpoint monitors the reachability of the secondary addresses of its peer so that it always knows which addresses are available for the failover. The monitoring is done by sending a heartbeat request (a **HEARTBEAT** chunk) to an idle destination address, which the peer acknowledges. The default frequency for the heartbeats is every 30 s. The implementation may start

sending heartbeat requests immediately after the association has been established but it is not required to do so.

Moreover, a proposed SCTP extension, **Dynamic Address Reconfiguration**, enables dynamic multi-addressing. This proposal defines a new SCTP chunk type `ASCONF`. An endpoint uses the `ASCONF` chunk to notify its peer about changes to its address set. The chunk contains one or more instructions for adding and deleting addresses and for setting the primary address. The recipient executes these instructions in the order in which they appear in the chunk [7]. Further details can be found in [51].

2.6 Considerations about multihoming protocols

Multiaccess and multihoming are yet to become prevalent in network deployments despite years of research and development in the area. Indeed, the corresponding support is often missing from state-of-the-art protocols. For example, modern mobility management protocols, such as Mobile IPv6 (MIPv6) are not capable of handling multihoming natively and must be combined with other protocols, such as SHIM6, to enable enhanced multihoming support. Furthermore, in some proposals new layers are introduced to perform specific functionalities and aim at reducing the ensuing complexity due to multihoming mechanisms in the original protocol stack [50].

Moreover, multihoming and mobility affect the security of protocols in several ways. First, existing security mechanisms are often based on implicit assumptions of a static network topology and unchanging addresses. When the assumptions are invalidated, the existing security mechanisms may become ineffective. Second, it is possible to misuse mobility signaling. Potential attacks include DoS by preventing legitimate communication, connection hijacking, spoofing and intercepting data, and redirecting packet flows to the target of a flooding attack [7].

Even if the multihoming literature is relatively rich of solutions and proposals [3, 37, 40, 6], also due to the large set of different possibilities and the

Protocol	MH	R	U	L	F	S	P	Sec
NEMO	End-host	✗	✓	✗	✗	✗	✗	✓
SHIM6	End-host	✓	✗	✗	✗	✓	✗	✗
HIP	End-host	✓	✓	✗	✗	✓	✗	✓
ILNP	End-site	✓	✓	✓	✓	✗	✗	✗
LISP	End-site	✓	✓	✓	✗	✓	✓	✗
SCTP	End-host	✓	✓	✓	✓	✗	✓	✓

Table 2.1: Primary Features of most Widespread Multihoming Protocols.

R: Resilience, **U:** Ubiquity, **L:** Load balancing/sharing, **F:** Flow distribution, **S:** Scalability, **P:** Policy, **Sec:** Security, **T:** Transport layer, **N:** Network layer

Protocol	Pros	Cons
NEMO	Mobility, Handover latency	Requires changes to hosts
SHIM6	Deployment	Mobility, Security
HIP	Compatibility, Security	Deployment, Policy
ILNP	Semantics, H-MH support	Requires changes to DNS
LISP	Scalability, Flexibility	Encapsulation overhead
SCTP	Security, Flow control	Requires changes to hosts

Table 2.2: Multihoming Protocols Pros and Cons

complexity of efficient implementation of the above features, multihoming is supported neither by any widespread user application yet, nor by standard network devices. Table 2.1 concisely reports some main features of the considered protocols, while Table 2.2 their pros and cons.

It is well recognized that protocols like SHIM6 or HIP may provide articulated multihoming features along with relative easy implementation, failure detection and recovery, or ubiquity and security support. However, none of them can be considered as the ideal solution [38]. In fact, while SHIM6 may break the functionality of other protocols [50], HIP implementation complexity is the reason of its recent de facto cooldown of the researcher community in the field.

Another relevant solution in the literature is the ILNP. Even if it has shown some significant pros, ILNP has demonstrated non negligible disadvantages in terms of *a*) required changes to standard DNS and the associated deployed equipment; and, even more relevant, *b*) the “philosophy” of disruptive approach behind the protocol itself, calling for significant re-deployment and non-back-compatible evolution of the existing installation base. Nevertheless, as the history of IPv6 teaches, Internet infrastructure does not evolve as quickly as it should. So, ILNP risks to rest a wonderful concept but with no practical usage.

The practical experience of experimentation of the SCTP shows that it would represent a feasible multihoming solution by itself [15, 28], but with some non-negligible restrictions. Its pros are many and valuable, and its independence to the network layer is crucially important. But it has to be supported and enabled with ad hoc modules running at hosts’ OS kernel layer. So far, for Linux users it is possible since kernel version 2.4 as well as in Mac OSX 10.7 and in Solaris 10, but Microsoft seems to have not plans to add native SCTP support because of the lack of customer demand, and Android supports it only in rooted mode (activating the `lksctp` library in the Linux kernel) and it seems not to be in Google developers’ plans too [4].

Nowadays LISP appears as the best and most promising tradeoff solution available. Sponsored by Cisco Systems, LISP is born to solve a number of problems risen up after more than a decade of relatively unchanged Internet infrastructure and very slow wearying transition to IPv6, which in fact is still ongoing. LISP has demonstrated to be the most efficient solution in terms of scalability [54]—a reason that alone could justify its future usage. Moreover, LISP does not require any substantial change to the existing legacy infrastructure, except for the entities which specifically support it, even if for both outgoing and incoming packets a processing latency is added at the edge of the network [19]. Hence, considering that the most significant LISP performance limitations relate to network devices’ hardware technology, I claim that LISP can play the role of the most widespread multihoming solution in next generation networks, possibly (but not necessarily) in association with SCTP.

Chapter 3

Android and Networking

3.1 Android Market Share and Evolution

Google acquired Android Inc. on August 17, 2005. Not much was known about Android Inc. at the time, but many assumed that Google was planning to enter the mobile phone market with that move.

When in 2007 Symbian OS obtained 60.1 % of the world-wide market share while its best competitor—Blackberry—only 10.5 %, it didn't know that it was already eclipsing. When Apple released the first iPhone, both Symbian and Windows Mobile started losing attractiveness to the public eye as well as to developers'.¹ In Q4 2008, the market still was in Symbian's hands (47.1 %, -7.5 % YOY avg.) but Blackberry (19.5 %, 6.0 %), Windows Mobile (12.4 %, -1.2 %), Apple (10.8 %, 4.1 %) and others (8.7 %, -2.8 %) divided up all the rest. In this setting Android 1.0 took the first steps.

Q3 2009 registered a peak for Blackberry and iOS (thanks to the release of the iPhone 3GS) while Android, in Q4 2009, almost reached the level of Windows Mobile, which in effect was steadily decreasing (7.4 %, 2.0 % vs 7.8 %, -3.1 %).

Then 2010 was the year of the boom. Android overtook Windows Mobile, iOS and Blackberry in a little more than two quarters, and then Android 2.2

¹The first generation of iPhone was announced on January 9, 2007 after years of rumors and speculation, and introduced in the United States on June 29, 2007.

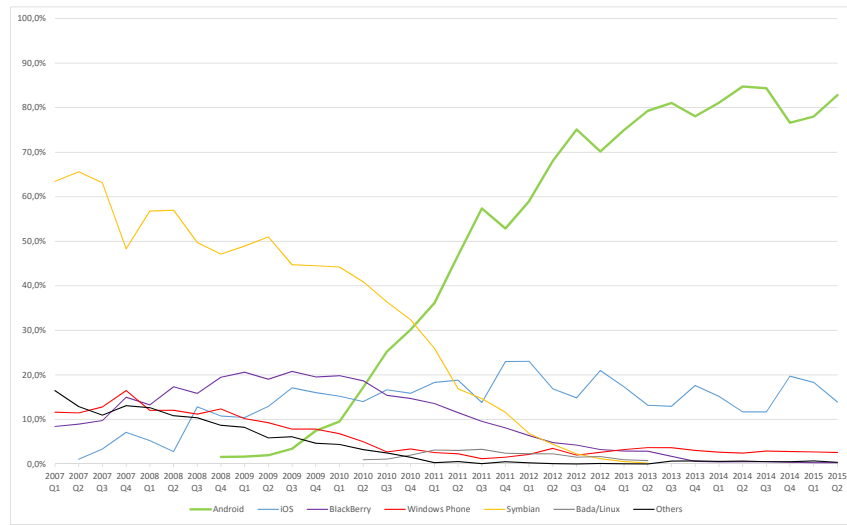


Figure 3.1: History of Mobile OS Market Share (% on all devices)

Source: 2007 Q1 → 2010 Q4, Gartner World-Wide Smartphone Sales.

Source: 2011 Q1 → 2015 Q2, IDC World-Wide Smartphone Shipments.

“*Froyo*” (API level 8) was released. For the Q4 2010 Android had acquired 30.2% (17.0%) of the market share while Symbian had 32.4% (−8.8%).

So, as shown in Figure 3.1, since Q1 2010, while the trend for Android was positive, all the others gradually lost market share, with the sole exception of Apple, which still now has a swinging periodic trend. It is important to notice that with every new Apple iPhone, Android lost market share only in the release quarters, but regained it shortly afterwards. Windows Mobile, eventually, after BlackBerry’s vanishing, has placed itself as the third competitor on the market, swinging around 2.9% since Q1 2012.

In Q3 2011 Android reached the important quota of 57.4% (26.2%) of the world-wide market share on the wave of Android 3.0 “*Honeycomb*” (API level 11), released in February and then, already in October 2011, Android 4.0 “*Ice Cream Sandwich*” (API level 14) which was the last version that officially supported Adobe Flash.

Furthermore, looking at the Figure 3.2 another important fact emerges: every new Apple iPhone release has not affected Android devices’ sales. Moreover, while the trend of iOS device sales is slowly growing, in truth

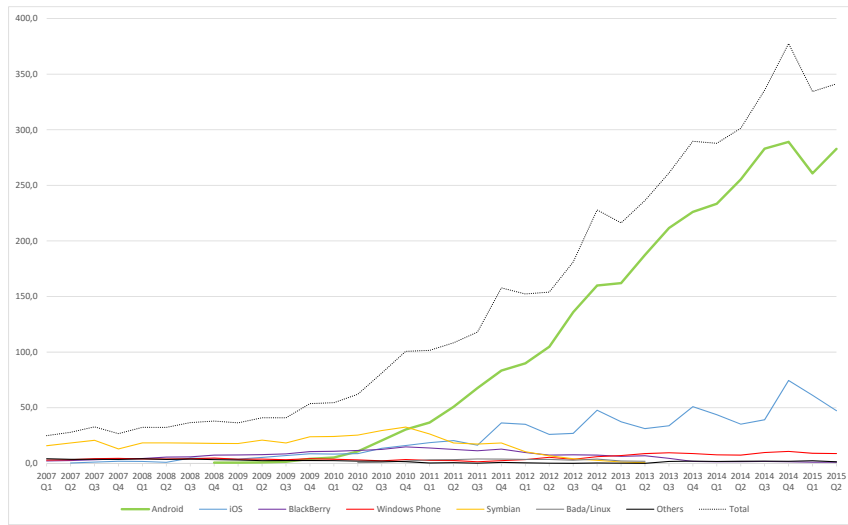


Figure 3.2: History of Mobile OS Market Share (Millions of units)

Source: 2007 Q1 → 2010 Q4, Gartner World-Wide Smartphone Sales.

Source: 2011 Q1 → 2015 Q2, IDC World-Wide Smartphone Shipments.

Apple is losing market share as a result of the outstanding amount of world-wide Android device sales. October 21, 2013 Google released Android 4.4 “*KitKat*” and shortly after, in Q2 2014, Android reached an unprecedented level with 84.7% (4.6%) of the market share.

At this point it is hard to get more market share without the disappearance of one of the major competitors. Microsoft Mobile and its perfectly stable average of 2.9% is far from disappearing because Microsoft and Nokia’s brands still have a great appeal despite years of failing marketing choices. Even Apple, which indeed has a swinging but decreasing trend, seems to be increasing rates in the recent past and the successful release of iPhone6 and iPhone6+ is the proof. On the other hand, Android is always gaining popularity in the Eastern market where new companies are producing mobile devices for all kind of customers, but overall, the base share now is so wide that in the worst case it will require years, or even more than a decade, for Android to disappear.

This scenario is very promising for Android developers and in fact this is the main reason why I have chosen Android as the reference Operating Sys-

tem (OS). But not only: just looking at the connectivity features introduced with every new Android release (see Appendix A) their prominent role in the Android developing ecosystem is evident, so my choice may be defined as just pragmatic.

3.2 Android Architecture

The Android OS is based on a modified Linux 2.6 kernel. Compared to a Linux 2.6 environment though, several drivers and libraries have been either modified or newly developed to allow Android to run as efficiently and as effectively as possible on mobile devices (such as smartphones or tablets). Some of these libraries have their roots in open source projects. Due to some licensing issues, the Android community decided to implement their own C library (called *Bionic*), and to develop an Android specific Java runtime engine, the Dalvik Virtual Machine (VM).

With Android, the focus has always been on optimizing the infrastructure based on the limited resources available on mobile devices. To complement the operating environment, an Android specific Application Framework was designed and implemented. Therefore, Android can best be described as a complete solution stack, incorporating the OS, middleware components, and applications. In Android, the modified Linux 2.6 kernel acts as the Hardware Abstraction Layer (HAL). Finally, the Android Software Development Kit (SDK) provides the tools and APIs necessary to begin developing applications on the platform using the Java programming language.

3.2.1 Linux Kernel

As shown in Figure 3.3, Android relies on a Linux kernel which operates as the HAL for core system services such as security, memory management, process management, network stack, and device drivers. Some of the Android specific kernel enhancements include:

- *Alarm Driver*, which provides timers to wakeup devices;
- *Shared Memory Driver* (`ashmem`);

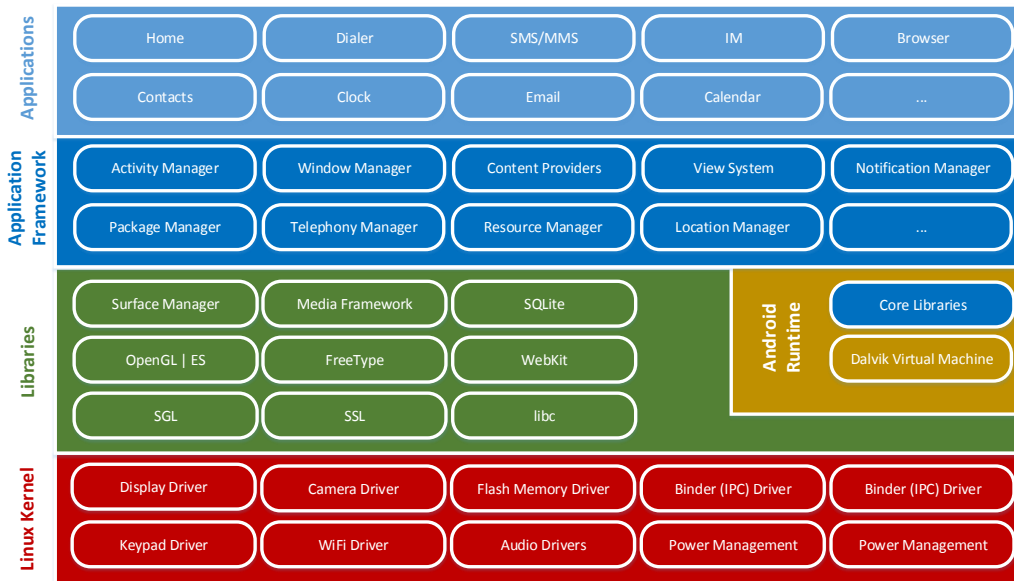


Figure 3.3: Android Architecture

- *Binder*, for Inter-Process Communication (IPC);
- *Power Management*, which takes a more aggressive approach than the Linux one;
- Low-memory killer;
- Kernel debugger and logger;
- etc.

During the Android boot process, the Android Linux kernel component first calls the `init` process. The `init` process accesses the files `init.rc` and `init.device.rc`.² Out of the `init.rc`, a process labeled `zygote` is started.

The `zygote` process loads the core Java classes and performs the initial processing steps. These Java classes can be reused by Android applications and hence, this step expedites the overall startup process. After the initial load process, `zygote` idles on a socket and waits for further requests.

Every Android application runs in its own process environment. A special driver labeled the `binder` allows for efficient IPCs. Actual objects are stored in Shared Memory which means that IPC is being optimized, as less data has

²The `init.device.rc` is device-specific.

to be transferred. Compared to most Linux environments, Android does not provide any swap space. Hence, the amount of virtual memory is governed by the amount of physical memory available on the device.

3.2.2 Libraries

Android includes a set of C/C++ libraries used by various components of the system. These capabilities are exposed to developers through the Android Application Framework as the Libraries layer is interfaced through Java (which deviates from the traditional Linux design). It is in this layer that the Android specific `libc` Bionic is located. This library is not compatible with the Linux `glibc` but compared to it, the Bionic library has a smaller memory footprint. More in detail, the Bionic library contains a special thread implementation that firstly optimizes the memory consumption of each thread and then reduces the startup time of a new thread.

Also, Android provides runtime access to kernel primitives and hence, user-space components can dynamically alter the kernel behavior. Only processes/threads though that do have the appropriate permissions are allowed to modify these settings. Security is maintained by assigning a unique User ID (UID) and Group ID (GID) pair to each application. As mobile devices are normally intended to be used by a single user only (compared to most Linux systems), the Linux `/etc/passwd` and `/etc/group` settings have been removed. In addition, to boost security, `/etc/services` was replaced by a list of services maintained inside the executables.

To summarize, the Android C library is especially suited to operate under the limited CPU and memory conditions common to the target Android platforms. Further, special security provisions were designed and implemented to ensure the integrity of the system.

3.2.3 Android Runtime

Android includes a set of *Core Libraries* which provide most of the functionality available in the Java programming language. Every Android application

runs in its own process, with its own instance of the *Dalvik VM*. Dalvik has been written so that a device can run multiple VMs efficiently. The *Dalvik VM* executes files in the Dalvik Executable (`.dex`) format which is optimised for minimal memory footprint. The VM is register-based, and runs classes compiled by a Java language compiler that have been transformed into the `.dex` format by the included “`dx`” tool. The *Dalvik VM* relies on the Linux kernel for underlying functionality such as threading and low-level memory management.

3.2.4 Application Framework

By providing an open development platform, Android offers developers the ability to build applications. Developers have full access to the same framework APIs used by the core applications. The application architecture is designed to simplify the reuse of components; any application can publish its capabilities and any other application may then make use of those capabilities (subject to security constraints enforced by the framework). This same mechanism allows components to be replaced by the user.

In this layer, the *Activity Manager* governs the application life cycle; the *Content Providers* enable applications to either access data from other applications or to share their own data, the *Resource Manager* provides access to non-code resources (such as graphics), while the *Notification Manager* enables applications to display custom alerts.

3.2.5 Applications

Android applications are bundled into an Android package (`.apk`) via the Android Asset Packaging Tool (AAPT). To streamline the development process, Google provides the Android Development Tools (ADT) which streamlines the conversion from `.class` to `.dex` files, and creates the `.apk` during deployment. In a very simplified manner, Android applications are in general composed by the entities explained as following.

