uc3m | Universidad Carlos III de Madrid

BACHELOR IN TELEMATICS ENGINEERING

(2019-2020)

Bachelor Thesis

Implementation of Protocol for Assignment of Local and Multicast Addresses

---

Miguel González Saiz

Tutor:

Antonio de la Oliva

Leganés September 1, 2020

Abstract

Not many years ago, due to the growth of the Internet infrastructure and the appearance of an immense number of new devices on the market, each one of them requiring a global and unique IPv4 address to be connected to the Net, we came up to the problem of IPv4 address shortage. This one had to be solved with a complex address translation mechanism (NAT) and the development of a new standard with greater addressing capacity (IPv6).

Lately, this problem has reappeared. This time referring to the scarcity of media access control addresses, also called MAC addresses. However, in this case, there are solutions that do not need the use of unique global addresses, but local addresses, defined in the standard IEEE 802c-2017, instead.

Recently, this association has been developing a new protocol that consists on the assignation of unicast and multicast MAC addresses in level 2 networks. This protocol is called PALMA (*Protocol for Assignment of Local and Multicast Addresses*) and allows the temporary assignment of MAC addresses to multiple devices, even if they lack of a global one.

The purpose of this work is the elaboration of the first PALMA implementation, based on the last published draft, as well as the analysis of its performance.

The result obtained is a compact and efficient "software" package, which has been correctly functional validated and could serve as a reference for future developments.

Keywords

IEEE, PALMA, Implementation, Protocol, Performance, MAC, Local Address, Network, Link-Layer, SLAP, MAAP, DHCPv6

Resumen

No hace muchos años, con el crecimiento de la infraestructura de Internet y la aparición de una inmensa cantidad de dispositivos nuevos en el mercado, que usarían cada uno de ellos una dirección IPv4 global y única en el mundo para conectarse a la red, se nos presentó el problema de la escasez de direcciones IPv4. Este tuvo que ser solventado con un complejo mecanismo de traducción de direcciones (NAT) y con el desarrollo de un nuevo estándar con mayor capacidad de direccionamiento (IPv6).

Últimamente, el problema vuelve a aparecer. Esta vez referido a la escasez de las direcciones de control de acceso al medio, también denominadas direcciones MAC, aunque, en este caso, existen soluciones que pasan por la no utilización de direcciones globales únicas, sino locales, tal como fueron definidas en el standard IEEE 802c-2017.

Recientemente, esta asociación ha estado desarrollando un nuevo protocolo que consiste en la asignación de direcciones MAC, tanto de unidifusión como de multidifusión, enfocado a redes de nivel 2. Se trata de un protocolo llamado PALMA (*Protocol for Assignment of Local and Multicast Addresses*) que permite la asignación temporal de direcciones MAC a múltiples dispositivos, sin la necesidad de que estos ya dispongan de una propia global.

El objeto de este trabajo ha sido la elaboración de la primera implementación de este protocolo, basada en el último borrador publicado, así como el análisis de sus prestaciones.

El resultado obtenido ha sido un paquete de "software" compacto y eficiente, el cual ha sido correctamente validado y podrá servir como referencia para futuros desarrollos.

Palabras Clave

IEEE, PALMA, Implementación, Protocolo, Rendimiento, MAC, Dirección local, Red, Nivel 2, SLAP, MAAP, DHCPv6

Table of Contents

List of Figures

List of Tables

Listings

Acronyms and Abbreviations

AAI	Administratively Assigned Identifier
CID	Company ID
ELI	Extended Local Identifier
IEE	Institute of Electrical and Electronics Engineers
MAC	Medium Access Control
MAAP	MAC Address Acquisition Protocol
OUI	Organizationally Unique Identifier
PALMA	Protocol for Assignment of Local and Multicast Addresses
SAI	Standard Assigned Identifier
SLAP	Structured Local Address Plan
U/L	universal/local

Chapter 1

Introduction

1.1 Motivation

Over the years, technology has improved our lives and it's completely doubtless that we tend to a more advanced and sophisticated one. In particular the telecommunications sector has been increasing the number of laptops, mobiles and IoT gadgets, among others, which all need a unique IP address to communicate with each other over the internet. This fact has lead to the commonly known problem of the lack of addresses in the internet, and it is directly related to the issue this work is expected to cover.

These devices mentioned before, need not only IP address[5] but also a unique MAC (*Medium Access Control*)[6] address for a proper functioning in layer 2 networks. So far, what it had been done until now is to assign a global unique MAC address to every manufactured telecommunication device following some rules to guarantee uniqueness within the network. In spite of this, it is easy to predict that a future lack of these type of addresses is happening, as well as it has occurred with IP[7].

To overtake this incoming issue, the IEEE is designing a protocol called PALMA (*Protocol for Assignment of Local and Multicast Addresses*), which consists in the dynamic assignment of local and multicast MAC addresses. However, this IEEE project has started recently and there is no more than several drafts in that regard without a final specification.

The purpose of this thesis is to implement the code of the PALMA latest version draft[8]

published by the IEEE and deal with the many complexities and uncertainties found in the multiple possibilities to achieve the same goal.

1.2 Objectives

The main objective of this project is to develop an implementation following the PALMA specification, which can be served for the testing of the protocol and as reference for future versions. The implementation is pretended to be as lightweight, portable and efficient as possible to make it compatible with embedded systems, including bare metal ones. Moreover, it will be subjected to rigorous tests in order to confirm the expected behavior and check its scalability.

To accomplish this we identify the next partial objectives:

- Remove all the obviousness and inconsistency of the PALMA draft.
- Develop a clear and comprehensive code that reflects a perfect structure based on objects and classes.
- Subject the final program to multiple tests which make up the functional validation plan.
- Elaborate some other testing to check performance and study the scalability parameter mainly.

1.3 Methodology

We have divided the development of the PALMA software in different intermediate phases, presented as follows:

- Read thoroughly and explore the contents of the last version draft related to PALMA. Once understood the fundamental concepts of it, eliminate all possible inconsistencies

and transform those into different functionalities that have sense with the rest of the specification.

- After the draft review, evaluate different coding options, such as the code language and structure, in order to choose the most appropriate bearing in mind the main objective. Afterwards, once the software is characterized, we proceed to develop the implementation of the different classes and coding modules required to achieve the wanted structure.

- Build a functional validation plan to guarantee the correct functioning of the protocol, focusing on the main functionalities as well as small details of the specification. This plan consists in validation tests that, in our case, must show not only that the software we have just made matches the draft specification but, also the possible changes made as a result of the inconsistencies mentioned before.

- Finally, elaborate more tests capable to show the code performance and some other parameters such as scalability.

1.4 Outline of Research

The present document is structured as indicated below:

- Chapter 2: State of the Art.

  A background of the evolution of link-layer local address assignments over the years is presented, as well as PALMA similar operation protocol's such as MAAP or DHCPv6 with the link-layer extension. At the end of the chapter, it is explained why PALMA appeared and what differs it with the rest of the protocols mentioned.

- Chapter 3: Assumptions to PALMA Draft

  We present our assumptions to the undefined behaviors, and clarify all the possible contradictions or inconsistencies found in the PALMA last version draft.

- Chapter 4: Implementation

  During this chapter, we face the programming paradigm and explain the resolution of the different difficulties and complexities found in our main objective, the implementation.

- Chapter 5: Tests and Results

  A complete set of tests that form a functional validation plan is shown, as well as some graphs resulting from scalability and protocol convergence time studies.

- Chapter 6: Conclusions and Future Works

  Conclusions of the implementation made and the matching between this one and the proposed objectives are commented, as well as some ideas for future PALMA versions.

- Chapter 7: Regulatory Frame

  An overview of the main regulations and normative this project and, above all, the draft specification is subjected to.

- Chapter 8: Socio-Economical Background

  A brief summary of all expenses related to the implementation, divided into personnel and equipment costs, followed by some conclusions about the socio-economical impact of PALMA.

Chapter 2

State of the Art

In this chapter we will explain all the necessary mechanisms in order to understand properly the basics of PALMA (*Protocol for Assignment of Local and Multicast Addresses*), or more precisely the technologies on which PALMA protocol was based.

Moreover, in order not to miss any part of the explanations described in this memory, PALMA latest version draft[8], whose implementation this project is trying to achieve, should be read before continue reading this thesis.

2.1 Background of Local and Multicast Address Assignment

To be able to understand the background of local and multicast addressing it is necessary to date back and read about the history of the IEEE (*Institute of Electrical and Electronics Engineers*) association in relation to this.

