UNIVERSIDAD CARLOS III DE MADRID
Escuela Politécnica Superior
Ingenieria Tecnica en Informatica de Gestion

Jaime Antonio Jiménez Bolonio

Implementation and performance of REsource LOcation And Discovery (RELOAD) Parser and Encoder

Thesis submitted in partial fulfillment of the requirements for the degree of Bachellor of Science in Technology.

Helsinki, June 02, 2009

Supervisor:	Marcelo Bagnulo
	Professor
Instructor:	Jouni Mäenpää
	M.Sc. (Tech)

A mi familia y a Valentina ♡.

`01010110 69 01110110 61 WU8h!`

UNIVERSIDAD CARLOS III DE MADRID

RESUMEN DEL PROYECTO DE FIN DE CARRERA

Autor:	Jaime Antonio Jiménez Bolonio		
Título del proyecto:	Implementación y pruebas de REsource LOcation And Discovery (RELOAD) Parser and Encoder		
Date:	June 02, 2009	**Number of pages:**	94
Faculty:	Ingenieria Técnica en Informática de Gestión		
Supervisor:	Prof. Marcelo Bagnulo		
Instructor:	Jouni Mäenpää, M.Sc.		

El ampliamente utilizado paradigma cliente/servidor está siendo complementado e incluso reemplazado por otros planteamientos de tipo Peer-to-Peer (P2P). Las redes P2P ofrecen un sistema descentralizado de distribución de la información, son más estables, y representan una solución al problema de la escalabilidad.

Al mismo tiempo, el Session Initiation Protocol (SIP), un protocolo de señalización diseñado inicialmente para arquitecturas de tipo ciente/servidor, ha sido ampliamente adoptado para servicios de comunicación tipo Voice-over-IP (VoIP).

El actual proceso de estandarización llevado a cabo por el Peer-to-Peer Session Initiation Protocol (P2PSIP) Working Group del IETF se está acercando al desarrollo de aplicaciones que puedan utilizar tecnologías P2P junto con SIP.

RELOAD es un protocolo P2P de señalización, que está todavía en desarrollo. RELOAD trabaja en entornos en los que existen Network Address Translators (NATs) o firewalls. RELOAD soporta diferentes aplicaciones y proporciona un marco de seguridad, también permite el uso de diversos algoritmos para las Distributed Hash Tables (DHTs) mediante los llamados "topology plugins".

Esta tesis tiene como objetivos la implementación de un codificador y decodificador para mensajes de RELOAD, y el análisis de su rendimiento. Para este último punto se implementará un programa de prueba ejecutable en un teléfono móvil y en un servidor para la simulación de una red RELOAD.

Keywords: P2P, SIP, P2PSIP, CHORD, RELOAD

UNIVERSIDAD CARLOS III DE MADRID	ABSTRACT OF THE BACHELLOR'S THESIS

Author:	Jaime Antonio Jiménez Bolonio		
Name of the thesis:	Implementation and performance of REsource LOcation And Discovery (RELOAD) and Encoder		
Date:	June 02, 2009	**Number of pages:**	94
Faculty:	Ingenieria Técnica en Informática de Gestión		
Supervisor:	Prof. Marcelo Bagnulo		
Instructor:	Jouni Mäenpää, M.Sc.		

The widely used classic client/server paradigm is being complemented and sometimes replaced by current Peer-to-Peer (P2P) approaches. P2P networks offer decentralized distribution of information, are more stable, and represent a solution to the problem of scalability.

At the same time the Session Initiation Protocol (SIP), a signalling protocol initially designed for client/server architectures, has been widely adopted for Voice-over-IP (VoIP) communication.

The current standardization process of the Peer-to-Peer Session Initiation Protocol (P2PSIP) working group of the IETF is moving towards the development of applications that can use both P2P and Session Initiation Protocol (SIP) technologies in conjuntion.

RELOAD is a P2P signalling protocol, which is still under development. RELOAD works in environments where there are Network Address Translators (NATs) or firewalls. RELOAD can support various applications and provides a security frameworks. RELOAD also allows the use of various Distributed Hash Table (DHT) algorithms in the form of topology plugins.

This thesis aims at implementing a parser and encoder for RELOAD messages, and analyzing its performance by implementing a test program that will run on a mobile phone and on a server simulating a RELOAD overlay network.

Keywords: P2P, SIP, P2PSIP, CHORD, RELOAD

Acknowledgments

I would like to thank Jouni Mäenpää for all his time and help with the implementation, specially for developing the CryptoEngine used in RELOAD, and also for his astonishing patience when explaining and correcting the thesis. This work would not have been possible without his help. I would also like to thank Veera Andersson for her help with the many doubts I had on Java and Ari Keranen for his help on the ones I had on LaTeX and several other tools; also Darwin Valderas for his help with the Perl's script.

I would like to express my gratitude also to my supervisor Professor Marcelo Bagnulo and the University Carlos III for allowing me to do the thesis abroad. My especial gratitude goes to Gonzalo Camarillo for giving me the opportunity of being here in the first place, in such a fantastic environment surrounded with so many brilliant people.

I cannot finish without saying how grateful I am with my parents and brother, I can feel them with me despite being 3000 kilometers apart.

Helsinki, June 02, 2009

Jaime Antonio Jiménez Bolonio

Contents

Abbreviations and Acronyms

3G	Third Generation
ACK	Acknowledgement Packet
AHHP	ARPANET Host-to-Host Protocol
ARPA	Advanced Research Projects Agency
ARPANET	Advanced Research Projects Agency Network
API	Application Programming Interface
BCP	Best Current Practice
CPU	Central Processing Unit
DHT	Distributed Hash Table
DNS	Domain Name System
DTLS	Datagram Transport Layer Security
HSDPA	High-Speed Downlink Packet Access
HTTP	Hypertext Transfer Protocol
IAB	Internet Architecture Board
IANA	Internet Assigned Numbers Authority
ICE	Interactive Connectivity Establishment
ID	Identification
IESG	Internet Engineering Steering Group
IETF	Internet Engineering Task Force
IM	Instant Messaging
ISOC	Internet Society
IP	Internet Protocol
ISP	Internet Service Provider
IRTF	Internet Research Task Force

MMUSIC	Multiparty Multimedia Session Control
NAT	Network Address Translation
NCP	Network Control Program/Protocol
NWG	Network Working Group
P2PSIP	Peer-to-Peer Session Initiation Protocol
P2P	Peer-to-Peer
PSTN	Public Switched Telephone Network
RELOAD	REsource LOcation And Discovery
RFC	Request For Comments
RSA	Rivest, Shamir, & Adleman
SCIP	Simple Conference Invitation Protocol
SHA	Secure Hash Algorithm
SIP	Session Initiation Protocol
SIPPING	Session Initiation Proposal Investigation
TCP	Transmission Control Protocol
TLS	Transport Layer Security
TTL	Time To Live
UA	User Agent
UDP	User Datagram Protocol
URL	Uniform Resource Locator
VoIP	Voice Over IP
WG	Working Group
XML	eXtensible Markup Language

List of Figures

List of Tables

Chapter 1

Introduction

Signaling has evolved from circuit-switching, in which the resources were dedicated during communication, to packet-switching networks, in which the resources are shared. The basics of Internet Protocol (IP) networks lay on the idea of a shared medium.

Changes and innovations affect the widespread client/server paradigm, too. A seamless way to communicate between various end-user applications and systems is also desired; as a matter of fact P2P systems achieve this, allowing communication between peers, behind NATs and Firewalls. P2P's great innovation is that it uses the relatively small computing power of personal computers and allows creating networks with immense processing capabilities. In a word, it has changed the way information and culture are approached, making it inexpensive and widely available.

As P2P networks are ubiquitous in the Internet panorama and thanks to protocols like SIP - that help to interact with various end users and merge different technologies - communication systems are being redefined. P2PSIP and the protocol it uses -RELOAD- are an example of towards where the networking protocols are moving.

RELOAD can be used for any application that needs to distribute its resources; indeed some of its current uses are Instant Messaging (IM) and Voice over IP but there can be many more. Although this technology is still under development by the IETF, there already are some similar and successful commercial applications such as Skype. Many more - commercial and non commercial - applications are envisaged, which will have the possibility to interconnect with each other thanks to the standardization process carried out by IETF Working Groups.

1.1 Problem Statement

The Session Initiation Protocol was designed to work in a client-server architecture. While the appearance of P2P technologies opened a new field for the development of SIP, also SIP helped improving current P2P systems; Peer-to-Peer SIP appears to be the best way to follow as it offers the possibility to create robust, well distributed, cost efficient and scalable networks.

SIP, SIPPING and P2PSIP Working Groups perform the difficult task of standardizing the protocols that will be used in P2PSIP communications. The development is currently focused on RELOAD: as it maintains a connection of nodes in an overlay, it enables real-time communication using SIP; moreover, it is able to communicate through barriers such as NATs or Firewalls, and to provide security even among malicious peers.

1.2 Goal and scope of the Thesis

This thesis aims at implementing the RELOAD Message Structure and a working simulation to analyze how the traffic in a RELOAD overlay is. It will focus on the analysis of a test application on a mobile phone, in order to acknowledge if this type of devices can support the traffic they would have on a RELOAD network.

The scope of the implementation is limited to RELOAD's message structure and the simulation focuses on imitating the message load that a single peer would receive and send in a real overlay.

Chapter 2

Background

In this chapter an introduction to the various technologies that are used or that are related to the thesis will be given.

The IETF is an organization concerned with the standardization of Internet protocols. As the motor of the development of new standards for the Internet, the IETF is a key-contributor to the development of the Internet.

SIP has been relatively recently developed but it is already being used in many new communication applications and it is used to set-up a broad range of sessions. At the same time the use of P2P models has risen and now many systems use it as core architecture.

P2PSIP can be considered the last milestone in this field as it aims at merging P2P and SIP in order to create decentralized P2P networks using SIP as messaging system. The technology is currently taking form as RELOAD, a P2P Signaling protocol. RELOAD can be used for a great number of possible applications, P2PSIP is one example.

RELOAD creates a P2PSIP network that can have several usages. The one on in which this thesis focuses is the SIP usage. RELOAD can be used to store and retrieve data, as a service discovery tool or to form direct connections in P2P environments.

2.1 The IETF

The *Internet Engineering Task Force (IETF* [1]*)* is a loosely self-organized organization who contributes to the engineering and evolution of Internet technologies[28]. In fact, it promotes Internet Standards and sets certain guidelines on the development of the Internet architecture.

2.1.1 The Origins of the IETF

A brief outline of the development of the Internet is needed to understand the origins of the IETF.

Without going too far back in time, we could say that the origins of the Internet lie on the ARPANET. The ARPANET started in 1965 with the connection of just two computers, one in Massachusetts and the other in California, that created the first wide-area computer network ever built [33]. Initially ARPANET used the switched telephone system, which proved to be inadequate, and thus confirmed the need for packet switching.

In December 1970, the Network Working Group (NWG) working under S. Crocker finished the initial ARPANET Host-to-Host protocol (AHHP), whose first implementation was called the Network Control Program (NCP) and later popularized as the Network Control Protocol [46]. The work on the IP protocol took off in the mid 1970s, finally substituting NCP in 1983 [19]. Commercialisation and introduction of privately run Internet Service Providers (ISPs) first started in the 1980s, and the expansion of IP into popular use in the 1990s; since then it has become the largest network in the world.

Researchers used to typewrite their work and circulate hard copies among colleagues carrying out a similar work in ARPA, requesting feedback and ideas. The basic ground rules were that *"anyone could say anything and that nothing was official"*, thus the notes were labeled *"Request for Comments"* (RFCs). The appearance of the e-mail made the wide distribution of RFCs possible. The IETF infrastructure grew from half a dozen people and about 100 RFCs in 1968, to hundreds involved and 5392 RFCs at the time this thesis is being written.

2.1.2 The Structure of the IETF

It has to be noted that, at the beginning, the authors of the RFCs were graduate students and anyone could participate [46]. The IETF is still unusual in that it exists as a collection of reunions, but is not a corporation and has no board of directors, no members, and no dues [12]. Although the principles remain largely the same, and its goal remains unchanged: *"to make the Internet work better"* [12], some things have changed. As an example, a web-based tool replaced email as the mechanism to distribute the RFCs. Also the structure of the IETF changed; at the beginning it depended directly on the Internet Activities Board

[1]See http://www.ietf.org/

(IAB [2]), but with the increase of the interest in the engineering of the Internet in the mid 1980s, it was divided into working groups [28]. IETF was mainly formed by academics in the past, but nowadays in most areas of the IETF, equipment manufacturers are the ones writing the protocols and leading the working groups [28].

The current structure of the various standardization bodies (see Figure 2.1) lies mainly in the Internet Society (ISOC [3]) [5], which provides all the financial support and administers the standardization process. The Internet Architecture Board (IAB) [3] is the committee that oversights the technical and engineering development of the Internet and serves as a liaison between the IETF and the ISOC. As the IETF was divided into various working groups [28], the area directors and various liaisons with different organizations such as IAB, IANA an RFC Editor liaisons formed the Internet Engineering Steering Group (IESG) [19]. The remaining working groups formed the Internet Research Task Force (IRTF [4]), whose duty is to work on long term plans regarding Internet protocols, applications, architecture and technology [4]. The IETF mission includes facilitating technology transfer from the IRTF to the wider Internet community and making recommendations to the IESG regarding the standardization of protocols and protocol usage in the Internet [28].

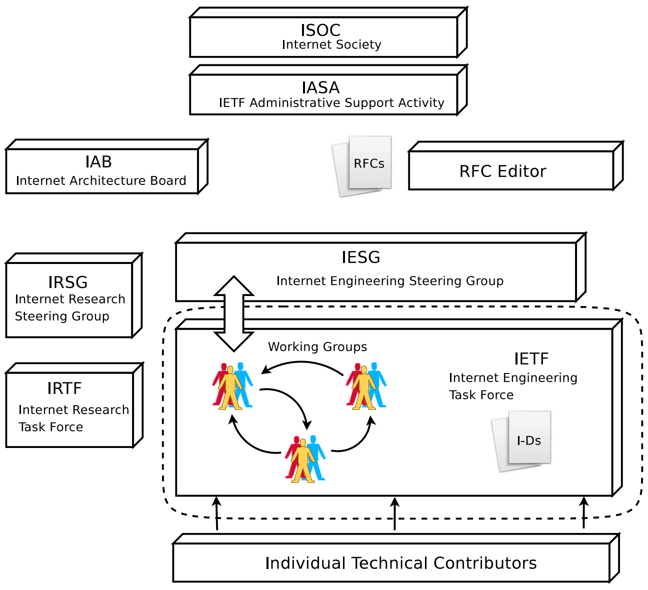

ISOC
Internet Society
IASA
IETF Administrative Support Activity
IAB
Internet Architecture Board
RFCs
RFC Editor
IRSG
Internet Research Steering Group
IESG
Internet Engineering Steering Group
IRTF
Internet Research Task Force
Working Groups
IETF
Internet Engineering Task Force
I-Ds
Individual Technical Contributors

Figure 2.1: Hierarchy of the IETF

The IETF is divided into various working groups [5], each of them directly dependent from

[2]See http://www.iab.org/
[3]See http://www.isoc.org
[4]See http://www.irtf.org
[5]For a exhaustive and actualized list of the current Working Groups it is recommended to see http://www.ietf.org/html.charters/wg-dir.html

the IESG. The areas each group covers are: Applications Area, General Area, Internet Area, Operations and Management Area, Real-time Applications and Infrastructure Area, Routing Area, Security Area and Transport Area. Each working group has a free, unrestricted, official mailing list where most of the work is carried out. Apart from the mailing lists, the IETF meetings help gathering information from various individual technical contributors. The meetings are held three times a year and anyone may register for and attend any meeting.

Within the Real-time Applications and Infrastructure Area, there are various working groups. Among them we have the *Session Initiation Protocol (SIP)* Working Group, chartered to maintain and continue the development of SIP. The Session Initiation Proposal Investigation (SIPPING) Working Group, chartered to document the use of SIP for several applications related to telephony and multimedia, and to develop requirements for extensions to SIP needed for those applications. Peer-to-Peer Session Initiation Protocol (P2PSIP) Working Group, chartered to develop protocols and mechanisms for the use of the Session Initiation Protocol (SIP) in decentralized rather than centralized settings where SIP is currently deployed.

2.1.3 Publication Process

Internet Drafts (I-Ds)

Internet Drafts are versions of the future RFCs. During the development stages, they are available for informal review and comment. An Internet-Draft that is published as an RFC, or that has remained unchanged in the Internet-Drafts directory for more than six months without being recommended by the IESG for publication as an RFC, is simply removed from the Internet-Drafts directory. At any time, an Internet-Draft may be replaced by a more recent version of the same specification, restarting the six-month timeout period [28], [4].

RFCs

RFCs represent the main documentation that the IETF publishes. In Figure 2.2 we can see how an RFC is written. The process starts when an IETF working group or an individual submits an Internet Draft to the IESG. The IESG may have some concerns or suggest some changes to the draft and give it back in order to apply those changes. Once all the concerns are resolved the IESG the draft to the general Internet community for review so that some comments or suggestions arise. This *"Last Call"* notification shall be announced via electronic mail to the IETF Announce mailing list and is a period no shorter than two weeks [15]. If no further comments arise and the draft is approved, a notification to the RFC Editor is sent with instructions to publish the specification as an RFC. In between the process the IESG may decide to create new Working Groups in the case of controversy, change the publication category or commission independent technical reviews at its discretion.

It must be noticed that not all RFC's are standards (for example rfc 1976). There are differ-

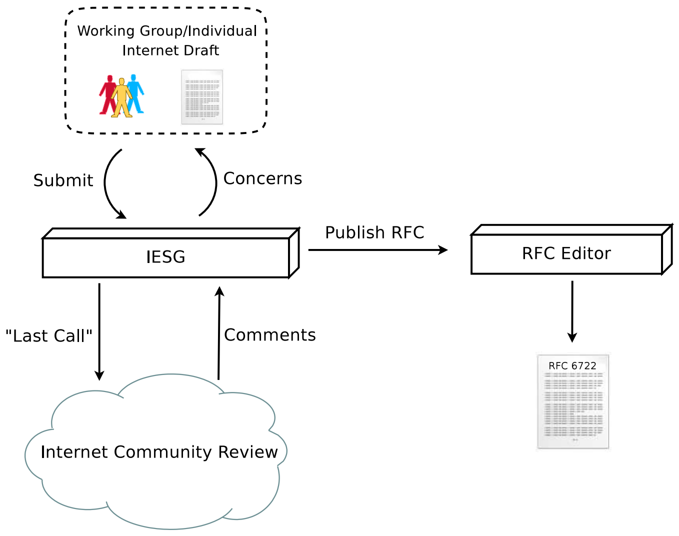


Figure 2.2: Process of creating an RFC

ent types (see Figure 2.3): Standards Track RFCs, Informational and Experimental RFCs and Best Current Practice (BCP) RFCs [19].

Standards Track RFCs

Standards Track RFCs have three maturity levels: a standards track is on the proposed standard (PS) level, when the specification is believed to be well understood and there are no known problems. It becomes a Draft standard (DS), once it is a proposed standard and there are multiple interoperable implementations in the real world. It becomes a standard (STD), when the draft implementation is widely used and operational experience appears within the community [28], [4].

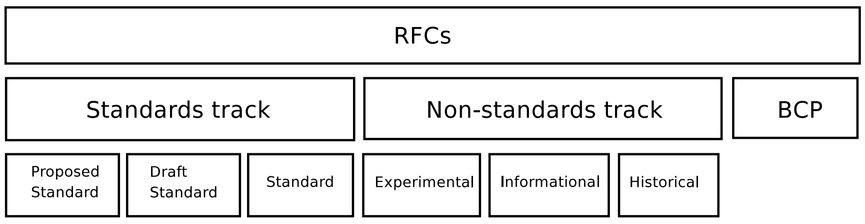


Figure 2.3: Types of RFCs

Informational, Experimental and BCP RFCs

Informational RFCs are used to spread general information to the community. They are not reviewed as thoroughly as the ones in the standards track since they do not include a proposal. Experimental RFCs are for specifications that may be interesting, but for which implementation is still unclear. If, later, the specification becomes popular or is proved to work properly, it can be re-issued as a standards-track RFC [28] [4]. BCP documents describe the application of various technologies in the Internet.

A special comment should be done for these RFCs which purpose is solely humorous. These RFCs are always submitted the April Fool's day, they have the same design of the normal RFCs and are usually very elaborated hoaxes. The tradition is quite old, dating back to June 1973, when RFC 527 [41] was written.

2.2 The Session Initiation Protocol (SIP)

Before the Internet was as common and widespread as it is nowadays, there was a separate network for each service: for data, for voice, mobile and TV networks [54]. Each of these was specific and had country or even regional variants which were incompatible between each other since they used different protocols.

The advent of the Internet has enabled the merging of these dissimilar networks. While voice networks have become optimized for voice and voice only communications, they are scarcely used when it comes to data transfer. In the case of modern mobile networks, it may turn out that they will become a means for accessing the Internet, especially in the case of the so-called 4th generation network (4G). TV providers are focusing more and more on the Internet portability of their services while TV preponderance is fading away against Internet video services [43].

The rise of the Internet has forced mobile, TV and data companies to search for new business models to take advantage of Internet technologies and protocols. In particular, companies are likely to seek for those protocols that merge different technologies thus helping to interact with various end users and merge various technologies: SIP is one of those because of its impact on the telecommunication industry.

2.2.1 The History of SIP

In February 1996, Mark Handley and Eve Schooler submitted their Internet Draft regarding the Session Invitation Protocol (SIPv1). The very same day Henning Schulzrinne submitted his proposal for the Simple Conference Invitation Protocol (SCIP). Although both protocols were text based, SIPv1 only handled session establishment: once the user joined the session the signalling stopped. SCIP signalling remained after session establishment in order to enable changing parameters during the sessions and closing previous sessions. Moreover, Schulzrinne's SCIP was based on Hypertext Transfer Protocol (HTTP).

During the 35th IETF meeting, still in 1996, both protocols were merged into one, which was called the Session Initiation Protocol (SIPv2, just SIP from now on). Its first draft was submitted in December 1996. Like SCIP, it was based on HTTP and could use also TCP, while thanks to SIPv1 it could use UDP, too. Like the two previous it was text based.

Back then the intention was to set up multicast groups for multimedia conferences, and therefore the task of developing SIP was given to the Multiparty Multimedia Session Protocol (MMUSIC) working group. In December 1997, due to the large size of the SIP specification, it was decided to split the specification into base and various extensions of SIP.

In February 1999, SIP reached the level of proposed standard and was published as RFC 2543. The importance of SIP grew in the IETF and thus a new SIP working group was created in September 1999 to focus solely on the development of SIP. Due to the enormous amount of contributions to SIP, it was decided in March 2001 to split the SIP working group in two. The SIP working group would focus on the main specification and its main extensions, while the new group Session Initiation Proposal Investigation (SIPPING) was to test the use of SIP for several applications related to telephony and multimedia, and to develop requirements for SIP extensions. One year later, in June 2002, the current SIP specification was published as the RFC 3261 [54], [49] (see Figure 2.4).