Activities provide visible screens that mobile users can interact with. An

Activity is also responsible for monitoring and reacting to the operations that a user have performed on the screen.

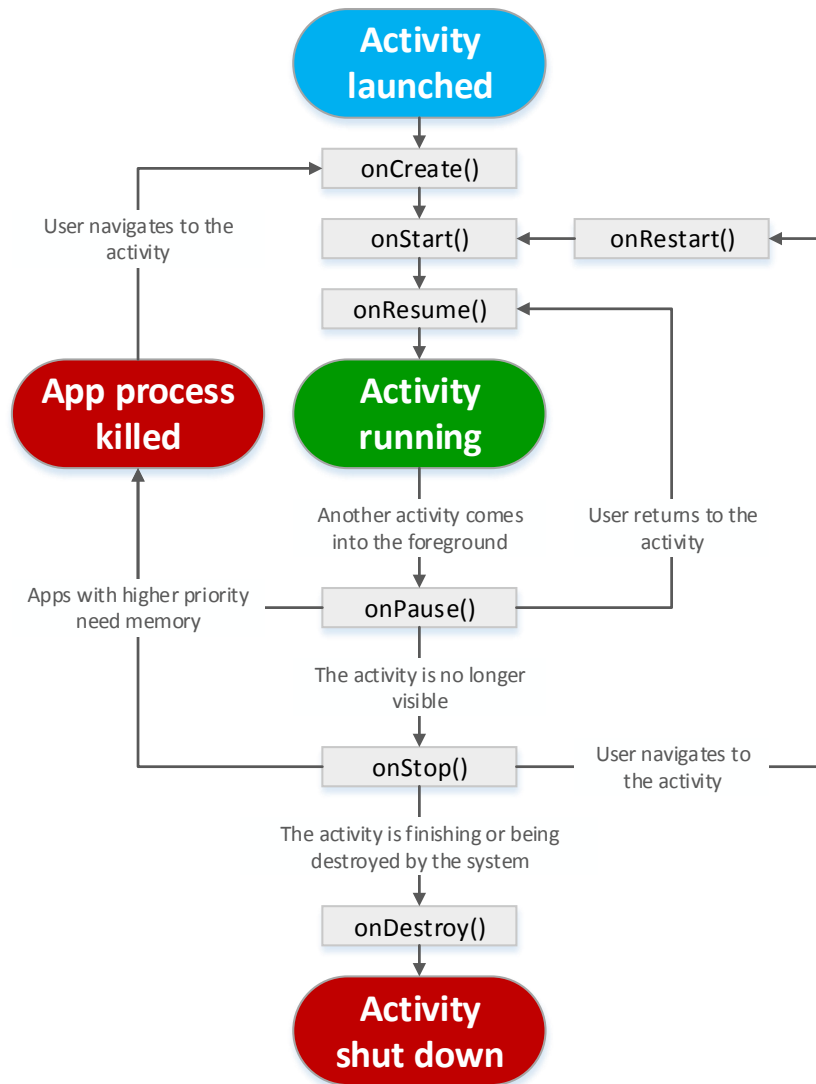


Figure 3.4: The Activity life-cycle

The life-cycle of an activity includes several states. It begins from `onCreate()` and ends at the time when `onDestroy()` is called. After an Activity has been created, `onStart()` is the point that the Activity becomes visible to users. The method `onResume()` also shows a state the Activity is visible, however different from `onStart()`, it restores a pre-

vious state. The `onPause()` represents a state that the current Activity is placed in the background, it is active and ready to be brought back into focus at any time. Though the activity at the state of `onStop()` is still alive, it is unattached from the *Window Manager* and can no longer be restored.

The Activity which is started at the application launch time is called the *Main Activity*. An application can have a series of Activities and one Activity is capable of creating another one. When a new Activity is started, the old one is not killed; instead, its state is pushed into the stack. The old Activity will be restored by retrieving its state and regain the focus if the user navigates back.

Services work quite similar to Activities, the only difference is that the Service usually runs in the background and performs a long term task; As a result, it does not provide any graphic interfaces.

Services can be started in two different ways. Calling the method `startService()` allows us to run an independent task, the Service quits automatically when the task is finished. The other way is through application bindings: a bound service is subjected to an application, thus the application has to decide when to active it and when to kill it.

Content Providers work as the database for the application. The data in Content Providers can be shared across applications but only when the access is allowed. The application is also able to use the public Content Providers managed by Google.

When storing data into the Content Provider, the user needs to specify the name of the data by following the Uniform Resource Identifier (URI) scheme so that the data can be identified and retrieved by name.

Broadcast Receivers let the application listen to a particular state of either the system or other applications. They are especially useful in order to activate some service at a specific point. Let us suppose that the application has to get started as soon as the phone is finished with the initialization. If one fully registers for receiving the broadcast of the phone boots, one will be notified at that specific point and then it is possible to ask the system to launch the application.

The notification message sent is called **Intent** and it is serialized when it is sent. The message consists of the data together with the operation that will be performed. Intent filters are used to filter out unwanted intents so that users are informed by relevant ones only.

Android Manifest

The `AndroidManifest.xml` file is the configuration file of the Android application. It specifies the components that the application owns and the external libraries it uses. As to the Android permissions, it declares permissions it requests as well as permissions that are defined to protect its own components [25]. The structure of `AndroidManifest.xml` is as following:

```
1 <?xml version="1.0" encoding="utf-8"?>
2
3 <manifest>
4
5     <uses-permission />
6     <permission />
7     <permission-tree />
8     <permission-group />
9     <instrumentation />
10    <uses-sdk />
11    <uses-configuration />
12    <uses-feature />
13    <supports-screens />
14    <compatible-screens />
15    <supports-gl-texture />
16
17    <application>
18
19        <activity>
20            <intent-filter>
21                <action />
22                <category />
23                <data />
24            </intent-filter>
25            <meta-data />
26        </activity>
27
28        <activity-alias>
29            <intent-filter> . . . </intent-filter>
30            <meta-data />
31        </activity-alias>
32
```



```
33     <service>
34         <intent-filter> . . . </intent-filter>
35         <meta-data/>
36     </service>
37
38     <receiver>
39         <intent-filter> . . . </intent-filter>
40         <meta-data />
41     </receiver>
42
43     <provider>
44         <grant-uri-permission />
45         <meta-data />
46         <path-permission />
47     </provider>
48
49     <uses-library />
50
51 </application>
52
53 </manifest>
```

3.3 Support to Connectivity

Android uses a wide set of packages for handling connectivity. In addition to the `java.net` and `javax.net` packages, Android has its own packages to provide connectivity APIs to developers. The main package is `android.net`, but in it there is a specialized hierarchy which includes `.http`, `.nsd`, `.rtp`, `.sip` and `.wifi`, where the latter has also his own hierarchy for peer-to-peer functionalities. Not included in `.net`—but considerably important for new Android devices’ connectivity—is the `android.nfc` package, obviously for Near Field Communication (NFC). Finally, other very important connectivity features are provided by classes in `android.app` and `android.content` packages. The complete hierarchy is shown in Figure 3.5.

The `android.net` package is a set of classes that help with network access, beyond the normal `java.net.*` APIs. It is structured as shown in Figure B.1. In addition, Android has a number of classes to achieve end-to-end service-oriented communications. The most important classes are `ConnectivityManager`, `Service`, `VpnService`, `Activity` and `Intent`.



Figure 3.5: Android Connectivity Packages

3.3.1 The `ConnectivityManager` class

This class answers queries about the state of network connectivity. It also notifies applications when network connectivity changes. It is possible to get an instance of this class by calling the method `Context.getSystemService(Context.CONNECTIVITY_SERVICE)` provided by `Context` class.³

The primary responsibilities of this class are to:

- Monitor network connections (WiFi, GPRS, UMTS, etc.);
- Send broadcast intents when network connectivity changes;
- Attempt to “fail over” to another network when connectivity to a network is lost;

³The `Context` is an interface to global information about an application environment. It formally is an abstract class whose implementation is provided by the Android system. It allows access to application-specific resources and classes, as well as up-calls for application-level operations such as launching activities, broadcasting and receiving intents, etc.

ConnectivityManager
<pre> -ACTION_BACKGROUND_DATA_SETTING_CHANGED: String -CONNECTIVITY_ACTION: String -DEFAULT_NETWORK_PREFERENCE: int -EXTRA_EXTRA_INFO: String -EXTRA_IS_FAILOVER: String -EXTRA_NETWORK_INFO: String -EXTRA_NETWORK_TYPE: String -EXTRA_NO_CONNECTIVITY: String -EXTRA_OTHER_NETWORK_INFO: String -EXTRA_REASON: String -TYPE_BLUETOOTH: int -TYPE_DUMMY: int -TYPE_ETHERNET: int -TYPE_MOBILE: int -TYPE_MOBILE_DUN: int -TYPE_MOBILE_HIPRI: int -TYPE_MOBILE_MMS: int -TYPE_MOBILE_SUPL: int -TYPE_WIFI: int -TYPE_WIMAX: int +getActiveNetworkInfo(): NetworkInfo +getAllNetworkInfo(): NetworkInfo[] +getBackgroundDataSetting(): boolean +getNetworkInfo(int): NetworkInfo +getNetworkPreference(): int +isActiveNetworkMetered(): boolean +isNetworkTypeValid(int): static boolean +requestRouteToHost(int, int): boolean +setNetworkPreference(int): void +startUsingNetworkFeature(int, String): int +stopUsingNetworkFeature(int, String): int </pre>

Figure 3.6: The `ConnectivityManager` class

Stricken out constants and methods are deprecated.

- Provide an API that allows applications to query the coarse-grained or fine-grained state of the available networks [5].

3.3.2 The `Service` class

A `Service` is an application component representing either an application's desire to perform a longer-running operation while not interacting with the user, or to supply functionality for other applications to use. Each service class must have a corresponding `<service>` declaration in its pack-

age's `AndroidManifest.xml`. Services can be started with `Context.startService()` and `Context.bindService()`.

It is important to note that services, like other application objects, run in the main thread of their hosting process. This means that, if a service is going to do any CPU intensive (such as music playback) or blocking (such as networking) operations, it should spawn its own thread in which to do that work. The `IntentService` class is available as a standard implementation of `Service` that has its own thread where it schedules its work to be done.

Service
-START_CONTINUATION_MASK: int
-START_FLAG_REDELIVERY: int
-START_FLAG_RETRY: int
-START_NOT_STICKY: int
-START_REDELIVER_INTENT: int
-START_STICKY: int
-START_STICKY_COMPATIBILITY: int
#dump(FileDescriptor, PrintWriter, String[]): void
+getApplication(): final Application
+onBind(Intent): abstract IBinder
+onConfigurationChanged(Configuration): void
+onCreate(): void
+onDestroy(): void
+onLowMemory(): void
+onRebind(Intent): void
+onStart(Intent, int): void
+onStartCommand(Intent, int, int): int
+onTaskRemoved(Intent): void
+onTrimMemory(int): void
+onUnbind(Intent): boolean
+startForeground(int, Notification): final void
+stopForeground(boolean): final void
+stopSelf(): final void
+stopSelf(int): final void
+stopSelfResult(int): final boolean

Figure 3.7: The `Service` class

Most confusion about the `Service` class actually revolves around what it is *not*. A `Service` is not a separate process. The `Service` object itself does not imply it is running in its own process; unless otherwise specified, it runs in the same process as the application it is part of. Also, a `Service` is not a thread: it is not a means to do work off of the main thread.

Thus a **Service** itself is actually very simple, providing two main features:

- A facility for the application to tell the system about something it wants to be doing in the background (even when the user is not directly interacting with the application). This corresponds calling to `Context.startService()`, which asks the system to schedule work for the service, to be run until the service or someone else explicitly stop it.
- A facility for the application to expose some of its functionality to other applications. It corresponds calling to `Context.bindService()`, which allows a long-standing connection to be made to the service in order to interact with it.

When a **Service** component is actually created, for either of these reasons, all that the system actually does is instantiate the component and call its `onCreate()` and any other appropriate callbacks on the main thread. It is up to the **Service** to implement these with the appropriate behavior, such as creating a secondary thread in which it does its work.

There are two reasons for a service to be run by the system. If someone calls `Context.startService()` then the system will retrieve the service (creating it and calling its `onCreate()` method if needed) and then call its `onStartCommand(Intent, int, int)` method with the arguments supplied by the client. The service will at this point continue running until `Context.stopService()` or `stopSelf()` is called. Multiple calls to `Context.startService()` do not nest (though they do result in multiple corresponding calls to `onStartCommand()`), so no matter how many times it is started: a service will be stopped once `Context.stopService()` or `stopSelf()` is called; however, services can use their `stopSelf(int)` method to ensure the service is not stopped until started intents have been processed.

Clients can also use `Context.bindService()` to obtain a persistent connection to a service. This likewise creates the service if it is not already running (calling `onCreate()` while doing so), but does not call `onStartCommand()`. The client will receive the `IBinder` object that the service returns from its `onBind(Intent)` method, allowing the client to then make calls back to the service. The service will remain running as long as the

connection is established (whether or not the client retains a reference on the service's `IBinder`). Usually the `IBinder` returned is for a complex interface that has been written in Android Interface Definition Language (AIDL).

A service can be both started and have connections bound to it. In such a case, the system will keep the service running as long as either it is started or there are one or more connections to it with the `Context.BIND_AUTO_CREATE` flag. Once neither of these situations hold, the service's `onDestroy()` method is called and the service is effectively terminated. All cleanup (stopping threads, unregistering receivers) should be complete upon returning from `onDestroy()`.

Global access to a service can be enforced when it is declared in its manifest's `<service>` tag. By doing so, other applications will need to declare a corresponding `<uses-permission>` element in their own manifest to be able to start, stop, or bind to the service.

When using `Context.startService(Intent)`, it is also possible to set `Intent.FLAG_GRANT_READ_URI_PERMISSION` and `Intent.FLAG_GRANT_WRITE_URI_PERMISSION` on the `Intent`. This will grant the `Service` temporary access to the specific URIs in the `Intent`. Access will remain until the `Service` has called `stopSelf(int)` for that start command or a later one, or until the `Service` has been completely stopped. This works for granting access to the other apps that have not requested the permission protecting the `Service`, or even when the `Service` is not exported at all.

The Android system will attempt to keep the process hosting a service around as long as the service has been started or has clients bound to it. When running low on memory and needing to kill existing processes, the priority of a process hosting the service will be the higher of the following possibilities:

- If the service is currently executing code in its `onCreate()`, `onStartCommand()`, or `onDestroy()` methods, then the hosting process will be a foreground process to ensure this code can execute without being killed.
- If the service has been started, then its hosting process is considered to be less important than any processes that are currently visible to the user on-screen, but more important than any process not visible.

Because only a few processes are generally visible to the user, this means that the service should not be killed except in extreme low memory conditions.

- If there are clients bound to the service, then the service's hosting process is never less important than the most important client. That is, if one of its clients is visible to the user, then the service itself is considered to be visible.
- A started service can use the `startForeground(int, Notification)` API to put the service in a foreground state, where the system considers it to be something the user is actively aware of and thus not a candidate for killing when low on memory.

Of course this means that most of the time that a service is running, it may be killed by the system if it is under heavy memory pressure. If this happens, the system will later try to restart the service. An important consequence is that if you want to implement `onStartCommand()` in order to schedule work, it must be done asynchronously. Other application components running in the same process as the Service can increase the importance of the overall process beyond just the importance of the service itself [5].

3.3.3 The `VpnService` and `VpnService.Builder` classes

`VpnService` is a base class for applications to extend and build their own Virtual Private Network (VPN) solutions, which specializes the `Service` class. In general, it creates a virtual network interface, configures addresses and routing rules, and returns a file descriptor to the application. Each read from the descriptor retrieves an outgoing packet which was routed to the interface. Each write to the descriptor injects an incoming packet just like it was received from the interface. The interface is running on IP, so packets are always started with IP headers. The application then completes a VPN connection by processing and exchanging packets with the remote server over a tunnel.

Letting applications intercept packets raises huge security concerns. A VPN application can easily break the network. Besides, two of them may

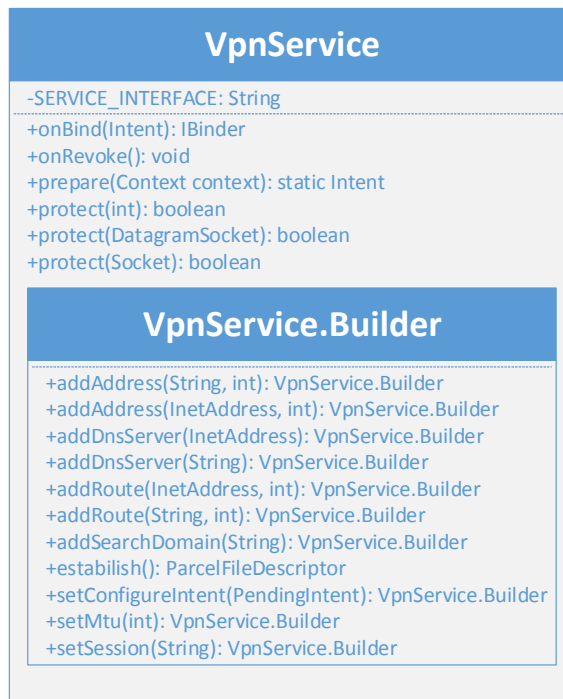


Figure 3.8: The `VpnService` and `VpnService.Builder` classes

conflict with each other. The system takes several actions to address these issues. Here are some key points:

- User action is required to create a VPN connection.
- There can be only one VPN connection running at the same time. The existing interface is deactivated when a new one is created.⁴
- A system-managed notification is shown during the lifetime of a VPN connection.
- A system-managed dialog gives the information of the current VPN connection. It also provides a button to disconnect.
- The network is restored automatically when the file descriptor is closed. It also covers the cases when a VPN application is crashed or killed by the system.

⁴This, as it will be explained in section 3.4, is however an insurmountable obstacle for multihoming implementation for Android.

There are two primary methods in this class: `prepare(Context)` and `establish()`. The former deals with user action and stops the VPN connection created by another application. The latter creates a VPN interface using the parameters supplied to the `VpnService.Builder`. An application must call `prepare(Context)` to grant the right to use other methods in this class, and the right can be revoked at any time. Here are the general steps to create a VPN connection:

1. When the user presses the button to connect, the system calls `prepare(Context)` and launches the returned intent. When the application becomes prepared, it starts the service.
2. It creates a tunnel to the remote server and negotiate the network parameters for the VPN connection.
3. It supplies those parameters to a `VpnService.Builder` and creates a VPN interface by calling `establish()`.
4. It processes and exchanges packets between the tunnel and the returned file descriptor.
5. When `onRevoke()` is invoked, it closes the file descriptor and shuts down the tunnel gracefully.

Services extended by this class need to be declared with appropriate permission and intent filter. Their access must be secured by `BIND_VPN_SERVICE` permission, and their intent filter must match `SERVICE_INTERFACE` action [5].