The initial standard addressing document of IEEE 802 Overview and Architecture, IEEE Std 802-1990, specified that the first address bit on the medium is the I/G Address Bit, used to identify the destination address either as an individual or as a group address. The second one is the Universally or Locally Administered (U/L) Address Bit, indicating whether the address has been assigned by a local or universal administrator.

Virtually no information regarding locally administered addresses was included. IEEE Std 802-1990 describes Organizationally Unique Identifiers as 24 bits in length, but the truth

is that the address space is of 22 bits. The first bit can be set to 1 or 0 depending on the application and the second bit for all assignments is 0, which means that the remaining 22 bits that cannot in fact be changed by the assignee, result in 2^22 (near 4 million) identifiers.

Regarding Universally Administered addresses, IEEE Std 802-1990 specifies that the "first 24 bits correspond to the Organizationally Unique Identifier as assigned by the IEEE, except that the assignee may set the first bit to 1 for group addresses or set it to 0 for individual addresses," such that "varying the last 24 bits allows the assignee approximately 16 million unique individual addresses and 16 million unique group addresses that no other organization can have."

A brief subclause on local MAC addresses was introduced in the revision IEEE Std 802-2014. This one states that local MAC addresses "need to be unique on a LAN or bridged LAN unless the bridges support VLANs with independent learning." Company ID (CID) was also introduces in that revision, referring to the IEEE RA for details. It did not specify the creation of local MAC addresses based on CID, but many hints in relation can be found on this statement: "A CID assignment has the X bit (the U/L address bit in a MAC address) set to one, which would place any address created with a CID in the locally administered address space."

The amendment of IEEE Std 802 by IEEE Std 802c-2017, on local MAC address usage, introduced the Structured Local Address Plan (SLAP). This one consists on the understanding of the Extended Local Identifier (ELI), the Administratively Assigned Identifier (AAI), and the Standard Assigned Identifier (SAI). Afterwards, the IEEE introduced an expanded version of its tutorial "Guidelines for use of the 24-bit Organizationally Unique Identifiers (OUI)" as "Guidelines for Use Organizationally Unique Identifier (OUI) and Company ID (CID)," including explanatory material regarding CID. Using language similar to that of IEEE Std 802-2014, the tutorial also suggested the possibility of building a local MAC address from a CID ("A CID has the X bit equal to one and consequently that places any address with the CID as its first 3 octets in the local address space (U/L = 1). Local ad-

dresses are not globally unique, but a network administrator is responsible for assuring that any local addresses assigned are unique within the span of use. If a CID is used to create MAC addresses, the X bit becomes the U/L bit. These addresses are by definition locally administered and consequentially may not be globally unique. CID, though, can be a useful tool in management of the local address space to help a network administrator keep local addresses unique for a specific local network environment.[8]

Finally in 2017, IEEE published the *IEEE Std 802c* a new standard which defines a new optional SLAP that specifies different assignment approaches in four specified regions of the local MAC address space. These ones were called SLAP quadrants and can be formed out of the combination of the IEEE 48-bit MAC address structure SLAP bits (see Figure 2.1 and Table 2.1 for more details. Furthermore, a briefly summary of each quadrant is explained below:

- In SLAP Quadrant 01, "Extended Local Identifier" (ELI) MAC addresses are assigned based on a 24-bit Company ID (CID), assigned by the IEEE Registration Authority (RA). The remaining bits are specified as an extension by the CID assignee or by a protocol designated by the CID assignee.[1]

- In SLAP Quadrant 11, "Standard Assigned Identifier" (SAI) MAC addresses are assigned based on a protocol specified in an IEEE 802 standard. For 48-bit MAC addresses, 44 bits are available. Multiple protocols for assigning SAIs may be specified in IEEE standards. Coexistence of multiple protocols may be supported by limiting the subspace available for assignment by each protocol.[1]

- In SLAP Quadrant 00, "Administratively Assigned Identifier" (AAI) MAC addresses are assigned locally by an administrator. Multicast IPv6 packets use a destination address starting in 33-33, so AAI addresses in that range should not be assigned. For 48-bit MAC addresses, 44 bits are available.[1]

- SLAP Quadrant 10 is "Reserved for future use" where MAC addresses may be assigned

using new methods yet to be defined, or until then by an administrator as in the AAI quadrant. For 48-bit MAC addresses, 44 bits would be available.[1]

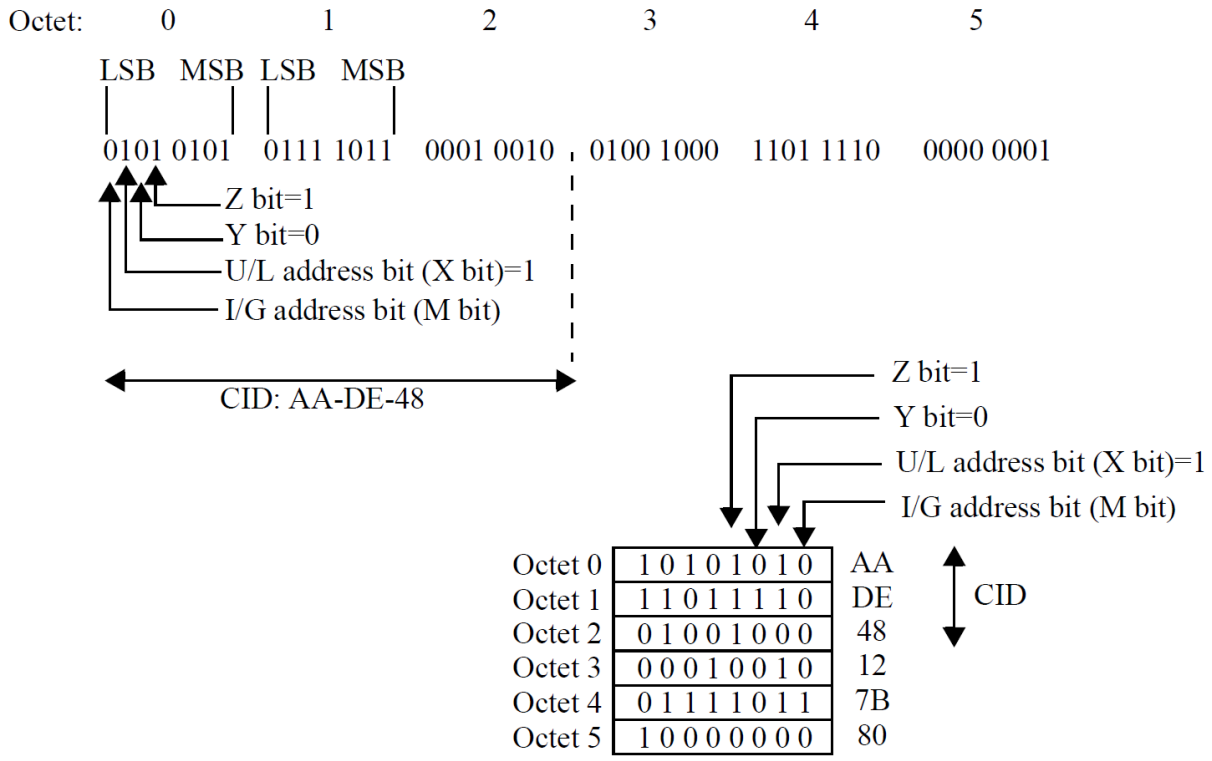


Figure 2.1: IEEE 48-bit MAC address structure[1]

Table 2.1: SLAP quadrants[1]

SLAP Quadrant	Y bit	Z bit	SLAP local identifier type	SLAP local identifier
01	**0**	**1**	Extended Local	ELI
11	**1**	**1**	Standard Assigned	SAI
00	**0**	**0**	Administratively Assigned	AAI
10	**1**	**0**	*reserved*	*reserved*

2.2 PALMA And Similar Protocols

This section introduces two protocols which serve for the acquisition of MAC addresses and their objective is similar to PALMA protocol's one. At the end, we will explain why PALMA arose and its differences with the other two protocols.