It may seem awkward that RFC 3261 has not been modified in seven years but it is actually quite common. Once the final details of the main specification of a protocol are settled, it is time to focus on the extensions of that protocol. As an example, the Internet Protocol (IP) on RFC 791 was written at the end of the 1970s and was documented as IPv4 in September 1981. Although IPv4 will one day be replaced by IPv6, RFC 791 is still a relevant document for defining Internet Protocol datagrams even at the moment. Also, we would be incorrect to think that no further investigation on the topic has been made since hundreds of drafts and RFCs directly related to SIP, and many more that use SIP in an indirect manner [7] have been published. More information about SIP and its extensions can be foundfrom the draft *"A Hitchhiker's Guide to the Session Initiation Protocol (SIP)"* which has a comprehensive list of the RFC specifications in addition to Internet Drafts which are still under development within.

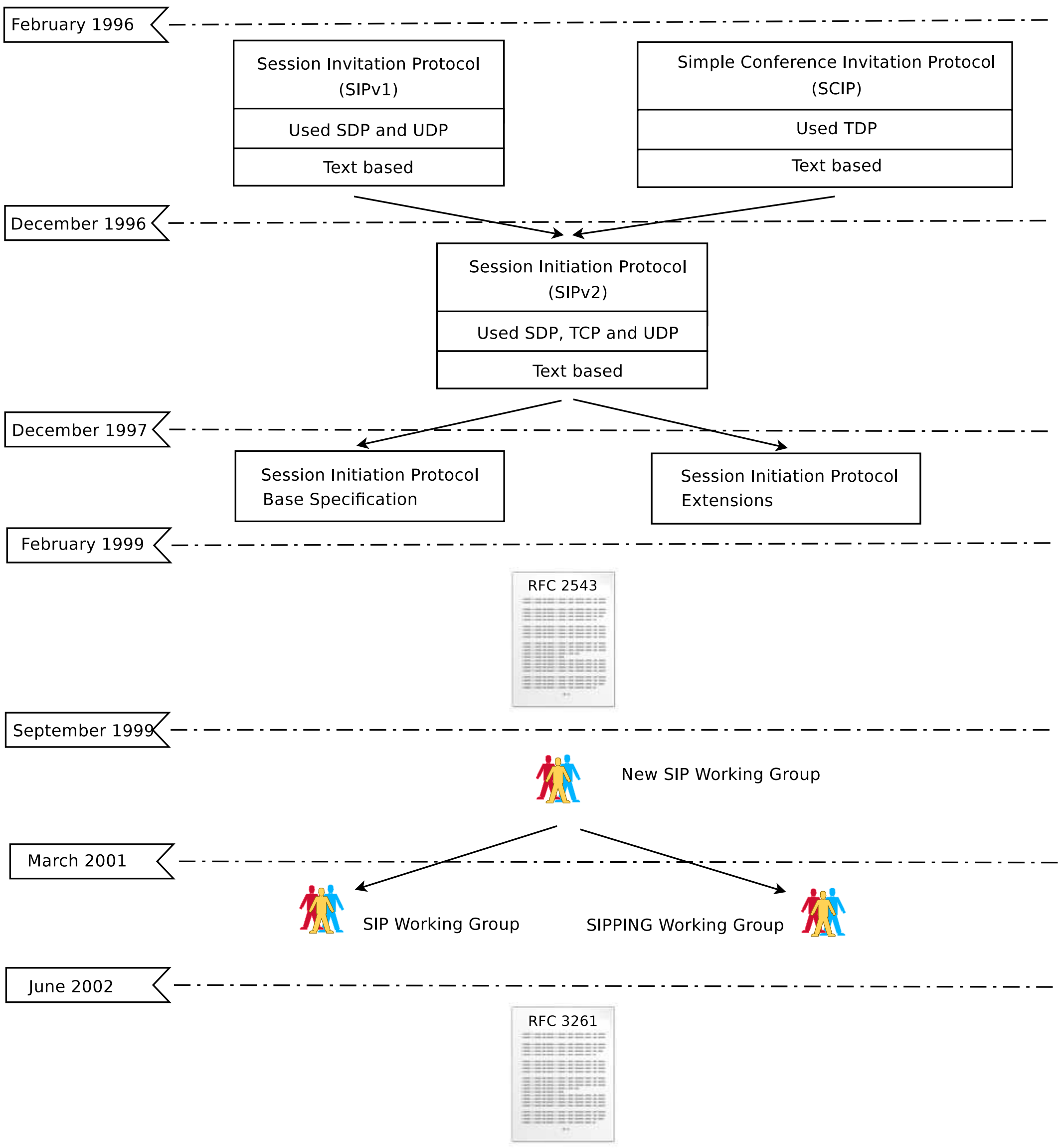


Figure 2.4: History of SIP

2.2.2 SIP Functionality and architecture

Functionality provided by SIP

SIP is a protocol for initiating, modifying, and terminating interactive sessions. SIP supports five facets of establishing and terminating multimedia communications [49]:

User location: determination of the end system to be used for communication, which does not mean a geographical location.

User availability: determination of the willingness of the called party to engage in communications.

User capabilities: determination of the media and media parameters to be used.

Session setup: establishment of session parameters at both called and calling party.

Session management: including transfer and termination of sessions, modifying session parameters, and invoking services.

Functionality not provided by SIP

SIP is not a protocol for device control or remote procedure calls. SIP is not a transport protocol, it can carry small attachments but it is not intended to carry large amounts of information [54]. SIP is also not a session-management protocol, but only a session-setup protocol. SIP is also not a VoIP protocol, although VoIP is one possible service that can be implemented over a SIP enabled network. SIP is purely a signaling protocol and makes no specification on media types, descriptions, services, and so on.

Entities of SIP

SIP has evolved to be able to set up a broad range of sessions, from multimedia (e.g., voice, video, etc) to gaming and instant messaging. SIP messages are simply either requests or responses. Some logical entities are required to carry those messages. It is important to understand the role of the entities in order to understand the architecture of SIP.

User Agents: An User Agent (UA) is the SIP entity that interacts with the user. Basically the UA is an endpoint in a SIP connection. An user agent can be a SIP phone or SIP client software running on a laptop or mobile phone, it can also be a gateway to another network, like an open gateway [49]. There are two types os SIP UAs; the User Agent Client (UAC) and the User Agent Server (UAS). The UAC initiates SIP requests and receives responses, while the UAS is the part of the user agent that generates responses to previous requests. Both user agents are part of every single SIP UA and both of them are used on a typical SIP session. In other words, if a piece of software initiates a request, it acts as a UAC for the duration of that transaction. If it receives a request later, it assumes the role of a user agent server for the processing of that transaction [20].

Servers: Servers such a proxy server are intermediaries within the SIP network. A server is a network element that receives requests in order to service them and sends back responses to those requests. Examples of servers are proxies, user agent servers, redirect servers, and registrars [49]. The proxy server acts as a link to connect two entities by forwarding the requests and responses it receives. In contrast the redirect server's response is a direction where the request should be retried. Finally, the registrar server updates the information of a database according to the information it receives in a request. [6] In the case of the proxy servers, various types exist depending on the state they keep.

Stateless proxy, one that does not keep any state when forwarding requests and responses, a simple message forwarder.

Stateful proxy, a proxy that stores state information during the transaction.

Call stateful proxy, a proxy that stores all the state regarding a session.

[6]It is important to note that SIP servers are not neededl on P2PSIP overlays, as we will see in the further chapters.

2.2.3 A typical SIP session

On Figure 2.5, we can see an example of a SIP session. We have two users, Maria and Javier, who want to start a VoIP call. Maria uses a mobile phone with a SIP application and Javier uses some software installed on his laptop. Both are using SIP Uniform Resource Locators (SIP URLs), whose format is very similar to an email addresses; SIP URLs consist of a hostname, username, a port number and other parameters. In the example in this case: *sip:maria@home.com* and *sip:javier@abroad.com*

The process of connecting and starting a media session is transparent to the user and is as follows:

1. Maria sends an INVITE request to the proxy located in Madrid, *sip:proxy.madrid.com*. The SIP URL of the proxy could also indicate the port and the transport method. For example: *sip:proxy.madrid.com:5060;transport=UDP*.
2. The *sip:proxy.madrid.com* proxy server locates Javier's proxy server in Helsinki, *sip:proxy.helsinki.com*.
3. The *sip:proxy.helsinki.com* consults a location service to find Javier's contact URL, which happens to be *sip:javier@abroad.com*.
4. The invite request is forwarded to Javier, and a message appears on the screen of his laptop informing him that Maria wants to start a VoIP call.
5. Javier accepts the session by clicking on a button in the VoIP application.
6. The call confirmation is forwarded through the proxies up to Maria's mobile phone, which automatically sends an ACK request. This time the request goes straight to Javier, bypassing the proxies.
7. The media session, which in this case is a VoIP conversation, starts. It could be another type of multimedia session, or any session requiring SIP.
8. The session over SIP finishes the moment Maria hangs up the mobile phone. The SIP application sends a BYE message and the OK response which is returned by Javier's application terminates the session.

2.2.4 Transactions in SIP

As we have seen before, SIP has different types of transactions. A transaction consist of a request invoking a particular method, or function, on the server, and at least one response. The transaction types are: INVITE-ACK transactions, CANCEL transactions and regular transactions.

The INVITE-ACK transactions involve an INVITE and the ACK confirmation of the response. The CANCEL transaction is connected to a previous transaction and cancels it. A regular transaction is any transaction apart from the INVITE-ACK and the CANCEL.

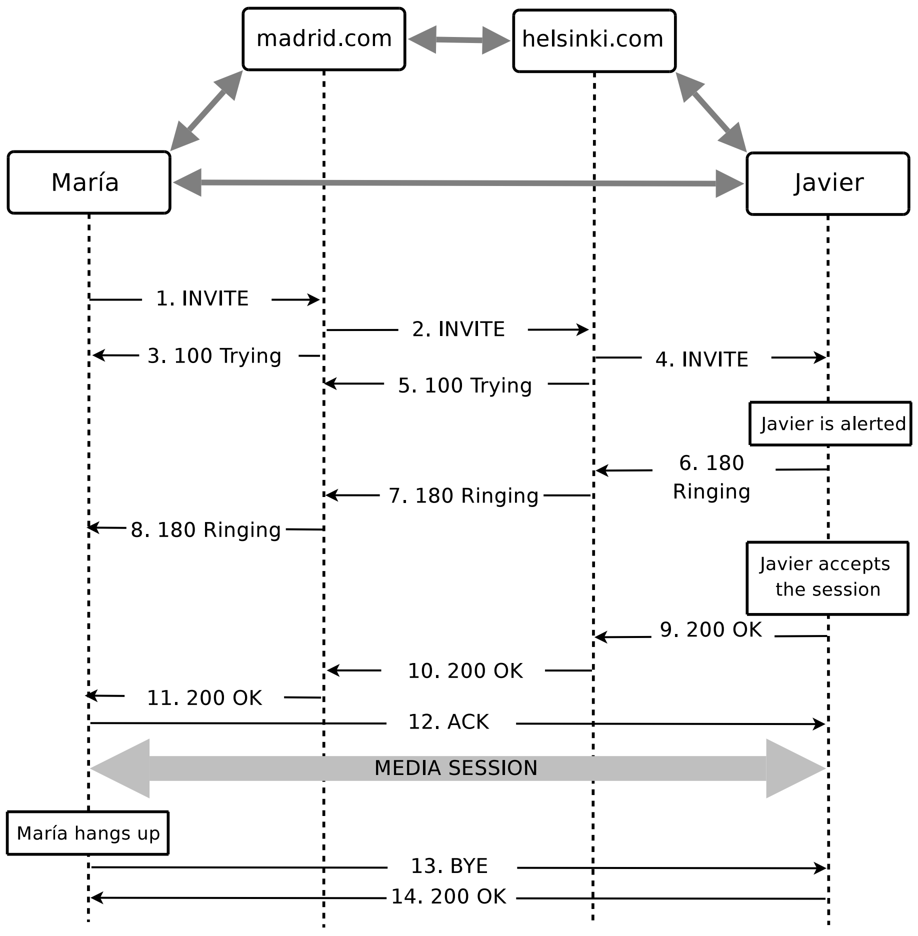


Figure 2.5: The SIP Trapezoid

2.3 Peer-to-Peer (P2P)

In recent years there has been a dramatic increase in the use of P2P systems, mainly because of their main characteristic which is their ability to provide a large combined storage and processing power while having low cost scalability capabilities.

2.3.1 Basic terminology

In order to understand what Peer-to-Peer computing or Client-Server computing means,it is necessary to review some common terminology.

Overlay Network: An overlay network is a computer network on top of another IP network, or on top of the Internet [54]. The nodes are connected using the IP addresses of the underlying network. Discovery and routing is done on the application layer at the endpoints.

Centralized Systems: represent single-unit solutions, usually there is one machine and various terminals. An example of this type are supercomputers and mainframes.

Distributed Systems: are those in which the computers form a network and such network is coordinated only by passing messages.

Distributed Computing: is a computer system in which several interconnected computers share the computing tasks assigned to the system [26]. This term is frequently associated with P2P.

Resource: is any hardware or software entity being represented or shared on a distributed network [57]. For example, a resource could be a computer, a file, a network service, a web page, etc.

Node: is a term used to represent a device connected to an overlay. Can be a server, a client or a peer.

Client: is a type of node, consumer of information. The client can only initiate requests but is not able to serve them. An example of a client is a mobile phone accessing the Internet.

Server: is a type of node, source of information. The server can only respond requests but is not able to initiate them. An example of a server is a server serving a web page.

Service: is a network-enabled entity that provides some capability [25]; An example of a service is a server is providing the capability of file retrieval.

Peer: a peer can be any device acting as both a consumer and a source of information. An example of a peer is a computer connected to a p2p network by means of a file-sharing application.

2.3.2 Client-Server Architecture

The client-server model is an example of distributed networking. In it the client receives the information from the server which serves various clients. The typical characteristics of the client are that it initiates requests, waits for replies and that it usually does not connect to a large number of servers at once. On the other hand, the server never initiates the communication, it only waits to receive a request and then replies to the client.

In this model, the information concentrates on the server side instead of being in the clients and it is not to be confused with a centralized sytem (see Figure 2.6). If the server stops working, then the whole network loses its source of information and therefore disconnects. The need to overcome this weakness lead to the creation of new ways to distribute the information without entirely depending on one single entity.

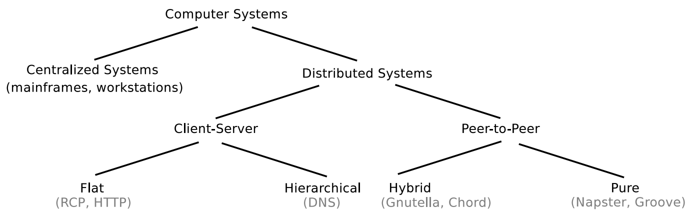


Figure 2.6: Taxonomy of Computer Systems Architectures.

A client-server architecture, like the one in Figure 2.7, also characterizes on the way the resources communicate with each other; when the information has to go from one client to another one, it always goes through a server. The clients can not locate each other without the information stored in the server.

2.3.3 Peer to Peer Architecture

Another type of distributed systems are P2P networks, in which the information is mainly stored in the peers, that is, in the network itself.

A P2P network can be defined as the *"shared provision of distributed resources and services"* [52]. Decentralization and autonomy are also characteristical of these networks [42], like the one in Figure 2.7.

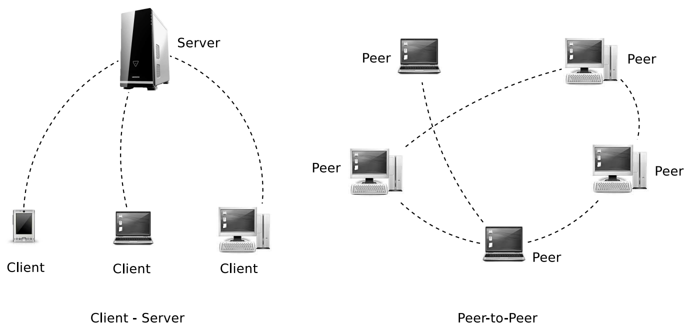


Figure 2.7: Typical client-server and peer-to-peer systems.

Sharing of distributed resources and services: In pure decentralized networks, there is no such thing as a client or server since these roles are played by each node, which become thus peers.

Decentralization: When considering pure p2p networks, there is no central coordination, since each of the nodes operates at the same level [51]. In between fully centralized and fully decentralized architectures, there is a range of different structures, whose boundaries are not clearly delimited. For example, there are P2P networks that use a central server to store the security certificates like in P2PSIP (see Section 2.4).

Autonomy: Usually each node of the network can choose wether to share its resources or not. In addition, some parts of the network may form a separate p2p network.

One may ask how the change from centralized networking into decentralized occurred. Actually, the Internet started as a P2P system. If the ARPANET is analyzed according to the previous characteristics it can be noticed that it operated as a peer to peer network as its aim was to share resources among different computers throughout the United States in a decentralized fashion. The firsts nodes of the ARPANET were various universities, and they were connected as peers of the same network architecture rather than as clients of an unique server [57]. In the early stages of the Internet there was no NAT traversal problem [24], since each machine was permanently connected to the Internet and had a static IP address. Firewalls were unknown until the late 1980s. Generally, any two machines on the Internet could send packets to each other. The appearance of Network Address Translators (NATs) and firewalls caused a lack of connectivity for applications that did not use the client-server paradigm [18]. Meanwhile, the use of the Internet spread massively; users connected from their own computers with dial-up connection and due to the decrease of static IP addresses Internet Service Providers (ISPs) started using dynamic IP addresses. Each PC had relatively brief sessions instead of being permanently connected, and had a different IP in each of the sessions. In the last twenty years, the Internet grew to unimaginable levels, also accompanied with the decrease of the computer components, bandwidth and storage modules. But it was the design of standard NAT traversal mechanisms [18] such as Interactive Connectivity Establishment (ICE) [47], together with the ease of scalability and interoperability of P2P networks [34] that there has been a striking increase in P2P-based computing [11], such as: search engines, games, file-sharing software (as well as a large sharing community), communication and security systems.

Unstructured P2P

An unstructured P2P network is one in which there is no clear definition of what the topology of the network should be. Since there is no definition of the location of the nodes, the searching process sometimes consumes a lot of bandwidth and processing power. For instance, if a peer needs to find one particular resource, it has to send a request to one or multiple adjacent nodes which will forward the request to their adjacent peers and so on. This can cause many problems regarding security since an exponential request attack can be easily carried on. Therefore, each message carries a time stamp, named time-to-live (TTL), which forces the elimination of each request after a certain number of hops. Another major drawback is that this structure does not guarantee that the search will be successful at all.

Still, it is very stable and far more simple than structured alternatives.

Structured P2P

A structured P2P network is an overlay network that uses a defined protocol to ensure the reachability of all peers, so that any node can efficiently route a search to any other peer in the overlay.

Distributed Hash Tables (DHTs) are commonly used for this purpose. In structured P2P architectures, each node is assigned an unique ID when it joins the overlay and the data it wants to store is assigned another ID (i.e., key). That key indicates where the data will be stored in the overlay, and is created by means of a hashing function. Later, when a search is performed, each node will consult its hash table, which contains a range of keys indicating the responsible nodes for each data. If the node doesn't find the information of the searched key, it will simply forward the request to another node, which will be determined using the specific overlay algorithm of the network. In the case of Chord, which is probably the most widely known DHT algorithm, the request is forwarded to the node in the routing table that is closest to the target ID.

The topics of the Distributed Hash Tables and Chord are more thoroughly explained in section 2.4.2.

2.3.4 Examples of well known P2P Systems

Among the many examples of different P2P systems, this section presents a few but relevant examples. Nowadays the largest P2P systems are used for file sharing, but there are also developments in other fields [58] such as: research, communications and even military projects.

File Sharing

Examples of P2P file sharing networks include Gnutella, Kad and Bittorrent.

Gnutella [7] is an unstructured system that is used mainly for file sharing. Its main focus is on decentralization (see Figure 2.8). The Gnutella network had 40.5% [9] of the market share in 2007, being thus the most widely used P2P sharing network in that year.

Kad is a P2P network that implements the Kademlia P2P overlay protocol. Once the file transfer starts, peers connect directly to each other using the standard IP network. It is often used in conjunction with some applications like emule or edonkey, although they have their own P2P network.

BitTorrent [8] is another of the heavy-weights in the P2P sharing world, with 28.2% [9] of the market share in 2007, and probably more in 2008.

Research

Examples of P2P research networks include LionShare, SETI@home and Folding@home.

LionShare [9] is a fairly new network, the first release of which dates back to October 2007. Its aim is to provide a P2P sharing tool for academical purposes. Its main innovation is that it searches also on academic databases in addition to the files of the peers. It includes a method to control the access to the files in order to ensure security.

SETI@home [10] which is hosted by the Space Sciences Laboratory at Berkeley University, was extremely popular in the early years of the 21st century. It distributes computation among numerous personal computers, which are ofen idle. This type of distributed computing has also been used to crack encryption challenges.

Folding@home [11] which was launched in October 2000, is another example of a distributed computing cluster, probably the largest in the world [1]. The goal of this project is to research protein folding and misfolding to gain an understanding of how these are related to disease. Targeted diseases include (but are not limited to) Alzheimer's, Parkinson's, and many forms of cancer.

[7] see http://rfc-gnutella.sourceforge.net/
[8] see http://www.bittorrent.org/
[9] see http://lionshare.its.psu.edu
[10] see http://setiathome.berkeley.edu/
[11] see http://folding.stanford.edu/

Military

There has been some research, at least by the US government, on the possibilities to use P2P on tactical military networks, though most of this research is still inaccessible for the general public. An example of this is the Small Unit Operations - Situational Awareness and Information Management (SUO SAIM) project. SUO SAIMs information management layer is a self-organizing P2P system. The US military had to develop a fault tolerant architecture, capable of adapting to changes in the communication architecture and efficiently utilize the bandwidth available [27]. Therefore they used a P2P approach instead of a client-server one. There has also been some development on collaborative networks of sensors [13] which aim is to share the information between these sensors which can be either fixed on the ground, mobile on the ground, or located on air carriers or satellites.

Communication

Examples of P2P communication networks include Skype and Jabber

Skype [12] is the celebrated software that allows users to make telephone calls over the Internet. It uses a type of P2P network, with probably (since the protocol has not been made publicly available) a centralized server to handle authorization and user information. Skype can autodetect NAT and firewall settings and has super peers, which are peers that have been promoted to this status because of their capacity or connectivity (see Figure 2.8). Skype is available on different devices and has around 405 million users [8].

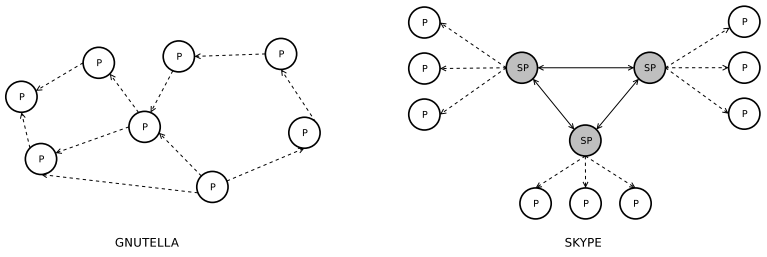


Figure 2.8: Comparison between Gnutella and Skype networks

Jabber [13] is an open source project that combines instant messaging (supporting many popular systems) with XML, creating a decentralized network of IM services. However, clients do not speak directly with each other. Jabber uses an unique Jabber ID (such as *username@domain.com*) to avoid the need for a central server for identification.