3.4 State-of-the-Art Multihoming Support

The core of Android networks handling is the `ConnectivityManager`. Added in API level 1—as already said—the class is responsible for all the operations between the system and the networks. Until API level 18⁵, Android platform officially was able to simultaneously manage only two networks: a default network and another network. This was actually the result of having two main network interfaces (generally a WiFi and a 3G/4G) and so the `ConnectivityManager` had to manage the data communications over two

⁵Android 4.3 Jelly Bean, release July 24, 2013.

possible networks but with the limitation of using only one of these interfaces at once. This scenario justified also the existence of an `int` flag like a `DEFAULT_NETWORK_PREFERENCE` that effectively pointed out which network is the default one and which one is not. Naturally the default network is the one with the best signal, and in order to know that, a `NetworkInfo` object that knows the network(s) state is required. This is important also because the `ConnectivityManager` takes care of the *fail over* (also known as *handover*) procedure, that is when a connection with a network is lost for some reason and a new one is consequently set up with another network. If so, calling the `NetworkInfo`'s `isFailover()` will return `true`.

But since Android supports many more networks now, the single “default network preference” cannot really express the hierarchy. At this point, the default network is defined by the `networkAttributes` in `config.xml`⁶, giving in this way more flexibility to developers, but the `ConnectivityManager`'s method `getNetworkPreference()` is definitely going to be deprecated. It retrieves the current preferred network type but—as specified in [5]—«this made sense when we only had two network types, but with more and more default networks we need an array to list their ordering. This will be deprecated soon.»

Hence the direction for the next API levels is to dismiss the “default network preference” principle for having, instead, a sorted array of networks. This actually has a good impact on ubiquity and on efficiency of system interfaces' management, but it eventually does *not* provide multihoming.

A possibility is represented by enabling all the network interfaces on the device but this is not allowed by default Android policies [24, 55]. In general, the WiFi interface has priority, which is good as it has better performances and the data consumption on 3G is not free. When the user turns on the WiFi, the 3G goes off; when the user turns on the 3G, it will be shut down if the WiFi is already on; if the user shuts down the WiFi, the 3G can be automatically turned on (if the user allows it in the settings). With these

⁶At the time of writing, the developer can determine the current value by calling `getNetworkPreference()`.

rules, it is impossible to get the two interfaces running at the same time and since there is no user interface to change this, it has to be changed directly in the source. The simplest way to get over the policies is to comment in the network **Connectivity Service** of the Android application framework the code lines which shut down the interface which does not have priority. The expected resulting behaviour is that turning on an interface will not shut down the other [24].

Of course, advanced IP routing, multiple routing tables and network filtering must be enabled too. In Android `goldfish` kernel:

```
1 CONFIG_IPV6=y
2 CONFIG_IPV6_MULTIPLE_TABLES=y
3 CONFIG_IP_ADVANCED_ROUTER=y
4 CONFIG_IP_MULTIPLE_TABLES=y
5 CONFIG_NETFILTER=y
6 CONFIG_NETFILTER_ADVANCED=y
7 CONFIG_NETFILTER_XTABLES=y
8 CONFIG_NETFILTER_XT_MARK=y
9 CONFIG_NETFILTER_XT_TARGET_MARK=y
10 CONFIG_IP_NF_IPTABLES=y
11 CONFIG_IP_NF_MANGLE=y
12 CONFIG_IP6_NF_IPTABLES=y
13 CONFIG_IP6_NF_MANGLE=y
```

And so, after a VPN set-up, an App could use multiple interfaces to route IP streams. Obviously it will be responsibility of the app's developer to manage the interfaces, the networks and the data through them, so this may represent a Pandora's crate in terms of potentialities as well as security issues.

Another option is to implement an ad hoc multihoming solution. As I have explained at the end of chapter 2, this may be achieved in different ways but generally a multihoming protocol has to be supported somehow.

So at this point a choice must be made: either to use a non-rooted system or a rooted one? If the choice is to root the device, it is possible to enable **SCTP**⁷ in Android's `goldfish` kernel and hence provide a good multihoming support for the host:

```
14 CONFIG_IP_SCTP=y
```

⁷See Section 2.6 for protocols' pros and cons.

In this case all multihoming features will be provided directly by SCTP with almost no effort for the developer.⁸

As an alternative, so far HIP is available on Android platforms thanks to an experimental porting of **HIP for Linux (HIPL)**. See section 3.4.1 for further details.

Otherwise, if the choice is a non-rooted Android because it is wanted to use it as a “black box”⁹, consequently it is necessary to implement a more complex solution like **LISPMob**.

3.4.1 HIP for Linux

HIPL is an open source software project initially developed at Helsinki Institute for Information Technology, Helsinki University of Technology and lately at Aalto University, Aachen University and Tampere University of Technology.

Basically HIPL is used for encrypting and protecting all TCP and UDP connections similarly to Transport Layer Security (TLS), but without requiring changes in applications. The access control is based on public key as demanded by HIP specifications in order to protect the end-to-end traffic. Furthermore HIPL lets to set-up a server behind NAT, to store and lookup hostnames and hosts’ address information (like DNS, but based on free and distributed technology), and guarantees that Internet connections survive to short-time address changes (e.g. when DHCP address lease expires).

HIPL uses a HIP daemon (**hipd**) whose main task is to handle the HIP base exchange. More in detail, the **hipd** manages all the core activities of the protocol: HIP packet generation and handling, HIT-to-IP mappings, HIP packets communication between the user level and the kernel, as well as all cryptographic-related features like cookies, Diffie-Hellman, signatures, etc. In

⁸Java supports SCTP in JDK 7 as of milestone 3, but it does not mean that *Dalvik VM* supports it. Though Android does have SCTP support by **goldfish** kernel version 2.5.67, it is not in Dalvik 4.2 or previous versions. Anyway developers may create a Java class that would wrap a native library with SCTP calls.

⁹Basically I want to come to a more “user-friendly” solution because no-one should expect that the average user would root his Android device.

addition, it provides a simple command-line administrative Graphical User Interface (GUI) [52].

HIPL does not require high-performance hardware to work or any other third-parties software tool. It runs on a Linux OS with a kernel 2.6.27 or patched previous versions but recently HIPL has a partial experimental support for the Android platform too.

The components that currently work on Android are `hipd` (the HIP daemon), and `hipconf` (the configuration tool). The `hipd` does require root privileges, and the running kernel must support the IPSec BEET mode, the dummy network driver and the null crypto algorithm. Often one or more of these are not compiled into the stock kernel, and it is likely that it is needed to compile and install a custom kernel. It must be said that currently HIPL only supports compiling under Linux but to help the user there is a script that downloads and extracts the toolchain needed to compile `hipd` and `hipconf` which has been confirmed to work at least on Ubuntu 12.04. After the HIPL source code is downloaded and the toolchain is installed, the steps to compile it for Android are almost similar to any standard Linux builds.

On Android, `hipd` needs to be run with the `'-a'` parameter. Additionally, it supports the same parameters as the normal Linux version does (e.g. `'-k'` kills an already running instance and `'-b'` starts `hipd` in the background). By example, running `'hipd -ab'` and configuring hosts in `/etc/hosts` one should be able to use HIP on any app that supports IPv6 [1].

3.4.2 LISPmob

LISPmob is an open-source LISP and LISP-MN implementation for Linux, Android and OpenWRT. With LISPmob, hosts can change their network attachment point without losing connectivity, while maintaining the same IP address.

Since its inception, LISP has gained significant traction because of its inherent architectural advantages: fully featured traffic engineering capabilities, minimal configuration needs, very low deployment cost and benefits to early adopters. Institutions (both commercial and academic) across the

globe have demonstrated their interest in LISP. At the time of writing LISP has been deployed in a *beta-network* that includes more than twenty countries and hundreds of institutions. In fact, the LISPmob audience, composed of both individuals and companies, has an estimated user-base of almost two hundreds. Mainly, individuals are interested in LISPmob to obtain IPv6 connectivity over IPv4-only providers (and sometimes the other way around) and to set up simple multihoming deployments. Instead, companies use LISPmob in different ways, mostly as a tool for proof-of-concept LISP deployments and to provide LISP connectivity on client-side devices. There is also an increasing interest for LISPmob in academic projects and other open-source communities.

The LISPmob project aims to bring a full-featured LISP open-source implementation to Linux-flavoured systems. Currently it is available for standard Linux, Android and OpenWRT. More in detail, since version 0.4.0, LISPmob includes support for Android devices operating as LISP-MN. There are two different editions of the LISPmob Android application: for rooted devices and for non-rooted devices. It is expected that in the future the root version will provide features beyond those available on the non-root version, however on LISPmob 0.4.1 there is just one root-only feature, the support for IPv6 RLOCs. In both editions, there is a limit of one IPv4 EID and one IPv6 EID mapped to one or more RLOC interfaces.

Even though several interfaces can be managed by LISPmob at the same time, on Android platform they can only be used in an “active-backup” fashion (that is, no more than one interface used at once), as consequence of the reasons previously explained.

Chapter 4

State-of-the-Art Android LISPmob and Research Project

4.1 LISPmob Versions and Functionalities

At the time of writing two versions of LISPmob for Android exist: the non-root and the root versions. These two different versions of LISPmob for Android have different requirements. LISPmob for non-rooted devices requires Android 4.0 or higher, while LISPmob for rooted devices requires root access and Android version 2.3.6 or higher. However, the main difference between the two versions is that the non-root version allows to select several interfaces but it only will use the one that Android has set as default¹; the root version also allows to select several interfaces by modifying the configuration file and it will generate the appropriate routing tables, but due Android networking limitations, only one interface is effectively active.

LISPmob for Linux implements the following features:

- Full IPv6 and IPv4 support;
- Router-mode and Mobile Node-mode, as well as other LISP devices;
- Handling multihoming scenarios with traffic balancing among links;
- Interface management, including handover events;
- NAT traversal capabilities.

¹See section 3.4.2 for further details on “active-backup”.

Instead, the LISPmob Android application basically allows to start and stop `lispd` daemon and edit the most important parameters of the configuration file. To access to the full list of features it is necessary to edit manually the configuration file located in `/sdcard/lispd.conf`. Manually edited parameters not present in the configuration form are overwritten when using the application configuration editor. In LISPmob 0.4.1 there is just one root-only feature, that is the support for IPv6 RLOCs. Very important to notice is that in both editions there is a limit of one IPv4 EID and one IPv6 EID mapped to one or more RLOC interfaces.

Moreover, due to a bug in VPN APIs on Android 4.4.0 and onwards, the non-rooted version of LISPmob will not work on Android 4.4.0, 4.4.1, 4.4.2 or 4.4.3. The bug was fixed on Android 4.4.4.

4.1.1 Reference Version

The LISPmob reference version for this research is the one for non-rooted devices. The motivations that lead to this choice are the following:

From the user point-of-view: People who root their devices are few because the rooting process requires skills uncommon to the average Android user. One of the LISP features in comparison with other protocols like HIP or SCTP is that it does not require changes to the user's devices, so this advantage must be absolutely preserved.

From the developer point-of-view: It is a fact that Google APIs not always offer complete control over the Android OS, specially when Native Development Kit (NDK) is used as much as in LISPmob. Nevertheless, since developers gain new powers with every new API level, the general rule for Android developers is to use the APIs in order to produce more reusable and robust code.

This research started using LISPmob 0.4.1.4 by forking the `lispmob-master` repository from [GitHub.com](https://github.com) in October 2014. Version 0.4.1.6 of the source files including the bug-fixing patch described in section 5.1.2 was released in late December 2014.

4.2 High-Level Software Architecture

Since our reference version of LISPmob is the one for non-rooted devices,² in this section its software architecture is presented from the software engineering perspective.

4.2.1 Package Organization

The `org.noroot.lispmob` package consists of several Java classes and, like any Android project, it extensively uses external resources as layout files, string libraries and drawables, all properly organized. But, since LISPmob uses the NDK because the `lispd` code is written in C/C++, it has also a Java Native Interface (JNI) responsible of making the link between the Java code and the native code.

So the whole project may be seen as organized in activities, services, utility libraries, a native interface (all in Java), the `lispd` native code (C/C++), external resources (XML and images), and build files—which includes configuration files (XML) and makefiles.

4.2.2 Activities

LISPmob

This is the Main Activity. The User Interface (UI) is very simple—it is composed of four objects: a checkbox and three buttons. The `lispCheckBox` starts/stops the VPN and basically shows if LISP protocol is used (or, more correctly, if the `lisp` is currently running). Each button creates an `Intent` and starts the related Activity: the `confActivity`, the `logActivity` and the `updateConfActivity`.³

²See section 4.1.1 for details on motivations.

³See the classes' description in this section for further details.

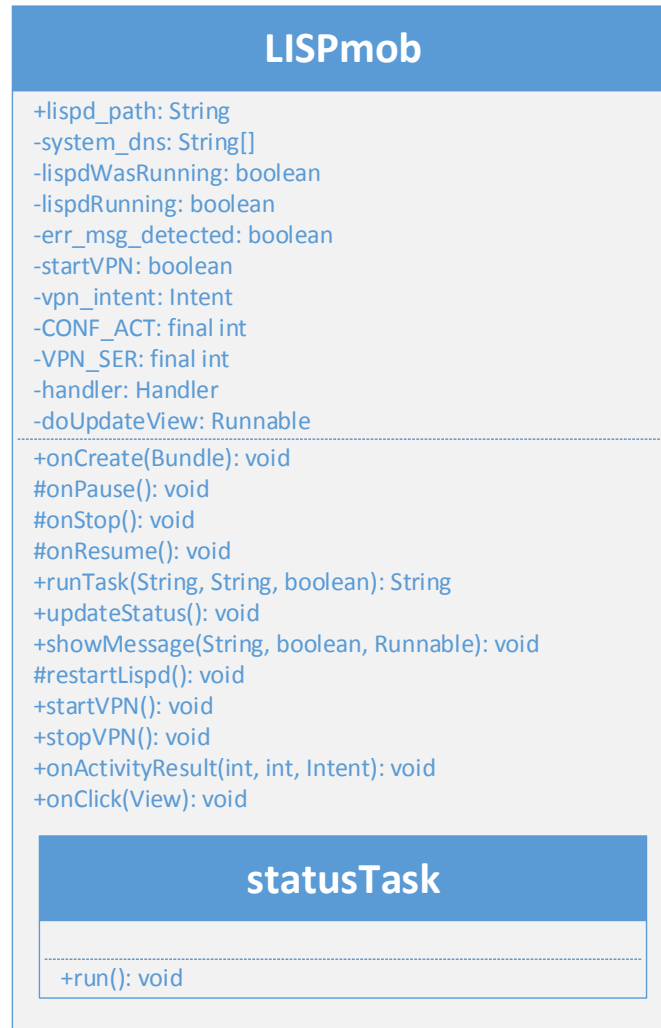


Figure 4.1: The LISPmob Activity



Figure 4.2: The confActivity Activity

confActivity

The `confActivity` is the Input/Output (I/O) Activity responsible of reading the configuration file `lispd.conf`, if it does exist. This Activity is created by the Main Activity, that is `LISPmob`.

logActivity

The `logActivity`'s unique responsibility is to provide a real-time updated UI for the `lispd.log` file. The user can set four levels of logging details (from 0 to 3) in the `lispd.conf` file.

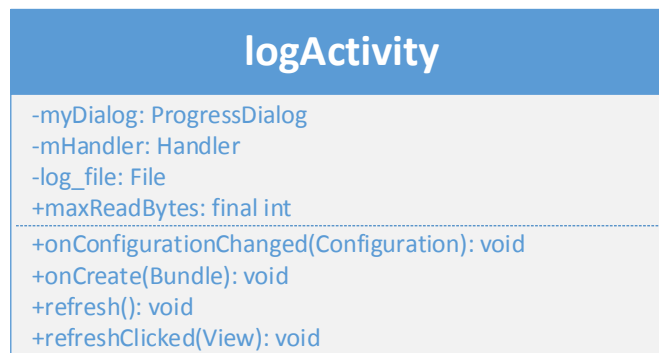


Figure 4.3: The `logActivity` Activity

updateConfActivity

This is the Activity responsible of the automatic building of the `lispd.conf` file. The user fills the empty fields with the parameters provided by the *beta network* and the `updateConfActivity` writes the `lispd.conf` file. The Activity also retrieve the parameters from the `lispd.conf` file in order to allow the editing by the user. Any manual edit of the `lispd.conf` file is lost when the user uses this Activity because the `lispd.conf` file is just overwritten.

updateConfActivity
+confFile: final String
+eidIPv4: String
+eidIPv6: String
+ifaces: List<String>
+MR: String
+MS: String
+MSKey: String
+proxyETR: String
+DNS1: String
+DNS2: String
+overrideDNS: boolean
+nat_aware: boolean
+rloc_prob_interval: int
+rloc_prob_retries: int
+rloc_prob_retries_interval: int
+logLevel: String
+CONFIG_UPDATED: final int
+conf_file: File
+iface_list: List<String>
+getNewDNS(): String[]
+isOverrideDNS(): boolean
+createConfFile(): void
+displayMessage(String, boolean, Runnable): void
+get_and_validate_parameters(): boolean
+onCreate(Bundle): void
+readConfFileAndFillParameters(): void
+updateConfClicked(View): void
+updateConfDNSClicked(View): void
+updateConfFile(): void
+updateConfNATAwareClicked(View): void

Figure 4.4: The updateConfActivity Activity

LISPmobVPNService
-TAG: final String
-mConfigureIntent: PendingIntent
-mThread: Thread
-mInterface: ParcelFileDescriptor
+vpn_runngin: boolean
+err_msg_code: int
-ipc_channel: IPC
+onStartCommand(Intent, int, int): int
+onDestroy(): void
+handleMessage(Message): boolean
+run(): synchronized void
-configure(): void
+notify_msg(String): void

Figure 4.5: The LISPmobVPNService Service

4.2.3 Services

LISPmobVPNService

This class is the VPN Service manager. `LISPmobVPNService` actually *is* a `VPNService` and in fact handles the IPC's channel and put the `lispd` over it. It also uses the `ConfigTools` class and handles log messages and notifications.

IPC

The IPC class is the engine of the LISPmob networking services. It setup a `LISPmobVPNService` and opens a `DatagramChannel` on which a socket is binded. While running, IPC understands if the packets over the `DatagramChannel` are VPN log messages or if the socket is effectively protected (that means that the VPN is on) or not.