2.2.1 DHCPv6 with Link-Layer Addresses Assignment

The Dynamic Host Configuration Protocol version 6 (DHCPv6) is a network protocol for configuring Internet Protocol version 6 (IPv6) hosts with IP addresses, IP prefixes and other configuration data required to operate in an IPv6 network.[9]

However, there are several deployment types that deal with a large number of devices that need to be initialized. One of them is a scenario where virtual machines (VMs) are created on a massive scale. Typically the new VM instances are assigned a link-layer address, but random assignment does not scale well due to the risk of a collision.[10] The huge number of such devices could strain the IEEE's available OUI (Organizationally Unique Identifier) global address space. As there is typically no need to provide global link-layer address uniqueness for such devices, a link-layer assignment mechanism avoids conflicts to be generated inside an administrative domain. That is why it is desired to have some form of mechanism that would be able to assign locally unique MAC addresses.[2]

Since DHCPv6 is a protocol that can allocate various types of resources (non-temporary addresses, temporary addresses, prefixes, as well as many options) and has the necessary infrastructure to maintain such allocations it is a good candidate to cover the desired functionality.

DHCPv6 is a server based protocol, so for this new functionality, clients will have to ask for some MAC addresses to be assigned to the DHCPv6 server. For a better understanding of this new mechanism, it is important to know that "IA_LL" is the Identity Association for Link-Layer Addresses and that "LLADDR" is the Link-layer address option used to request or assign a block of link-layer addresses. Both nomenclatures are going to be used in the

brief text explaining the procedure of the new functionality found below, which includes as well a summarized diagram of the exchanged packets between a DHCPv6 server and a client who wants some link-layer addresses (see Figure 2.2):

1. Link-layer addresses, MAC addresses are assigned in blocks, with The smallest block being a single address. To request an assignment, the client issues a Solicit message with an IA_LL option in the message. The IA_LL option MUST contain a LLADDR option. In order to indicate the preferred SLAP quadrant(s), the IA_LL option includes the new OPTION_SLAP_QUAD option in the IA_LL-option field (alongside the LLADDR option).[1 2]

2. The server inspects the Solicit message contents. For each of the entries in OPTION_SLAP_QUAD the server checks if it has a configured MAC address pool matching the requested quadrant identifier, and an available range of addresses of the requested size. If suitable addresses are found, the server sends back an Advertise message with an IA_LL option containing an LLADDR option that specifies the addresses being offered.[1 2]

3. The client waits for any server to send Advertise responses and picks a server that advertise an address in the requested quadrant(s). The client then issues a Request message that includes the IA_LL container option with the LLADDR option copied from the Advertise message sent by the chosen server. It includes the preferred SLAP quadrant(s) in a new QUAD IA_LL-option.[1 2]

4. The server after receiving the Request message, assigns the requested addresses and generates a Reply message, which will be sent back to the client. Upon receiving a Reply message, the client parses the IA_LL container option and may start using all provided addresses.[1 2]

5. The client is expected to periodically renew the link-layer addresses but this behavior can be administratively disabled in the server. To do this renewal, the client must

send a Renew option that includes the preferred SLAP quadrant(s) in the new QUAD IA_LL-option. This is because, in case the server is unable to extend the lifetime on the existing address(es), the preferred quadrants are known for the allocation of any new addresses.[1 2]

6. Finally the server responds with a Reply message, with an IA_LL option that includes an LLADDR option with extended lifetime.[1 2]

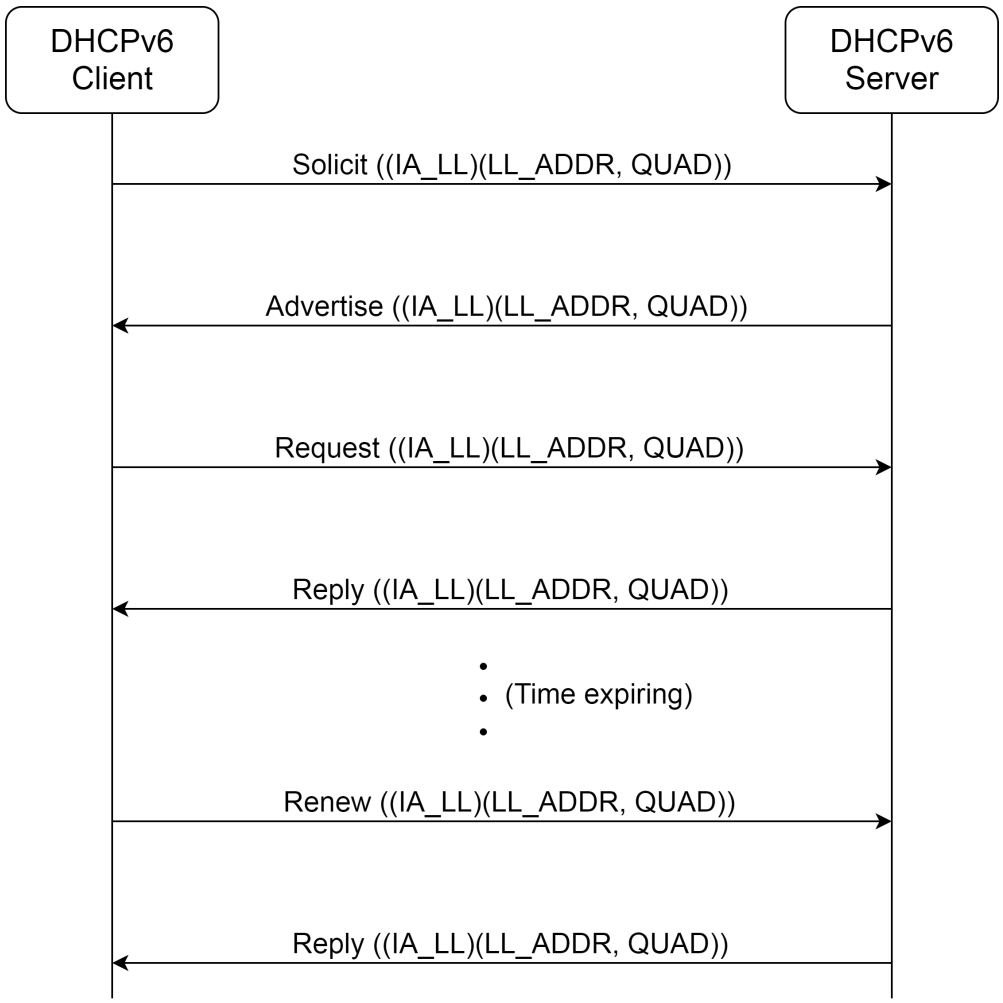


Figure 2.2: DHCPv6 signaling flow between client and server[2]

2.2.2 MAC Address Acquisition Protocol

The media access control (MAC) Address Acquisition Protocol (MAAP) is designed to provide a way to allocate dynamically the multicast MAC addresses needed by AVTP (Audio Video Transport Protocol). This whole section information is a summary of the IEEE 1722 document[11 3].

The MAAP specifies a mechanism to allocate multicast MAC addresses dynamically in a specified address range. Any application that uses addresses from the MAAP dynamic allocation pool shall implement the MAAP and MAAP shall be used to allocate these addresses. MAAP can be used to request a single address or a range of consecutive addresses.It uses a probe, announce, and defend mechanism in a peer-to-peer configuration to acquire addresses to be used by AVTP. To acquire an address range for use, an implementation of MAAP executes the following steps:

- Select an address range from the MAAP dynamic allocation pool.
- Send a series of MAAP_PROBE messages to determine whether the address range is already in use.
- Listen for MAAP_DEFEND messages indicating the address range is in use.
- Repeat the above steps until an unused address range has been found.

If everything goes right, the client executing the MAAP protocol acquires an address range and enters in a defending mode, where it announces and defends the address range as follows:

- Send MAAP_ANNOUNCE messages periodically to inform the network that those address ranges are currently in use. This allows new joined clients in the network to be aware of this information and avoid future possible conflicts.
- Listen for MAAP_PROBE messages and send MAAP_DEFEND messages in response if any address ranges conflict with the previously acquired address range.

- Listen for MAAP_ANNOUNCE messages that conflict with previously acquired address ranges, and in such case, discontinue the use of that address range and restart the protocol.

Furthermore, regarding MAAP messages, some specified information can be found below as well as in Table 2.2, which shows the MAAP probe and announce messages constant values.

- All MAAP_PROBE and MAAP_ANNOUNCE frames are sent with a multicast destination MAC address set to the reserved MAAP multicast address defined in the specification (Table B.10 of IEEE 1722).
- All MAAP_DEFEND frames are sent with a destination MAC address set to the source MAC address received in the MAAP_PROBE frame that caused the sending of the MAAP_DEFEND frame.
- The source MAC address shall be set to the MAC address of the sender.