[12]see http://www.skype.com
[13]see www.jabber.org/

2.4 Peer-to-Peer SIP (P2PSIP)

While P2P networks offer higher scalability and reliability than client-server networks, they trade off with higher delay in lookup service, especially in large-scale networks. When the P2P model is used on these networks, there are some mechanism to organize the information, so that the search for data has deterministic results, like Distributed Hash Tables (DHTs).

Depending on the topology of the network, different approaches can be used, currently the efforts are concentrating on REsource LOcation And Discovery (RELOAD), a peer-to-peer (P2P) signaling protocol for use on large networks like the Internet.

2.4.1 The Topology Problem

P2P architectures differ greatly in their structure, varying from hierarchical structures to ring or decentralized ones [57] or a hybrid combination of them. These variations and compositions are made to ensure the combination of the best features of each structure in order to optimize the topology according to its use.

P2P architectures can be divided into simple topologies (centralized, decentralized, ring, hierarchical) and composed or hybrid topologies (client/ring topology, centralized/centralized, centralized/decentralized). From the last type, we will provide just a few examples that suit best for this thesis since there are numerous possible combinations.

Simple Topologies

Among the most commonly used topologies we have the *Centralized Topology* (see Figure 2.9), which is used in client/server architectures, with the clients connecting to a single peer. We also have the *Ring Topology* which is a loop of nodes. It is usually used on large server farms because of its ease of escalation and load balance possibilities. Sometimes ease of search is what is looked for in a network, for which *Hierarchical Topologies* are used. They are tree-like structures depending on usually one or few nodes. A typical example are authorisation entities such as DNS. Another basic structure is *Decentralized Topologies*, they are used because of their robustness against random peer failures, but the search is extremely inefficient.

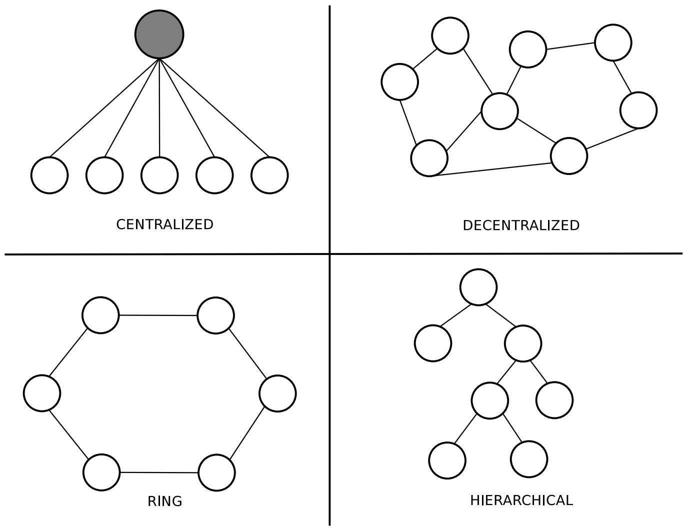


Figure 2.9: Most common simple network topologies

Composed Topologies

The next step of a client/server architecture is when instead of just one server there are various, so that they serve and organize the information more efficiently [22]. This is achieved in the *Centralized/Ring Topology*, used in robust web servers (see Figure 2.10) combining the simplicity of a client-server system with the robustness of a ring. *Centralized/centralized* topologies are also used in some web applications, when there is need for a central control but the features of a central server are distributed among other machines. This is the case of Google as for instance, the web server contacts a distributed database of links. *Centralized/decentralized* topologies are used to enable fault tolerance and centralization at the same time. It is similar to a hierarchical topology, where just some nodes belong to the upper centralized class. It is used in Skype, on other social networks, and various P2P systems.

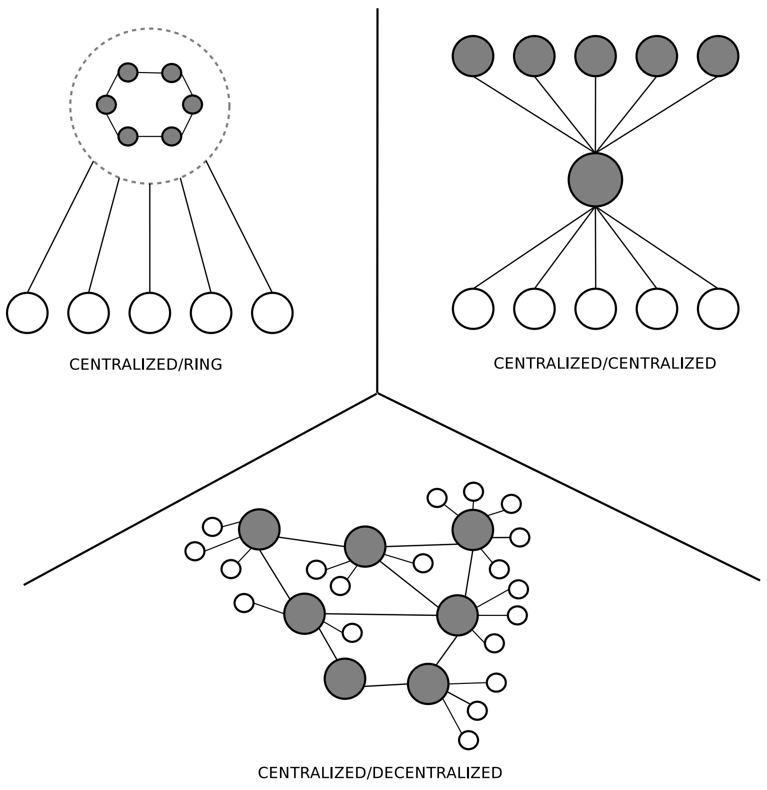


Figure 2.10: Some composed network topologies

2.4.2 Use of Distributed Hash Tables (DHTs)

A *hash function or hash algorithm* is a mathematical function that transforms a large amount of data into a small value, which is also usually encrypted. These functions are used on *hash tables* or hash maps, which are structures made for referencing, as they associate the hashes or keys with real values. Hash tables have various purposes; one of them is indexing, keeping an array of *(key, value)* data types. They are useful for lookup of information since the indexing makes the process very efficient.

On P2P networks, the information is stored at the endpoints of the network (i.e. peers) rather than on a single or various servers. The hash table approach is not possible here since the data has to be distributed more or less equally among the peers. Instead, *Distributed Hash Tables (DHTs)* are used. They are created by storing the contents of the hash table across a set of peers on a P2P network according to certain algorithms to ensure optimal distribution. Each of the peers will be responsible for part of the key space.

The distribution of the information through the network of peers is not random, it has to be ensured that it is equally distributed among the peers and it has to change according to the changes in the network, since nodes can join or leave it at any time. An example of routing performance in a DHT overlay network can be seen at [39]. To distribute the information,

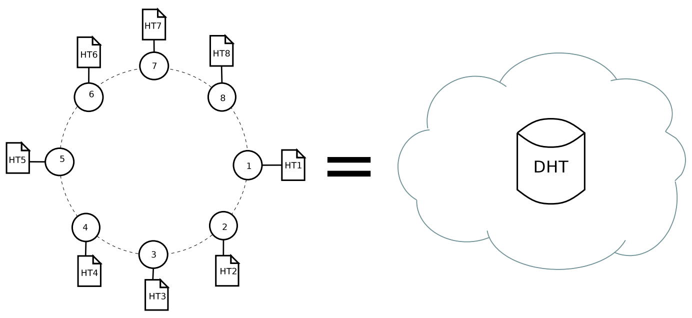


Figure 2.11: Hash Tables of an Overlay forming a Distributed Hash Table

most of structured P2P architectures use a DHT algorithm. Among the many algorithms, like Kademlia [40], Pastry [50], CAN [45] and Chord [56], we are going to focus on the last one, mainly because it is the one most widely used and is also the default algorithm used by RELOAD. A detailed comparison of these algorithms can be found in [35].

Chord

A fundamental problem that confronts peer-to-peer applications is locating the node that stores a desired data item. On a typical P2P network, a peer will forward a request for certain data to other peers, which in turn forward it to several other connected peers. This type of search is called flooding the network, and causes a problem on the network actually inhibiting it from retrieving the data.

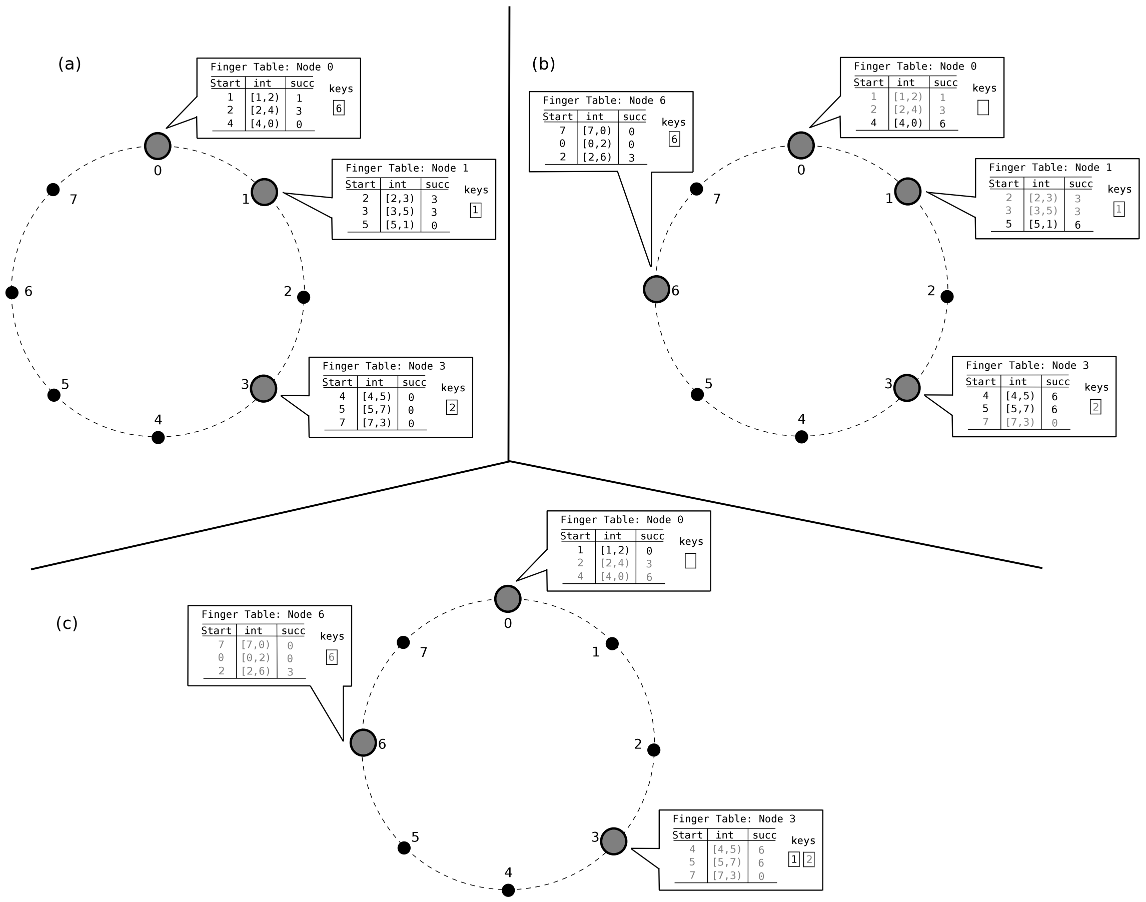
A solution to this problem comes with Chord. It is a fully distributed peer-to-peer lookup algorithm [56] used to map a key to a specific node. It can be used in various topologies [53]: centralized/ring, the classic server farm, hierarchical centralized/decentralized structure with some super-nodes among the rest of the peers (e.g. the one used on Skype), ring where the nodes are equals and their only difference is some resource limitation (see the types of topologies in Section 2.4.1).

Chord has just one operation: *providing that you give a key, Chord can map it onto a node*. Chord uses a distributed hash table for routing. If we have N nodes and K keys, each node is responsible at most for $(1 + O(logN))K/N$ keys. It has been demonstrated [56] that Chord can solve any given lookup by sending information to a maximum of $O(log\ N)$ nodes. This means that even when N is a large number, the number of messages sent by a node using Chord will still be relatively small. The nodes are arranged on a ring. On the Chord ring each node has a successor and a predecessor. Since P2P nodes join or leave the network freely, nodes maintain multiple successor pointers to improve robustness.

In Figure 2.12, we can see an example of a Chord ring. In Figure 2.12(a) Nodes 0,1 and 3 are connected while nodes 2,4,5,6,7 are not. The size of the network is 8 and the currently available files/keys map to nodes 1,2 and 6. Since node 6 does not exist, the file (key=6) is mapped to the first available node, in this case node 0. The file (key=2) is mapped onto node 3. Chord continually maps the files along the Chord ring as peers join and leave the network. For instance, In Figure 2.12(b), if the Node 6 joined the overlay, the file (key=6) would be stored in that node, and also the successors of the nodes would vary according to this new topology [55]. In Figure 2.12(c) when node 1 leaves the overlay, the finger tables are adjusted. And the key node 6 is responsible for are assigned to the next peer, in this case Node 3.

In Chord, there are two routing modes: *iterative and recursive* [56]. In *recursive routing*, the request is forwarded to the next hop until it reaches the responsible node who will reply [14] to the initial node. In Figure 2.13(a) we can see an example in which node 1 asks for key 12. Node 1 first checks its own table but is not responsible for that key so it forwards the request to the next hop, which is node 10. Node 10 checks its routing table, and forwards the request to the node that is supposed to have the information. this node is node 14. Finally, node 14 replies to node 1 with the value associated with key 12. Figure 2.13(b) shows an example of *iterative routing*. The request involves the very same nodes but the message is not forwarded to the next hop. Instead, each node along the path returns a response to node 1. Although recursive routing has a better performance, iterative is usually recommended for security reasons, since the message flow can be properly controlled. The routing used in

[14] In Chord, the response is not returned directly, but along the reverese path followed by the query



Finger Table: Node 0

Start	int	succ
1	[1,2)	1
2	[2,4)	3
4	[4,0)	0

keys 6

Finger Table: Node 1

Start	int	succ
2	[2,3)	3
3	[3,5)	3
5	[5,1)	0

keys 1

Finger Table: Node 3

Start	int	succ
4	[4,5)	0
5	[5,7)	0
7	[7,3)	0

keys 2

Finger Table: Node 6

Start	int	succ
7	[7,0)	0
0	[0,2)	0
2	[2,6)	3

keys 6

Finger Table: Node 0

Start	int	succ
1	[1,2)	1
2	[2,4)	3
4	[4,0)	6

keys

Finger Table: Node 1

Start	int	succ
2	[2,3)	3
3	[3,5)	3
5	[5,1)	6

keys 1

Finger Table: Node 3

Start	int	succ
4	[4,5)	6
5	[5,7)	6
7	[7,3)	0

keys 2

Finger Table: Node 0

Start	int	succ
1	[1,2)	0
2	[2,4)	3
4	[4,0)	6

keys

Finger Table: Node 6

Start	int	succ
7	[7,0)	0
0	[0,2)	0
2	[2,6)	3

keys 6

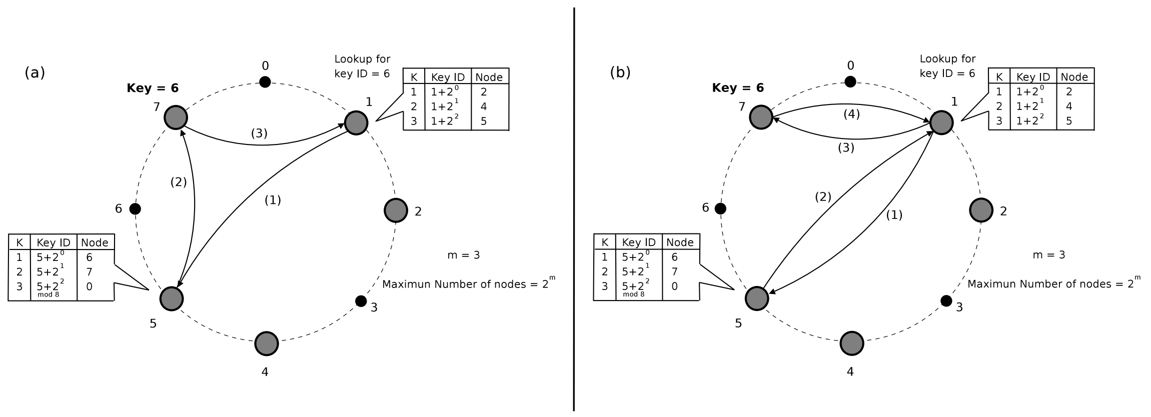
Finger Table: Node 3

Start	int	succ
4	[4,5)	6
5	[5,7)	6
7	[7,3)	0

keys 1 2

Figure 2.12: Example of a Chord ring (a), with the finger tables of the connected nodes. Examples of a node joining (b) and leaving (c) the overlay

RELOAD is slightly different from the recursive mode and it is explained in section 2.5.2.



K	Key ID	Node
1	$1+2^0$	2
2	$1+2^1$	4
3	$1+2^2$	5

K	Key ID	Node
1	$5+2^0$	6
2	$5+2^1$	7
3	$5+2^2$ mod 8	0

K	Key ID	Node
1	$1+2^0$	2
2	$1+2^1$	4
3	$1+2^2$	5

K	Key ID	Node
1	$5+2^0$	6
2	$5+2^1$	7
3	$5+2^2$ mod 8	0

Figure 2.13: Examples of recursive (a), and iterative routing (b)

2.5 REsource LOcation And Discovery (RELOAD)

In traditional SIP architectures, there is a hierarchy of SIP routing proxies and SIP user agents that follow a client/server structure. To start a communication session using SIP, a SIP user agent will send an SIP invite request to a proxy which will send it to the destination UA of the message. In P2P networks, this type of communication is not possible, since there are no proxy nodes that centralize the mapping. Instead, the mapping function is distributed among the peers of the network. This P2P overlay network is organized and maintained using the P2PSIP peer protocol, which provides for user and resource location in a SIP environment with no or minimal centralized servers [16], [17].

Nowadays Skype is among the firsts to adapt a similar technology on VoIP communications, but it does not use standard protocols like RELOAD and SIP and most probably uses a central server for indexing and authorization purposes even if this can not be ascertained since the protocol is closed source. Also, Skype is not compatible with other VoIP systems, making it useless for the research community. The IETF P2PSIP Working Group [6] is chartered to develop standard protocols and mechanisms for P2PSIP systems. Currently the P2PSIP WG has focused its efforts on the development of the *REsource LOcation And Discovery (RELOAD)* protocol, which is the peer protocol to be used in P2PSIP networks.

The RELOAD protocol is still under development and there is no implementation available at the time of writing this thesis. Its characteristics are defined in [30].

In 2007, there were several competing proposals for the P2PSIP peer protocol; RELOAD [30], Peer-to-Peer Protocol (P2PP) [14], Address Settlement by Peer-to-Peer (ASP) [31], Service Extensible P2P Peer Protocol (SEP) [32], Extensible Peer Protocol (XPP) [38] and Host Identity Protocol HOP (HIPHOP) [21]. At the end of the year, RELOAD and ASP were merged and by February 2008 also P2PP was merged to the combined RELOAD/ASP protocol. This version then became an official working group item of the P2PSIP working group.

Some of RELOAD's features are that it works on environments where there are NATs or firewalls. RELOAD can support various applications and provides a security framework, since connections in a P2P network will often be established among peers that do not trust each other. RELOAD also allows the use of various DHT algorithms in the form of topology plugins. The one that is being currently implemented is Chord.

2.5.1 RELOAD Architecture

In Figure 2.14, we can see the basic architecture of RELOAD [30].

Usage Layer: Each application that wishes to use RELOAD defines a RELOAD usage; examples include a SIP Usage, XMPP Usage or any other. These usages all talk to RELOAD through a common Message Transport API. The applications can use RELOAD to store and retrieve data, as a service discovery tool or to form direct connections in P2P environments. Currently defined usages are the SIP usage, the certificate store usage, the Traversal Using Relays around NAT (TURN) [48] server usage and the diagnostics usage. Message

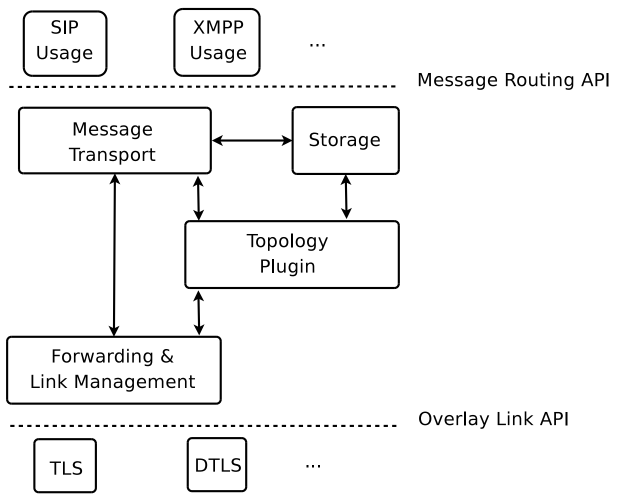


Figure 2.14: Major components of RELOAD.

Transport layer provides a generic message routing service for the overlay, that is sending and receiving messages from peers. The Storage component is responsible for processing messages relating to the storage and retrieval of data.

Topology Plugin: RELOAD is specifically designed to work with a variety of overlay algorithms. However, for interoperability reasons, RELOAD defines one algorithm, Chord, that is mandatory to implement. The topology plugin defines the content of the messages that will be used in RELOAD, the various procedures to join and leave an overlay, the hash algorithm used (the default algorithm used is Secure Hash Algorithm 1 (SHA-1)), the replication strategy, and the routing procedures. Forwarding and Link Management Layer provides packet forwarding services and sets up connections across NATs using Interactive Connectivity Establishment (ICE) [47].

Overlay Link Layer: Currently Transport Layer Security (TLS) over TCP and and Datagram Transport Layer Security (DTLS) over UDP are the "link layer" protocols supported by RELOAD for hop-by-hop communication.

2.5.2 RELOAD Network

The RELOAD Network is very similar to any other P2P network. However, one of the differences is that not all the members of the network are peers, in fact there are also clients (see Section 2.3.1). A node might act as a Client simply because it does not have the resources or does not implement the topology plugin required to act as a peer in the overlay. Still, a client uses the same RELOAD protocol as the peers, knows how to calculate Resource-IDs, and signs its requests in the same manner as peers. RELOAD is going to have a credential and an enrollment server too, although the role of such servers in a full P2P network is still under debate.

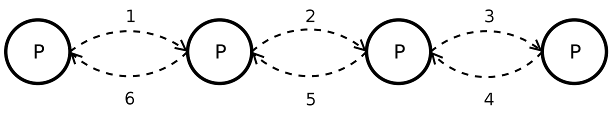


Figure 2.15: Example of symmetric recursive routing.

The routing mode in RELOAD is *symmetric recursive*, which is similar to recursive routing but with the difference that the return path of the response is the same as the path followed by the request in reverse. For instance, if a message goes from A to D through B and C, the returning path will be D,C,B,A. RELOAD uses this routing because it is the only available option (see Figure 2.15).