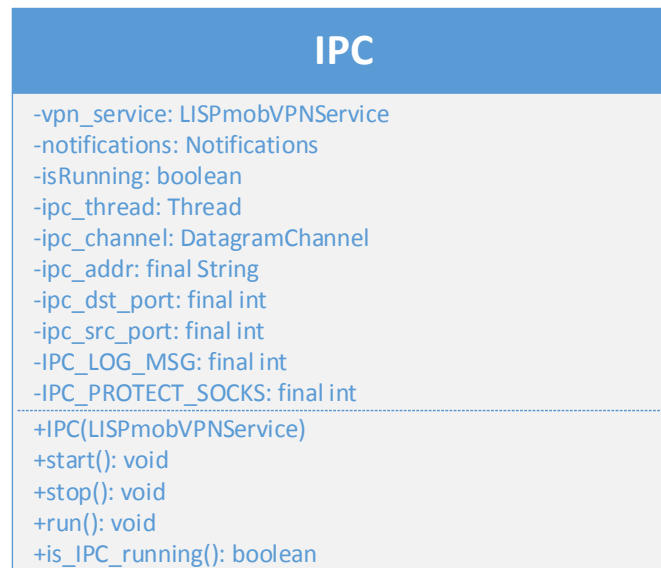


Figure 4.6: The IPC class

Obviously, the IPC constitutes a thread itself called “IPC” which can be started, run or stopped by appropriate methods.

4.2.4 Utility Libraries

ConfigTools

This is an utility library. It contains the parser methods to get the EIDs and the DNS from the configuration file, and the two very important methods `get_ifaces_list()` and `validate_IP_Address(String)` which respectively get the interface list from `/proc/net/xt_qtaguid/iface_stat_all` and verify the syntax correctness of IPv4 and IPv6 addresses.



Figure 4.7: The ConfigTools class

MultiSelectionSpinner

This `MultiSelectionSpinner` is a custom specialization of the `android.widget.Spinner` class, which basically is a multichoice UI panel. It shows the list of the device's network interfaces and allows the user to select one or more of them. This component is used by the `updateConfActivity` in order to create the “database-mapping” between the EID(s) and the network interface(s).

Notifications

A simple class that builds a log message as a notification and forwards it to the `Android NotificationManager`.

MultiSelectionSpinner
-_items: String[] -mSelection: boolean[] #simple_adapter: ArrayAdapter<String>
+MultiSelectionSpinner(Context) +MultiSelectionSpinner(Context, AttributeSet) -buildSelectedItemString(): String +getSelectedIndicies(): List<Integer> +getSelectedItemsAsString(): String +getSelectedStrings(): List<String> +onClick(DialogInterface, int, boolean): void +performClick(): boolean +setAdapter(SpinnerAdapter): void +setItems(List<String>): void +setItems(String[]): void +setSelection(int): void +setSelection(int[]): void +setSelection(List<String>): void +setSelection(String[]): void

Figure 4.8: The MultiSelectionSpinner class

Notifications
-context: Context
+Notifications(Context) +notify_msg(String): void

Figure 4.9: The Notifications class

4.2.5 Java Native Interface

LISPmob_JNI

JNI defines a way for managed code (written in the Java programming language) to interact with native code (written in C/C++). Native code accesses Java VM features by calling JNI functions. JNI functions are available through an *interface pointer* that is a pointer to a pointer. This pointer points to an

array of pointers, each of which points to an interface function. Every interface function is at a predefined offset inside the array. [41] In this way

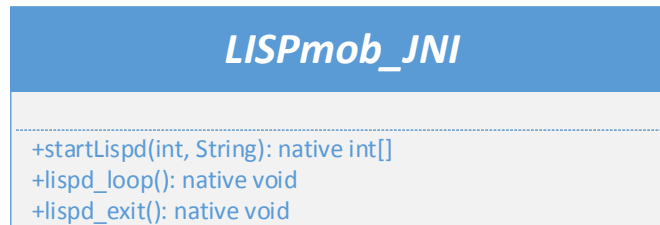


Figure 4.10: The LISPmob_JNI class

the LISPmob_JNI is the link between the Android LISPmob Java code and the `lispd` which is written in C++. This developing paradigm is smart and efficient—the `lispd`'s code is the same of the Linux one: modularity, extensibility and performances are so guaranteed.

4.3 Implementation Guidelines

4.3.1 The Beta Network

As explained in LISPmob documentation [47], running LISPmob host on the public Internet requires the following:

- an EID from a Mapping Service Provider (MSP);
- the RLOC of the Map Server that will register of this EID;
- an authentication token to register the EID with the Map Server;
- the RLOC of a Map Resolver;
- the RLOC of a PxTR;⁴
- a publicly routable RLOC for the host, which is neither firewalled, nor behind NAT.

Other than the last item, the above information is used for the `lispd` configuration file. The parameters I got from Lori Jackab of LISPmob.org are:

⁴Which is, more specifically, an Egress Tunnel Router.

EIDs: 153.16.51.48/32, 153.16.51.49/32

Map Server: 217.8.98.42

Map Resolver: 217.8.98.46

Password: `nattrav-test`

PxTRs: 217.8.98.33, 217.8.98.35

4.3.2 LISP-MN and NAT traversal

When used in a MN, the EID will be used by the applications on the host for establishing communications while the RLOC will differ, depending on the network point of attachment (e.g. it will be the IP address assigned to the host in the visited network).

Since version 0.3.3, LISPmob includes experimental NAT traversal capabilities. In order to use NAT traversal with LISPmob it is necessary a Map Server and a Re-encapsulating Tunnel Router (RTR) that are NAT traversal capable. Unfortunately, at time of writing, not all devices on the beta network have been updated to support NAT traversal yet. However, if NAT traversal feature is enabled, LISPmob is configured to send all data traffic through RTRs even if the interface has been provisioned with a public address. This behavior is a consequence of the lack mechanisms to update the cache of peers when there is an RTR involved in the data exchange. On its current form, NAT traversal support on LISPmob ignores IPv6 addresses of RLOC interfaces, besides, the current NAT traversal implementation in the beta network only supports the registration of a single EID per interface [29].

4.3.3 LISPmob build and install from source code

Linux

To build LISPmob for a standard Linux is necessary:

- a Linux hosts with kernel 3.2.0+;
- a C compiler (tested with `gcc`);
- GNU `make`;
- `git` (strongly recommended);

- `libConfuse`.

On Debian-derived Linux distributions (including Ubuntu), installing the following packages will provide almost all necessary dependencies⁵:

- `build-essential`;
- `git-core`;
- `libconfuse-dev`.

The latest version of the LISPmob source code can be obtained from GitHub:

```
git clone git://github.com/LISPmob/lispmob.git
```

To build and install the code, just run the following in the top-level directory:

```
make
sudo make install
```

This will build the executable files, installing them into `/usr/local/sbin`.

Android

Building the source code for Android is supported on Linux and Mac OSX only. Moreover, some extra packages are required:

- Android SDK;⁶
- Android NDK;⁷
- Apache Ant.

Since the Android code uses `git` submodules, it is important to build from a `git` repository checkout. To get the latest version of the LISPmob source from GitHub:

```
git clone git://github.com/LISPmob/lispmob.git
cd lispmob
git submodule init
git submodule update
```

⁵Any package that is required for LISPmob building is pointed out by the shell, if missing. So, an `apt-get install`, obviously followed by the list of missing packages, is enough to solve all the package dependencies.

⁶From <http://developer.android.com/sdk/>.

⁷From <http://developer.android.com/tools/sdk/ndk/>.

But also, it is necessary to install `androgenizer` in order to compile the `json-c`:

```
cd android/jni/androgenizer
autoreconf -i
./configure
make
make install
```

and then:

```
cd android/jni/json-c
./autogen.sh os=-linux-android
make Android.mk
```

To build the code, go to `android/` (located in the top-level directory) and modify the `local.properties` file with the correct path of our Android SDK and Android NDK. In the Android SDK Manager it should either have been installed the API level 17 (Android 4.2.2), or update the `project.properties` file to specify the currently installed API.⁸

Then, to compile the code:

```
cd android
./select_appl.bash
ant debug
```

This command generates an Android Package (APK) file called `lispmob-debug.apk` in the `android/bin/` directory. To install it: copy the file to the Android device and install it using the Application Manager.

4.4 Research Project

4.4.1 Project Definition and Phases

Due the Android limitations explained in section 3.4, the main operative goal of this research is to test the *seamless roaming* from an Android device's network interface to another, which may be considered as a multihoming's

⁸Regardless of the target API, LISPmob for rooted devices should still work on all Android releases from API level 9 (Android 2.3) and above, and the non-rooted version should work on releases from API level 14 (Android 4.0) and above.

goal,⁹ using LISPmob. Subsequently, a wide number of tests is made in order to estimate from experimental results some valuable LISP performance parameters, and then to draw forth how real is the chance of using it as an effective multihoming solution for the final user. The reference version is the `noroot` package, in accordance with what said in section 4.1.1. More specifically, the project consists in:

Development phase: Implementing, if possible, a solution in order to support multiple EIDs over the same network interface. It consists in the following sub-tasks:

1. Download LISPmob `noroot` from the Play Store;
2. Use LISPmob `noroot` and start the `lispd`;
3. Select a random active EID;
4. Check that the IP traffic is LISP-tunnelled \mapsto *Goal #1*.
5. Fork the LISPmob master repository from `GitHub.com`;
6. Study and analyze the `noroot` package code;
7. Edit the Java classes in order to support multiple EIDs;
8. Build the whole project;
9. Test the new features \mapsto *Goal #2*.

Goal #1: Check the IPv4-to-IPv4 LISP tunnelling (*success* ✓ or *fail* ✗).

Goal #2: Support multiple EIDs (*success* ✓ or *fail* ✗).

Test phase: The test phase consists of two distinct parts:

Seamless roaming: Testing seamless roaming from WiFi to 3G networks, and vice versa. It consists in:

1. Use LISPmob `noroot` with both WiFi and 3G enabled;
2. Select and start the download of a remote file of adequate size;
3. Increase the distance between the device and the WiFi Access Point (AP) until the connection is lost or manually switch off the WiFi interface;
4. Check if the download fails or slows down;
5. Re-enable the WiFi interface and reconnect to the WiFi AP.

⁹See “end-site multihoming”, in section 1.1.1 for details.

6. Check if the download fails or slows down.

Goal: Seamless roaming (*success* ✓ or *fail* ✗).

LISP performances: Testing LISP performances in comparison with standard IPv4. It consists in:

1. Choose a valuable set of n active EIDs;
2. Select a new EID;
3. Ping the selected EID m times;
4. Save the data;
5. Start the `lispd` on the device;
6. Ping the selected EID m times;
7. Stop the `lispd` on the device;
8. Save the data, and again \circlearrowright 2.

Goal: Obtain all the ping times (*success* ✓ or *fail* ✗).

Analisis phase: Analisis of the collected data and synthesis of the results.

It consists in:

1. Import all collected data in Microsoft Excel;
2. Calculate the most important statistical parameters;
3. Draw valuable charts spotting out LISP performances;
4. Make evaluations and considerations about data.

Goal: Estimate the LISP delay and relative speed difference.

4.4.2 Project Testbed

The testbed for the project is the following:

Development platform: AMD Athlon II X2 240 2.81 GHz, RAM 4 GB, Samsung HD105SI SATA-II 1 TB, Ubuntu 14.10.

Android testing device: *Sony Xperia go (ST27i)*, Cortex-A9 1 GHz, RAM 512 MB, HD Device 2 GB + Internal 4 GB, WiFi 802.11 b/g/n, 3G (HSDPA 14.4 Mb/s, HSUPA 5.76 Mb/s), Android 4.1.2, LISPmob 4.0.1.

ISP: WiFi Fastweb, 3G Wind Telecommunications.

Development IDE: Eclipse Kepler SR2, SDK 20 (Android 4.4W).

Networking tools: Ping & DNS 2.3/86 (Android), Wireshark 1.12.2.

Analysis software: Microsoft Excel 2013.

Chapter 5

Implementation Insights on LISPmob and Experimental Results

5.1 Development phase

The main goal of the development phase is implementing, if possible, a solution in order to support multiple EIDs over the same network interface. This may be considered a feasible implementation of the *end-host multihoming*, as explained in section 1.1.1. The alternative option—that is enabling two or more network interfaces at once with the same EID, which may be considered an implementation of the *end-site multihoming*—also has taken in consideration, but it has been set aside due the Android’s kernel limitations explained in section 3.4.

5.1.1 Configuration and first run

The first step was the download and install of LISPmob `noroot` from the Play Store. This was accomplished very easily, without any problem.

The second step, quite obviously, was to start using LISPmob. In order to do that, was necessary to first accomplish two sub-tasks: *a)* setup of the

`lispd` configuration file; and *b*) run the `lispd`.

The `lispd.conf` setup can be easily done using the proper activity, the `updateConfActivity` (button `Edit LISP Configuration` of the Main Activity): at this point all the parameters given for the beta network¹ were properly set up. The resulting `lisp.conf` is included in Appendix C.

The run of the `lispd` rised the following exception:

```
java.lang.ExceptionInInitializerError
at org.lispmob.noroot.LISPmobVPNService.run(LISPmobVPNService.java:123)
at java.lang.Thread.run(Thread.java:856)
Caused by: java.lang.UnsatisfiedLinkError: Cannot load library:
link_image[1891]: 1416 could not load needed library 'libjson-c.so' for
'liblispd.so' (load_library[1093]: Library 'libjson-c.so' not found)
at java.lang.Runtime.loadLibrary(Runtime.java:370)
at java.lang.System.loadLibrary(System.java: 535)
at org.lispmob.noroot.LISPmob_JNI.<clinit>(LISPmob_JNI.java:12)
```

and then LISPmob fatally crashed.

At this point any significant use of LISPmob was denied, so the priority was to fix the error.

5.1.2 Error fixing

The application error was an `ExceptionInInitializerError`, raised by the `LISPmobVPNService` while running (starting) the `lispd` thread. As reported by the stack trace², the problem was the loading of the `libjson-c.so` library which is needed to start the `lispd`. More in detail, the `UnsatisfiedLinkError` in Android is distinctive of problems occurred throughout the Java and native code interactions. Indeed, in this case, the `libjson-c.so` was not found because not loaded by the proper entity, that is the `LISPmob_JNI` interface, as explained in section 4.2.5.

Hence, once that the problem was localized, I edited the `LISPmob_JNI` as following:

```
1 package org.lispmob.noroot;
2
3 public class LISPmob_JNI {
```

¹See section 4.3.1 for the complete list.

²See section 5.1.1


```
4
5 public static native int[] startLispd(int tunFD, String storage_path);
6
7 public static native void lispd_loop();
8
9 public static native void lispd_exit();
10
11 static {
12     System.loadLibrary("json-c");
13     System.loadLibrary("lispd");
14 }
15 }
```

Moreover, as a direct consequence, the `Android.mk`'s code section related to local and shared libraries changed as following:

```
55 LOCAL_STATIC_LIBRARIES := libconfuse libjson-c
56 LOCAL_SHARED_LIBRARIES := libcutils
57 LOCAL_STATIC_LIBRARIES := libconfuse
58 LOCAL_SHARED_LIBRARIES := libcutils libjson-c
59 LOCAL_MODULE = lispd
60 #include $(BUILD_EXECUTABLE)
61 include $(BUILD_SHARED_LIBRARY)
```

These edits constitute the main changes introduced with LISPmob (version 0.4.1.6) released in December, 20 2014 on [GitHub.com](https://github.com) by the software maintainers.

5.1.3 Building and second run

The build of the fixed code followed the procedure described in section 4.3.3. The resulted APK was then copied into the device and installed with the Android Application Manager. The configuration file was not changed.

The run of the `lispd` made Android asking the user about the trusting of the VPN set up by LISPmob. Then the LISP tunnel was up and debug information was available in real time through the Main Activity. After the successful registration of the EID to the Map Server,³ all the IP traffic was effectively routed over the `tun0` LISP tunnel interface.

In order to confirm that, a ping from the device to a random host showed the following (ping details omitted):

³See section 2.4 for the related theory.


```

151         if (tmp.length < 2)
152             continue;
153         String[] tmp_1 = tmp[1].split("/");
154         if (tmp_1.length < 2)
155             continue;
156         if (tmp_1[0].contains(":")){
157             eidIPv6 = tmp_1[0];
158             EditText e = (EditText) findViewById(R.id.updateConfeid6Text);
159             e.setText(eidIPv6);
160         }else if (tmp_1[0].contains(".")){
161             eidIPv4 = tmp_1[0];
162             EditText e = (EditText) findViewById(R.id.updateConfeid4Text);
163             e.setText(eidIPv4);
164         }
165     }
166     if (sub_line.contains("interface")){
167         String[] tmp = sub_line.split("=");
168         if (tmp.length < 2)
169             continue;
170         String iface_name = tmp[1];
171
172         Iterator <String>iface_it = iface_list.iterator();
173         while (iface_it.hasNext())
174         {
175             if (iface_it.next().equals(iface_name)){
176                 if (!ifaces.contains(iface_name)){
177                     ifaces.add(iface_name);
178                 }
179                 break;
180             }
181         }
182     }
183 } // ...

```

And later on, the createConfFile():

```

472     if (ifaces != null){
473         Iterator <String> it = ifaces.iterator();
474         while (it.hasNext()){
475             String iface_name = it.next();
476
477             if (!eidIPv4.equals("")){
478                 defText= defText.concat("database-mapping {\n")
479                 .concat("     eid-prefix     = "+eidIPv4+"/32\n")
480                 .concat("     interface     = "+iface_name+"\n")
481                 .concat("     priority_v4   = 1\n")
482                 .concat("     weight_v4     = 100\n")
483                 .concat("     priority_v6   = 1\n")
484                 .concat("     weight_v6     = 100\n")
485                 .concat("}\n\n");

```

```
486     }
487     if (!eidIPv6.equals("")){
488         defText= defText.concat("database-mapping {\n")
489         .concat("         eid-prefix      = "+eidIPv6+"/128\n")
490         .concat("         interface     = "+iface_name+"\n")
491         .concat("         priority_v4   = 1\n")
492         .concat("         weight_v4     = 100\n")
493         .concat("         priority_v6   = 1\n")
494         .concat("         weight_v6     = 100\n")
495         .concat("}\n\n");
496     }
497 }
498 } // ...
```

The code is written to support one EID IPv4 and one EID IPv6 only, eventually over one or more interfaces. That is not a developer's project choice but a mere consequence of the limits imposed by the beta-network infrastructure. Indeed, tests made by the LISPmob maintainers highlighted that multi-EID is not well supported on the NAT traversal infrastructure of the beta-network, and since a MN will be likely behind a NAT, that is a strong drawback. However, this is not an architecture flaw and the infrastructure could—and hopefully shortly will—be easily updated to support multiple EIDs while doing NAT traversal operations, but at this moment that constitutes an insurmountable limit.

For the reason explained so far, editing the `updateConfActivity` in order to support multiple EIDs on the same interface is just pointless. Hence, this goal of the developing phase *failed* ❌.

5.2 Test phase

The test phase consisted of two main tests: *a)* the seamless roaming from WiFi to 3G and from 3G to WiFi; and *b)* the ping of a wide set of EIDs from the device. In the following sections both are described in detail.

5.2.1 Seamless roaming

As a reminder, performing a **seamless roaming** means that users can utilize their mobile applications over two or more wireless networks without any no-

ticeable interruption as they move throughout the respective signal coverage areas.