Table 2.2: MAAP probe and announce constant values[3]

Constant	**Value**
MAAP_PROBE_RETRANSMITS	3 s
MAAP_PROBE_INTERVAL_BASE	500 ms
MAAP_PROBE_INTERVAL_VARIATION	100 ms
MAAP_ANNOUNCE_INTERVAL_BASE	30 s
MAAP_ANNOUNCE_INTERVAL_VARIATION	2 s

In order to get a more familiar understanding of MAAP protocol's operation, Figure 2.3 and Figure 2.4 are representing two complete scenarios for the requesting and defending addresses respectively, in a MAAP operative network environment.

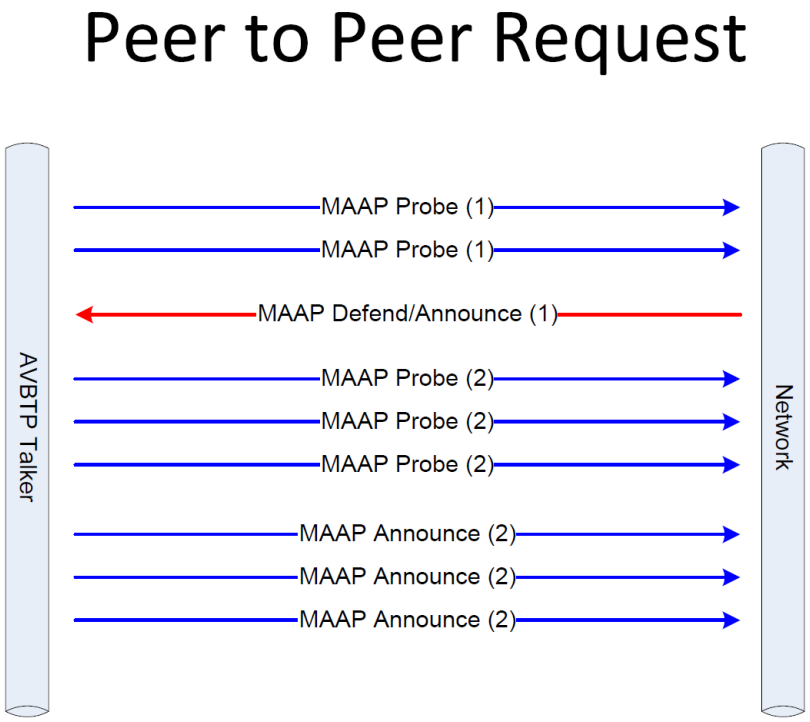


Figure 2.3: MAAP peer to peer requesting scenario[3]

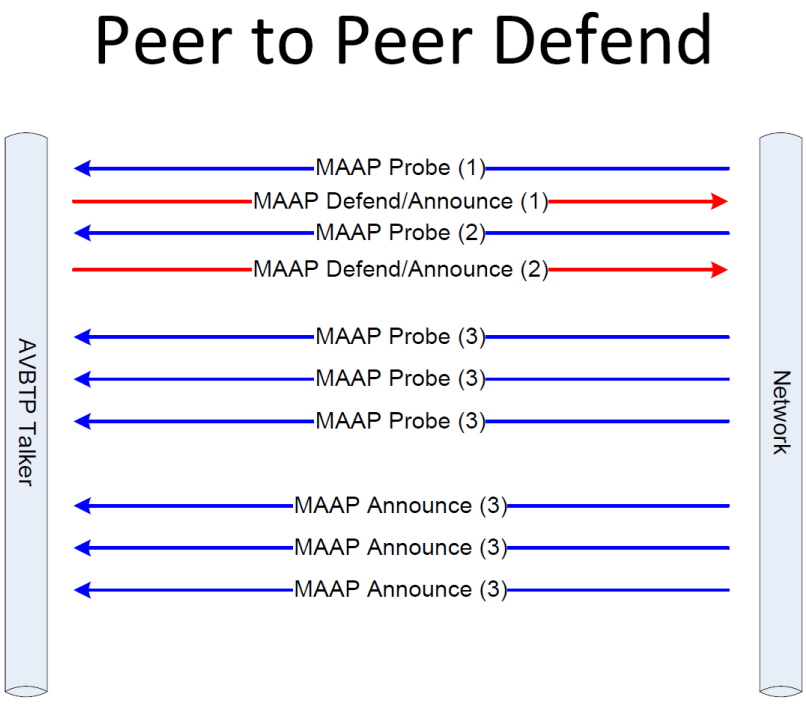


Figure 2.4: MAAP peer to peer defending scenario[3]

2.2.3 PALMA Summary

The Protocol for Assignment of Local and Multicast Addresses (PALMA) is summarized herein. PALMA is specified as a single protocol supporting both server allocation and server-less peer-to-peer claiming-based self-assignment. Both claiming-based and server-based PALMA modes handle the case in which the station lacks of a valid MAC unicast address assignment prior to the execution of the protocol. If we recapitulate, we have seen two protocols useful for the MAC address assignation before and one of them, DHCPv6, handle the case in which the client station lacks of a MAC unicast address previously to being executed but not the case of multicast MAC address allocations. The other one, MAAP, does allow the allocation of multicast addresses but does not cover the case of missing a unicast MAC address prior to the execution of it. Moreover, DHCPv6 works only in a server based mode whereas MAAP in a client based mode.

As we can see PALMA protocol is a mixture of MAAP and DHCPv6 with the link-layer extension. It takes the server based idea and the lack of a preassigned unicast address incident from DHCPv6 protocol, and the peer-to-peer claiming based self-assignment allocation from MAAP. In conclusion, two useful protocols mixed in one even more practical and multi-functional.

Chapter 3

PALMA Draft Assumptions and Clarifications

This chapter explains the inconsistencies and contradictions the draft contains, and moreover the proposed alternatives to these that will be as well included in the implementation. As in every draft paper, no matter the subject, many holes or even mistakes are probably found. That is why the previous step of analyzing and understanding well the editor's purpose, is essential. In our case, several careful reads and reflections are needed in order to achieve a complete understanding of the protocol itself and pinpoint any possible error.

We have made assumptions for every mistake or inconsistency found and divided them into the following subchapters that are named like the PALMA draft[8] chapter in which they were located. For that reason is very convenient to have read the document before continuing.

3.1 PALMA Theory of Operation

3.1.1 PALMA Message flows with self-assignment

In this section of draft, the message flows in the self-assignment PALMA mode is explained including a diagram (Figure 1 in the draft). In our implementation we will assume that for the multicast claiming the client can send DISCOVER messages without a prior assigned unicast address and without including any unicast addresses in the claimed set. The two

reasons of this are, first one because when a client claims either unicast or multicast addresses it can only specify one of these types, it cannot claim both a unicast set and a multicast set. The second reason is because when a client claims a multicast set without having a prior assigned unicast address, which means it uses a random DISCOVERY source address, it can be responded by a server with an OFFER, and this OFFER message will include a unicast address for the client to use it as source. In any case if a client claims a multicast set and no server replies are found within the network, the client cannot self-assign the claiming set unless it has a preassigned MAC address.

3.1.2 PALMA Message flows with server-based assignment

This section in the draft explains the message flows in the server-assignment PALMA mode and includes a figure to represent it (Figure 2 in the draft). The problem here is that the figure represented is not very accurate and some clarifications must be done. Whenever a client without a prior assigned unicast source address claims a set with a DISCOVERY message, it self-assigns for a short period of time (DISCOVER interval time) a random address from the PALMA random unicast source addresses. When the time between DISCOVER messages expires, the client must self-assign a new random address and so on. This fact implies that if a server sends an OFFER message to the client offering an address set, this one has to reach the client before it self-assigns a new address, otherwise, the client's device network card will dismiss the packet. Furthermore, in our implementation, several OFFER packets can be received by the client before the DISCOVER interval timeout, because the client will not respond to the OFFER message until the timeout arises.

3.2 PALMA Message Addressing and Protocol Type

In this section we will make some clarifications about the source address and destination address of some of the PALMA messages. Those who are not considered here is because there is no assumption to it with respect to the original PALMA draft.

3.2.1 PALMA DISCOVER Message Addressing

Following the draft, the destination address of the DISCOVER message can be either the PALMA Multicast Address or the source address of a certain server when the client wants to do a renewal of the addresses offered by this one (as specified in line 1-2 of page 10 in the draft). This last possibility will not be taken into account in the implementation because we have thought that every message directly aimed to a server has to be done by means of a REQUEST message. Therefore, whenever a client wants a renewal of the addresses previously assigned by a particular server or even starts the protocol with a known server address and wants to claim an address set to this one, it must send a REQUEST to this server specifying its intentions.