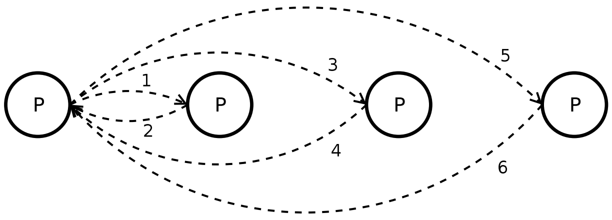


Figure 2.16: Example of iterative routing.

Iterative routing is not possible since a message may need to be sent to a peer that is not present in the routing table, which requires a new direct connection to be established, becoming the latency too high for the communnication to be efficient (see Figure 2.16).

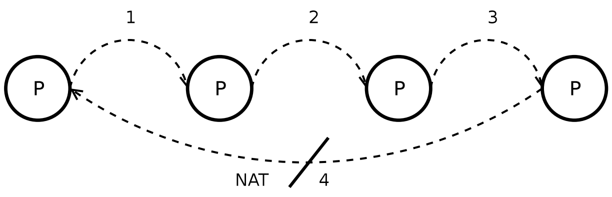


Figure 2.17: Example of pure recursive routing.

The pure *recursive routing* can not be applied either for similar reasons; if a node behind a NAT receives a message response that has not been previously requested, the NAT will drop the message, making the communication impossible (see Figure 2.17).

2.5.3 RELOAD Message Examples

For the purposes of this thesis, we are going to focus on RELOAD's message system, therefore it is important to understand how RELOAD's message flow works. We will focus on relevant procedures that a node has to perform in the overlay, such as: lookup, joining, and leaving.

The *Lookup* or Find process (see Figure 2.18) is done in a symmetric recursive fashion as was shown in Section 2.5.2:

1. A peer sends a Find Request to its neighbor peer to find a peer closer to a requested id.
2. If they are using UDP, an ACK will be sent upon receival of each message by each hop.
3. The lookup is forwarded until the node responsible of that information is found, then it responds with a Find Answer.

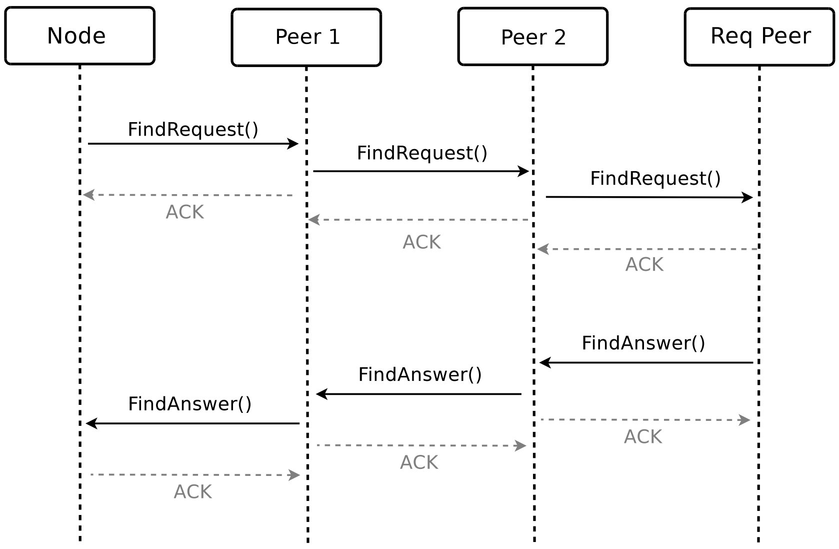


Figure 2.18: Example of Find Request using UDP.

The *Joining* process involves a Joining Party (JP) connecting and becoming part of an overlay [30] (see Figure 2.19). The different steps are:

1. JP connects to its chosen bootstrap node and sends a series of Probes to populate its routing table.
2. JP sends Attach requests to initiate connections to each of the peers in the connection table as well as to the desired finger table entries.
3. JP enters all the peers it contacted into its routing table.

4. JP sends a Join to its immediate successor, the admitting peer (AP) for Node-ID. The AP sends the response to the Join.

5. AP does a series of Store requests to JP to store the data that JP will be responsible for.

6. AP sends JP an Update explicitly labeling JP as its predecessor. It also sends an Update to all of its neighbors with the new values of its neighbor set (including JP).

7. JP sends Updates to all the peers in its routing table.

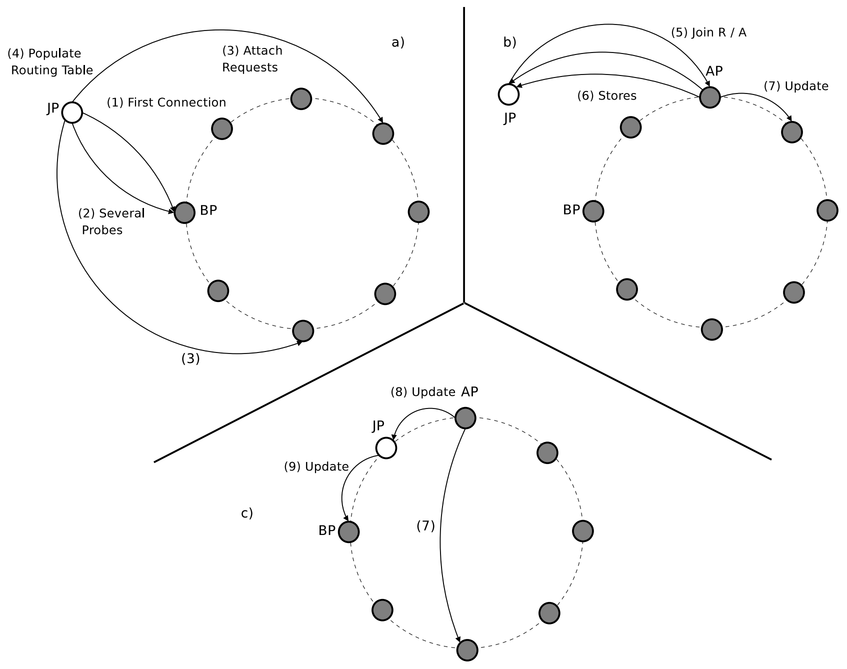


Figure 2.19: A node connects to a Bootstrap Peer (a). At this point (b) the node is already part of the overlay and (c) updates the peers.

The *Leaving* of an overlay is performed when a node departs from it. The node sends the Leave message to its predeccessor and its successor and they send the response back. Both nodes remove the leaving peer from their own table. The periodic stabilization will afterwards update the rest of the nodes [30] [36].

Chapter 3

Implementation

This chapter explains the implementation of the ReloadMessage structure and the rest of the classes needed to generate, send and receive messages, as well as several UML diagrams of important parts of the code and their explanation.

The implementation of the RELOAD Message Structure has been carried out following different drafts, thoroughly defined by the P2PSIP Working Group (see [30]) and other drafts (see [36]). This implementation can be reused in future versions of RELOAD since it has been implemented following the current standards.

The rest of the implementation regarding the sockets, the server, and the client has been done for the sole purpose of testing the messages and the traffic characteristics of RELOAD.

For notation purposes a presentation language has been commonly adopted for RELOAD. This presentation language is similar to C but RELOAD can be implemented in any different language.

3.1 RELOAD Message Structure

This section explains the general characteristics of the implementation. For each relevant part, a more detailed explanation of the source code is included, in addition to class diagrams and functionalities.

The structure of a RELOAD Message consists of a forwarding header, compulsory in all messages; the message content; and the security block. Each message will differ on the information contained and the purpose of the message, but the main parts will always be the same (see Figure 3.1).

The structure has been developed to satisfy the requirements of a request/response protocol. It is intended to be extensible, therefore it uses fields such as length, type and value to ease the process of adding new fields. In its design, interoperability is also a key factor, that is why some parts of the structure have been mandatory to implement and are compulsory in other implementations of RELOAD. The structure of every RELOAD message has three concatenated parts: forwarding header, message content, and security block (see Figure 3.1).

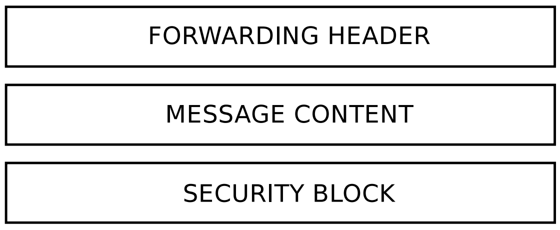


Figure 3.1: Parts of a RELOAD Message

The Forwarding Header is used for routing purposes within the overlay and it also allows peers to identify a message as a RELOAD message. The Message Content is opaque from the perspective of the forwarding layer but it is interpreted by higher layers. The security Block contains the certificates and the signature over the messages. The signature is also used in some messages related to storing information and it can be computed without parsing the message contents.

The different data types [1] and their equivalence in bytes are:

[1]The opaque values always contain a length field of 1, 2 or 3 bytes. The length field indicates the length of the opaque in bytes.

uint8	1 byte or 8 bits
uint16	2 bytes or 16 bits
uint24	3 bytes or 24 bits
uint32	4 bytes or 32 bits
Boolean	1 byte or 8 bits
byte[]	A byte array of undetermined length
byte[n]	A byte array of n-bytes length
vector<n..m>	A vector that can contain from n to m values
opaque value<0..$2^8 - 1$>	A byte array of undetermined length (up to 2^8 bytes)
opaque value<0..$2^{16} - 1$>	A byte array of undetermined length (up to 2^{16} bytes)
opaque value<0..$2^{32} - 1$>	A byte array of undetermined length (up to 2^{32} bytes)

3.2 Forwarding Header

Each message has a header which is used to forward the message between peers and to its final destination. This header is the only information that an intermediate peer (i.e., one that is not the target of a message) needs to examine. The structure of the forwarding header has the following components [30]:

```
struct {
        uint32              relo_token;
        uint32              overlay;
        uint16              configuration_sequence;
        uint8               ttl;
        uint8               reserved;
        uint32              fragment;
        uint8               version;
        uint32              length;
        uint64              transaction_id;
        uint16              flags;
        uint16              via_list_length;
        uint16              destination_list_length;
        uint16              route_log_length;
        uint16              options_length;
        Destination         via_list[via_list_length];
        Destination         destination_list[destination_list_length];
        RouteLogEntry       route_log[route_log_length];
        ForwardingOptions   options[options_length];
 } ForwardingHeader;
```

The different fields are: the relo_token, which contains the value 0xc2454c4f (the string 'RELO'); the overlay field or checksum of the overlay being used, which allows nodes to participate in multiple overlays and to detect accidental misconfiguration; the configuration_sequence, which is the sequence number of the configuration file; the ttl (time to live) field, which is a byte field that indicates the number of iterations, or hops, a message can experience before it is discarded; the fragment, which is used to handle fragmentation; the version field, which is the version of the RELOAD protocol being used; the length field, that indicates the size in bytes of the whole message; and the transaction_id, which is an unique 64 bit number that identifies the transaction being carried and also serves as a salt to randomize the request and the response. There are also control flags and the via_list_length, destination_list_length, route_log_length and options_length, which are the the length of their respective lists in bytes. The via_list field contains the sequence of destinations through which the message has passed, the destination_list contains a sequence of destinations which the message should pass through, the route_log contains a series of route log entries, and the options field contains a series of ForwardingOptions entries.

3.2.1 UML of the Forwarding Header

Of the Forwarding Header it is relevant to comment on the main class, that contains all the parts of the Forwarding Header (see Figure 3.2). In order to clarify the information, relations with primitive types (vector, byte, short, integer, long) are also shown.

The Forwarding Header implements the ReloadMessagePart and its main method is to encode the message by calling the encode() method. The equals() method makes a comparison between two Forwarding Headers and returns a Boolean value, which will be false if they are not equal and true if they are equal.

CalculateLengthForwardingHeader() calculates, as its name indicates, the length of the whole Forwarding Header by getting the lengths of the different fields and returning the total length. The length will be used in the length field of the Forwarding Header. Getters and setters, methods used to control changes to a variable, were also implemented.

The P2PSIP Logger has been used throughout the implementation for debugging purposes, it basically prints the relevant information in the console, printing also the possible errors that might arise.

There are several Vector (a growable array of objects) fields in the Forwarding Header; these fields have their own classes, for instance the DestinationList or the RouteLog.

The Destination List contains several entries of the type Destination (see Figure 3.3). The Destination class contains several values: the type of the Destination data which can be one of "peer", "resource", or "compressed"; the length of the destination data, which allows the addition of new DestinationTypes; and the destination value itself, which is an encoded Destination Data structure that depends on the value of "type".

A Destination Data can be one of three types: peer, which is a simple Node-ID; compressed, which is a compressed list of Node-IDs and/or resources represented in an opaque string; and resource, which is the Resource-ID of the desired resource.

In the diagram it can be noticed that there are three relevant methods, which are: encode(), parse() and equals(). There is no random() method since the way a new random destination is generated is by creating an instance of it filled with the pseudorandom values.

The RouteLog List contains several entries of the type RouteLog (see Figure 3.4).

The route log provides a means to store the information about the path taken by the message up to the receiving node. The route log class contains several values: version, which is a textual representation of the software version; the Overlay Link Layer protocol, for the test we carried out this field is "udp_dtls"; the Node-ID of the peer; the uptime of the peer in seconds; the address and port of the peer; the peer's certificate; and the Extensions, which have their own class (RouteLogExtension). The contents of that structure are: The type of the extension and the length of the rest of the structure.

The classes OpaqueByte and OpaqueShort, along with OpaqueInt have been implemented to ease the programming and they just contain a length field and then a byte array. This array will vary in length between 0 - 255, 0 - 65535 or 0 - 4294967295 bytes depending on whether their maximum length is represented by a byte, a short or an integer. These classes are used throughout the implementation.

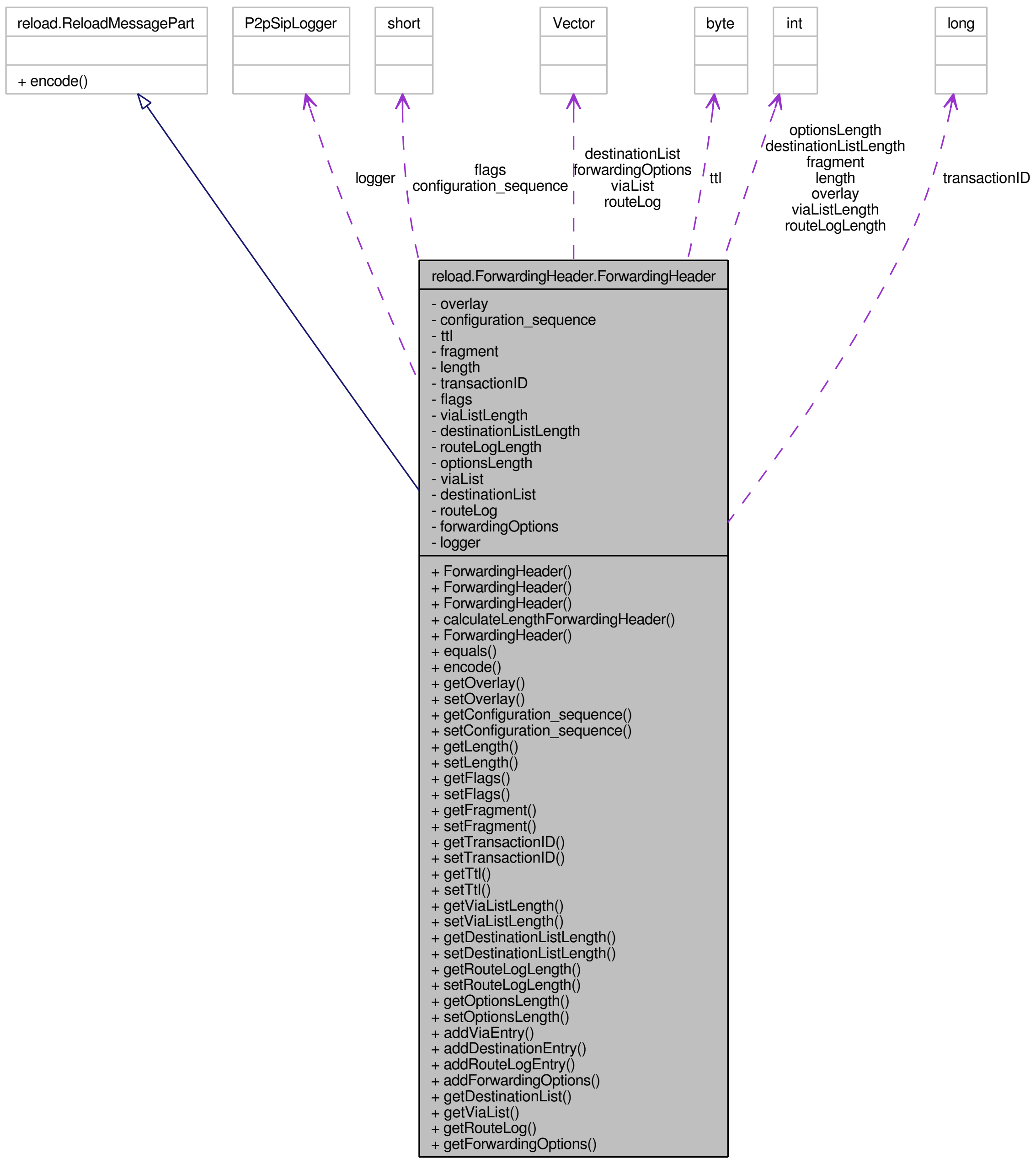


Figure 3.2: Class diagram of the Forwarding Header

The IpAddressPort class is interesting too, as it represents a network address and has been implemented in order to carry either an IPv6 or IPv4 address (see Figure 3.5). The first two fields in the structure are the same no matter what kind of address is being represented: the type of address (v4 or v6) and the length of the rest of the structure.

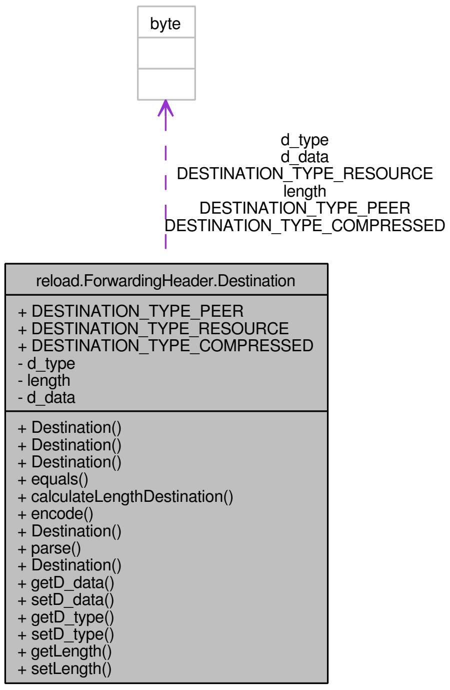


Figure 3.3: Class diagram of the Destination

The type and the length are at the beginning so that if an application using the RELOAD message Structure is parsing an IP Address and does not interpret the IP, it will still be able to parse the IpAddressPort field and then discard it, if it is not needed.

To represent the rest of the structure we have created two other classes, one for the IPv4 addresses and another one for the IPv6. They both contain a two-byte port value and a 4-byte (for the IPv4) or 32-byte (for the IPv6) value for the IP address. As an example:

FIELD	VALUE	HEX
type	IPv4	01
length	6	06
address	127.0.0.1	7F 00 00 01
port	5500	15 7C

The IPv4 address "127.0.0.1" with port "5500" would be sent through the wire, and its representation in hexadecimal values would be: 01 06 7F 00 00 01 15 7C.

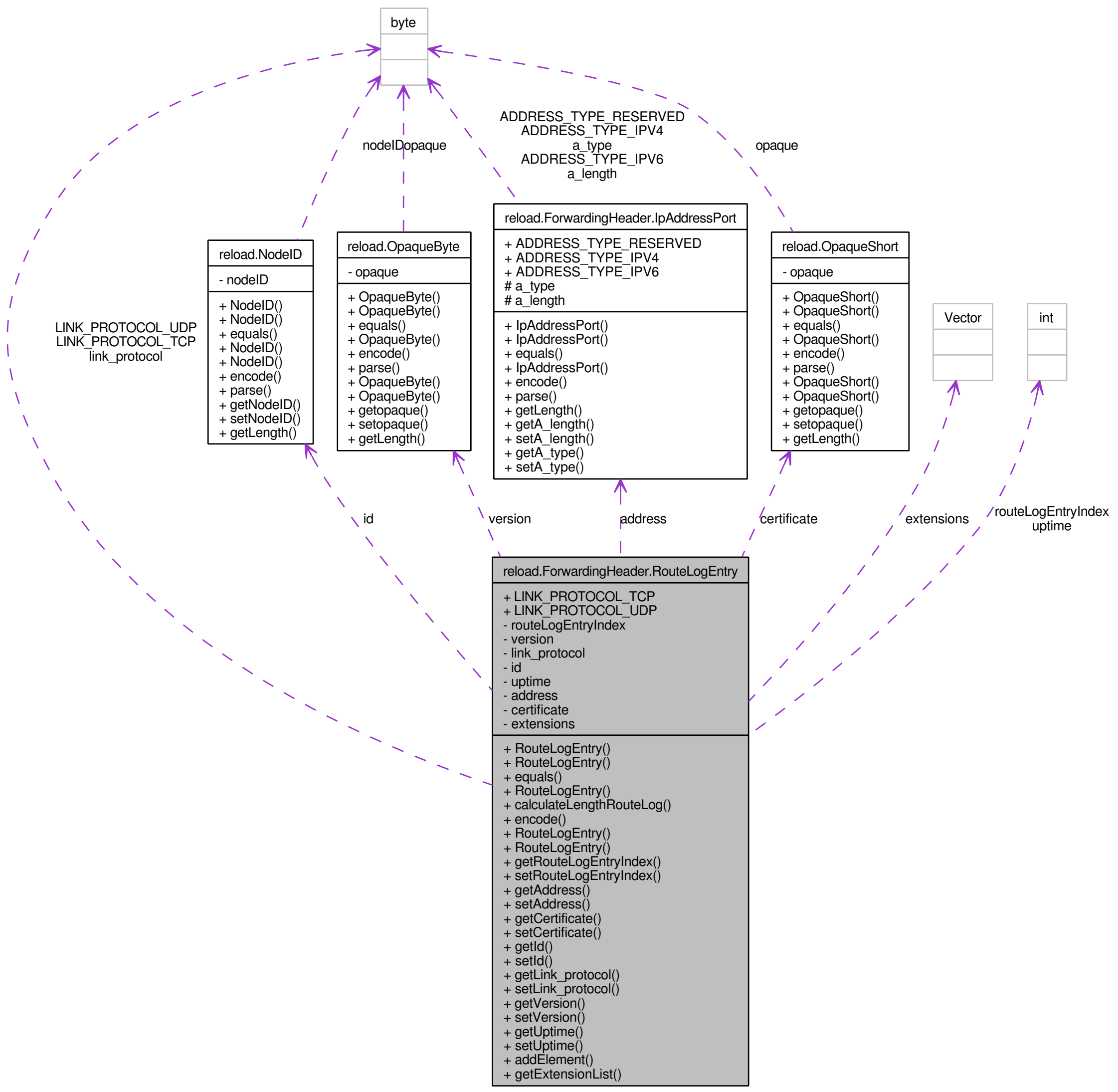


Figure 3.4: Class diagram of the RouteLog

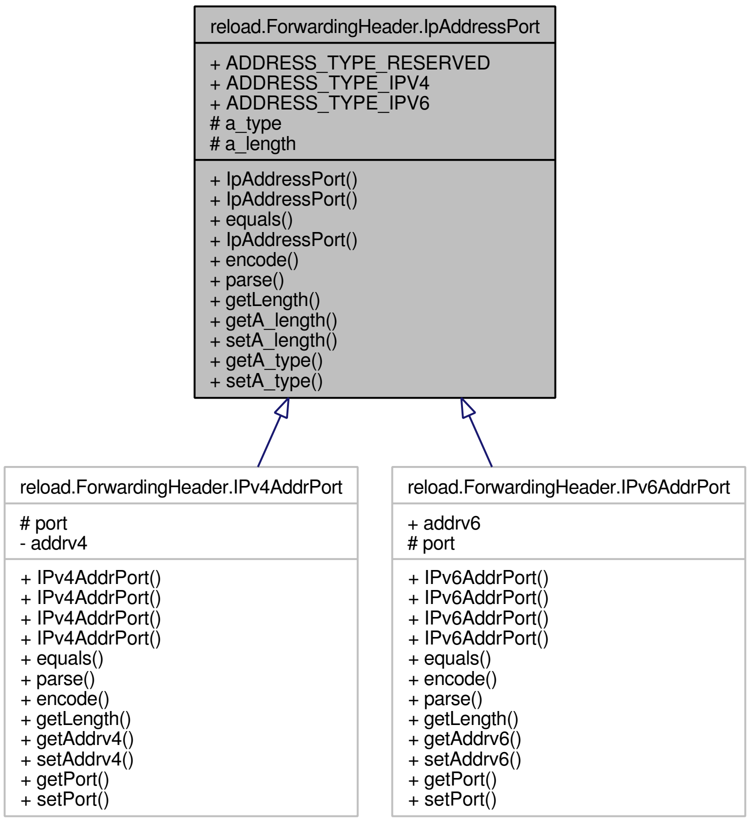


Figure 3.5: Class diagram of the IP Address Structure

3.3 Message Content

Each message has a generic header which is used to forward the message between peers and to its final destination. This header is the only information that an intermediate peer (i.e., one that is not the target of a message) needs to examine. The structure of the message content has the following components:

```
struct {
        MessageCode    message_code;
        opaque         message_body
        <0..2^24-1>;
} MessageContents;
```

The message_code indicates the message that is being sent. The codes that represent the type of message that has been implemented are the following:

NAME	VALUE
Probe Request and Answer	1 and 2
Attach Request and Answer	3 and 4
Store Request and Answer	7 and 8
Fetch Request and Answer	9 and 10
Find Request and Answer	13 and 14
Join Request and Answer	15 and 16
Leave Request and Answer	17 and 18
Update Request and Answer	19 and 20
Route Query Request and Answer	21 and 22
Ping Request and Answer	23 and 24
Stat Request and Answer	25 and 26
Attach Request and Answer	27 and 28
Error	Hexademical value 0xffff
Unused values	0, 5, 6, 11 and 12

The message_body field contains the message body itself, represented as a variable-length string of bytes. In the implementation the payload has the encoded value of each different message depending on the code of the message.