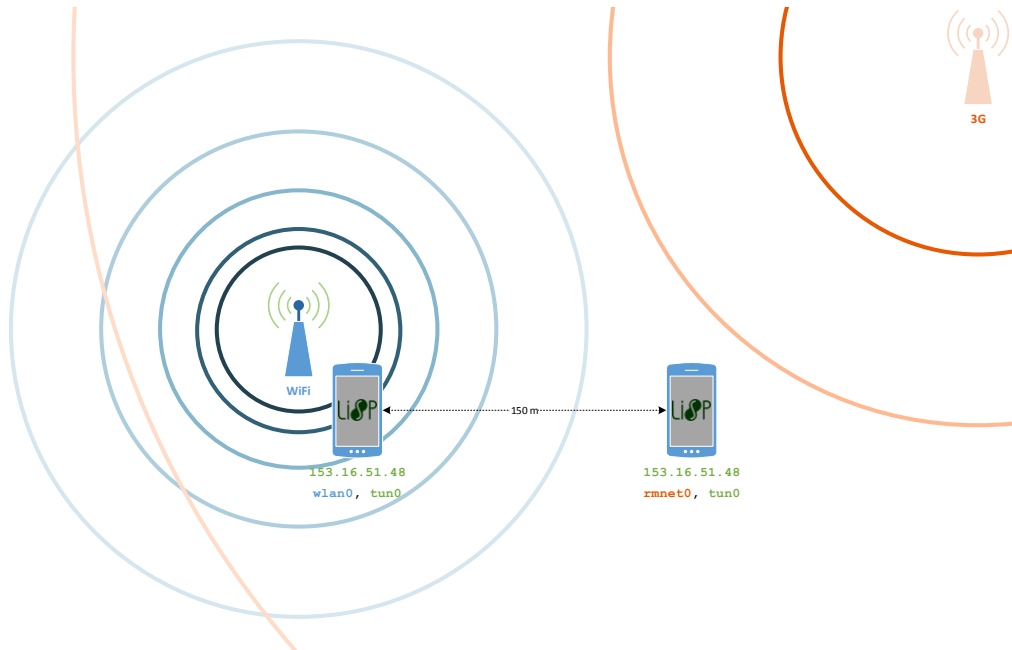


Figure 5.1: Seamless roaming

In other words, in a seamless roaming communication, a user should be able to make a Voice over IP (VoIP) call while walking through a city without experiencing drops or substantial hiccups in the conversation; or, an user that is downloading a file while driving should be able to do so in a timely manner without ever having to restart the download process. Nevertheless, roaming delays do not have always the same impact on different applications: a five or ten second roaming delay with a file download will likely go unnoticed, but that level of delay during a VoIP call will lead to a dropped call and annoyed users. Thus, the given application in use over the wireless networks impacts on how the term “*seamless*” should be interpreted.

The test-plan expects to enable both the network interfaces on the device (enabling the WiFi and waiting the successful connection to the AP first, then also the 3G interface is enabled). Then start LISPmob and run the `lispd` while browsing Internet, a remote file of about 100 MB is selected for the purpose.

Hence the file download is started. Since the LISPmob VPN is up, all the incoming and outgoing data traffic is tunneled over the `tun0` interface. I made the choice to overstep the AP's range, so the device was brought at a distance of about 150 m,⁴ so when the WiFi connection was lost, the 3G effectively replaced it without any stop or slowdown of the file downloading.

Later on, the device was brought again in the AP's range so the WiFi soon replaced the 3G and also in this case there was no impact on the file downloading, which after some more seconds was correctly finalized indeed.⁵ This was the proof I was investigating on, so the seamless roaming test was accomplished and *successful* ✓.⁶

5.2.2 LISP performance

This test was made to compare LISP-MN performance using LISPmob on our Android device with standard IP. More specifically, using LISPmob, I collected the ping times towards a set of active EIDs via LISP IPv4-in-IPv4 tunneling; then, switching off LISPmob, I also collected the respective ping times via standard IPv4. The list of selected active EIDs has been retrieved on the beta-network's website and is included in Appendix D. The selection process reduced the initial number of 81 EIDs to the definitive 53 active EIDs. An EID has been considered "active" if pingable with success at least one time both using LISP IPv4-in-IPv4 *and* IPv4.

The collected results are summarized and reported in the table 5.1 by the three most important statistical parameters—mean value, standard deviation and percentage of packet loss.

⁴At a distance of 120 m the WiFi's signal was absent but the connection still up. At this point Android did not switch to the 3G automatically: until the WiFi connection is completely lost the WiFi interface is *always* preferred. As expected, a considerable slowdown of the download due the WiFi's signal reduction was registered at this stage.

⁵The test was also repeated switching off/on the WiFi interface manually. In this case the overall download speed was almost constant during the whole downloading process.

⁶The exact values of the download speed were impossible to track due the lack of specific testing equipment. Nevertheless, the scientific validity of the results is assured by the reproducibility of the experiment.

Thus, the whole test phase was *successful* ✓.

5.3 Analysis phase

Here all the results collected in the testing phase are fully analyzed. From the data recorded in the LISP performance test it is possible to extract some valuable information as statistical parameters.

The starting point is represented by 2 sets of $k = 53$ active EIDs. For each active EID we collected $N = 100$ ping times. So we eventually have k average values related to the standard IPv4, and k related to the LISP IPv4-in-IPv4, for a total amount of 10600 ping samples. For our purpose, each set may be considered as a discrete random variable.

So I define the general **sample mean vector** $\bar{\mathbf{h}}$ as a column vector whose j -th element \bar{h}_j is the average value of the N observations of the j -th variable:

$$\bar{h}_j = \frac{1}{N} \sum_{i=1}^N h_{i,j} \quad (5.1)$$

where $j = \{1, 2, \dots, k\}$.

Thus, I define $\bar{\mathbf{x}}$ as the vector which contains the average of all the observations for each variable related to the standard IPv4, and $\bar{\mathbf{y}}$ as the one related to the LISP IPv4-in-IPv4:

$$\bar{\mathbf{x}} = \begin{bmatrix} x_1 \\ \vdots \\ x_j \\ \vdots \\ x_k \end{bmatrix}, \quad \bar{\mathbf{y}} = \begin{bmatrix} y_1 \\ \vdots \\ y_j \\ \vdots \\ y_k \end{bmatrix}. \quad (5.2)$$

5.3.1 Delay

As I have $\bar{\mathbf{x}}$ and $\bar{\mathbf{y}}$ which are univariate discrete random variables, it is possible to know their expected values $\mu_{\bar{\mathbf{x}}}$ and $\mu_{\bar{\mathbf{y}}}$ (that are the arithmetic means), by the general formula:

$$\mu_{\bar{\mathbf{h}}} = \frac{1}{N} \sum_{j=1}^k \bar{h}_j \quad (5.3)$$

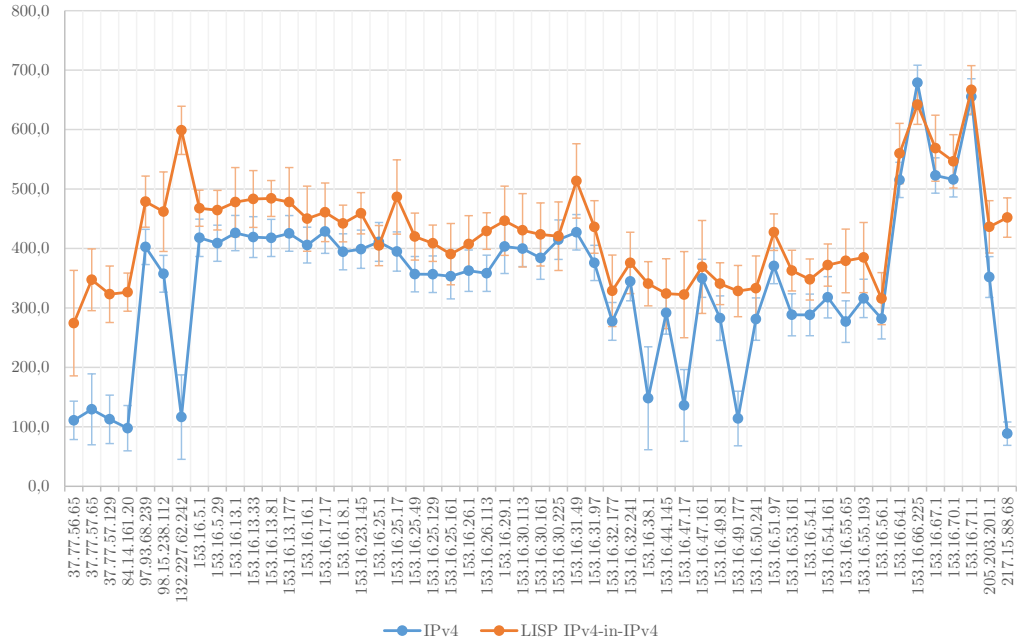


Figure 5.2: Plot of \bar{x} and \bar{y}

and their standard deviations $\sigma_{\bar{x}}$ and $\sigma_{\bar{y}}$, by the general formula:

$$\sigma_{\bar{h}} = \sqrt{\frac{1}{N} \sum_{j=1}^k (\bar{h}_j - \mu_{\bar{h}})^2} \quad (5.4)$$

obviously, again, where $j = \{1, 2, \dots, k\}$.

The results are the ones summarized in table 5.1, but these parameters alone give us poor information—I need also the **difference of sample means vector** $\bar{\Delta}$ as a column vector whose j -th element $\bar{\Delta}_j$ is the difference

	μ (ms)	σ (ms)	λ (%)
\bar{x}	342.7	130.1	1.2
\bar{y}	426.8	84.6	7.0
$\bar{\Delta}$	84.1	89.7	

Table 5.1: Comparison of IPv4 and LISP IPv4-in-IPv4 performance

\bar{x} : IPv4, \bar{y} : LISP IPv4-in-IPv4.

μ : mean value, σ : standard deviation, λ : percentage of packet loss

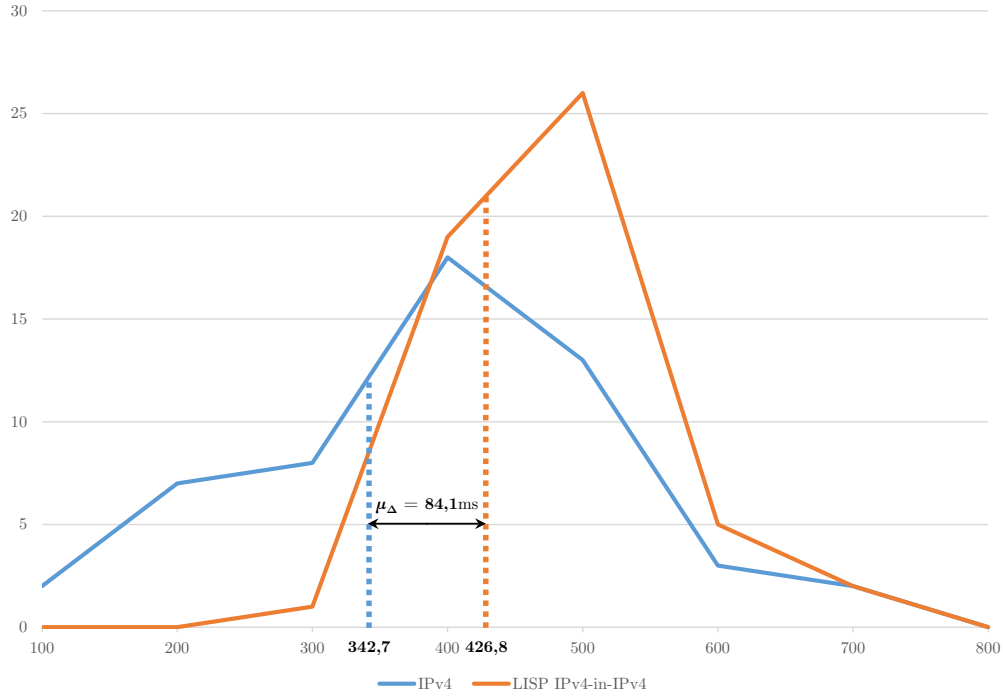


Figure 5.3: Frequency Distribution of \bar{x} and \bar{y}

value between y_j and x_j :

$$\bar{\Delta} = \begin{bmatrix} y_1 - x_1 \\ \vdots \\ y_j - x_j \\ \vdots \\ y_k - x_k \end{bmatrix} . \quad (5.5)$$

For it I calculate the expected value $\mu_{\bar{\Delta}}$ (that is again the arithmetic mean), and the standard deviation $\sigma_{\bar{\Delta}}$. The results are summarized in table 5.1.

These parameters are very important indeed. In particular the mean value $\mu_{\bar{\Delta}}$ represents exactly the **LISP delay** as the average time difference between using LISP or using it not, that is 84.1 ms. It is now possible to plot the frequency distribution chart of \bar{x} and \bar{y} (figure 5.3). It shows the comparison between IPv4 and LISP IPv4-in-IPv4: the average LISP delay is indeed 84.1 ms, that is a result which is fully in accordance with what expected.

Reasons behind the results

The reasons of the LISP delay are various. Figure 5.4 graphically shows the composition of the ping packets used in the test phase.

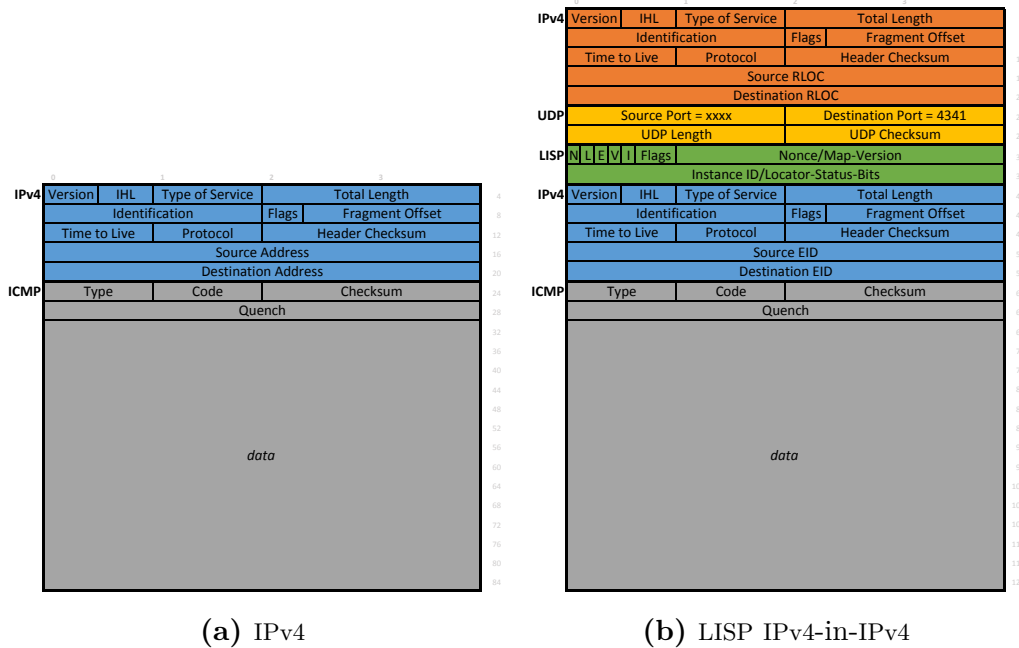


Figure 5.4: Composition of ping packets

In IPv4, the inner ICMP packet is 64B, while the IPv4 header is 20B, for a total size of 84B. In LISP, instead, the tunneling requires a change into the IPv4 header where the IP addresses are replaced with the source and destination EIDs, and moreover there is an overhead which is composed of 8B of LISP header, 8B of UDP header, and other 20B of IPv4 header where the IP addresses are the source and destination RLOCs, for a total size of 120B.

Obviously this overhead itself involves an increasing in the network latency, but also the encapsulation and decapsulation of the traffic require a time which may be short but cannot be null. On the other hand, the RLOCs' improved handling of routing tables and the consequent optimized paths, plus other fine-tuning techniques into the RLOC space, will considerably reduce the network latency.

5.3.2 Speed

Said that, it may be interesting to evaluate other parameters like the average traffic speeds $v_{\bar{x}}$ and $v_{\bar{y}}$, defined by the general formula:

$$v_{\alpha, \bar{h}} = \frac{q_{\alpha, \bar{h}}}{\mu_{\bar{h}}} \quad (5.6)$$

where $q_{\bar{h}}$ is the size of the single ping packet in terms of α which expresses if is considered the whole data “*in-the-wire*” (w) or only the payload (p). The speeds are then expressed in B/s.

	w (B)	v_w (B/s)	p (B)	v_p (B/s)
\bar{x}	84	245.1	54	157.6
\bar{y}	120	281.2	54	126.5

Table 5.2: IPv4 versus LISP speeds

\bar{x} : IPv4, \bar{y} : LISP IPv4-in-IPv4.

w : packet size “*in-the-wire*”, p : payload, v : average speed

The results summarized in table 5.2 are quite interesting: even if there is a considerable delay between LISP and standard IP as said in section 5.3.1, the speed of the whole data “*in-the-wire*” using LISP is greater. Instead, if the speed is considered in terms of the payload which is what the users experience, LISP’s speed performance suffer a **relative difference of -19.71%** .⁷

5.3.3 Precision

Another important consideration to make is about relative standard error and deviation. LISP is a very new technology in computer networks, but it has a very high potential. In fact, defined the standard error $SE_{\mu_{\bar{h}}}$, the

⁷The relative difference is defined as:

$$d_r = \frac{v_{p, \bar{y}} - v_{p, \bar{x}}}{\max(|v_{p, \bar{x}}|, |v_{p, \bar{y}}|)}.$$

relative standard $\mathbf{RSE}_{\mu_{\bar{h}}}$, and the relative standard deviation $\mathbf{RSD}_{\bar{h}}$ by the general formulas:

$$\mathbf{SE}_{\mu_{\bar{h}}} = \frac{\sigma_{\bar{h}}}{\sqrt{k}} \quad ; \quad \mathbf{RSE}_{\mu_{\bar{h}}} = \frac{\mathbf{SE}_{\mu_{\bar{h}}}}{\mu_{\bar{h}}} \quad ; \quad \mathbf{RSD}_{\bar{h}} = \frac{\sigma_{\bar{h}}}{\mu_{\bar{h}}} \quad ; \quad (5.7)$$

the obtained results are summarized in table 5.3.

	\mathbf{SE}_{μ} (ms)	\mathbf{RSE}_{μ} (%)	\mathbf{RSD} (%)
\bar{x}	17.9	5.2	38.0
\bar{y}	11.6	2.7	19.8

Table 5.3: IPv4 versus LISP statistics

\mathbf{SE}_{μ} : standard error, \mathbf{RSE}_{μ} : relative standard error,
 \mathbf{RSD} : relative standard deviation

Therefore from this I discovered that LISP has almost the *half* of the relative standard deviation in comparison with IP, and this is a very valuable result: it means that communications over LISP tunneling are definitely very stable and this may have obvious good consequences in terms of channel throughput.