3.2.2 PALMA REQUEST Message Addressing

If we continue reading the draft, at line 35-36 of page 10, we can find a brief text explaining the possible destination addresses of a REQUEST message. As a result of the previous subsection, we can affirm that the destination address of a REQUEST message can either be the source address of an OFFER message previously received by a client, or in addition, whatever source address of an operative PALMA server in the network, whenever the client's protocol software has been executed with a known server address beforehand.

3.2.3 PALMA DEFEND Message Addressing

Regarding the specification about the DEFEND message, it is said that this message type shall be send in response to a DISCOVER message and should have as destination address the source address of this one (as specified in line 6-7 of page 11). To this, we will just add that PALMA DEFEND message can be as well a response to an ANNOUNCE and in that case it would have as destination address the source address of this ANNOUNCE message.

3.2.4 Protocol Identifier

In the draft we can find the protocol identifier of all PALMA frames, which shall be the PALMA Ethertype, in Table 4 of the draft, but this one is not defined yet. Consequently, in our implementation, we will consider the PALMA Ethertype as 0x33ff read in hexadecimal.

3.3 PALMA Message Format and Content

3.3.1 PALMA message parameters

The draft subchapter 6.5.2 describes the format of the parameter subfields for each of the message parameters. This format depends on the Parameter ID (an indispensable subfield for every parameter), but every parameter has a subfield called Length, where it is contained the complete length of the whole parameter including all of its subfields. We can understand from this that the maximum length that the Length subfield can represent is 255 octets, from where one octet is for the Parameter ID subfield and another one for the Length subfield itself.

Having this fact clear, we introduce now the inconsistencies located in the draft in relation to it.

Station ID Parameter subfield

The Station ID Parameter subfields are specified in Table 11 of the draft. The inconsistency we can find here is that Station ID subfield occupies up to 254 octets, as shown in the table. This is impossible, it must occupy up to a maximum of 253 octets and in this is the way it is going to be implemented.

Network ID Parameter subfield

The Station ID Parameter subfields are specified in Table 13 of the draft. The inconsistency we can find here is the same one as in the Station Id Parameter specification. In this case, it is shown that the Network ID subfield can occupy 254 octets, which as mentioned before,

will be changed to 253 from now on.

3.3.2 PALMA Messages

In the chapter 6.5.3 the PALMA messages are specified and each one of these has some obligatory parameters and others that could be optionally included. As we have been doing until now, we will show some of the changes we have made for our implementation in order to achieve a better protocol functioning.

DISCOVER

Following the draft, the DISCOVER message is used for self-assign mode and for server based mode in the case of doing a renewal. As mentioned before, we discarded this last case and we will use the DISCOVER message only for self-assign mode. This means that a client sends DISCOVER messages over the network when it wants to self-assign a set of unicast or multicast addresses either in the case it already has a prior assigned unicast source address or it lacks of one.

For this purpose, the DISCOVER message can be formed without any parameter, in which case it would be a "null DISCOVER" that does not specify any claiming address set and only if a server offers some addresses, the client will finally get those assigned, in any other case it has nothing to do.

Nevertheless, the DISCOVER message can also have the expected parameters, those are, the Station ID parameter and the MAC Address Set parameter. It could optionally include as well a Vendor Specific parameter, but for our implementation it will never have a Client Address parameter, because we have thought that it would be better to make the server the only one in charge of this functionality. As a result, only OFFER messages will be capable of including a Client Address parameter.

REQUEST

Regarding REQUEST message, we have also mentioned sections before that it will be used not only as an OFFER response but also when a client starts the protocol with a known server address. In that case, no DISCOVER messages would be sent but REQUEST messages instead directly to the server whose source address is known.

REQUEST messages, as opposed to DISCOVER messages, must include at least a MAC Address set parameter, otherwise it would be useless to send a "null REQUEST" to a certain server without specifying the set of addresses you request from it. In such case, if a client wants to send a "null REQUEST", this one should be indicated by the claiming set initial address and then a number 0 in the specified count addresses to request. Additionally, REQUEST messages can optionally include the Station ID parameter.

ACK

We have made many changes with respect to the draft we were given to implement and that causes even more changes for the whole to have a complete sense.

ACK messages are sent in response to REQUEST messages, that is a fact that will not be changed, but we have established just before that REQUEST messages could be sent in the beginning of the program, without having a prior received OFFER. For that reason, ACK messages must include a MAC Address Set parameter and a Lifetime parameter. This is because in the case of a server receiving a REQUEST from a particular client to whom the server has not sent an OFFER before, we can either reject the requesting set or offer an alternative one including the lifetime of this one, all by means of the ACK message.

Moreover, ACK messages will continue having the chance of including a Station ID parameter, which will be compulsory in case the REQUEST message has this option.

3.4 PALMA Constants

One of the most important aspects when facing the implementation is having defined all the PALMA constants, that is for example, the timers duration or the counter initial values. The draft's chapter 6.6 covers this part with two tables, but unfortunately values are not defined yet in this draft version. To encounter this problem, we have taken the example of the MAAP (*MAC Address Acquisition Protocol*) protocol constants defined in Table 2.2. Results for timer values can be seen in Table 3.1 as well as for counter initial values in Table 3.2.

Table 3.1: PALMA timer values

Purpose	Timer Name	Duration
DISCOVER interval	dsc_timer	500 ms
ANNOUNCE interval	ann_timer	30 s
REQUEST interval	ack_timer	500 ms
self-assignment lifetime	life_timer	600 s

Table 3.2: PALMA counter initial values

Purpose	Name	Value
maximum number of DISCOVER messages	dsc_count_max	3
maximum number of ANNOUNCE messages	req_count_max	3

Furthermore, for the implementation we have add an interval variation for the PALMA messages shown in Table 3.1, replicating the example of the MAAP protocol, which will be useful for a more proper functioning. In the case of the DISCOVER and REQUEST messages, the value of this interval variation is 100 milliseconds, whereas for the ANNOUNCE message is 2 seconds.

3.5 PALMA flow diagrams

In this last section of this chapter we will describe in details the differences between the diagrams presented in the draft and the real diagrams used to write the code. We have made changes in every client state except on the Start State. This means that if we do not specify any changes in relation to any other part of the draft but the ones that appear in this section, is because we consider that the rest not named here is consistent, as we have been doing through this whole chapter.

3.5.1 PALMA Client DISCOVERY State

We have analyzed the flow diagram of the Client DISCOVERY State and everything seemed consistent. However, we have changed the part of the diagram where the behavior when an OFFER is received is explained. From our point of view, and for the ease and consistency of the code, it is better to allow the client to decide whether it agrees with the OFFER or not. This is, in the case it agrees, the next step would be change the protocol mode to the server based mode and start sending a REQUEST message. Moreover, if the client does not agree with the hypothetical OFFER, it will not go to the START state, otherwise it will be nonsense. We have thought that instead, it will ignore this OFFER and continue with the self-assignment mode. In this case, as we are in the DISCOVERY state, it will continue issuing DISCOVER messages. The resulted diagram for this newness is shown below, in Figure 3.1.