As it was previously shown, there are many different possible messages that a peer can use to communicate within the overlay. Some of them are similar and some others are rather different. Among the many possible, we are going to focus just on a few representative examples: the Ping Answer, the Fetch Request, and the Update Request.

The **Ping Answer** is the simplest message after the Ping Request, basically an empty message:

```
struct {
         uint64 response_id;
         uint64 time;
} PingAns;
```

It only has the response_id and the time. The first is randomly generated and used to distinguish Ping responses in cases where the Ping request is multicast. The later is the time when the ping response was created (In absolute time, which is the milliseconds since midnight Jan 1, 1970).

The **Fetch Request** retrieves one or more data elements stored at a given Resource-ID, in pseudocode it is as follows:

```
enum {reserved(0), single_value(1), array(2),
          dictionary(3), (255)} DataModel;
struct {
         uint32 first;
         uint32 last;
} ArrayRange;

struct {
         KindId(uint32) kind;
         DataModel      model;
         uint64         generation_counter;
         uint16         length;
         select(model) {
                case single_value:
                        /*Empty*/
                case array:
                        ArrayRange      indices<0..2^16-1>;
                case dictionary:
                        DictionaryKey   keys<0..2^16-1>;
        } model_specifier;
} StoredDataSpecifier;

struct {
        ResourceId              resource;
        StoredDataSpecifier     specifiers<0..2^16-1>;
} FetchReq;
```

The resource is the resource ID of the resources being fetched from and the specifiers are a sequence of StoredDataSpecifier (see Subsection 3.3.1) values. Each StoreDataSpecifier specifies some of the data values to retrieve. Each specifier will be different depending on the model that it is in use: it can be a single value, an array type, or a dictionary type. The model_specifier is a reference to the data value being requested within the data model specified for the kind. For instance, in the case in which the specifier is an ArrayRange, we will get a series of indices containing two integer values each, these values specifying respectively the beginning and the end of the range of the array. In case the data is of data model dictionary then the specifier contains a list of the dictionary keys being requested. If no keys are specified, then this is a wildcard fetch and all keyvalue pairs are returned. If the data is of data model single value, the specifier is empty.

The generation-counter is the last generation-counter received by the peer, it is used to indicate the requester's expected state of the storing peer. The length value indicates the length of the rest of the structure.

The **Update Request** is used along with the Update Answer (see Subsection 3.3.1) in the process of stabilizing the predecessors and the successors. The RELOAD Base specification [30] and the Self-tuning DHT draft for RELOAD [36] have been used on the implementation, the pseudocode is as follows:

```
enum {reserved (0),notify(1), succ_stab(2),
          pred_stab(3), full(4), (255)} ChordUpdateType;

struct {
        ChordUpdateType type;
        NodeId  sender_id;

        select(type) {
                case notify:
                        uint32  uptime;
                case pred_stab:
                    /* Empty */;
                case succ_stab:
                    /* Empty */;
                case full:
                        uint32  uptime;
                        NodeId  predecessors<0..2^16-1>;
                        NodeId  successors<0..2^16-1>;
                        NodeId  fingers<0..2^16-1>;
          };
} UpdateReq;
```

The type field indicates which type of ChordUpdateType was sent. It can be either 'notify', in which case the sender wishes to notify the recipient of the sender's existence; 'succ_stab'; or a 'pred_stab' depending if the Update request is related to the successor or predecessor stabilization routine. Finally, it can even contain the entire routing table of the sender in the 'full' case. The rest of the fields are: the sender_id, that contains the sender's Peer-ID; and the actual information depending on the ChordUpdateType. If the type of the Update request is pred_stab or succ_stab, the request does not have more information. If it is a notify, it will contain the sender's current uptime in seconds. If it is a full request it will contain all the sender's predecessor and successor list as well as the uptime and the finger table.

3.3.1 UML of the Message Content

The second major part of a RELOAD message is the contents part. The message contents contain two fields that have been already explained: the message code and the message body. The message body array can be filled with several different messages (see Figure 3.6). Since there is no need to represent all the possible messages here, two of them will be used: the Attach Request and the Update Answer.

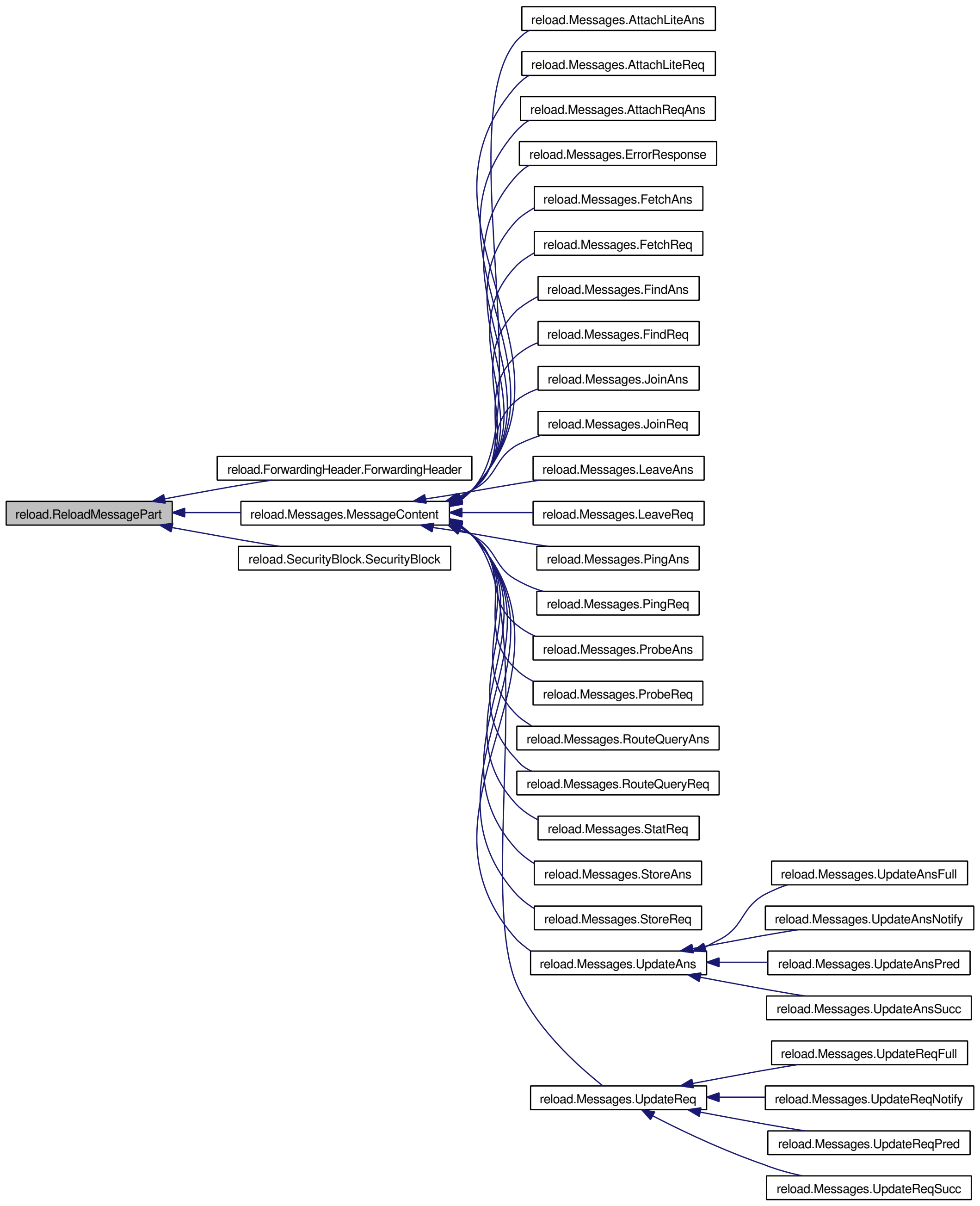

reload.ReloadMessagePart
reload.ForwardingHeader.ForwardingHeader
reload.Messages.MessageContent
reload.SecurityBlock.SecurityBlock
reload.Messages.AttachLiteAns
reload.Messages.AttachLiteReq
reload.Messages.AttachReqAns
reload.Messages.ErrorResponse
reload.Messages.FetchAns
reload.Messages.FetchReq
reload.Messages.FindAns
reload.Messages.FindReq
reload.Messages.JoinAns
reload.Messages.JoinReq
reload.Messages.LeaveAns
reload.Messages.LeaveReq
reload.Messages.PingAns
reload.Messages.PingReq
reload.Messages.ProbeAns
reload.Messages.ProbeReq
reload.Messages.RouteQueryAns
reload.Messages.RouteQueryReq
reload.Messages.StatReq
reload.Messages.StoreAns
reload.Messages.StoreReq
reload.Messages.UpdateAns
reload.Messages.UpdateReq
reload.Messages.UpdateAnsFull
reload.Messages.UpdateAnsNotify
reload.Messages.UpdateAnsPred
reload.Messages.UpdateAnsSucc
reload.Messages.UpdateReqFull
reload.Messages.UpdateReqNotify
reload.Messages.UpdateReqPred
reload.Messages.UpdateReqSucc

Figure 3.6: Reload Message Structure, with the different types of messages, the Forwarding Header and the Security Block

The Attach Request is used to establish a direct TCP or UDP connection to another node for the purposes of sending RELOAD messages or application layer protocol messages. Its structure is the same as that of the Attach Answer and the fields of the class (see Figure

3.7) are: the ufrag or username fragment, which is used in ICE; the password for ICE; the application, which is a 16-bit port number; the role, which is interpreted as a simple byte; and the list of candidates, which is a Vector of different IceCandidate values. Also like all the messages, it includes the encode(), equal() and the random generator of the message.

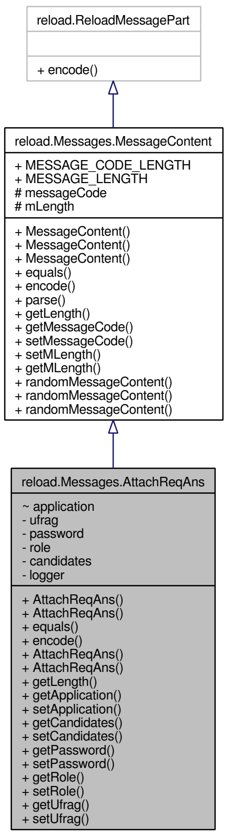


Figure 3.7: Class diagram of the Attach Request Message

The Update Answer of this implementation is based on the specification of the Self-Tuning DHT for RELOAD draft (see [36]) and not on the main RELOAD specification (see [30]). The Update implements the predecessor stabilization and successor stabilization procedures of a RELOAD message. The Update Answer class (see Figure 3.8) differs depending on the type of Update Answer: if it is 'full', the answer is empty; if is 'notify', the answer will contain the sender's current uptime in seconds; if it is 'pred_stab', the answer will carry the predecessor list of the responding peer; if it is 'succ_stab', the answer will include the predecessor and successor lists of the responding peer.

Another interesting class is the Stored Data Specifier (see Figure 3.9), which is used in the Store messages to store a particular information on a node. The information will vary for different subclasses depending on the type of information: "single_value", "array" or "dictionary". For the latter, there is a specific class that will handle its information.

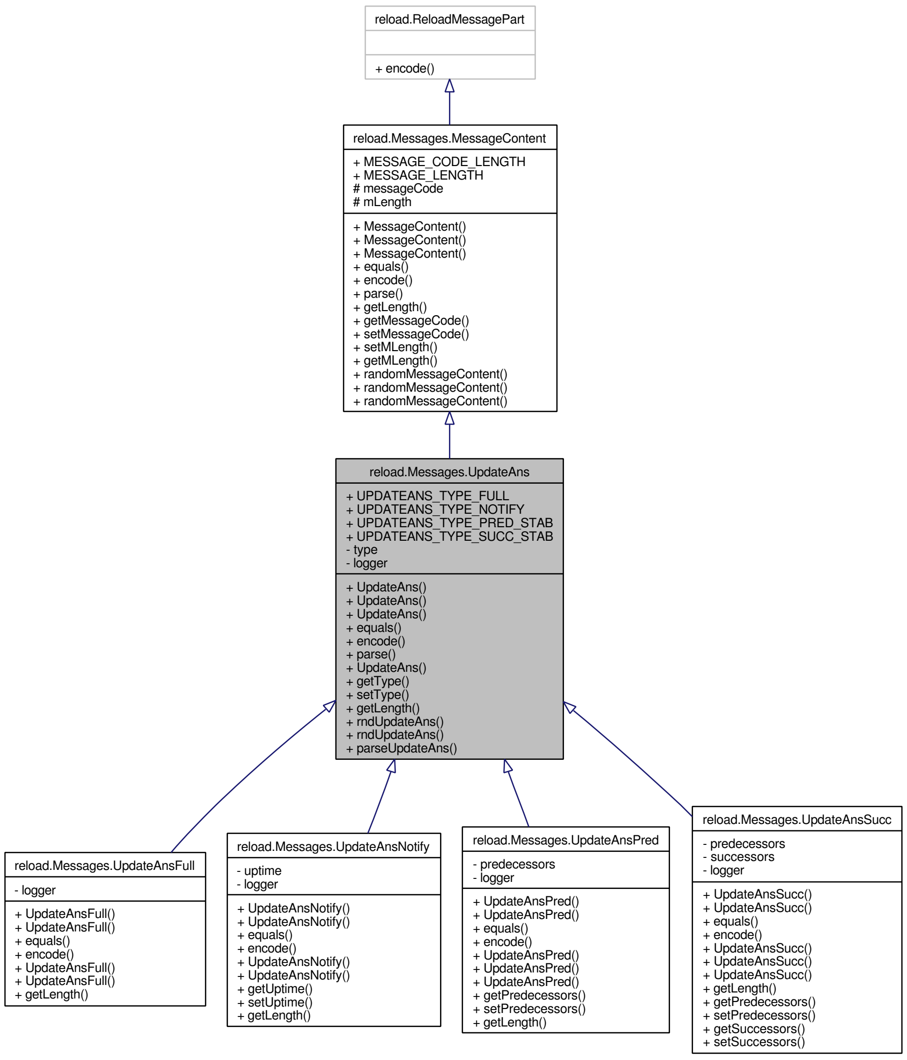


Figure 3.8: Class diagram of the Update Answer Message

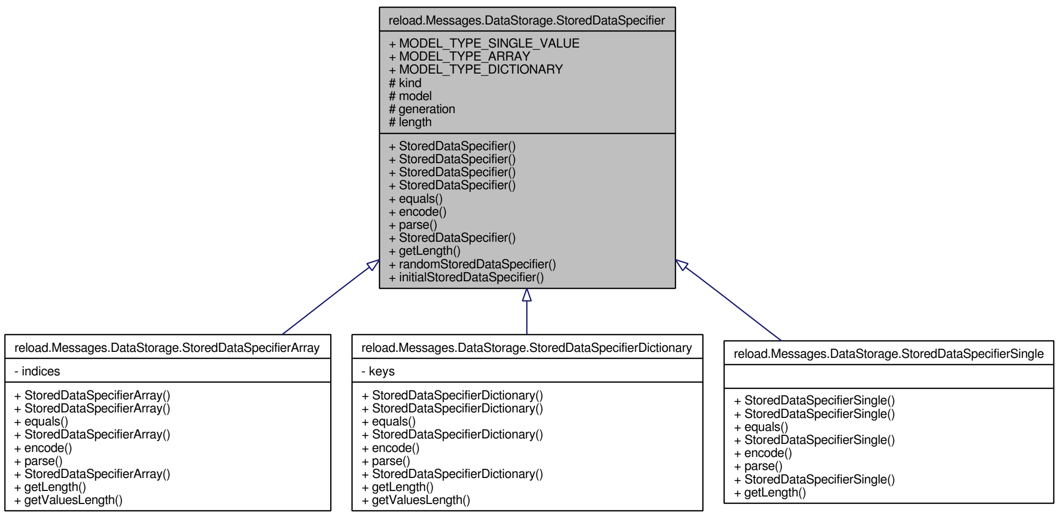

reload.Messages.DataStorage.StoredDataSpecifier
+ MODEL_TYPE_SINGLE_VALUE
+ MODEL_TYPE_ARRAY
+ MODEL_TYPE_DICTIONARY
# kind
# model
# generation
# length
+ StoredDataSpecifier()
+ StoredDataSpecifier()
+ StoredDataSpecifier()
+ StoredDataSpecifier()
+ equals()
+ encode()
+ parse()
+ StoredDataSpecifier()
+ getLength()
+ randomStoredDataSpecifier()
+ initialStoredDataSpecifier()
reload.Messages.DataStorage.StoredDataSpecifierArray
- indices
+ StoredDataSpecifierArray()
+ StoredDataSpecifierArray()
+ equals()
+ StoredDataSpecifierArray()
+ encode()
+ parse()
+ StoredDataSpecifierArray()
+ getLength()
+ getValuesLength()
reload.Messages.DataStorage.StoredDataSpecifierDictionary
- keys
+ StoredDataSpecifierDictionary()
+ StoredDataSpecifierDictionary()
+ equals()
+ StoredDataSpecifierDictionary()
+ encode()
+ parse()
+ StoredDataSpecifierDictionary()
+ getLength()
+ getValuesLength()
reload.Messages.DataStorage.StoredDataSpecifierSingle
+ StoredDataSpecifierSingle()
+ StoredDataSpecifierSingle()
+ equals()
+ StoredDataSpecifierSingle()
+ encode()
+ parse()
+ StoredDataSpecifierSingle()
+ getLength()

Figure 3.9: Class diagram of the Stored Data Structure

3.4 Security Block

A security block contains certificates and a digital signature over the message. For this implementation of RELOAD we are using self-signed certificates, the creator of the certificate is the one signing it on its legitimacy. The problem of self-signed certificates is that they are not revocable, which may allow an attacker to masquerade an identity after a private key has been compromised. Since all stored data is signed by the owner of the data, the storing peer can verify that the storer is authorized to perform a store at that Resource-ID and also allows any consumer of the data to verify the provenance and integrity of the data when it retrieves it. The specification of the main class is as follows:

```
enum {x509(0),(255)} certificate_type;

struct {
        certificate_type        type;
        opaque                  certificate<0..2^16-1>;
} GenericCertificate;

struct {
        GenericCertificate      certificates<0..2^16-1>;
        Signature               signature;
} SecurityBlock;
```

The certificates (see Subsection 3.3.1) are stored in the certificates bucket, which contains all the certificates that are needed to verify every signature in both the message and the internal message objects.

Each certificate is represented by a GenericCertificate structure, which has a type indicating the certificate used and the certificate itself. In RELOAD, the defined certificate type is X.509 [29]. The encoded certificate itself is generated using the BouncyCastle API [10] [2].

The signature in this implementation of RELOAD has also been calculated using the Bouncy Castle API [10]; it is computed over the message content and parts of forwarding header. It is structured as follows:

```
enum {reserved(0), cert_hash(1), (255)} SignerIdentityType;

select (identity_type) {
        case cert_hash;
        HashAlgorithm           hash_alg;
        opaque                  certificate_hash<0..2^8-1>;
} SignerIdentityValue;

struct {
        SignerIdentityType      identity_type;
        uint16                  length;
        SignerIdentityValue     identity[SignerIdentity.length];
} SignerIdentity;
```

[2]The Bouncy Castle Crypto package is a free Java implementation of cryptographic algorithms. The package is organized so that it contains a light-weight API suitable for use in any environment (including J2ME).

```
struct  {
        SignatureAndHashAlgorithm        algorithm;
        SignerIdentity                   identity;
        opaque                           signature_value<0..2^16-1>;
} Signature;
```

The signature contains the algorithm in use. The algorithm definitions are found in the IANA TLS SignatureAlgorithm Registry [2] and in The Transport Layer Security (TLS) Protocol (RFC5246 [23]). The identity is the one of a particular node, and it is used to form the signature. The signature value is calculated over several fields of the Forwarding Header (overlay and transaction id), all the Message Contents and the SignerIdentity field in the Security Block, as specified in [30]. The engine of Bouncy Castle's Crypto API uses the private RSA key to calculate the signature. As it was previously shown in section 2.5.1, the RELOAD messages are normally sent using an encrypted transport (TLS or DTLS), therefore the communication is always encrypted. Within the RELOAD message, there is the possibility of encrypting also the Message Contents.