Nevertheless, probably due also to the very recent release of the exploited LISP implementation, LISP has shown to be affected by 7.0 % of packet loss⁸, which is undoubtedly non-negligible for almost all Internet applications. It must be said, however, that in this experiments (and in the results reported above) I have considered as lost the packets with ping times greater than 800.0 ms, as reasonable in many delay-sensitive applications. Furthermore, it must be taken into account that this value refers to ICMP ping packets exchanged without supporting/implementing any additional error-recovery strategy: hence the reported results can be considered the lowest upper bound. Still, this LISP under-performance of 7.0 % is too far from the most usual IP 1.2 % to take into consideration the adoption of LISP on a world-wide scale very soon.

⁸See table 5.1.

Conclusions and future work

Despite years of research and development, multihoming is yet to become widespread in network deployments. Indeed, the corresponding support is often missing from state-of-the-art protocols. Where Linux may take advantage of enabling the SCTP support in its kernel, Android due its limitations must find more complex solutions in order to provide multihoming. In recent history, many protocols have been proposed with the purpose of assert themselves as acceptable solutions, but in truth, all with inadequate outcomes because of their incomplete and unoptimized implementations.

Sponsored by Cisco Systems, LISP is born to solve a number of problems risen up all together after decades of paralysis of the Internet infrastructure's update and a very slow and wearying transition to IPv6, which in fact is still ongoing. HIP as well was proposed almost a decade ago but now seems almost abandoned for its implementation difficulties, while IETF endorsed ILNP as the recommended choice for the future routing architecture, and so the resulting effect is that no significant steps have been done in order to give multihoming to final users yet.

So my goal was to describe what is the state-of-the-art multihoming support for Android and to make some evaluations on what is effectively available and usable now by the Android's users, without requiring the device rooting. This choice has been taken primarily because expecting the device rooting from average users is quite unrealistic, and also because one of the established prerequisites is the non-needing of modifications of the host devices.

After the developing phase in which I fixed the error responsible of the crash of the whole LISPmob application, the test phase has been a long as

well as engaging turnover of emotions while the theory were matching with the collecting data. And the final results are very interesting and promising indeed. The analysis phase shows that the comparison of LISP in MN mode with standard IP points out a noticeable difference in terms of time delay and a too high rate of packet loss. LISP values are too far from the IP ones to consider a world-wide use of LISPmob right now: their reasons are many and various, but there are also some very good points in using LISP. Mobility and multihoming are obviously the most important ones, but resilience, ubiquity, load balancing/sharing, scalability and policy are coveted features as well. In addition, I prove that LISP IPv4-in-IPv4 tunneled traffic is very stable in comparison with standard IP.

This thesis seems to be the first attempt ever to compare what might be the main technology of the future Internet with the actual one through real data, and I feel proud for it. At the same time I recognize that it is only a tiny piece of the puzzle and still there are many aspect to consider and deepen: *a)* I tested the LISP IPv4-in-IPv4 but it would be very important to test the IPv6-in-IPv6 case and the others hybrid combinations; *b)* My research is focused on LISP-MN in order to test the performance in a mobility and multihomed environment, but when they are unnecessary, the advantages offered by PxTRs' optimized paths should be tested too; *c)* Rooting or not rooting? I chose to not root the device for the reasons explained in section 4.1.1, but also the rooted version should be tested in depth—in this case in fact I would expect much better performance, so it would be very interesting to know how wide is the gap with IP.

Finally, the future outlooks for a world-wide use of LISP depend by many factors, including the true willing of the ISPs about investing in the Internet infrastructure modernisation. But actually as it is known, this is determined by the potential revenue which requires far-sightedness and remarkable investments, so it is just the well-known “dog chasing its own tail”. Otherwise LISP will remain a wonderful research project with no practical use for no-one, like many others before it. Thus, in my opinion, more than any quick and significant improvement of its performance, LISP's success will be determined almost exclusively by market logic.

Appendix A

Android Connectivity Features Evolution

Here are presented in chronological order all Android releases and their main connectivity features.

Android 1.0	API level 1
September 23, 2008	Linux kernel 2.6.25
<ul style="list-style-type: none">• Web browser to show, zoom and pan full HTML and XHTML web pages;• Access to web email servers, supporting POP3, IMAP4, and SMTP;• Google Sync, allowing management of over-the-air synchronization of Gmail, People, and Calendar;• Instant messaging, text messaging, and MMS;• WiFi and Bluetooth support.	

Android 1.1	API level 2
February 9, 2009	Linux kernel 2.6.25
<ul style="list-style-type: none">• —	

Android 1.5 Cupcake	API level 3
April 27, 2009	Linux kernel 2.6.27
<ul style="list-style-type: none"> • Auto-pairing and stereo support for Bluetooth; • Copy and paste features in web browser; 	
Android 1.6 Donut	API level 4
September 15, 2009	Linux kernel 2.6.29
<ul style="list-style-type: none"> • Updated technology support for CDMA/EVDO, 802.1x, VPNs, and a text-to-speech engine; 	
Android 2.0 Eclair	API level 5
October 26, 2009	Linux kernel 2.6.29
<ul style="list-style-type: none"> • Expanded Account sync, allowing users to add multiple accounts to a device for synchronization of email and contacts; • Microsoft Exchange email support, with combined inbox to browse email from multiple accounts in one page; • Bluetooth 2.1 support; • Ability to tap a Contacts photo and select to call, SMS, or email the person; • Refreshed browser UI with bookmark thumbnails, double-tap zoom and support for HTML5; 	
Android 2.0.1 Eclair	API level 6
December 3, 2009	Linux kernel 2.6.29
<ul style="list-style-type: none"> • — 	
Android 2.1 Eclair	API level 7
January 12, 2010	Linux kernel 2.6.29
<ul style="list-style-type: none"> • — 	

Android 2.2 Froyo	API level 8
May 20, 2010	Linux kernel 2.6.32
<ul style="list-style-type: none">• Integration of Chrome's V8 JavaScript engine into the browser application;• Support for the Android C2DM service, enabling push notifications;• Improved Microsoft Exchange support, security policies;• Auto-discovery, GAL look-up;• Improved Calendar synchronization and remote wipe;• USB tethering and WiFi hotspot functionality;• Option to disable data access over mobile network;• Support for Bluetooth-enabled car and desk docks;• Support for file upload fields in the Browser application;• Adobe Flash support;	

Android 2.2.1 Froyo	API level 8
January 18, 2011	Linux kernel 2.6.32
<ul style="list-style-type: none">• —	

Android 2.2.2 Froyo	API level 8
January 22, 2011	Linux kernel 2.6.32
<ul style="list-style-type: none">• Minor bug fixes, including SMS routing issues that affected the Nexus One.	

Android 2.2.3 Froyo	API level 8
November 21, 2011	Linux kernel 2.6.32
<ul style="list-style-type: none">• Two security patches.	

Android 2.3 Gingerbread	API level 9
December 6, 2010	Linux kernel 2.6.35
<ul style="list-style-type: none"> • Native support for SIP VoIP internet telephony; • Support for NFC; • New Download Manager, giving users easy access to any file downloaded from the browser, email, or another application; • Enhanced support for native code development; • Switched from YAFFS to ext4 on newer devices; • Native support for more sensors (such as gyroscopes and barometers). 	

Android 2.3.1 Gingerbread	API level 9
December, 2010	Linux kernel 2.6.35
<ul style="list-style-type: none"> • — 	

Android 2.3.2 Gingerbread	API level 9
January, 2011	Linux kernel 2.6.35
<ul style="list-style-type: none"> • — 	

Android 2.3.3 Gingerbread	API level 10
February 9, 2011	Linux kernel 2.6.35
<ul style="list-style-type: none"> • — 	

Android 2.3.4 Gingerbread	API level 10
April 28, 2011	Linux kernel 2.6.35
<ul style="list-style-type: none"> • Support for voice or video chat using Google Talk; • OAL support; • Switched the default encryption for SSL from AES256-SHA to RC4-MD5. 	

Android 2.3.5 Gingerbread	API level 10
July 25, 2011	Linux kernel 2.6.35
<ul style="list-style-type: none">• Improved network performance for the Nexus S 4G, among other fixes and improvements;• Fixed Bluetooth bug on Samsung Galaxy S;• Improved Gmail application;	

Android 2.3.6 Gingerbread	API level 10
September 2, 2011	Linux kernel 2.6.35
<ul style="list-style-type: none">• —	

Android 2.3.7 Gingerbread	API level 10
September 21, 2011	Linux kernel 2.6.35
<ul style="list-style-type: none">• Fixed the side-effect of impairing the WiFi hotspot functionality of many Canadian Nexus S phones.	

Android 3.0 Honeycomb	API level 11
February 22, 2011	Linux kernel 2.6.36
<ul style="list-style-type: none">• Multiple browser tabs replacing browser windows, plus form auto-fill and a new “incognito” mode allowing anonymous browsing;• New two-pane Email UI to make viewing and organizing messages more efficient, allowing users to select one or more messages;• Ability to encrypt all user data;• HTTPS stack improved with SNI;• FUSE kernel module;	

Android 3.1 Honeycomb	API level 12
May 10, 2011	Linux kernel 2.6.36
<ul style="list-style-type: none"> • USB On-The-Go; • High-performance WiFi lock, maintaining high-performance WiFi connections when device screen is off; • Support for HTTP proxy for each connected WiFi access point. 	
Android 3.2 Honeycomb	API level 13
July 15, 2011	Linux kernel 2.6.36
<ul style="list-style-type: none"> • — 	
Android 3.2.1 Honeycomb	API level 13
August 30, 2011	Linux kernel 2.6.36
<ul style="list-style-type: none"> • Bug fixes and minor security, stability and WiFi improvements; • Improved Adobe Flash support in browser; 	
Android 3.2.2 Honeycomb	API level 13
September 20, 2011	Linux kernel 2.6.36
<ul style="list-style-type: none"> • — 	
Android 3.2.3 Honeycomb	API level 13
	Linux kernel 2.6.36
<ul style="list-style-type: none"> • — 	
Android 3.2.4 Honeycomb	API level 13
December, 2011	Linux kernel 2.6.36
<ul style="list-style-type: none"> • “Pay as You Go” support for 3G and 4G tablets. 	

Android 3.2.5 Honeycomb	API level 13
January, 2012	Linux kernel 2.6.36
<ul style="list-style-type: none"> — 	

Android 3.2.6 Honeycomb	API level 13
February, 2012	Linux kernel 2.6.36
<ul style="list-style-type: none"> Fixed data connectivity issues when coming out of airplane mode on the US 4G Motorola Xoom. 	

Android 4.0 Ice Cream Sandwich	API level 14
October 18, 2011	Linux kernel 3.0.1
<ul style="list-style-type: none"> Improved visual voicemail with the ability to speed up or slow down voicemail messages; Better voice integration and continuous, real-time speech to text dictation; Android Beam, a NFC feature allowing the rapid short-range exchange of web bookmarks, contact info, directions, YouTube videos and other data; WiFi Direct; AVF, and TUN (but not TAP) kernel module. Prior to 4.0, VPN software required rooted Android. 	

Android 4.0.1 Ice Cream Sandwich	API level 14
October 21, 2011	Linux kernel 3.0.1
<ul style="list-style-type: none"> — 	

Android 4.0.2 Ice Cream Sandwich	API level 14
November 28, 2011	Linux kernel 3.0.1
<ul style="list-style-type: none"> — 	

Android 4.0.3 Ice Cream Sandwich	API level 15
December 16, 2011	Linux kernel 3.0.1
<ul style="list-style-type: none"> • Social stream API in the Contacts provider; 	

Android 4.0.4 Ice Cream Sandwich	API level 15
March 29, 2012	Linux kernel 3.0.1
<ul style="list-style-type: none"> • Improved phone number recognition. 	

Android 4.1 Jelly Bean	API level 16
July 9, 2012	Linux kernel 3.0.31
<ul style="list-style-type: none"> • Bluetooth data transfer for Android Beam; • USB audio for external sound DACs. 	

Android 4.1.1 Jelly Bean	API level 16
July 11, 2012	Linux kernel 3.0.31
<ul style="list-style-type: none"> • — 	

Android 4.1.2 Jelly Bean	API level 16
October 9, 2012	Linux kernel 3.0.31
<ul style="list-style-type: none"> • — 	

Android 4.2 Jelly Bean	API level 17
November 13, 2012	Linux kernel 3.4.0
<ul style="list-style-type: none"> • Rewritten Bluetooth stack, switching from Bluez to Broadcom open source BlueDroid, allowing improved support for multiple displays and wireless display (Miracast); • Native RtL, always-on VPN and application verification; • New NFC stack; • SELinux; • Premium SMS confirmation; • Group Messaging. 	

Android 4.2.1 Jelly Bean	API level 17
November 27, 2012	Linux kernel 3.4.0
<ul style="list-style-type: none"> • Added Bluetooth gamepads and joysticks as supported HID. 	

Android 4.2.2 Jelly Bean	API level 17
February 11, 2013	Linux kernel 3.4.0
<ul style="list-style-type: none"> • Fixed Bluetooth audio streaming bugs; • Long-pressing the WiFi and Bluetooth icons in Quick Settings now toggles the on/off state; • New sounds for wireless charging and low battery; 	

Android 4.3 Jelly Bean	API level 18
July 24, 2013	Linux kernel 3.4.39
<ul style="list-style-type: none"> • Bluetooth low energy support; • Bluetooth AVRCP 1.3 support; • Many security enhancements, performance enhancements, and bug fixes; • System-level support for geofencing and WiFi scanning APIs; • Background WiFi location still runs even when WiFi is turned off; 	

Android 4.3.1 Jelly Bean	API level 18
October 3, 2013	Linux kernel 3.4.39
<ul style="list-style-type: none"> • — 	
Android 4.4 KitKat	API level 19
October 21, 2013	Linux kernel 3.10
<ul style="list-style-type: none"> • Wireless printing capability; • NFC host card emulation, enabling a device to replace smart cards; • WebViews now based on Chromium engine; • Public API for developing and managing text messaging clients; • SAF, an API allowing apps to retrieve files in a consistent manner; • Sensor batching, step detector and counter APIs; • Native infrared blaster API • Bluetooth MAP support; • Settings application no longer uses a multi-pane layout on devices with larger screens; • WiFi and mobile data activity indicators are moved to quick settings; 	
Android 4.4.1 KitKat	API level 19
December 5, 2013	Linux kernel 3.10
<ul style="list-style-type: none"> • — 	
Android 4.4.2 KitKat	API level 19
December 9, 2013	Linux kernel 3.10
<ul style="list-style-type: none"> • Further security enhancements and bug fixes; 	
Android 4.4.3 KitKat	API level 19
June 2, 2014	Linux kernel 3.10
<ul style="list-style-type: none"> • HTML5 Canvas hardware acceleration performance improvements; • HTML5 form validation and datalist; 	

Android 4.4.4 KitKat	API level 19
June 19, 2014	Linux kernel 3.10
<ul style="list-style-type: none"> • Eliminated an OpenSSL MitM vulnerability. • VPN APIs bug fixes. 	
Android 4.4W KitKat	API level 20
June 25, 2014	Linux kernel 3.10
<ul style="list-style-type: none"> • Same as Android 4.4 KitKat, but with wearable extensions added. 	
Android 4.4W.1 KitKat	API level 20
September 6, 2014	Linux kernel 3.10
<ul style="list-style-type: none"> • — 	
Android 4.4W.2 KitKat	API level 20
October 21, 2014	Linux kernel 3.10
<ul style="list-style-type: none"> • GPS support. 	
Android 5.0 Lollipop	API level 21
November 3, 2014	Linux kernel 3.16.1
<ul style="list-style-type: none"> • Audio input and output through USB devices; 	
Android 5.0.1 Lollipop	API level 21
December 2, 2014	Linux kernel 3.16.1
<ul style="list-style-type: none"> • — 	
Android 5.0.2 Lollipop	API level 21
December 19, 2014	Linux kernel 3.16.1
<ul style="list-style-type: none"> • — 	

Android 5.1 Lollipop	API level 22
March 9, 2015	Linux kernel 3.16.1
<ul style="list-style-type: none"> • Ability to join WiFi networks and control paired Bluetooth devices from quick settings; 	
Android 5.1.1 Lollipop	API level 22
April 21, 2015	Linux kernel 3.16.1
<ul style="list-style-type: none"> • Native WiFi calling support. 	
Android 6.0 Marshmallow	API level 23
October 5, 2015	Linux kernel 3.18.10
<ul style="list-style-type: none"> • — 	
Android 6.0.1 Marshmallow	API level 23
December 7, 2015	Linux kernel 3.18.10
<ul style="list-style-type: none"> • — 	