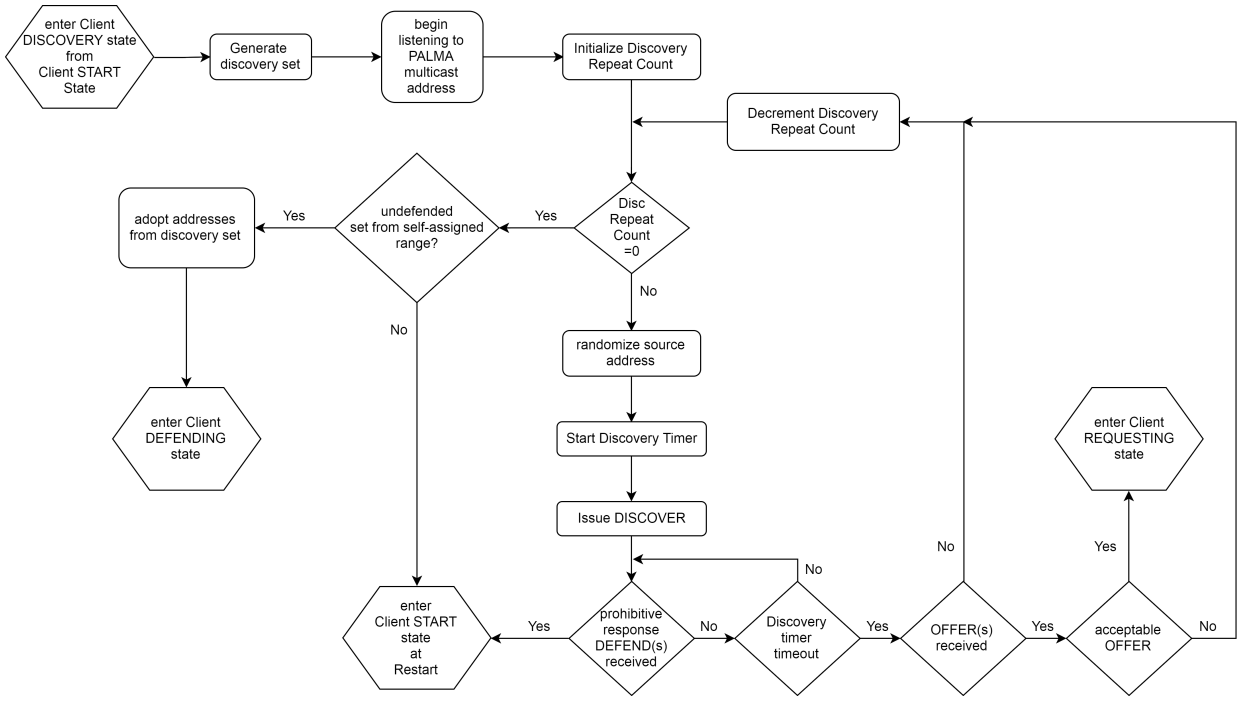


Figure 3.1: PALMA Client Flow Diagram: DISCOVERY State

3.5.2 PALMA Client REQUESTING State

In relation to the REQUESTING State flow diagram of the draft, we have corrected a small issue and added a new functionality barely described throughout the draft.

The issue was not a big one rather than the fact that we enter the REQUESTING State from the DISCOVERY, the BOUND, the DEFENDING and also the START State, incident which was not considered in the original draft, at least not in this diagram. Actually, this fact herein explained is corroborated with the START state flow diagram, where we can easily see that a client can directly enter the REQUESTING state from the START state only when it knows a server source address.

The new functionality refers to the one related to RELEASE messages. The client will send a RELEASE message to the server whenever the last ACK contains a MAC Address set which the client does not like or agree with. In other words, the client sends a RELEASE message when it receives a non acceptable ACK. An example to understand this rare case is the following:

1. A client starts the protocol's program with a known server address, so it sends a REQUEST message to this server, whose address is already known, requesting a set of addresses which are not administratively allowed.

2. The server receives this REQUEST message and replies with an ACK offering an alternative address set, due to the incompatible requested set.

3. The client receives this ACK message but it does not agree with the offered set and does not want to assign any of those offered addresses. To notify this and let the server know that it will not use this set, the client issues a RELEASE, indicating that these addresses are now free to be offered to another client.

For a more graphical understand of this new behavior we have modified the draft flow diagram and the result is shown below, in Figure 3.2.

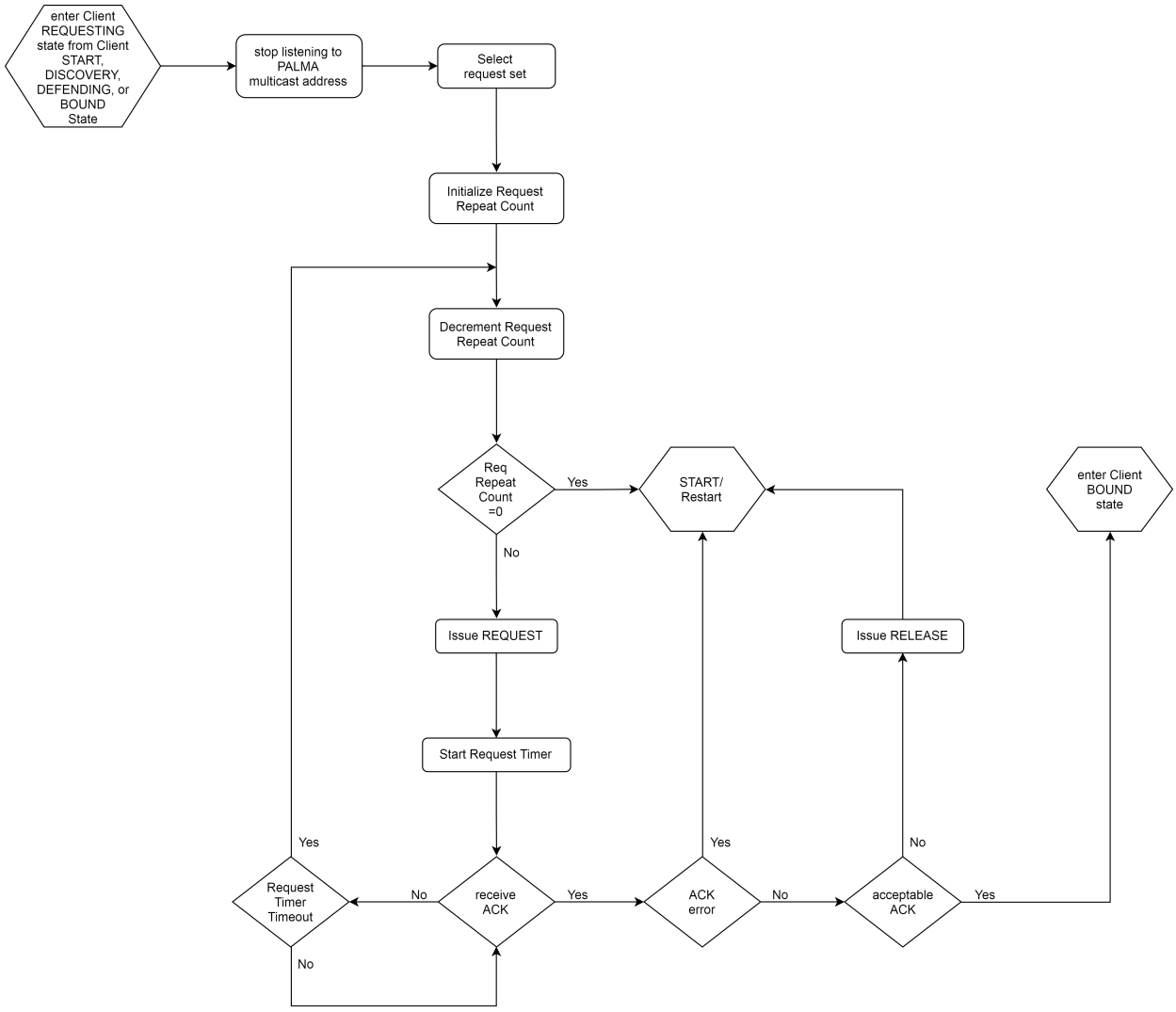


Figure 3.2: PALMA Client Flow Diagram: REQUESTING State

3.5.3 PALMA Client BOUND State

Regarding the Client BOUND State, we have added minor fixes to the flow diagram, one of them related to the functionality mentioned in the previous subsection.

When a client is in the BOUND state, it should stay there until the assignment timer runs off, or also until the user requests a release. This last possibility can happen for example when an application starts the program because it needs some MAC addresses, but minutes later it does not need those addresses anymore. In our implementation we handle this

functionality by exiting the program. Whenever this occurs, the client issues a RELEASE message, notifying the server that those addresses previously assigned are now free.

Furthermore, the renewal functionality is also included in the new diagram, Figure 3.3. When the assignment timer timeouts, the client can go either to the START state if the client has not been launched with the renewal option, or, to the REQUESTING state in the case of having the renewal option active. In our implementation, this option has to be enabled before launching the client.