When a RELOAD message is received, the first step is to verify the signature and the certificate to ensure that the sending node belongs to the overlay. Then the message itself will be parsed, and decrypted in case it is encrypted.

3.4.1 UML of the Security Block

The Security Block is an implementation of the ReloadMessagePart, it contains the Certificates and the Signature. In figure 3.10, we can see its class diagram. The class is actually really simple since the complexity is in the Certificate generator and in the Signature Generator; indeed it could be said that the Security Block is a wrapping class that holds the other two.

The GenericCertificate (see 3.11) contains the type of the certificates and an OpaqueShort object (byte array of a maximum of 65535 bytes of length). It contains the usual encode(), parse(), random generator, equals(), etc. methods but it has some particularities. In the parsing method, it also verifies the certificate using the BouncyCastle engine (the parsing of the Security Block is described in 3.5).

In the previous section there was an explanation of the components of the Signature and in the following there will be an explanation of how it is calculated (see 3.5); in this section we will concentrate on the classes of two of its components: the SignatureAndHashAlgorithm and the CryptoEngine.

The SignatureAndHashAlgorithm is a simple two-bytes structure specified in RFC 5246 [23]. The first byte is the hash algorithm used, in this case SHA-1. The second byte is the algorithm used to create the Signature, which is the RSA algorithm. According to the very same specification in RFC 5246 [23], the values corresponding to the SHA-1 and the RSA are 2 and 1. Therefore a encoded instance of this class will always be 02 01 for any given RELOAD message. In future implementations the values may change but the same structure can be reused.

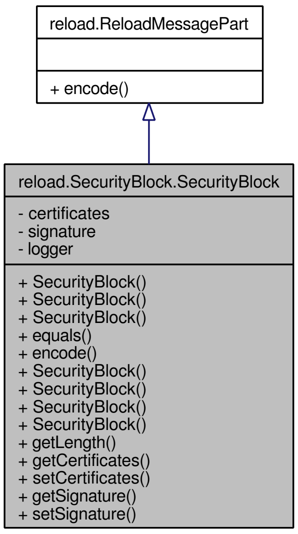


Figure 3.10: Class diagram of the Security Block

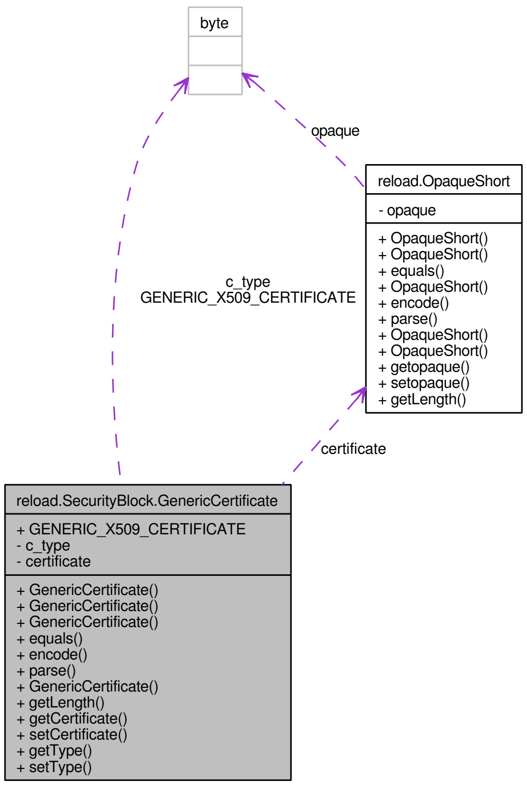


Figure 3.11: Class diagram of the Certificate

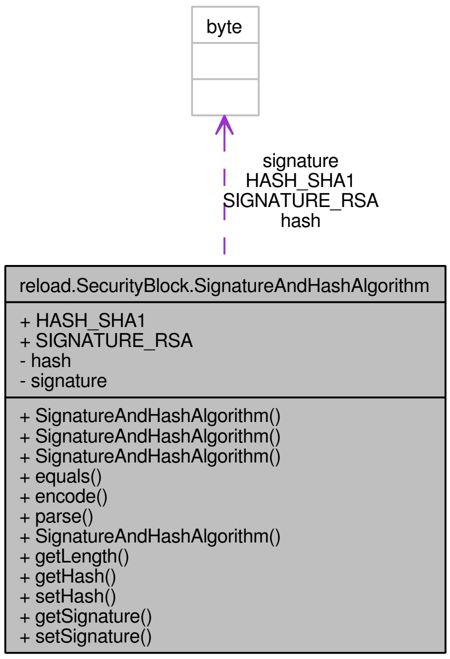


Figure 3.12: Class diagram of the Signature and hash algorithm

The Crypto Engine class serves as an "interface" class to communicate with the relevant classes of the Bouncy Castle API [10]. It contains all the methods related with encryption and decryption that need to be used in RELOAD (see 3.5): calculateSignature(), which calculates a hash over the data provided as an argument and then signs it using the specified private key; generateKeyPair(), which generates a public/private key pair; generateCertificate() which generates a self-signed X.509 certificate; and verifySignature() and verify SignedCertificate(). This class was not re-implemented for this thesis but an existing implementation was re-used.

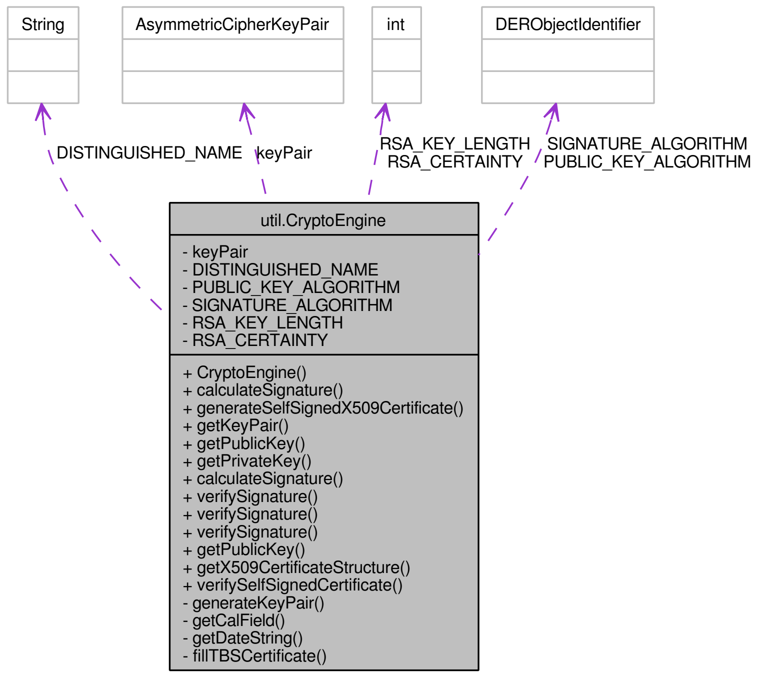


Figure 3.13: Class diagram of the Crypto Engine

3.5 General Implementation

Within this project and among the many classes, there are three main methods that have been programmed: To create a new random message, to encode the message, and to parse the message.

Generating a Message Creating a random message will generate any message among all the possible messages; this generator does not only fill the fields of a message but also the content included is realistic, as in a real RELOAD network. This has been done so that the measurements are accurate. The generator will create a new Forwarding Header and a new Message Content object. The Security Block will be created during the encoding of the message.

Encoding a Message The encoding of the message consists of creating a byte array that will contain the whole message in bytes. Therefore it has to encode all the classes and structures created in Java into the byte array, filling the required fields correctly. All low level operation methods (at byte level) have been implemented since the Java Micro Edition does not support most of them per default. The encoding is required in order to obtain the binary messages that will be sent to the peers. It is during the encoding that, due to performance considerations, the Security Block is created. Also at this point it is encoded into the byte array.

Parsing a Message In order to create a proper communication between peers in the RELOAD network, all messages have to be parsed correctly into a structure that can be interpreted by the receiving application, so that it can then take the correct actions regarding to the message. When parsing, the byte array is read and stored at byte level; this is possible since there are several fields on the message that indicate the length of the next part of the message. For instance, every time that an OpaqueByte structure which has a maximum length of 255 bytes is used, an extra byte is added before it is encoded to be sent on the wire. Therefore, the 5-byte value "value", would be encoded as: 05 76 61 6c 75 65. During parsing, as specified in RELOAD base [30] the 05 field is taken and interpreted as the length of the next field, so the next 5 bytes will be stored in the correspondent structure and the parsing will go on.

Encoding and parsing the Security Block On RELOAD a standard public/private key cryptography is used to verify and generate signatures. When parsing the message, also the Security Block is parsed, by using the following cryptographic procedure to verify the signature (see Figure 3.14).

First it is important to know whether the message is being sent with the certificate of the sender included, or if this is sent before. In case the certificate has been included in the message, it will be present in the certificate bucket of the Security Block, and when parsed, the receiver shall assume that the first certificate of the bucket is the one that corresponds to the sender [30]. If the certificate is sent beforehand so that the receiver can store it, the process of parsing will be exactly the same but the certificate used will be the one already stored in the computer (instead of the first of the certificate bucket that is in each message).

The process starts when the Security Block is generated. The elements that will be passed are the random generator, the values from which the certificate will be computed and the engine itself. The Security Block will call a method in the engine to calculate the signature. First the hash over the message will be calculated using SHA-1 and then the hash will be encrypted using the RSA algorithm and the private key of the sender. The certificate is also signed using the sender's private key (since it is self-signed). Then the certificate and the signature will be included in the message that will be sent.

When the receiver gets the message and arrives to the stage in which it has to parse the Security Block, it will first get the first certificate of the certificate bucket, or from the place where it has been stored if it was not sent with the message. In case the certificate is in the certificate bucket, the receiver will assume that the first certificate is the one corresponding to the sender. The receiver will verify the certificate and obtain the public key of the sender. Then it will parse the signature by first decrypting it. For that it needs the previous public key and the RSA algorithm.

Once the signature is parsed the original hash of the message will be obtained. Then another hash over the message will be calculated on the receiver side, using again the SHA-1 algorithm and the relevant fields of the message that were already parsed.

To conclude the receiver will compare both hashes and in case they are not equal it will imply a corruption of the data before arrival.

Usually the certificates are signed by a certification authority, but in this case self-signed certificates are used. By doing that we can ensure the authentication of the sender, since he is signing with his private key and it is physically stored in his machine.

3.5.1 UML of the Sockets

So far only the message generation and parsing has been presented, in this subsection we will comment the UML diagrams of the sockets used to communicate between peers in the network. The RELOAD implementation uses two different socket APIs: one for the Java Micro edition and the other for the Standard Edition. The sockets allow the RELOAD application to communicate with the transport layer and use the transport protocol to transmit the information on the wire. The protocol in our testing program will be the User Datagram Protocol (UDP), although RELOAD mandates the use of TLS or DTLS. Plain UDP was used since J2ME phones are not capable of performing as TLS servers and do not support DTLS, in adition to that UDP to simplifies the communication by not using the implicit hand-shaking dialogues present in other protocols

For the Java Standard Edition, on the server side, the class MyDatagramSockets (see Figure 3.15) will be used. In it a datagram socket will be created and binded to the specified port on the local machine. MyDatagramSockets class has the necessary methods to enable the server to send and receive incoming datagrams.

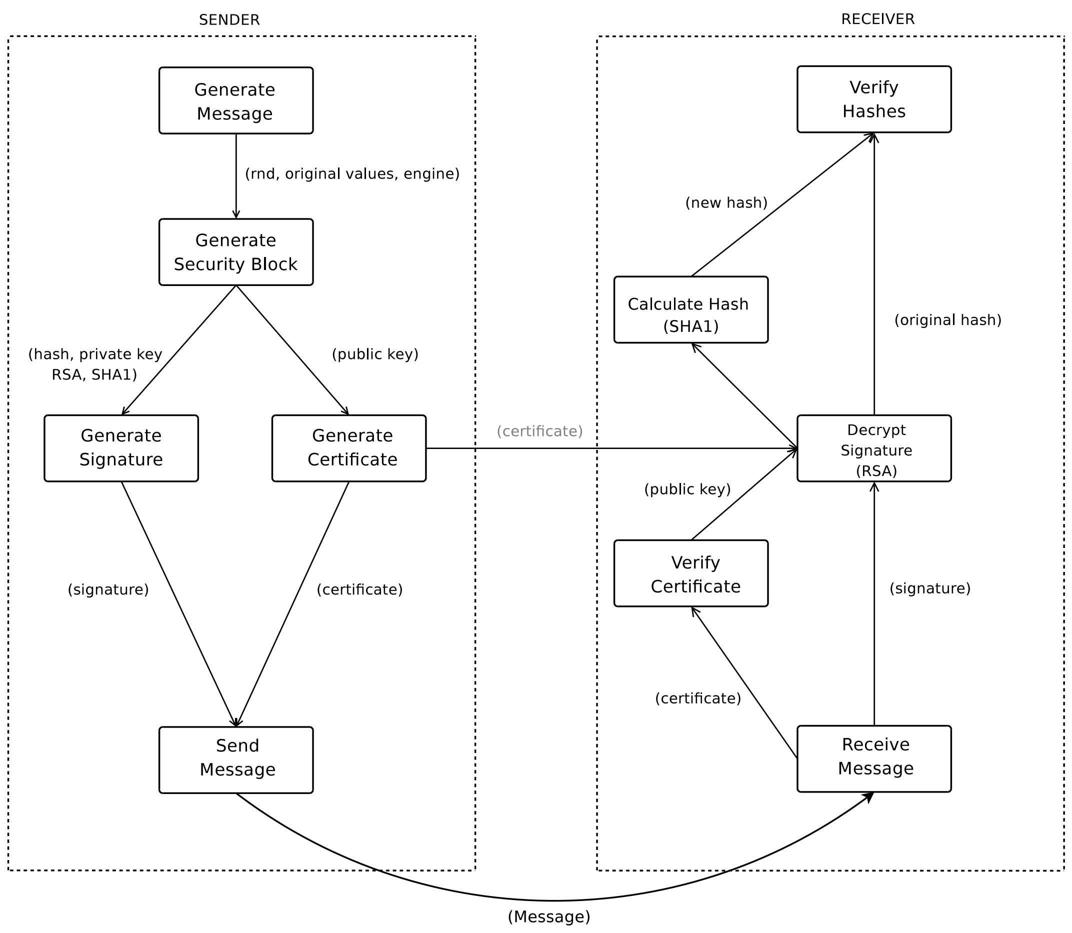


Figure 3.14: Process of generating, sending and parsing the Security Block

For the Java Micro Edition, there is another class: IO (see Figure 3.16), whose name comes from Input/Output. It is very similar to the Standard Edition's sockets since it enables sending and receiving datagrams, but it has the peculiarity of working in the restricted environment that J2ME introduces, and of being created to specifically work on UDP Datagram connections [37].

The two classes on which the main code is run are TestReloadSockets (see Figure 3.18) and MessageSender (see Figure 3.17). We are going to focus on the ones programmed for the mobile, although they have their homologue implementation on the server side. The TestReloadSockets is the main class of the whole implementation and it has several threads: one generates a different type of random message depending on a specific delay previously calculated for it, then it sends the message; another one listens for incoming messages or certificates, then parses them, verifies the signature and stores the relevant information. In order for the TestReloadSockets to send the information, it uses another class, the MessageSender.

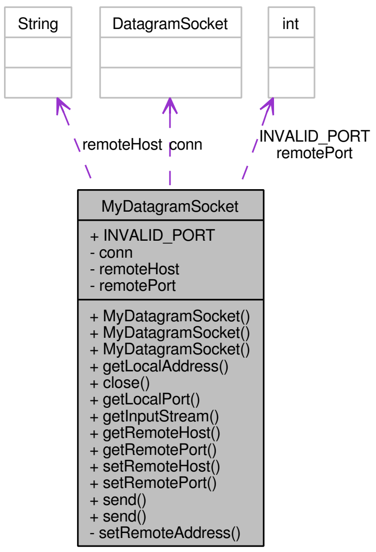


Figure 3.15: Class diagram of the Datagram Socket

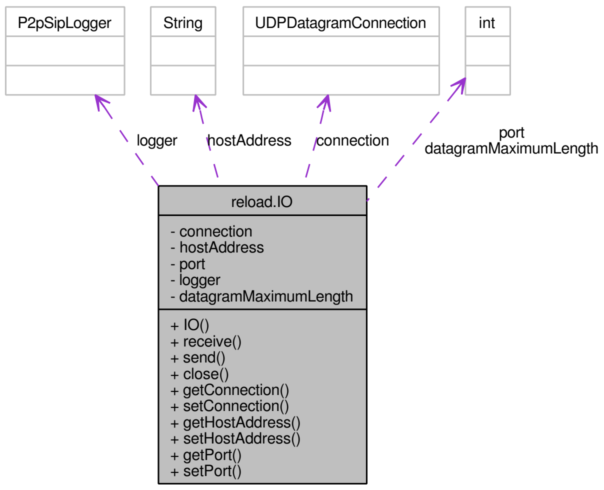


Figure 3.16: Class diagram of the IO

The MessageSender receives all the necessary information to generate and send a message (IO, IP, Port, Message and CryptoEngine fields) with a certain pre-established delay (delay field). The messages can be sent either periodically, for which we use Thread that contains a Periodic Timer; or they can be sent at an exponential rate to ensure randomness, for which we use the Exponential Timer thread. Both periodic and exponential messages are used to implement the traffic model used in the measurements.

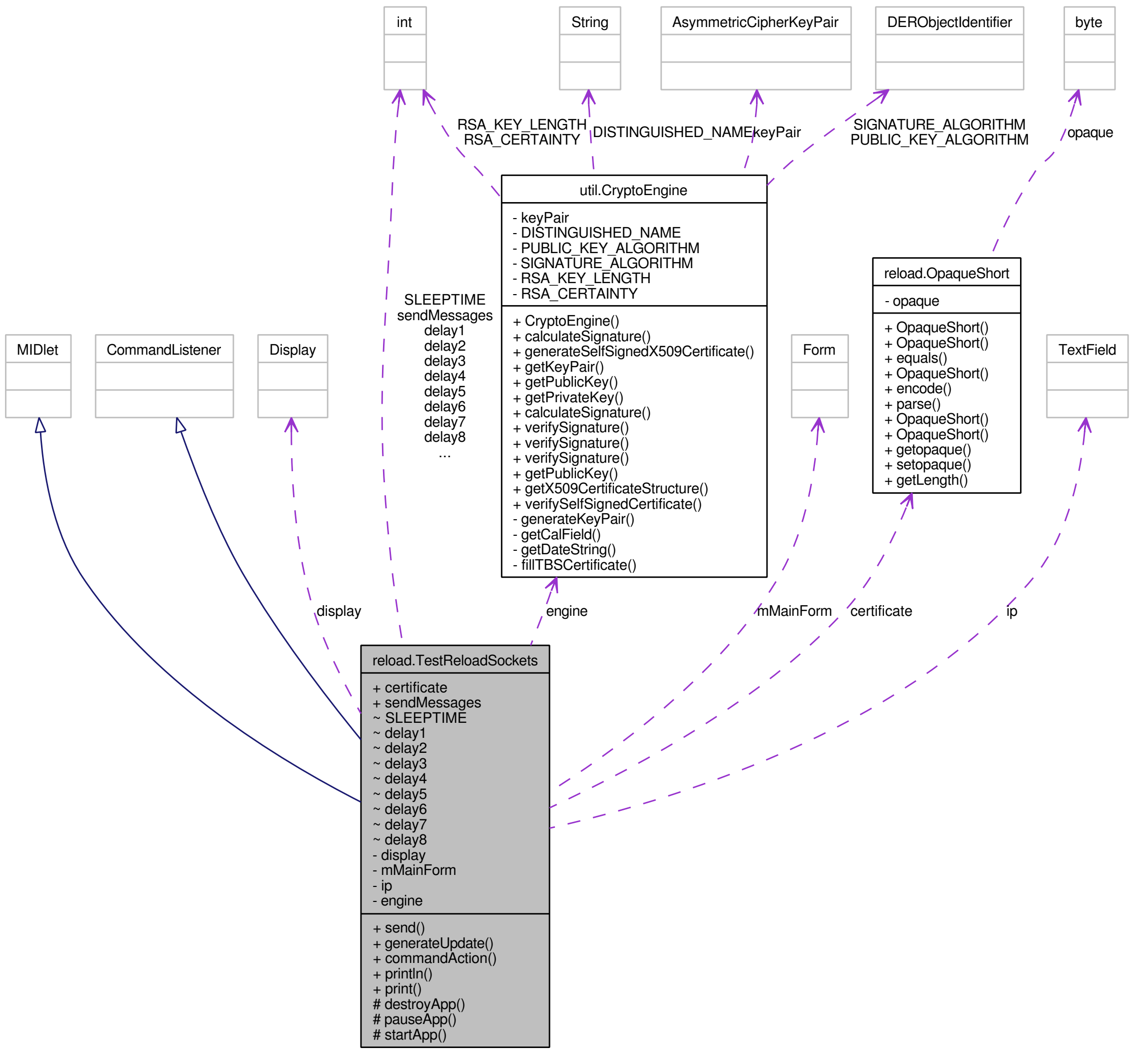


Figure 3.17: Class diagram of the TestReloadSockets

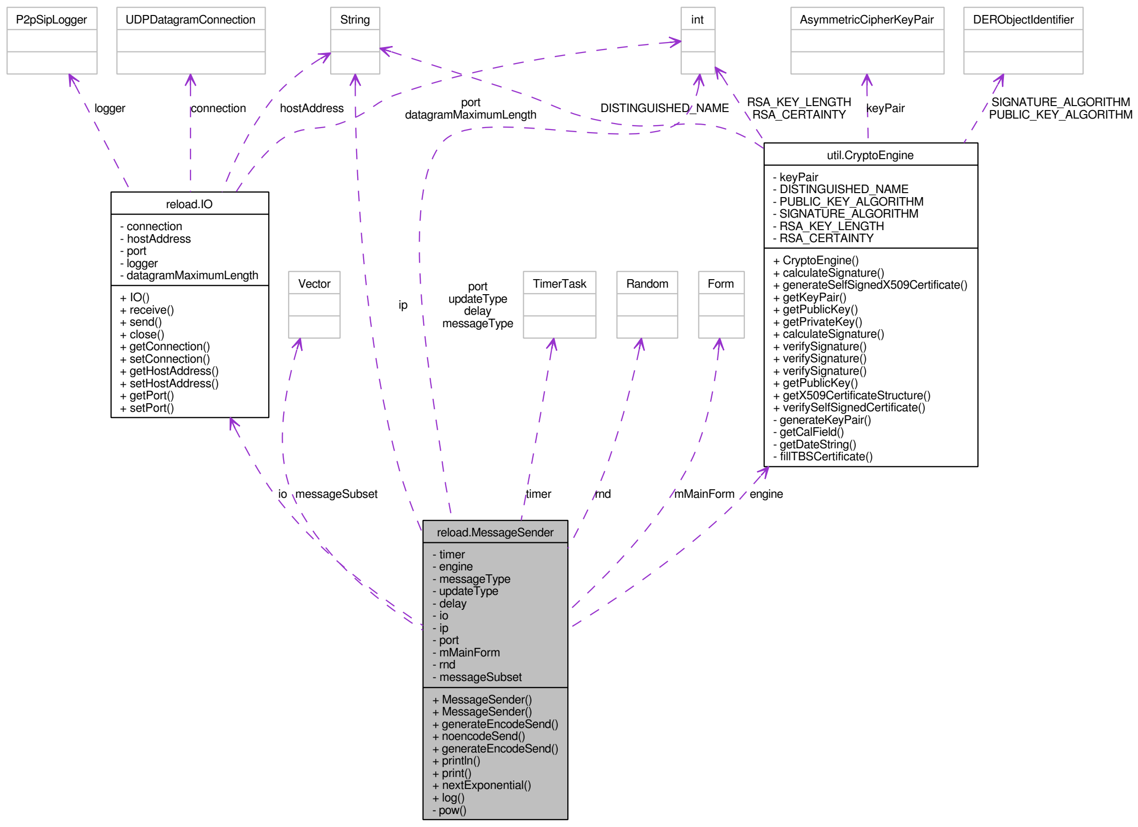


Figure 3.18: Class diagram of the Message Sender

Chapter 4

Measurements and Evaluation

The purpose of the experiment has been to measure the performance of a mobile device when receiving and sending a traffic similar to the one on a RELOAD network. In order to achieve that, several RELOAD messages were generated and sent according to a traffic model that attempts to imitate a real P2PSIP overlay.