Appendix B

The android.net package UML

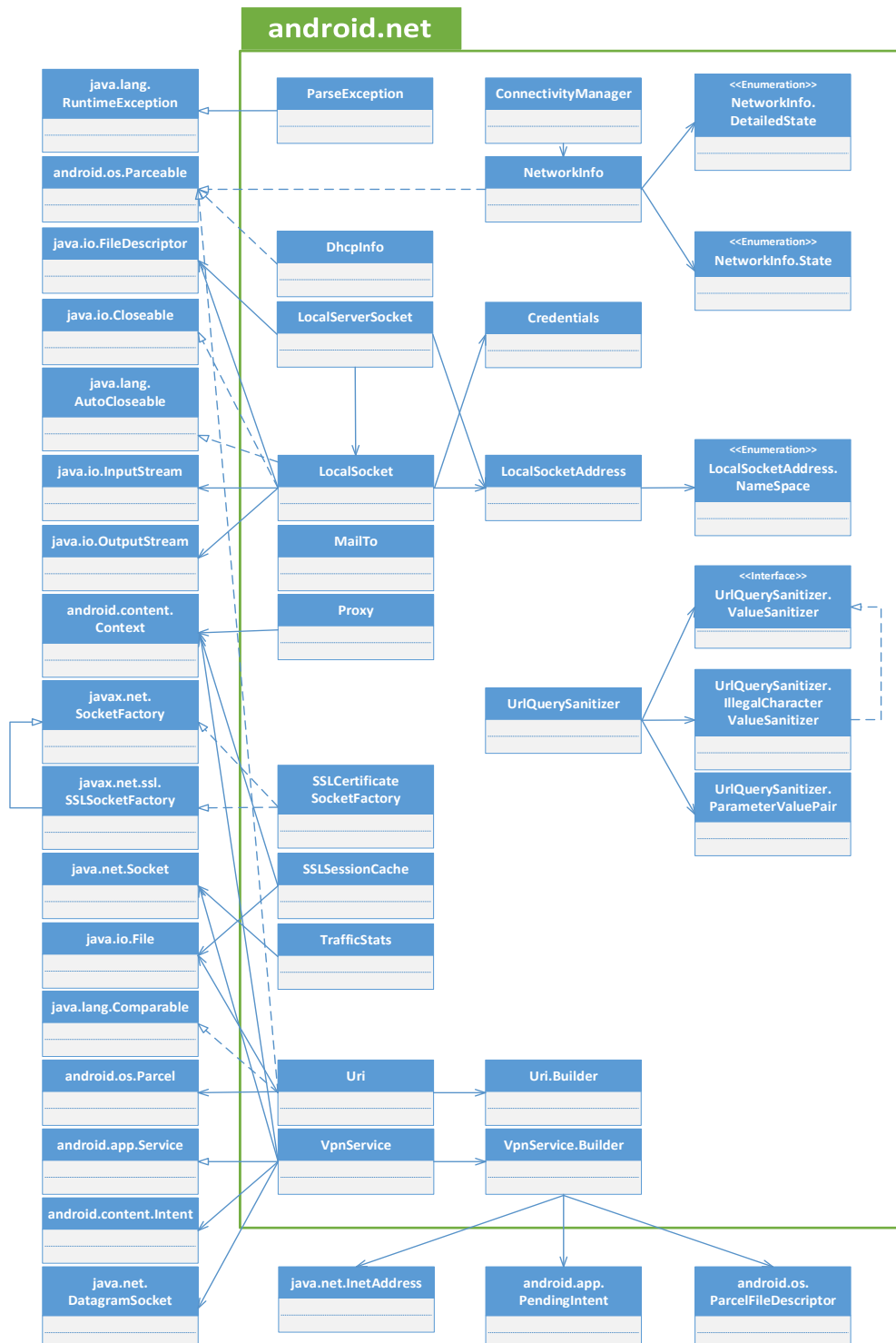


Figure B.1: The android.net package and relationships between its classes

Appendix C

LISPmob Configuration File (lispd.conf)

```
1 # General configuration
2 #     debug: Debug levels [0..3]
3 #     map-request-retries: Additional Map-Requests to send per map cache miss
4
5 router-mode          = off
6 debug               = 3
7 map-request-retries = 2
8
9 # RLOC Probing configuration
10 #   rloc-probe-interval: interval at which periodic RLOC probes are sent (seconds
11 #   ). A value of 0 disables RLOC Probing
12 #   rloc-probe-retries: RLOC Probe retries before setting the locator with status
13 #   down. [0..5]
14 #   rloc-probe-retries-interval: interval at which RLOC probes retries are sent (
15 #   seconds) [1..#rloc-probe-interval]
16
17 rloc-probing {
18     rloc-probe-interval          = 30
19     rloc-probe-retries           = 2
20     rloc-probe-retries-interval = 5
21 }
22
23 # NAT Traversal configuration.
24 #   nat_aware: check if the node is behind NAT
25 # Limitation of version 0.3.3 when nat_aware is enabled:
26 #   - Only one interface is supported.
27 #   - Only one Map Server and one Map Resolver
```

```
27
28 nat-traversal {
29     nat_aware    = true
30 }
31
32 # Encapsulated Map-Requests are sent to this map-resolver
33 # You can define several map-resolvers. Encapsulated Map-Request messages will be
34 #   sent to only one.
35 #   address: IPv4 or IPv6 address of the map resolver
36
37 map-resolver = {
38     217.8.98.46,
39 }
40
41 # DDT Client section.
42 # DDT configuration has preference over map-resolver configuration
43 #
44 #   ddt-client [on/off]: Obtain the mapping from EIDs to RLOCs through the DDT
45 #   tree
46 #ddt-client    = on
47
48 # DDT Encapsulated Map-Requests are sent to these ddt root node. You can define
49 #   several ddt-root-node. DDT Encapsulated Map-Request messages will be sent to
50 #   the ddt-root-node with higher priority (lowest value).
51 #
52 #   address: IPv4 or IPv6 address of the DDT root node
53 #   priority [0..255]: DDT root nodes with lower values are more preferable.
54 #   weight [0..255]: Not yet implemented
55
56 #ddt-root-node {
57 #     address    = <IPv4 or FQDN name>
58 #     priority   = 1
59 #     weight     = 100
60 #}
61
62 # Some LISP beta-network (lisp4.net/lisp6.net) DDT root nodes
63
64 ddt-root-node {
65     address    = 84.88.16.10
66     priority   = 1
67     weight     = 100
68 }
69
70 ddt-root-node {
71     address    = 2001:40B0:1::FOF2
72     priority   = 1
73     weight     = 100
74 }
75
```

```
72 # Current LISP beta-network (lisp4.net/lisp6.net) DDT root nodes:
73 # IPv4:
74 #     - csuc-ddt.rloc.lisp4.net: 84.88.16.10
75 #     - arin-ddt.rloc.lisp4.net: 192.149.252.136
76 #     - ripe-ddt.rloc.lisp4.net: 193.0.0.170
77 #     - vxnet-ddt.rloc.lisp4.net: 199.119.73.8
78 # IPv6:
79 #     - csuc-ddt.rloc.lisp6.net: 2001:40B0:1::FOF2
80 #     - arin-ddt.rloc.lisp6.net: 2001:500:4:12::5
81 #     - ripe-ddt.rloc.lisp6.net: 2001:610:240:5:193::170
82
83 # Map-Registers are sent to this map-server
84 # You can define several map-servers. Map-Register messages will be sent to all
      of them.
85 #   address: IPv4 or IPv6 address of the map-server
86 #   key-type: Only 1 supported (HMAC-SHA-1-96)
87 #   key: password to authenticate with the map-server
88 #   proxy-reply [on/off]: Configure map-server to Map-Reply on behalf of the xTR
89
90 map-server {
91     address      = 217.8.98.42
92     key-type     = 1
93     key         = natrav-test
94     proxy-reply = on
95 }
96
97 # Packets addressed to non-LISP sites will be encapsulated to this Proxy-ETR
98 # You can define several Proxy-ETR. Traffic will be balanced according to
      priority and weight.
99 #   address: IPv4 or IPv6 address of the Proxy-ETR
100 #   priority [0-255]: Proxy-ETR with lower values are more preferable.
101 #   weight [0-255]: When priorities are the same for multiple Proxy-ETRs, the
      Weight indicates how to balance unicast traffic between them.
102
103 proxy-etr {
104     address      = 217.8.98.33
105     priority     = 1
106     weight       = 100
107 }
108
109 # List of PITRs to SMR on handover
110 #   address: IPv4 or IPv6 address of the Proxy-ITR
111 #   Current LISP beta-network (lisp4.net/lisp6.net) Pitr addresses
112
113 proxy-itrs = {
114     69.31.31.98,
115     149.20.48.60,
116     198.6.255.37,
117     173.36.193.25,
```

```
118     129.250.1.63,
119     217.8.98.33,
120     217.8.98.35,
121     193.162.145.46,
122     193.34.30.222,
123     193.34.31.222,
124     147.83.131.33,
125     158.38.1.92,
126     203.181.249.172,
127     202.51.247.10
128 }
129
130 # IPv4 / IPv6 EID of the node.
131 # One database-mapping structure is defined for each interface with RLOCs
    associated to this EID
132 #   eid-prefix: EID prefspinner.setSelection(position);ix (IPvX/mask) of the
    mapping
133 #   interface: interface containing the RLOCs associated to this mapping
134 #   priority_vX [0-255]: Priority for the IPvX RLOC of the interface. Locators
    with lower values are more preferable. This is used for both incoming policy
    announcements and outgoing traffic policy management. (A value of -1 means
    that IPvX address of that interface is not used)
135 #   weight [0-255]: When priorities are the same for multiple RLOCs, the Weight
    indicates how to balance unicast traffic between them.
136
137 database-mapping {
138     eid-prefix      = 153.16.51.48/32
139     interface       = wlan0
140     priority_v4     = 1
141     weight_v4       = 100
142     priority_v6     = 1
143     weight_v6       = 100
144 }
145
146 database-mapping {
147     eid-prefix      = 153.16.51.48/32
148     interface       = rmnet0
149     priority_v4     = 1
150     weight_v4       = 100
151     priority_v6     = 1
152     weight_v6       = 100
153 }
154
155 override-dns       = true
156 override-dns-primary = 8.8.8.8
157 override-dns-secondary = 8.8.4.4
```


Appendix D

List of the selected active EIDs

37.77.56.65 gw.10ww.steffann.nl
37.77.57.65 gw.emile.sintact.nl
37.77.57.129 br.sitecom.lispnet.nl
84.14.161.20 20.161-14-84.ripe.coltfrance.com
97.93.68.239 97-93-68-239.static.rvsd.ca.charter.com
98.15.238.112 cpe-98-15-238-112.hvc.res.rr.com
132.227.62.242 xtr1.ipv6.lip6.fr
153.16.5.1 cisco-it-xtr-1.lisp4.net
153.16.5.29 lisp.cisco.com,lisp4.cisco.com
153.16.13.1 snoble-xtr.lisp4.net
153.16.13.33 dalvarez-xtr.lisp4.net
153.16.13.81 srin-xtr.lisp4.net
153.16.13.177 manishee-xtr.lisp4.net
153.16.16.1 isc-pxtr.eid.lisp4.net
153.16.17.17 gregg-xtr.lisp4.net
153.16.18.1 robert-cdw-xtr.lisp4.net
153.16.23.145 kleinart-xtr.lisp4.net
153.16.25.1 rymcdowe-xtr.lisp4.net
153.16.25.17 miles-xtr-1.lisp4.net
153.16.25.49 epulvino-xtr.lisp4.net
153.16.25.129 jemannin-xtr.lisp4.net

153.16.25.161 bluethunder-xtr.lisp4.net
153.16.26.1 mwhitley-xtr.lisp4.net
153.16.26.113 scalan-xtr.lisp4.net
153.16.29.1 jpcaron-xtr.lisp4.net
153.16.30.113 bguldan-xtr.lisp4.net
153.16.30.161 coopergeneral-xtr.lisp4.net
153.16.30.225 perry-xtr.lisp4.net
153.16.31.49 apparatus-xtr.lisp4.net
153.16.31.97 gar-xtr.lisp4.net
153.16.32.177 mironto-xtr.lisp4.net
153.16.32.241 experteach-xtr.lisp4.net
153.16.38.1 lisp6-fr-xtr.lisp4.net
153.16.44.145 lukasm-xtr.lisp4.net
153.16.47.17 ataf-xtr.lisp4.net
153.16.47.161 smirnov-xtr.lisp4.net
153.16.49.81 turnerhouse-xtr.lisp4.net
153.16.49.177 leander-xtr.lisp4.net
153.16.50.241 haase-xtr.lisp4.net
153.16.51.97 rjm-xtr.lisp4.net
153.16.53.161 aflorio-cz-xtr.lisp4.net
153.16.54.1 hulsmann-xtr.lisp4.net
153.16.54.161 aflorio-it-xtr.lisp4.net
153.16.55.65 eison-xtr.lisp4.net
153.16.55.193 raiffeisen-hu-xtr.lisp4.net
153.16.56.1 lukasm2-xtr.lisp4.net
153.16.64.1 iij-mr-ms.lisp4.net
153.16.66.225 alphawest-xtr.lisp4.net
153.16.67.1 apan-xtr.lisp4.net
153.16.70.1 yuyarin-xtr.lisp4.net
153.16.71.1 farhadsh-xtr.lisp4.net
205.203.201.1 router.millerad.com
217.15.88.68 68-88-15-217.reverse.alphalink.fr

Acronyms

AAC	Advanced Audio Coding	DNS	Domain Name System
AAPT	Android Asset Packaging Tool	DoS	Denial-of-Service
ADT	Android Development Tools	DDoS	Distributed DoS
AEP	Android Extension Pack	DRM	Digital Rights Management
AES	Advanced Encryption Standard	EID	Endpoint Identifier
A-GPS	Assisted GPS	ESP	Encapsulated Security Payload
AIDL	Android Interface Definition Language	EVDO	Enhanced Voice-Data Optimized
AJAX	Asynchronous Javascript And XML	ext	extended filesystem
AOT	Ahead-of-Time	FIB	BGP Forwarding Table
AP	Access Point	FLAC	Free Lossless Audio Codec
API	Application Programming Interface	FQDN	Fully Qualified Domain Name
APK	Android Package	FTP	File Transfer Protocol
ART	Android RunTime	FUSE	Filesystem in Userspace
ATA	Advanced Technology Attachment	GAL	Global Address List
AVF	Android VPN Framework	GC	Garbage Collector
AVRCP	Audio/Video Remote Control Profile	GID	Group ID
BGP	Border Gateway Protocol	GIF	Graphics Interchange Format
BID	Binding unique IDentification	GNU	GNU's Not Unix
C2DM	Cloud to Device Messaging	GPL	General Public License
CDMA	Code Division Multiple Access	GPRS	General Packet Radio Service
CGA	Cryptographic Generated Address	GPS	Global Positioning System
CoA	Care-of Address	GPU	Graphical Processor Unit
CPU	Central Processor Unit	GUI	Graphical User Interface
DAC	Digital-to-Analog Converter	H-MH	Host-Multihoming
DFZ	Default-Free Zone	HAL	Hardware Abstraction Layer
DHCP	Dynamic Host Configuration Protocol	HBA	Hash-Based Address
DLNA	Digital Living Network Alliance	HD	Hard Drive
		HDR	High Dynamic Range
		HI	Host Identifier

HID	Human Interface Device	MCoA	Multiple Care-of Addresses
HIP	Host Identity Protocol	MD	Message-Digest algorithm
HIPL	HIP for Linux	MIP	Mobile IP
HIT	Host Identity Tag	MIPv6	Mobile IPv6
HSDPA	High-Speed Downlink Packet Access	MitM	Man-in-the-Middle
HSUPA	High-Speed Uplink Packet Access	MMS	Multimedia Messaging Service
HTML	HyperText Markup Language	MN	Mobile Node
HTTP	HyperText Transfer Protocol	MPEG	Moving Picture Experts Group
HTTPS	HTTP over SSL	MPT	Multi-Path Transport
I-LV	Identifier-Locator Vector	MR	Mobile Router
IAB	Internet Architecture Board	MS	Mapping System
ICMP	Internet Control Message Protocol	MSP	Mapping Service Provider
ID	Identity	NAT	Network Address Translation
IDC	International Data Corporation	NEMO	Network Mobility
IDE	Integrated Development Environment	NFC	Near Field Communication
IEEE	Internet Engineering Task Force	NDK	Native Development Kit
IETF	Internet Engineering Task Force	NID	Node Identifier
ILNP	Identifier/Locator Network Protocol	OAL	Open Accessory Library
IKE	Internet Key Exchange	OS	Operating System
IMAP4	Internet Message Access Protocol 4	OSI	Open Systems Interconnection
I/O	Input/Output	PC	Personal Computer
IP	Internet Protocol	POP3	Post Office Protocol 3
IPC	Inter-Process Communication	ppi	Pixel per inch
IPSec	IP Security	PPP	Point-to-Point Protocol
IPv4	IP version 4	PSTN	Public Switched Telephone Network
IPv6	IP version 6	PxTR	Proxy Tunnel Router
ISO	International Organization for Standardization	QoS	Quality of Service
ISP	Internet Service Provider	QVGA	Quarter VGA
JDK	Java Development Kit	RAM	Random Access Memory
JIT	Just-In-Time	RC	Rivest Cipher
JNI	Java Native Interface	RFC	Request for Comments
LED	Light-Emitting Diode	RLOC	Routing Locator
LISP	Locator/Identifier Separation Protocol	RtL	Right-to-Left
LISP-MN	LISP Mobile Node	RTR	Re-encapsulating Tunnel Router
MAP	Message Access Profile	S-MH	Site-Multihoming
		SAF	Storage Access Framework
		SATA	Serial ATA

SCTP	Stream Control Transport Protocol	UI	User Interface
SDK	Software Development Kit	UID	User ID
SE	Standard Edition	ULID	Upper-Layer Identifier
SELinux	Security-Enhanced Linux	ULP	Upper-Layer Protocol
SHA	Secure Hash Algorithm	UML	Universal Model Language
SHIM6	Site Multihoming by IPv6 Intermediation	UMTS	Universal Mobile Telecommunications System
SIMA	Simultaneous Multi-Access	URI	Uniform Resource Identifier
SIP	Session Initiation Protocol	USB	Universal Serial Bus
SMR	Solicit-Map-Request	VGA	Video Graphics Array
SMS	Simple Message System	VM	Virtual Machine
SMTP	Standard Mail Transmission Protocol	VoIP	Voice over IP
SNI	Server Name Indication	VPN	Virtual Private Network
SNPA	Sub-Network Point of Attachment	WRT	Wireless Receiver/Transmitter
SSH	Secure Shell	WVGA	Wide VGA
SSL	Secure Sockets Layer	WXGA	Wide eXtended Graphics Array
TCP	Transmission Control Protocol	XHTML	eXtensible HyperText Markup Language
TLS	Transport Layer Security	XML	eXtensible Markup Language
TTL	Time-To-Live	YAFFS	Yet Another Flash File System
UDP	User Datagram Protocol	YOY	Year-Over-Year

Bibliography

- [1] AALTO UNIVERSITY AND RWTH AACHEN UNIVERSITY, *HIPL User Manual*, 2013.
- [2] JOE ABLEY, MARCELO BAGNULO, AND ALBERTO GARCIA-MARTÍNEZ, *Applicability Statement for the Level 3 Multihoming Shim Protocol (Shim6)*. IETF Internet-Draft, October 2010.
- [3] J. ABLEY, K. LINDQVIST, E. DAVIES, B. BLACK, AND V. GILL, *IPv4 Multihoming Practices and Limitations*. RFC 4116 (Informational), July 2005.
- [4] ANDROID DEVELOPERS, *Support SCTP*. Issue 3272, July 2009.
- [5] ———, *Android SDK*. API level 19, December 2013.
- [6] R. J. ATKINSON AND S. N. BHATTI, *Identifier-Locator Network Protocol (ILNP) Architectural Description*. RFC 6740 (Experimental), November 2012.
- [7] TUOMAS AURA, PEKKA NIKANDER, AND GONZALO CAMARILLO, *Effects of Mobility and Multihoming on Transport-Protocol Security*, in S&P 2004: IEEE Symposium on Security and Privacy, Berkeley, CA, USA, May 2004, pp. 12–26.
- [8] SÉBASTIEN BARRE, AMINE DHRAIEF, NICOLAS MONTAVONT, AND OLIVIER BONAVENTURE, *MipShim6: une approche combinée pour la mobilité et la multidomiciliation*, in CFIP 2009: 14ème Colloque Francophone sur l'Ingénierie des Protocoles, October 2009.
- [9] OLIVIER BONAVENTURE, *Scaling the internet with lisp*, August 2009.
- [10] MARCELLO BORSARI, *Multihoming su Symbian per VoIP*, master's thesis, Alma Mater Studiorum Università di Bologna, Bologna, Italy, 2010.
- [11] CATHARINA CANDOLIN, MIIKA KOMU, MIKA KOUSA, AND JANNE LUNDBERG, *An implementation of HIP for Linux*, Proceedings of the Linux Symposium, (2003).
- [12] YAN CHEN, TONI FARLEY, AND NONG YE, *QoS Requirements of Network Applications on the Internet*, Inf. Knowl. Syst. Manag., 4 (2004), pp. 55–76.
- [13] DUNG CHI PHUNG, STEFANO SECCI, DAMIEN SAUCEZ, AND LUIGI IANNONE, *The OpenLISP Control Plane Architecture*, IEEE Network, (2014), pp. 34–40.