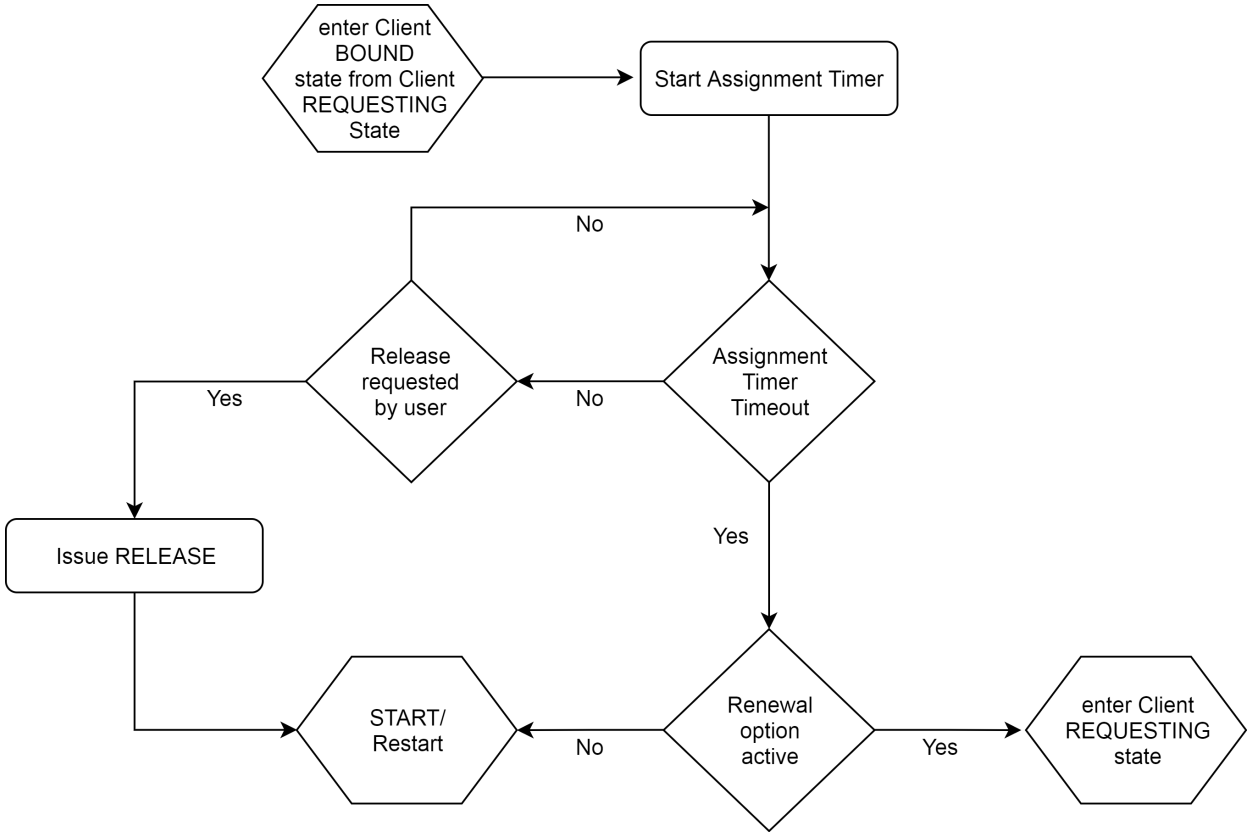


Figure 3.3: PALMA Client Flow Diagram: BOUND State

3.5.4 PALMA Client DEFENDING State

To conclude the client's states we have the DEFENDING State. The flow diagram found in the draft in relation to this state is very complete, but holds some inconsistencies that we are going to expose herein.

The first thing we have modified is the same as with the DISCOVERY State and its behavior when receiving an OFFER message. The performance here will be the same, this is when a client adopts an address set and afterwards receives a non acceptable OFFER, it keeps adopting and defending those addresses. There is still no sense on a dispose of every adopted address and going back to the Start state.

The last part of the diagram has been also changed in order to fix a major problem. This one appears when there exits two clients with adopted sets of addresses where one is a subset of the other. Moreover, if the two clients do not resolve internally the problem, they will be sending DEFEND messages to each other forever. For that reason we have contemplated this issue and came up to a working solution, which can be seen in the Figure 3.4.

Whenever a client in the DEFENDING State receives a DEFEND message, if this one alerts a full address conflict, then there is no doubt and this client must stop listening to the PALMA multicast address and go back to the START State. However, if the conflict is partial, the client tries to solve the conflict internally. For the implementation, a conflict can be resolved internally if the first address from the adopted set of the client receiving the DEFEND message does not match the first address of the conflicted set, and in addition, the client can remove 1 or more dispensable addresses from this adopted set. We have called dispensable addresses to those who are between a minimum and maximum values, previously configured before launching the program. We will get a deeper focus on these configuration values later on next chapter.

If a conflict can be resolved internally, then the client who received the DEFEND or ANNOUNCE message removes as many addresses from the adopted set until having the

minimum number of them. As mentioned before, this minimum value is compulsory to be defined before the execution of the protocol program. Once done that, the client checks if the conflict is resolved or there is still some addresses in its adopted set that match the ones in the conflict set. If there is no more conflict then, the client continues its DEFENDING state normally but this time without defending the previously removed addresses due to the conflict. Otherwise, the client must send a DEFEND message indicating the still existing conflict and defending its adopted set, which has now the minimum number of addresses.

The resulted flow diagram after all this changes in the client DEFENDING state behavior is shown below, in Figure 3.4.

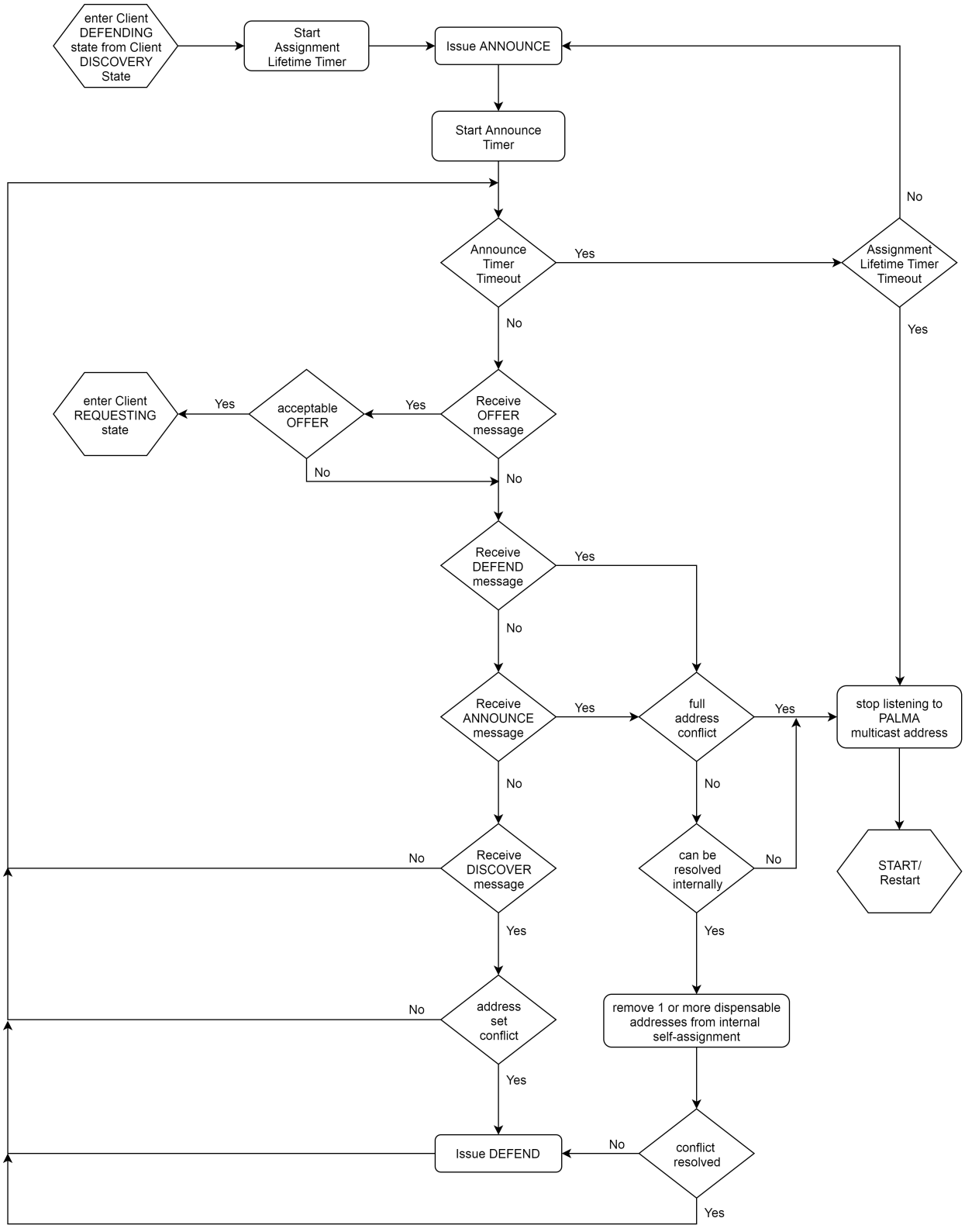


Figure 3.4: PALMA Client Flow Diagram: DEFENDING State

3.5.5 PALMA Server

The PALMA Server is basically a management tool to distribute addresses from a certain configured huge administratively accepted set and keep a monitoring of both the free and used ones. The diagram flow presented in the draft lacks of the new renewal functionality as well as of some more information we have just added to it. It is important to remark that we can have many servers running in a network, but these ones should have different set of addresses to distribute in order to avoid collision problems.