4.1 Methodology

The application has been tested for one hour, which is a reasonable time in order to get relevant battery usage information. The peer remains connected during the whole measurement period.

In the existing literature the maximum size of a network has been found to be of few tenths of thousands of nodes, therefore a reasonably large size of 10000 nodes has been chosen for the network in the experiments. The chosen network size would be the same as in an average big international company using P2PSIP for internal VoIP traffic.

The peer interarrival and departure time was chosen to be of 3 seconds. That means that it has been assumed that one peer will join or leave the network every three seconds on average. This time has been chosen considering that, if the average session time in a 10000-peer overlay is 8 hours (the average working time) and the duration of the experiment was 3600 seconds, then the peer interrarrival time is $(3600 \cdot 8)/10000$, which is approximately 3 seconds.

For this experiment we assume that the DHT in use is Chord. The time between its maintenance operations on ring-like networks, according to the draft on P2P Dynamic Evolution [44] is:

> "Our main result is that a Chord network in the ring-like state remains in the ring-like state, as long as nodes send $\Omega(log^2(N))$ messages before N new nodes join or $N/2$ nodes fail."

Based on this result, a DHT maintenance interval of $(N/2) \cdot 3/(log^2(N)$ seconds has been

Table 4.1: Data used in experiment

Field	**Value**
Measurement duration [s]	3600
Measurement duration [s]	3600
Peer uptime [s]	3600
Network size [nodes]	10000
Peer interarrival time [s]	3
Chord maintenance interval [s]	84.96
Average hop count	6.64
Finger table size	13
Neighbor table size	17
Number of resource records	10000
Number of replica copies of each resource	3
Self-look-up hop count	7.64
Self-look-up interval [s]	84.96

used.

According to the paper [56], the average number of hops needed to route a look-up through the overlay has been measured to be $(1/2)log_2(N)$ [44]. This information is relevant in order to calculate how many messages sent by other nodes the mobile phone will forward during the measurement.

The neighbor table (i.e. successor and predecessor lists) is carried on Update messages, their size consist on: the successor list size, which is the same as the finger table size, $log_2(N)$ [44]; and the predecessor list size. The predecessor list size is calculated using successor replication strategy, with 3 replica copies of each record. Because of this 4 predecessors are needed (3 for the replication plus the original predecessor) [36].

The total number of resource records in the overlay is assumed to be N since each node has one contact record. In addition to the Chord maintenance operations specified in [56], a self-look-up routine is used to detect loops within the overlay, in the process a node sends a "false" look-up to find itself, the number of hops will be the same as the average hop count plus one more to forward to the successor, $(1/2)log_2(N) + 1$ (a node sends a self-look-up request first to its successor because it cannot route a message to itself).

The table 4.1 shows the calculated values of the previous information.

4.2 Test environment

The environment consist on two sides, a computer as server that simulates the RELOAD overlay and a mobile phone as client. They both send and receive RELOAD messages, each of them at a different rate. The different delays have been obtained according to the table 4.1.

The delays have been *the same for each side*, the server was simulating a RELOAD overlay in which the mobile phone is a peer. The server sends traffic to the phone according to the traffic model. The phone both generates messages by itself and forwards traffic from other peers in the overlay according to the traffic model. The delays have also been the same regardless of being request or response. The difference has been on the messages themselves, since each message of the same type has been randomly generated and with a random interval.

Some of the messages included certificates and some others not. The find, store and attach messages did include the certificate of the sender. This is because in those type of messages the receiver is unknown and no communication has been previously established with it. In the case of the update, join and leave messages there has been a previous connection with the peer, therefore the peer does already have the certificate of the sender stored and there is no need to send it again.

In the same way, some messages must always be signed and in some others only a subset has to be signed. The mobile phone signs only messages that it initiates and does not sign messages that it forwards. In the case of update and leave messages they have always to be signed, in the case of the rest of the messages only a percentage is signed. That percentage depends on the number of nodes (hops) that the message has to go through before arriving to the destination ($100/hops$): in the case of the find, attach and store messages, it will be 6.64 hops, therefore 15% of the messages will be signed; in the case of the self-lockup messages it will be 7.74 hops, therefore 13% of the messages will be signed; and in the case of the join messages just one message will be sent and signed.

In order to send and receive the messages at the same pace as they would have been sent and received in a real overlay, some messages have been sent in a periodic fashion and some have been sent following a random distribution. The periodicity depends on whether the message is used in the predecessor-successor stabilization of the overlay as well as the look-up messages. Since the rest of the messages are supposed to be sent in different periods on the overlay and by different peers, to ensure randomness they are sent following the exponential distribution.

The 4.2 table shows the information that has been used for each type of message.

As we can see the Full Update, Join and Leave messages are sent every 14 hours, 7 hours, and every 45 minutes following an exponential scheme. Considering that the duration time of the experiment is 1 hour their probability of being sent at all is scarce.

Table 4.2: Messages and delays

Outgoing messages	**Number of messages**	**Periodicity**	**Certificate**	**Delay [s]**
Update (successor)	42.38	periodic	no	84.96
Update (predecessor)	42.38	periodic	no	84.96
Update (full)	0.12	exponential	no	30000
Find (finger stabilization)	281.53	exponential	yes	12.79
Join	0.8	exponential	no	4515.45
Leave	0.24	exponential	no	15000
Attach	58.2	exponential	yes	61.86
Store	2.24	exponential	yes	1609.11
Self-look-up	323.89	exponential	yes	11.11

4.2.1 Procedure

The process commences by executing the server application, which has a thread in listening mode waiting on port 50501 for datagrams to be received through the UDP channel. When the client is started, it calculates the key pair to be used on the signature and certificate and sends the certificate to the server. This is only done once, then it remains in a listening state on port 50500.

When the server receives the datagram it first checks whether the message is a RELOAD message or a Certificate, if the later happens it stores the certificate and sends back its own certificate to the IP address from which the first certificate was received. Then it remains in listening mode. When the client receives the certificate from the server it stores it and commences the transmission of the messages. The moment the server receives the first RELOAD message, it commences the transmission (See figure 4.1).

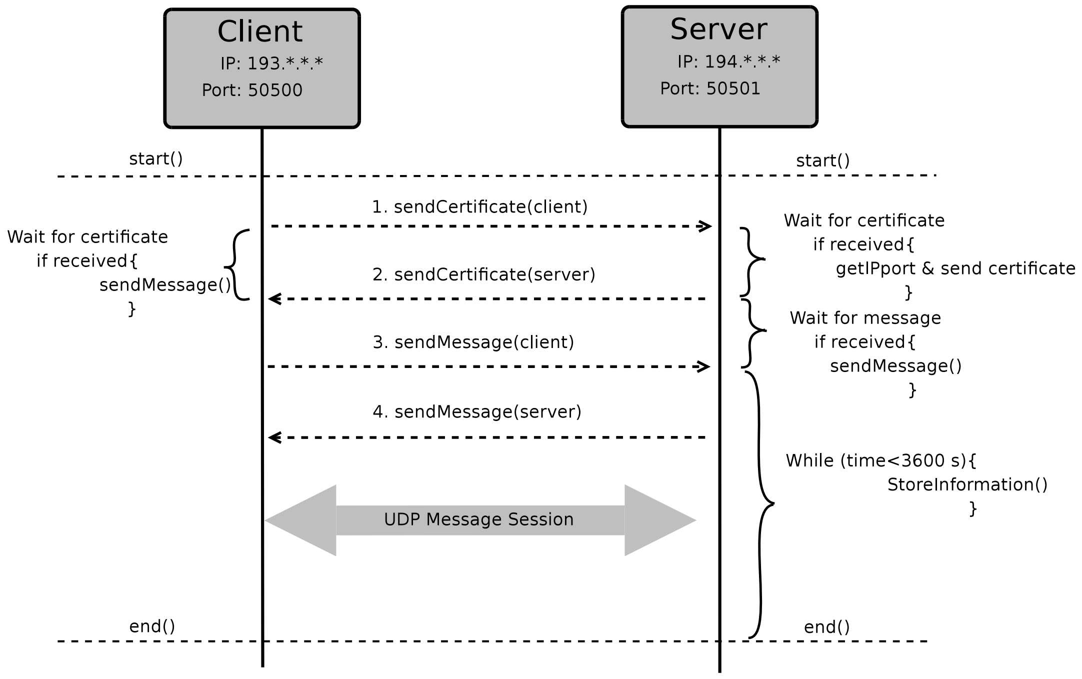


Figure 4.1: Model of the experiment

While both the server and the client are sending and receiving messages, the server stores relevant information into a vector structure. When the experiment is over the data is introduced into a spreadsheet to be analyzed later. On the client side the measured values include memory, CPU efficiency, and battery life.

Since storing that information on the client itself would have caused to alter the experiment by overloading the phone, the measurements of the client where taken by connecting it to a laptop, which was storing that information.

4.3 Evaluation

The table 4.3 shows a resume of the sizes and number of messages [1].

The size of the different parts of the message vary greatly depending on the message, on average the Forwarding Header and the Message Contents represent a smaller part of the whole message when compared to the Security Block (see figure 4.2).

The actual percentage is of 7.3% for the Forwarding Header, 8.6% for the Message Contents and 84.1% for the Security Block, therefore the Security Block is on average five times bigger as the rest of the message.

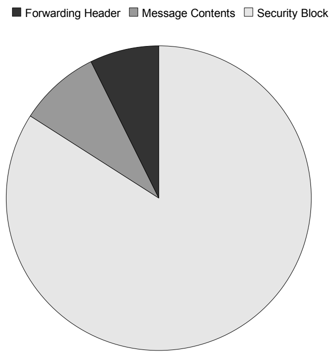


Figure 4.2: Size of the different parts of a RELOAD message

Some messages were not sent at all (join request, join answer, leave request and leave answer) this is because it is a matter of probability, and in a real overlay they are seldom sent. For instance the join and leave messages are just sent once for each peer.

Table 4.3: Result number of messages and sizes

Message	NOM	FH	MC	SB	SOM
Attach Req	60	42	39	685	767
Attach Ans	50	43	40	710	794
Store Req	3	57	29	813	899
Store Ans	1	57	145	830	1032
Find Req	535	56	8	828	893
Find Ans	555	56	7	789	853
Update Request	82	55	22	271	348
Update Answer	70	58	248	271	578

In the case of the **Attach and Find messages**, the non-signed messages were generated and stored at the beginning of the experiment. Then, if a non-signed message was required,

[1]The initials FH, MC, SB, SOM stand for the respective Average sizes of the Forwarding Header, Message Contents, Security Block and Size of Message. NOM stands for Number of Messages.

which was in the 85% of the cases, the same previously generated message was sent. Therefore the median corresponds to the value of this message. That is also the reason why their graphics show certain homogeneity.

The average size of the Attach Request and Attach Answer message (see figure 4.3) was 767.63 and 794.92 bytes. In the case of the Find Request and Find Answer (see figure 4.4) it was 893.25 and 853.02 bytes.

The variance in the the message sizes is caused because of the randomness in which they are generated. Despite some fields have always the same length, although they may contain different values, some other may have an array of values. In the case of the Attach Request for instance, the IceLiteCandidate array varies in number of elements from one message to another.

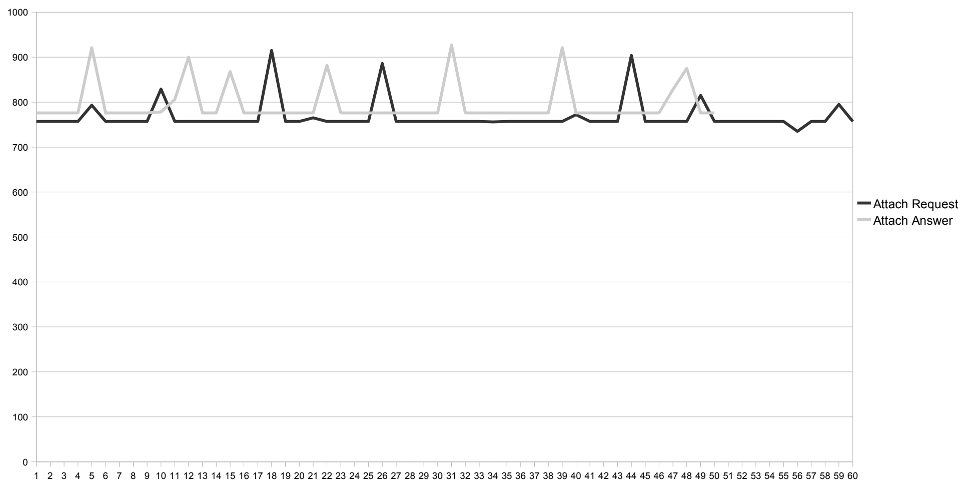


Figure 4.3: Graphic of the sizes of the Attach Request and Attach Answer messages

In the case of the **Update Request and Update Answer messages**, it can be noticed in figure 4.5 that the size is sensibly smaller to the previous two types. This is because the Update Messages did not include the Certificates, being their size smaller than in the other cases.

The average size of the Update Request was 348.87 bytes and its median 345 bytes. In the case of the Update Answer, the average size was 578.07 bytes and the median 577 bytes. This result can be understood by looking again at the table 4.3 in there we can see that the average size of the Message Content of the Update Request message is 22 bytes while the Update Answer has an average size of 248 bytes, being thus the message bigger. The reasons why the message content is smaller in the Update Request are two: first the probability of sending a Full Update Request message, which is the biggest one, is very rare; second, the size of the rest of the Update Request types (Notify, Predecessor and Successor) is relatively small when compared with its counterparts in the Update Answer.

While the messages were being received by the server, also the delay upon receival of

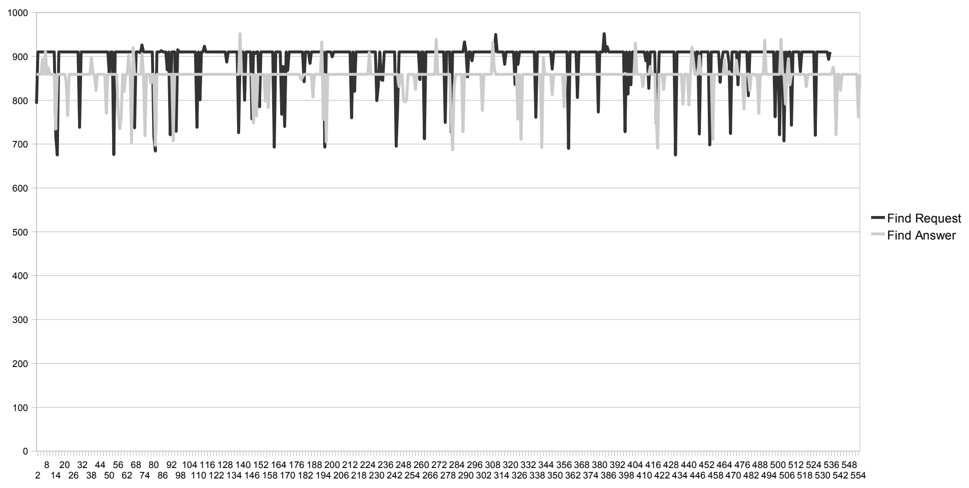


Figure 4.4: Graphic of the sizes of the Find Request and Find Answer messages

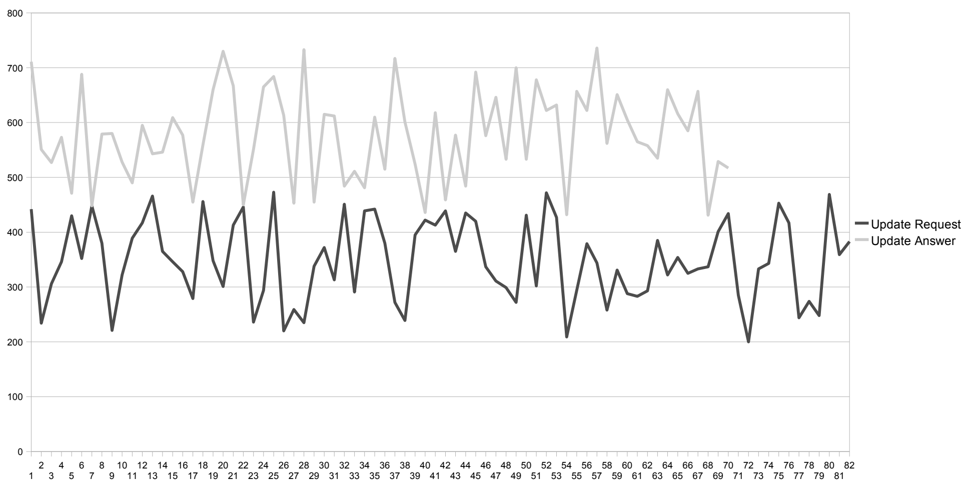


Figure 4.5: Graphic of the sizes of the Update Request and Update Answer messages

the Update Predecessors and Successors was measured (see figure 4.6). The reason for measuring only the delay of the Update messages is that they always are a on-hop message, therefore they can be simulated in the current scheme.

The measuring was done by including a timestamp in each of the Update messages that was sent to the phone, then the phone copied that timestamp in the Update messages it generated and sent it back to the server. Once in the server the timestamp was read again and substracted to the time upon receival, calculating thus the delay of the message.

The results show that the Average delay for the Update Predecessor message is of 2608.61

milliseconds, and the median is 1964 milliseconds. This of course is the roundtrip delay, sending and receiving a message.

For the Update Successor, the average delay for the is of 3731.88 milliseconds, and the median is 3462 milliseconds.

The messages experimented a great delay due to the 3G HSDPA connection that was used for the phone, being the downlink bandwidth 2Mbit/s and uplink bandwidth 1Mbit/s

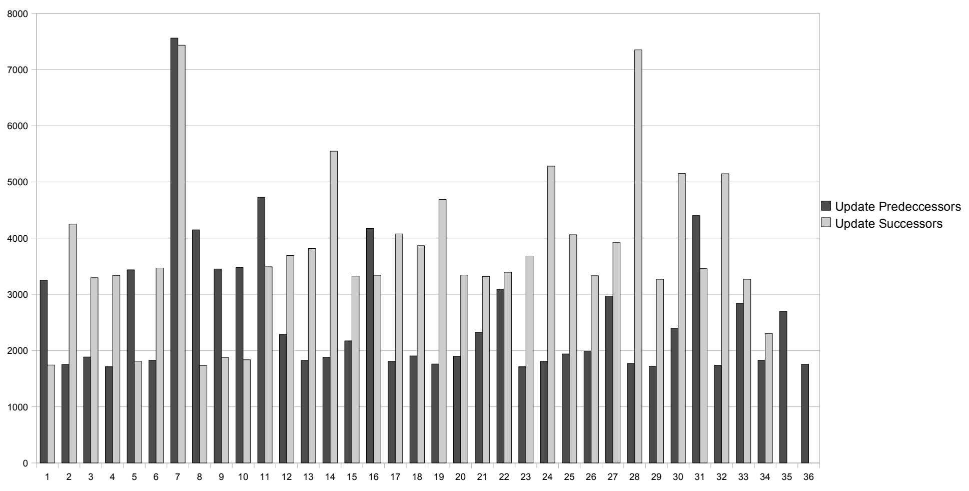


Figure 4.6: Graphic of the delays of the Update Predecessor and Update Successor messages

The time it took the mobile phone to **calculate the public and private keys**, a process done at the beginning of the experiment was 57 seconds. This is because the amount of processing power necessary to calculate them is high for a mobile phone. In other testings this time has gone down up to 37 seconds and up to 179, since the process of generating the keys has a fair amount randomness in it.

The number of received messages were 1356 and the total amount of bytes received were 1109922 bytes which is around 1 megabyte of information in total.

Regarding **memory consumption**, in 4.7 we can see how the used Java memory remains stable throughout the experiment, ranging from values of 547 to 767 kilobytes and being the average consumption of 660.91 kilobytes. In the case of the native memory of the phone the consumption is stable too (see figure 4.8), the minimum used memory in a certain instant is of 3.65 megabytes and the maximum of 4.19 megabytes, being the median of 4.04 megabytes and the initial memory of the phone of 32 megabytes. It is noticeable also that the load is stable during the measurements, varying only in a maximum of 555 kilobytes, depending on the Java Memory consumption, the initial consumption of 4 megabytes is caused by the high load that the CryptoEngine generates.

The battery consumption was the 18% of the phone's battery, which is quite high. There

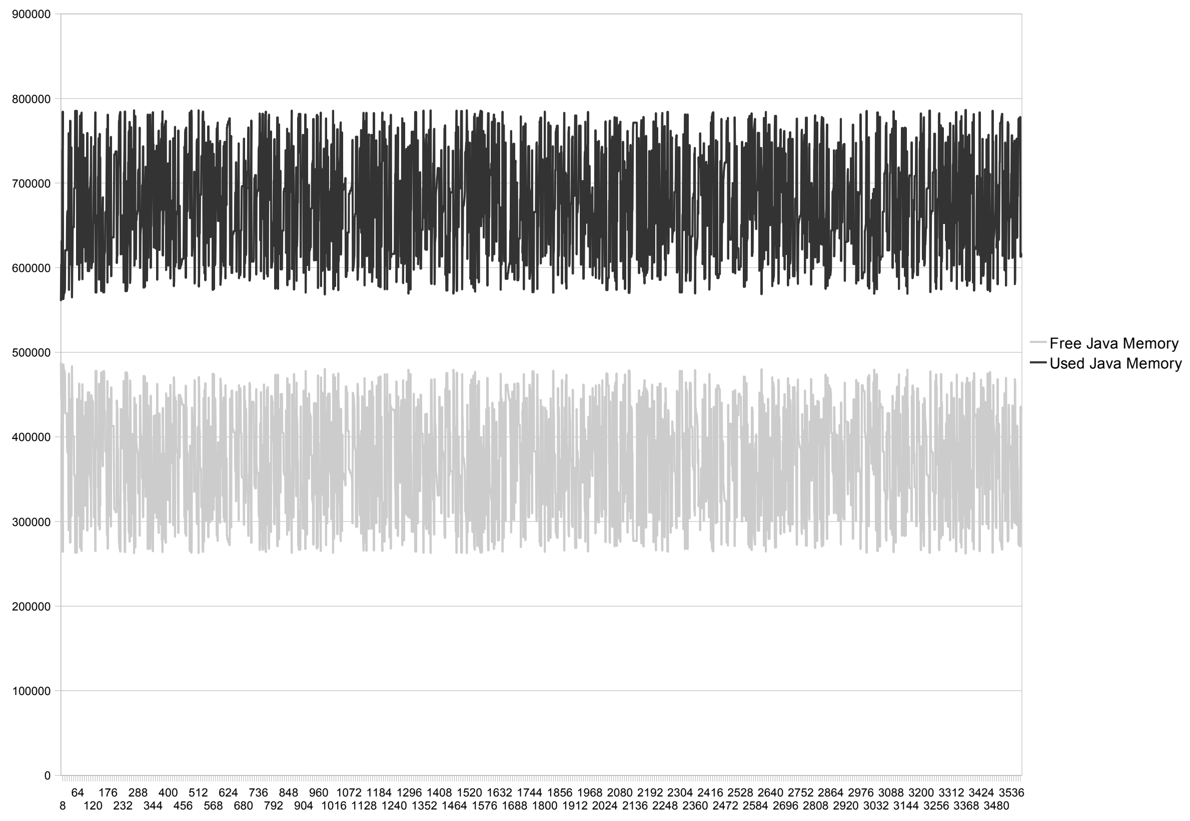


Figure 4.7: Graphic of the Java memory consumption

are two factors to be taken into account when analyzing this data, the first one is that having a network connection open for long periods of time consumes a lot of battery on a mobile phone and the other is that the cryptographic features introduce a high load on the CPU.