-
- [14] CISCO SYSTEMS, *IP Application Services Configuration Guide, Cisco IOS Release 12.4*, Cisco Systems, 2011.
- [15] P.T. CONRAD, G.J. HEINZ, JR. CARO, A.L., P.D. AMER, AND J. FIORE, *SCTP in Battlefield Networks*, in Military Communications Conference, 2001. MILCOM 2001. Communications for Network-Centric Operations: Creating the Information Force. IEEE, vol. 1, October 2001, pp. 289–295 vol.1.
- [16] F. CORAS, D. SAUCEZ, L. JAKAB, A. CABELLOS-APARICIO, AND J. DOMINGO-PASCUAL, *Implementing a bgp-free isp core with lisp*, in Global Communications Conference (GLOBECOM), 2012 IEEE, Dec 2012, pp. 2772–2778.
- [17] DENNIS DALLA TORRE, *Architetture per Network Mobility*, master’s thesis, Alma Mater Studiorum Università di Bologna, Bologna, Italy, 2011.
- [18] V. DEVARAPALLI, R. WAKIKAWA, A. PETRESCU, AND P. THUBERT, *Network Mobility (NEMO) Basic Support Protocol*. RFC 3963 (Proposed Standard), Jan. 2005.
- [19] AMINE DHRAIEF AND ABDELFETTAH BELGHITH, *Multihoming support in the Internet: A state of the art*, in MICS 2010: International Conference on Models of Information and Communication Systems, Rabat, Morocco, November 2010.
- [20] HASSNA EL FILALI, *Locator Id Separation Protocol*, master’s thesis, Alma Mater Studiorum Università di Bologna, Bologna, Italy, 2013.
- [21] ALEXANDER GLADISCH, ROBIL DAHER, AND DJAMSHID TAVANGARIAN, *Survey on Mobility and Multihoming in Future Internet*, *Wirel. Pers. Commun.*, 74 (2014), pp. 45–81.
- [22] SOROUSH HAERI, RAJVIR GILL, MARILYN HAY, TOBY WONG, AND LJILJANA TRAJKOVIC, *Multihoming with Locator/ID Separation Protocol: An Experimental Testbed*, February 2015.
- [23] M. HOEFLING, M. MENTH, AND M. HARTMANN, *A Survey of Mapping Systems for Locator/Identifier Split Internet Routing*, *Communications Surveys Tutorials*, IEEE, 15 (2013), pp. 1842–1858.
- [24] FRANCOIS HOGUET, *Network mobility for multi-homed Android mobile devices*, master’s thesis, Nicta – Université de Technologie de Compiègne, Eveleigh, Sydney, NSW, Australia – Compiègne, France, September 2012.
- [25] QING HUANG, *An extension to the android access control framework*, master’s thesis, Linköpings University, Linköpings, Sweden, October 2011.
- [26] J.R. IYENGAR, P.D. AMER, AND R. STEWART, *Concurrent multipath transfer using sctp multihoming over independent end-to-end paths*, *Networking*, IEEE/ACM Transactions on, 14 (2006), pp. 951–964.

- [27] LORÁND JAKAB, ALBERT CABELLOS, FLORIN CORAS, DAMIEN SAUCEZ, AND OLIVIER BONAVENTURE, *Evaluating the Performance of LISP Mapping Systems*.
- [28] APARNA KANUNGO, T. JANANI, AND P. NARAYANASAMY, *Dynamic IP reconfiguration in Stream Control Transmission Protocol*, in IEEE TENCON 2003, 2003.
- [29] DOMINIK KLEIN, MATTHIAS HARTMANN, AND MICHAEL MENTH, *Nat traversal for lisp mobile node*, in ACM ReArch 2010, Philadelphia, USA, Nov. 2010.
- [30] FABIAN KUHN, NANCY LYNCH, AND ROTEM OSHMAN, *Distributed Computation in Dynamic Networks*, in Proceedings of the Forty-second ACM Symposium on Theory of Computing, STOC'10, New York, NY, USA, 2010, ACM, pp. 513–522.
- [31] T. LI, *Recommendation for a Routing Architecture*. RFC 6115 (Informational), February 2011.
- [32] XIAOMEI LIU AND LI XIAO, *A Survey of Multihoming Technology in Stub Networks: Current Research and Open Issues*, Network, IEEE, 21 (2007), pp. 32–40.
- [33] SHIWEN MAO, SHUNAN LIN, YAO WANG, S. S. PANWAR, AND YIHAN LI, *Multipath video transport over ad hoc networks*, Wireless Communications, IEEE, 12 (2005), pp. 42–49.
- [34] DAVE MEYER, LIXIA ZHANG, AND KEVIN FALL (ED.), *Report from the iab workshop on routing and addressing*, RFC 4984, (2007).
- [35] KOSHIRO MITSUYA, ROMAIN KUNTZ, SHINTA SUGIMOTO, RYUJI WAKIKAWA, AND JUN MURAI, *A Policy Management Framework for Flow Distribution on Multihomed End Nodes*, in Proceedings of 2nd ACM/IEEE International Workshop on Mobility in the Evolving Internet Architecture, MobiArch '07, New York, NY, USA, 2007, ACM, pp. 10:1–10:7.
- [36] R. MOSKOWITZ AND P. NIKANDER, *Host Identity Protocol (HIP) Architecture*. RFC 4423 (Informational), May 2006.
- [37] R. MOSKOWITZ, P. NIKANDER, P. JOKELA, AND T. HENDERSON, *Host Identity Protocol*. RFC 5201 (Experimental), April 2008. Updated by RFC 6253.
- [38] HABIB NADERI AND BRIAN E. CARPENTER, *A Review of IPv6 Multihoming Solutions*, in ICN 2011, 10th International Conference on Networks, 2011.
- [39] PEKKA NIKANDER, JUKKA YLITALO, AND JORMA WALL, *Integrating Security, Mobility, and Multi-homing in a HIP Way*, in Proceedings of the Network and Distributed Systems Security Symposium, NDSS'03, February 2003, pp. 87–99.
- [40] ERIK NORDMARK AND MARCELO BAGNULO, *Shim6: Level 3 Multihoming Shim Protocol for IPv6*. RFC 5533 (Proposed Standard), June 2009.
- [41] ORACLE JAVA SE DOCUMENTATION, *Java Native Interface Specification*, 2014.

- [42] JEN-YI PAN, JING-LUEN LIN, AND KAI-FUNG PAN, *Multiple Care-of Addresses Registration and Capacity-Aware Preference on Multi-Rate Wireless Links*, in Advanced Information Networking and Applications - Workshops, 2008. AINAW 2008. 22nd International Conference on, March 2008, pp. 768–773.
- [43] S. PIERREL, P. JOKELA, J. MELEN, AND K. SLAVOV, *A Policy System for Simultaneous Multiaccess with Host Identity Protocol*, ACNM Proceedings, (2007), pp. 71–77.
- [44] R. PRAVEEN, J. PRASHANT, AND R.G. HEMANT, *A Multihoming solution for medium sized enterprises*, November 2005.
- [45] ALBERTO RODRÍGUEZ-NATAL, *Privacy extensions for LISP-MN*, master’s thesis, Universitat Politècnica de Catalunya, June 2012.
- [46] ALBERTO RODRÍGUEZ-NATAL, LORAND JAKAB, VINA ERMAGAN, PREETHI NATARAJAN, FABIO MAINO, AND ALBERT CABELLOS-APARICIO, *Location and Identity Privacy for LISP-MN*, in Proceedings of the IEEE International Conference on Communications, ICC’15, London, UK, June 2015.
- [47] ALBERTO RODRÍGUEZ-NATAL, LORÁND JAKAB, MARK PORTOLÉS, VINA ERMAGAN, PREETHI NATARAJAN, FABIO MAINO, DAVID MEYER, AND ALBERT CABELLOS-APARICIO, *LISPMob: Mobile Networking through LISP*, Proceedings of Springer Wireless Personal Communications Journal, (2012).
- [48] KEIICHI SHIMA, KOSHIRO MITSUYA, RYUJI WAKIKAWA, TSUYOSHI MOMOSE, AND KEISUKE UEHARA, *SHISA: The Mobile IPv6/NEMO BS Stack Implementation Current Status*, in Asia BSD Conference 2007, March 2007.
- [49] KEIICHI SHIMA, YOJIRO UO, NOBUO OGASHIWA, AND SATOSHI UDA, *Operational Experiment of Seamless Handover of a Mobile Router using Multiple Care-of Address Registration*, Journal of Networks, 1 (2006).
- [50] BRUNO MIGUEL SOUSA, KOSTAS PENTIKOUSIS, AND MARILIA CURADO, *Multihoming Management for Future Networks*, Mobile Networks and Applications, 16 (2011), pp. 505–517.
- [51] R. STEWART, Q. XIE, M. TUOXEN, S. MARUYAMA, AND M. KOZUKA, *Stream Control Transmission Protocol (SCTP) Dynamic Address Reconfiguration*. RFC 5061 (Proposed Standard), September 2007.
- [52] LUCA STORNAIUOLO AND ISRAA FREJ, *Testing Host Identity Protocol functionalities on HIP for Linux implementation*, March 2009.
- [53] NUNO VEIGA, MÁRIO ANTUNES, VÍTOR SANTOS, AND ALEXANDRE SANTOS, *Experiments with IPv6 Network Mobility Using NEMO Protocol*, in IADIS International Telecommunications, Networks and Systems, 2007.

-
- [54] M. WATARI, A. TAGAMI, AND S. ANO, *Evaluating the Performance of Locator/ID Separation Based on LISP Map Cache Emulation*, in Applications and the Internet (SAINT), 2012 IEEE/IPSJ 12th International Symposium on, July 2012, pp. 296–301.
- [55] KOK-KIONG YAP, TE-YUAN HUANG, MASAYOSHI KOBAYASHI, YIANNIS YIAKOUMIS, NICK MCKEOWN, SACHIN KATTI, AND GURU PARULKAR, *Making Use of All the Networks Around Us: A Case Study in Android*, CellNet’12, (2013).

Acknowledgements

Here I express all my gratitude to my academic supervisor Prof. **Paolo Bellavista** for his accurate guidance, invaluable constructive criticism and friendly and generous advice during the whole thesis work. Thanks to him I began my academic path at University of Bologna and thanks to him I accomplish it.

My warm thanks go to **Alberto Rodríguez-Natal** of the Universitat Politècnica de Catalunya for his kind helpfulness, accurate clarifications and for the sharing of very interesting unpublished results; and to Prof. **Arnold I. Davidson** of the University of Chicago for his punctual advice and the thesis' review.

I must thank also **Loránd Jakab** of the Universitat Politècnica de Catalunya and **Darrel Lewis** of Cisco Systems for their enthusiastic welcome into the LISPmob project, and **Seppo Heikkinen** of the Tampere University of Technology for his assistance into the discovering of HIP during my Erasmus program.

I also thank the **Librairie Française de Florence** for the testing equipment and broadband connection supply.

Lastly, my deepest heartfelt gratitude goes to **Angela Righi** for her all-constant support, sharp hints and engaging exchanges. Without her not a single word of this thesis would have been written.

Ringraziamenti

Un percorso lungo come il mio richiede degli adeguati ringraziamenti perché le persone che hanno contribuito a questo ambito risultato sono tante, e meritano tutte di essere incluse. Per arrivare fin qui ho dovuto superare 30 esami, ovvero 66 prove: 21 scritte, 24 orali, 15 laboratori, 5 progetti, e infine la tesi. Ho iniziato a Napoli e terminato a Bologna, passando per Tampere con tappa a Barcellona. Ho imparato tanto, dai libri ma soprattutto dalle persone.

Il mio primo e più importante ringraziamento va ai miei genitori, ai quali dedico questa tesi di laurea. **Elio** e **Anna**, che mi hanno insegnato come il sacrificio e la perseveranza, a fronte di qualsiasi difficoltà, prima o poi premiano sempre. Ci ho messo tanto, troppo, lo ammetto, sprecando tempo e risorse, e di questo vi chiedo scusa. Può sembrare assurdo, ma ingegneria purtroppo è così: per arrivare al traguardo sono richiesti studio, applicazione, memoria, dedizione, omologazione... tutto fuorché –salvo rare occasioni– l'*ingegno*. Ma oggi siamo qui, dies aureo signanda lapillo, ed è tutto ciò che conta.

Quindi per tutte quelle volte che pigiato i tasti R-U-N sul Commodore 64 per giocare ai Ghostbusters o a Frogger; per tutte quelle volte che ho sparato a un nazista in Wolfenstein o saltato qua e là con Aladdin o salvato Azeroth dall'Orda con il primo Pentium; per tutte quelle volte che ho risolto un enigma con Lara Croft o segnato un gol a FIFA o fondato una città a Age of Empires con la potenza dell'AMD K6; per tutte quelle volte che ho compilato i miei primi codici C++ grazie al Centrino del Toshiba M40-281 e per tutte quelle altre che ho creato un sito web o quelle altre ancora che ho sviluppato i miei progetti Java e C# sull'Athlon X2... *Grazie.*

Grazie, perché avete avuto davvero tanta tanta pazienza e mi avete sempre supportato, perché solo voi sapete quanto sia stato difficile per me in certi momenti, perché mi avete capito e sostenuto quando ho raggiunto i miei limiti e indirizzato e incoraggiato quando ho capito che potevo superarli.

Einstein nel 1943 scrisse ad una studentessa:

«Do not worry about your difficulties in Mathematics.
I can assure you mine are still greater.»¹

Ed un proverbio giapponese recita:

七転び八起き
«Nana-korobi ya-oki.»²

Dunque focalizzarsi sul problema e non arrendersi mai: è la lezione che ho imparato da voi e vi assicuro che ne farò tesoro. Anzi, quanto prima cercherò anche di tramandarla...

Grazie babbo, grazie mamma.

Ringrazio **Angela Righi**, mia amata compagna e mia fonte primaria di motivazione e d'ispirazione. Avere una donna così brillante accanto è un vero privilegio.

«C'est le temps que tu as perdu pour ta rose qui fait
ta rose si importante.»³

Ricordi?... Per il nostro futuro insieme, spero tanto di essere capace di aiutarti e di sostenerti così come tu hai sempre fatto con me. *Grazie Amore.*

Ringrazio **Claudio Paoella**, mio best friend fin dai tempi del liceo. Il futuro è incerto, viviamo in un mondo fluttuante, turbolento, imprevedibile. Ma tu ci sei stato sempre, indipendentemente dalle distanze e dagli impegni personali, senza “se” e senza “ma”, e so che sarà sempre così. *Grazie 'O Biò.*

¹ «Non preoccuparti delle tue difficoltà in matematica. Ti assicuro che le mie sono ancora maggiori.»

² «Cadi sette volte, ma rialzati otto volte.»

³ «È il tempo che hai speso per la tua rosa che fa la tua rosa così importante.»

Ringrazio **Maria Grazia Cusatis**, mia carissima amica, che nei miei primi giorni bolognesi mi ha aiutato più di quanto potessi mai chiedere o sperare. Senza il tuo generoso contributo l'ultima e più importante parte dell'avventura non sarebbe mai nemmeno cominciata. *Grazie Mary.*

Ringrazio **Paolo Schipani**, mio stimatissimo amico, con il quale ho condiviso alcune delle esperienze più intellettualmente stimolanti della mia vita. Tanti fanno l'Erasmus, ma solo io ho avuto la fortuna di averti proprio al piano di sopra: due colpi alla parete ed è già ora di cena. *Grazie Paolo.*

Ringrazio **Piero Zito**, mio amico e compagno di banco fisso a Bologna, con il quale ho condiviso davvero dei bellissimi momenti durante questi mesi all'Alma Mater: dalla scomodità dei banchi alla progettazione di improbabili regolatori, dai soporiferi teoremi in \mathbb{R}^n alle più cervelotiche callback di AJAX, dai puntuali suggerimenti di "zio Enrico" alle calde passeggiate serali per Bologna... *Grazie tante collega.*

Ringrazio **Israa Frej**, con la quale ho affrontato gran parte dell'esperienza finlandese in Erasmus. Se è stato un anno così folgorante e ricco di successi, è sicuramente perché studiare con te è stato molto più easy. *Thank you, Israa.*

Ringrazio **Francesco Murolo**, mio cugino, che per due estati a Castel Volturno ha seguito me nella preparazione degli esami anziché divertirsi al mare con la famiglia. *Grazie Franco.*

Ringrazio di cuore **Michela Baranello**, **Saygın Arkan**, **Ilaria Cenacchi**, **Stefano Natali** e **Joe Raiola** con i quali ho condiviso tante esperienze: sì certo, ogni tanto anche qualche lezione deprimente, ma soprattutto tanti esaltanti successi in esami impossibili. E ringrazio anche **Davide Maccaferri**, **Enrico Casoni**, **Alessandro Vadruccio**, **Alessandro Stega**, **Jan Svoboda**, **Vlad Tabus**, **Fabiana Nappi**, **Leonardo Cappabianca** e **Federica Virgilio**, con i quali ho condiviso percorsi più brevi ma comunque imprescindibili.

Inoltre ringrazio per le tante parole di incoraggiamento, gli insegnamenti di vita, le mille risate, i discorsi profondi, i confronti politici, gli allenamenti, i film, le saune, i fantacalcio, gli allenamenti, i viaggi, le passeggiate sotto le stelle e l'aurora boreale, gli Erasmus party, le partite a scacchi, le bevute, le cene, e le tantissime altre esperienze fatte insieme in questi anni gli amici già citati e **Emanuele De Falco**, **Stanislav Radomskiy**, **Christian Hanisch**, **Maren Moggale**, **Silvia Bagnale**, **Angela Mariotti**, **Francesco Ariano**, **Domenico Schiavulli**, **Armando Falato**, **Marco Cirello**, **Raffaele Meo**, **Ivano Esposito**, **Renato Romanello**, **Mirko Cortese**, **Giovanni Riccardi**, **Raffaele Muto**, **Emilia Santoro**, **Cecilia Dalla Negra**, **Astrid Del Frate**, **Filippo Turrini**, **Sergio Cricca**, **Ylenia Napoli**, **Omar Bosani**, **Valerio Garcea**, **Claudia Giacobbe**, i miei cugini **Marco Gioiello** e **Davide Gioiello**, e i miei nonni **Anna**, **Adelaide** e **Adolfo**, che saranno sicuramente orgogliosi di me.

Infine un ringraziamento speciale va ai miei cari zii **Francesca Lettieri**, **Patrizio Civetta** e **Andrea Lettieri** che hanno sempre trovato il modo di dimostrarmi la loro stima oltre che il loro affetto; agli onnipresenti **Bianca Torricelli** e **Roberto Righi** che mi hanno accolto davvero come un figlio, da subito; al carissimo **Salvatore Ascione** e famiglia che mi hanno visto crescere e che per me rappresentano il legame con la mia terra, le mie origini e la mia gioventù; al mio maestro di judo **Raffaele Parlati** e la gloriosa Nippon Club Napoli che io considero la mia seconda casa; e naturalmente i miei fratelli **Ivo** e **Matteo** che sono il mio vero team—e insieme siamo invincibili.

A *tutti voi* rivolgo il mio più sincero augurio di una brillante riuscita in tutti i vostri progetti.