The CPU load was varying from idle until up to a 99% of load when processing the messages. In 4.9 the swift changes can be noticed. It is also noticeable that despite the sudden changes and the lack of homogeneity in the graphic, the CPU was mostly idle during the experiment, being the median of the values of 6% of load.

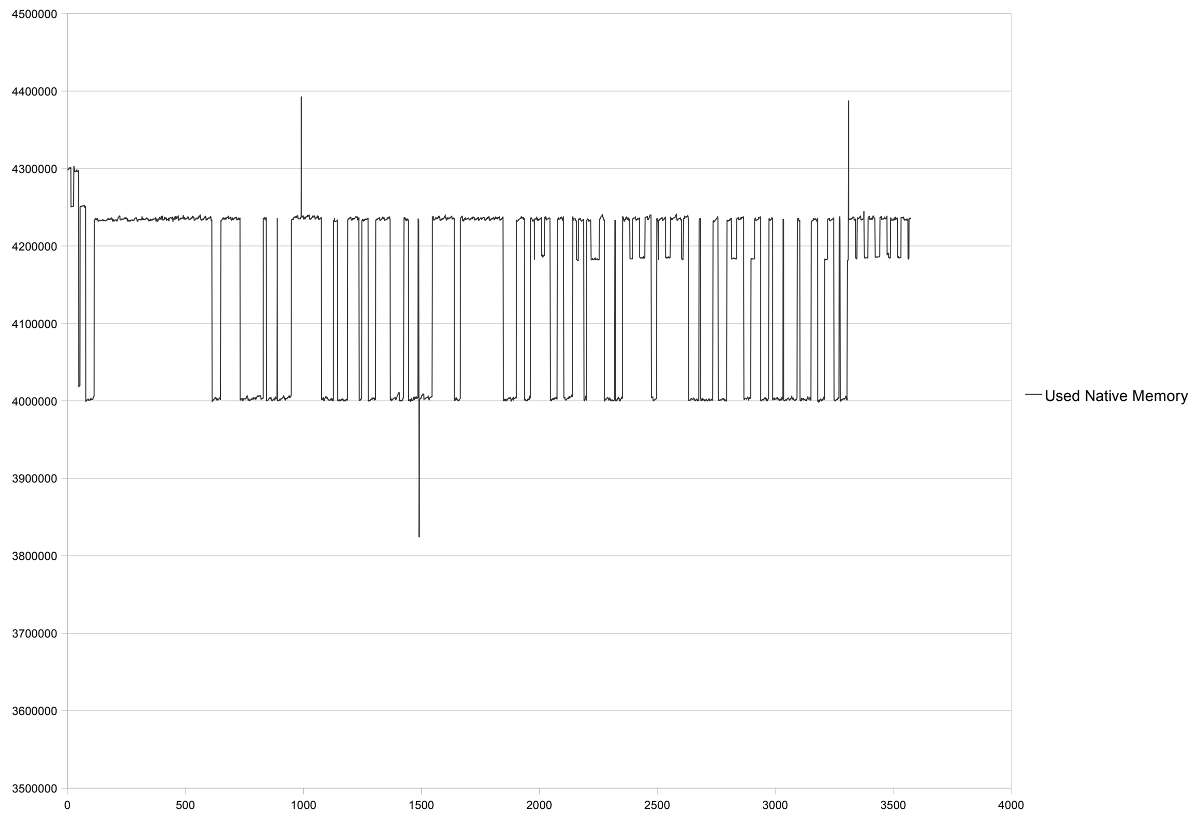


Figure 4.8: Graphic of the phone's native memory consumption

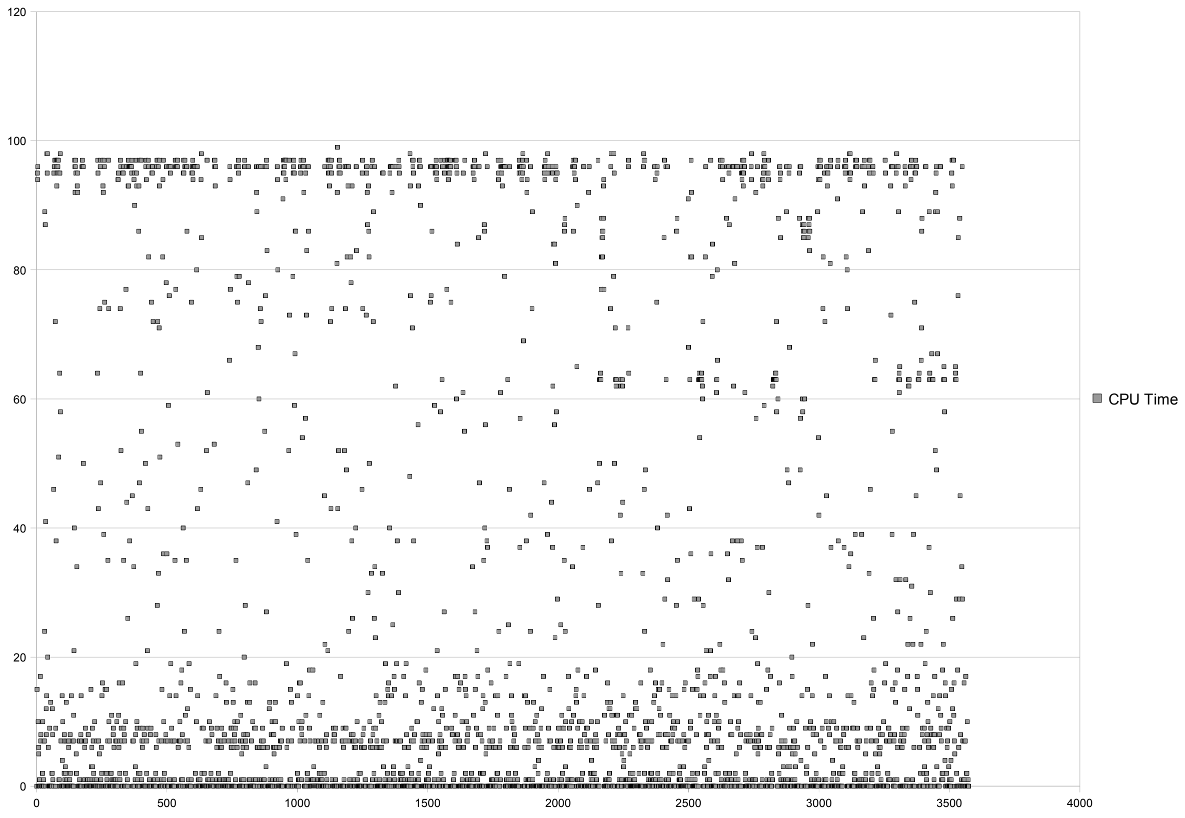


Figure 4.9: Graphic of the CPU Time

Chapter 5

Conclusions and Future work

The thesis consists of the implementation of the RELOAD message structure and on the testing of whether a mobile phone can participate in a RELOAD network. The testing consisted of a server simulating a RELOAD overlay and sending similar traffic to the one in an overlay to the mobile phone. The phone behaved as if in a real overlay.

This project constitutes the first measurement of this kind, and it positively proves the possibility of using mobile devices on a P2PSIP communication.

In the future it would be interesting to evaluate the performance of RELOAD on other types of mobile devices with better performance (i.e. PDAs or Smartphones for instance). It would also be interesting to continue with the implementation of RELOAD, realize the experiments on real networks and eventually implement a fully functional RELOAD application.

There is some room for improvement on battery consumption on the phone (18%), despite being as expected perhaps even higher since the screen was on all the time. The CPU load was relatively high when processing the messages but there where some idle periods in between, with no apparent overload of the phone.

The native memory used on the mobile device was relatively low, with a lot of consumption caused by the cryptographic procedures followed to generate the keys used to sign the messages. It would be therefore be interesting to optimize this consumption if possible.

There is also room for improvement on the average delay of the update messages: while some of this delay may be due to the cryptographic algorithms used and to the connection on the mobile phone's side, it is also necessary to re-evaluate the whole testing process in order to improve the performance.

Bibliography

[1] *Guinness World Record*, http://folding.stanford.edu/English/Awards, Referenced on 27.01.2009.

[2] *IANA TLS SignatureAlgorithm*, http://www.iana.org/assignments/tls-parameters/, Referenced on 12.05.2009.

[3] *Internet Architecture Board*, http://www.iab.org/, Referenced on 19.01.2009.

[4] *Internet research Task Force*, http://www.irtf.org/, Referenced on 19.01.2009.

[5] *Internet Society*, http://www.isoc.org/, Referenced on 19.01.2009.

[6] *P2PSIP IETF Working Group*, http://www.ietf.org/html.charters/p2psip-charter.html, Referenced on 22.02.2009.

[7] *SIP RFCs & Drafts*, http://www.tech-invite.com/Ti-sip-IDs-SIP.html, Referenced on 22.01.2009.

[8] *Skype Fast Facts Q4 2008*, http://ebayinkblog.com/wp-content/uploads/2009/01/skype-fast-facts-q4-08.pdf, Referenced on 27.01.2009.

[9] *Study: BitTorrent sees big growth, LimeWire still first P2P app*, http://arstechnica.com/old/content/2008/04/study-bittorren-sees-big-growth-limewire-still-1-p2p-app.ars, Referenced on 27.01.2009.

[10] *The Legion of the Bouncy Castle*, http://www.bouncycastle.org/, Referenced on 11.05.2009.

[11] *The O'Reilly P2P directory lists companies, projects and initiatives related to peer-to-peer technologies*, http://www.openp2p.com/pub/q/p2p_category/, Referenced on 23.01.2009.

[12] H. Alvestrand, *A Mission Statement for the IETF*, RFC 3935 (Best Current Practice), October 2004.

[13] Walter S. Baer, *Military Applications of P2P*, http://conferences.oreillynet.com/presentations/p2pweb2001/Moini_A_2021.ppt, November 2005, Referenced on 27.01.2009.

[14] S. Baset and H. Schulzrinne, *Peer-to-peer protocol (p2pp)*, March 2007, Work in progress.

[15] Scott Bradner, *IETF Structure and Internet Standards Process*, http://www.ietf.org/proceedings/07mar/slides/newcomer-0.pdf, 2007.

[16] D. Bryan, P.Matthews, E. Shim, D. Willis, and S. Dawkins, *Concepts and terminology for peer to peer sip*, July 2008, Work in progress.

[17] David A. Bryan, Bruce B. Lowekamp, and Cullen Jennings, *Sosimple: A serverless, standards-based, p2p sip communication system*, Advanced Architectures and Algorithms for Internet Delivery and Applications, 2005. AAA-IDEA 2005. First International Workshop on (2005).

[18] G. Camarillo, *Peer-to-peer (p2p) architectures*, November 2008, Work in progress.

[19] Gonzalo Camarillo, *Sip demystified*, McGraw-Hill Professional, 2001.

[20] Gonzalo Camarillo and Miguel-Angel Garcia-Martin, *The 3g ip multimedia subsystem (ims): Merging the internet and the cellular worlds, second edition*, John Wiley & Sons, 2006.

[21] E. Cooper, A. Johnston, and P. Matthews, *A distributed transport function in p2psip using hip for multi-hop overlay routing*, June 2007, Work in progress.

[22] Kshemkalyani Ajay D. and Singhal Mukesh, *Distributed computing: Principles, algorithms, and systems*, Cambridge University Press, New York, NY, USA, 2008.

[23] T. Dierks and E. Rescorla, *The Transport Layer Security (TLS) Protocol Version 1.2*, RFC 5246 (Proposed Standard), August 2008.

[24] Jeffrey L. Eppinger, *TCP Connections for P2P Apps: A Software Approach to Solving the NAT Problem*, http://reports-archive.adm.cs.cmu.edu/anon/anon/home/ftp/isri2005/CMU-ISRI-05-104.pdf, January 2005, Referenced on 26.01.2009.

[25] Ian Foster, Carl Kesselman, Jeffrey M. Nick, and Steven Tuecke, *The Physiology of the Grid:An Open Grid Services Architecture for Distributed Systems Integration*, http://www.globus.org/alliance/publications/papers/ogsa.pdf, 2002, Work in progress.

[26] Anne Geraci, *Ieee standard computer dictionary: Compilation of ieee standard computer glossaries*, Institute of Electrical and Electronics Engineers Inc, 1991.

[27] F. Hinchion, P. Mulgaonkar, D. Wilkins, and S. Galuga, *Peer to peer information management for tactical situation awareness systems*, Military Communications Conference, 2003. MILCOM 2003. IEEE **1** (2003), 179–185 Vol.1.

[28] P. Hoffman and S. Harris, *The Tao of IETF - A Novice's Guide to the Internet Engineering Task Force*, RFC 4677 (Informational), September 2006.

[29] R. Housley, W. Polk, W. Ford, and D. Solo, *Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile*, RFC 3280 (Proposed Standard), April 2002, Obsoleted by RFC 5280, updated by RFCs 4325, 4630.

[30] C. Jennings, B. Lowekamp, E. Rescorla, S. Baset, and H. Schulzrinne, *Resource location and discovery (reload) base protocol*, `http://www.ietf.org/internet-drafts/draft-ietf-p2psip-base-02.txt`, March 2009, Work in progress.

[31] C. Jennings, J. Rosenberg, and E. Rescorla, *Address settlement by peer to peer*, July 2007, Work in progress.

[32] XingFeng. Jiang, HeWen. Zheng, C. Macian, and V. Pascual, *Service extensible p2p peer protocol*, February 2008, Work in progress.

[33] Barry M. Leiner, Vinton G. Cerf, David D. Clark, Robert E. Kahn, Leonard Kleinroch, Daniel C. Lynch, Jon Postel, Larry G. Roberts, and Stephen Wolff, *A Brief History of the Internet*, `http://www.isoc.org/internet/history/brief.shtml`, Referenced on 20.12.2008.

[34] Eng Keong Lua, Jon Crowcroft, Marcelo Pias, Ravi Sharma, and Steven Lim, *A Survey and Comparison of Peer-to-Peer Overlay Network Schemes*, March 2004.

[35] Keong Lua, J. Crowcroft, M. Pias, R. Sharma, and S. Lim, *A survey and comparison of peer-to-peer overlay network schemes*, Communications Surveys & Tutorials, IEEE (2005), 72–93.

[36] J. Maenpaa, G. Camarillo, and J. Hautakorpi, *A self-tuning distributed hash table (dht) for resource location and discovery (reload)*, February 2009, Work in progress.

[37] Qusay H. Mahmoud, *J2ME Low-Level Network Programming with MIDP 2*, `http://developers.sun.com/mobility/midp/articles/midp2network/`, Referenced on 21.05.2009.

[38] E. Marocco and E. Ivov, *Extensible peer protocol (xpp)*, June 2007, Work in progress.

[39] Isaias Martinez-Yelmo, Ruben Cuevas, Carmen Guerrero, and Andreas Mauthe, *Routing performance in a hierarchical dht-based overlay network*, Parallel, Distributed, and Network-Based Processing, Euromicro Conference on **0** (2008), 508–515.

[40] Petar Maymounkov and David Mazieres, *Kademlia: A peer-to-peer information system based on the xor metric*, (2002), 53 – 65.

[41] R. Merryman, *ARPAWOCKY*, RFC 527, May 1973.

[42] Michael Miller, *Discovering p2p*, SYBEX Inc., Alameda, CA, USA, 2001.

[43] John Naughton, *Internet: The age of permanent net revolution*, http://www.guardian.co.uk/business/2006/mar/05/newmedia.broadcasting, Referenced on 21.01.2009.

[44] David Liben Nowell, Hari Balakrishnan, and David Karger, *Observations on the dynamic evolution of peer-to-peer networks*, (2001), Referenced on 22.05.2009.

[45] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenke, *A scalable content-addressable network*, (2001).

[46] J.K. Reynolds and J. Postel, *Request For Comments reference guide*, RFC 1000, August 1987.

[47] J. Rosenberg, *Interactive connectivity establishment (ice): A protocol for network address translator (nat) traversal for offer answer protocols*, October 2007, Work in progress.

[48] J. Rosenberg, R. Mahy, and P. Matthews, *Traversal using relays around nat (turn): Relay extensions to session traversal utilities for nat (stun)*, November 2008, Work in progress.

[49] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler, *SIP: Session Initiation Protocol*, RFC 3261 (Proposed Standard), June 2002, Updated by RFCs 3265, 3853, 4320, 4916, 5393.

[50] Antony Rowstron and Peter Druschel, *Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems*, (2001), 329–350.

[51] Detlef Schoder and Kai Fischbach, *P2P: Anwendungsbereiche und Herausforderungen*, 2002.

[52] Detlef Schoder, Kai Fischbach, and Christian Schmitt, *Core Concepts in Peer-to-Peer Networking*, 2005.

[53] Kundan Singh and Henning Schulzrinne, *Peer to peer telephony using SIP*, http://www.cs.columbia.edu/IRT/p2p-sip, April 2005, Referenced on 30.01.2009.

[54] Henry Sinnreich and Alan B. Johnston, *Internet communications using sip: delivering voip and multimedia services with session initiation protocol*, John Wiley & Sons, Inc., New York, NY, USA, 2001.

[55] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan, *Chord: A scalable peer-to-peer lookup service for internet applications (v1)*, (2001).

[56] Ion Stoica, Robert Morris, David Liben-Nowell, David Karger, M. Frans Kaashoek, Frank Dabek, and Hari Balakrishnan, *Chord: A scalable peer-to-peer lookup service for internet applications (v2)*, IEEE Transactions on Networking **11** (2003).

[57] Ian J. Taylor, *From p2p to web services and grids: Peers in a client/server world*, Springer, 2004.

[58] Georg Wittenburg, *Recent Trends in Peer-to-Peer Research*, http://page.mi.fu-berlin.de/wittenbu/studies/p2p_trends.pdf, July 2004, Referenced on 27.01.2009.

Appendix A

Tools and Materials

In this chapter several requirements necessary for the development the project are explained: knowledge base, software and hardware.

A.1 Knowledge Base

The knowledge required and adquired for this thesis has been mainly on the fields of protocol implementation on Java, and theoretical knowledge of several P2P systems and SIP.

General It has been necessary to learn about P2P networks, current trends and future planned developments. Among the extensive studied bibliography that is referred at the end of this thesis, "REsource LOcation And Discovery (RELOAD) Base" draft [30] and "Self-tuning Distributed Hash Table (DHT)" draft [36], which have been the model on which the project has been implemented, are worth noting.

Programming The code has been developed mainly in Java Micro Edition (J2ME), with special emphasis on low-level classes and on sockets since they are both used in the implementation. Java Standard Edition has been use for the server side.

The Cryptography has been done using the Bouncy Castle API [10]. It implements all the underlying cryptographic algorithms used in RELOAD, in order to self-sign the messages and generate the certificates. It has been chosen because it is suitable for memory constrained devices (J2ME).

Some basic scripts have also been used for data management, the language in that case has been Perl.

A.2 Software

The software used refers not only to the different applications but also Operative Systems,

The implementation has been carried out in the Eclipse IDE, version 3.4.2. Also some of its plugins have been used: Mobile Tools for Java SDK version 0.9.1, the Subclipse Client version 1.6.2 and the Sun Java Wireless Toolkit (WTK) version 2.5.2.

The main operative system has been Ubuntu versions 8.04 and 8.10. The server for the mobile phone is also an Ubuntu 8.04 with ssh installed. The desktop interface has been GNOME version 2.22.3 and the Kernel Linux version is the 2.6.24.

Windows XP has also been used in order to obtain the relevant data from the software used to measure CPU load and memory usage, which is the Sony-Ericsson SDK.

This thesis has been written on LaTeX, specifically using the Texmaker version 1.6 application.

The images are all original and have been done on DIA version 0.96.1, being later exported to .eps format to include them in this thesis. In the case of the UML diagrams they have been done using Doxygen version 1.5.9.

The OpenOffice Spreadsheet application has been used for the data management, and also to create the graphics from the data.

A.3 Hardware

The phones in which the application has been tested have been a Sony Ericsson C905 and a SonyEricsson K660, being both of them suitable for the application.

The programming and testing on the server side has been done on a laptop with 2 GB of memory, and an Intel Pentium M processor having 42 Gb of hard drive space.

Appendix B

Budget Estimation

In this part it is analysed the several steps in which the project is divided along with the theoretical cost that they would have. The costs have been calculated by determining the number of hours dedicated to the project, and depending on each of its tasks.

B.1 Tasks in the implementation of the project

Documentation The documentation part has consisted on the study of several books, and RFCs on P2P, SIP and RELOAD as well as learning Java and the use of the different tools required to start the implementation.

Implementation & Testing The implementation has had two parts, the implementation of the RELOAD message structure and the implementation of the testing environment. The testing has been doneat the same time as the implementation, at the end of every significative step of the development of the final application.

Measurements and Analysis The measurements have been analyzed after the final experiments, the ammount of data generated required the use of different applications to transform them into readable data and visually comprehensible in order to analyze them.

Writing Thesis Writing the thesis represents unifying the several parts it is composed of, as well as finding the references, learning to use the tools (LaTeX, Dia, Doxygen, ...) and revising the thesis.

B.2 Project costs

This section shows the total costs of the project, it does not include the cost of the hardware nor the software. Although the software costs is close to 0 € since the development has been carried out almost entirely on free software (i.e. Ubuntu, Dia, Eclipse, OpenOffice, LaTeX, . . .).

The table B.1 indicates the number of hours per task and the cost of the particular task in Euros. It has been taken the cost of a programmer as 20 €per hour. The total time taken to do the thesis is of 960 hours (i.e. 6 months).

The cost is calculated with the following formula:

$$Cost = hours \cdot C_h \tag{B.1}$$

Task	**Hours**	**Cost [€]**
Documentation	300	6000
Implementation	340	6800
Measurements	140	2800
Writing	180	3600
Total	960	**19200 €**

Table B.1: Tasks and costs