First of all, before issuing any OFFER, the server must check if it can offer any addresses. To do this, the server has a database where it stores all the information once the server program is launched. In this database, we could find either the free addresses and the used ones, as well as the lifetime left for the used ones.

If a server receives a DISCOVER message it replies with an OFFER, as long as there is enough free addresses to give away in the server's database. The same happens when it receives an ANNOUNCE and the "AutoassignObjection" option in the server is active. This is a new option we have designed for the server's configuration. The user configuring the server can decide if this one will offer addresses to the clients which already have adopted some addresses in the self-assignment mode or not.

Finally, if a server receives a REQUEST message, this one could be originated by the response to an OFFER, due to a renewal or simply by a client that starts the PALMA protocol with this known server address. In the case it is due to a renewal, the server must check that this particular client has been offered before those addresses it is requesting to renew. In a normal situation, the server will accept the renewal and reset the lifetime of this address set in the server's database. If the REQUEST message comes from a new client then the server has to analyze whether the request set is valid or not. Here we have added a new functionality which consists in offering an alternate set in the case the alternate set option is enabled in the server and the requested set by the client is not valid. Every change made with respect to the original Server flow diagram can be seen below, in Figure 3.5.

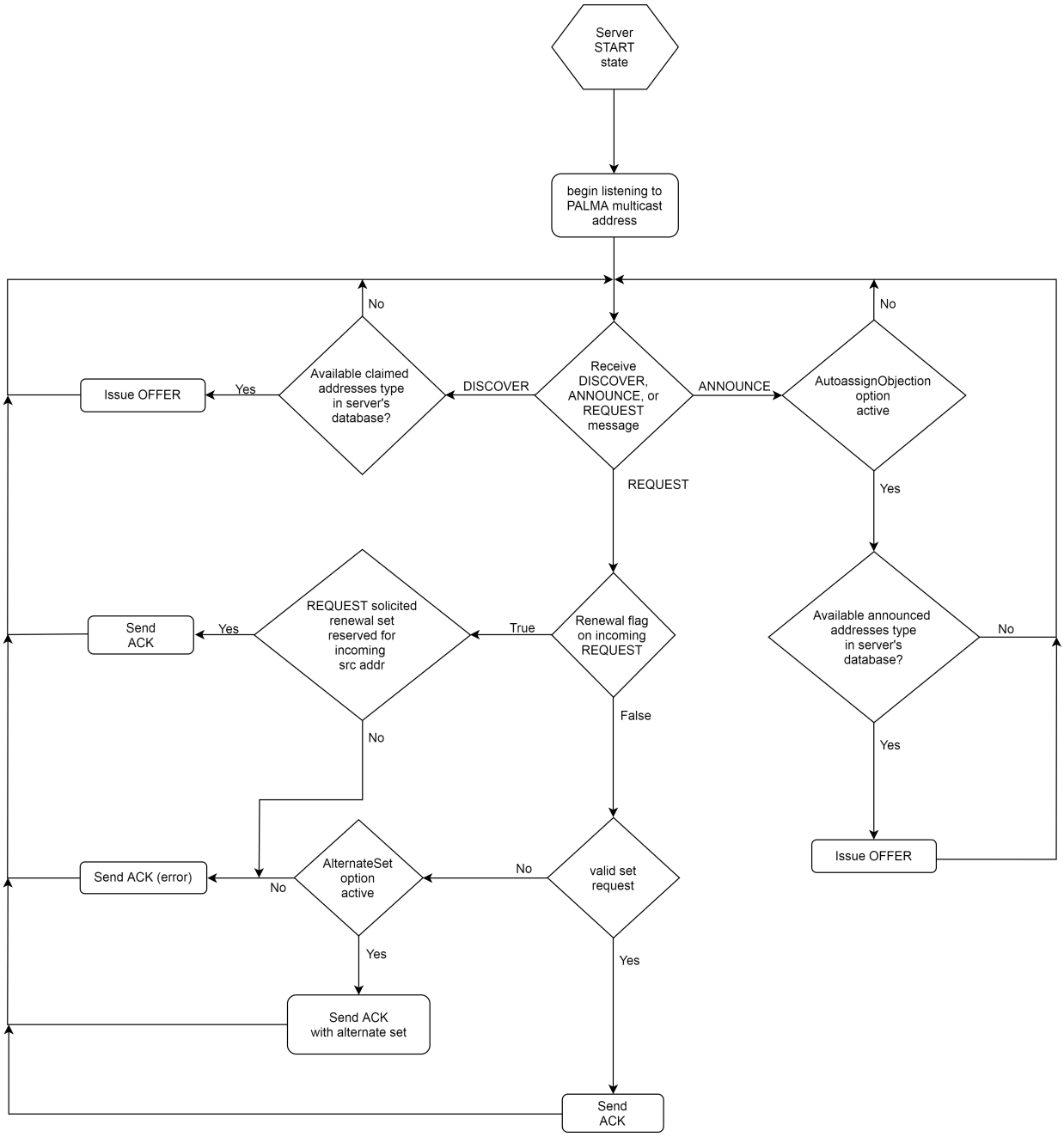


Figure 3.5: PALMA Server Flow Diagram

Chapter 4

Implementation

4.1 Programming Paradigm

Once we have understood the PALMA specification and made all the possible clarifications and assumptions, programming takes the next step, but before start writing code we firstly have to define a multiple set of uncertainties:

- Functional or object oriented programming.
- User level or kernel level programming.
- Programming language.
- Single thread or multi thread program.

Regarding the first point, we have chosen an object oriented programming for the implementation of this protocol, because it reflects an architectural model and increases the run-time performance compared to the functional programming. One more reason for this decision is that when programming in any object oriented language, we avoid layering penalties, which gives us a great advantage over the functional ones.[12]

In relation to the second issue, we were absolutely doubtless in choosing a user level programming for multiple reasons. When programming at user level, you can customize the code and make this one read configuration files, which is one important functionality our

code has. The ease of debugging and code maintenance were two essential characteristics we could not ignore. Moreover, programs at user level can easily coexist with others, and in this case we want this to happen with PALMA.[13]

It is easy to define the programming language once we have got clear that we want a user level and object oriented program. Java and C++ are two languages that fulfill these characteristics and came up to our minds. It is not new that Java[14] has lots of libraries but it is much more inefficient that C++[15]. In addition, our main objective is to make the implementation the more lightweight and not library dependent as possible, so as to be capable for running in embedded systems. Java needs a Java Virtual Machine (JVM)[16] to be executed and it is difficult to find such processor in embedded systems, while C++ compilers are available for any kind of processor and the result of compiling C++ is machine code, interpretable by any system.

Finally, facing the last item of the list above, probably the most technical one, we have thought on doing a single thread implementation. This is because PALMA protocol allows the execution in a single thread and this option is more efficient. Furthermore, single thread programs, as ours, can be run without any problem in such kernels that do not support multitasking, as for instance embedded systems.

Before proceeding to the next section and get a deeper focus on the different implementation code modules, it is important to say that we have made the code to be working on Linux operative system. We have chosen this system rather than others because it is open source, it is very well documented and it has extensible uses.[17] Tools such as "gdb"[18] command for debugging or the program "Mininet"[19] for the posterior code tests had a lot to do with this decision as well.

4.2 Network Packet Management

An essential part of the implementation is the module that manages the network packets, both the incoming and the outgoing ones.

In Linux systems it is common to use sockets[20] for this management. In our case we have used raw sockets[21] and, as we had defined a single thread program, writing is thought to be blocking for the socket, whereas reading is not. Any operation done with sockets can be blocking or not depending on the behavior we want these to have. The reason for writing tasks to be blocking is not other than the fact we want outgoing packets to be sent as fast as possible, but when reading from incoming packets other protocol signals or timeouts could be more important.

To handle the waiting of incoming packets we use the Posix[22] function called "pselect()"[23], which is going to handle as well all the timeouts of the different timers in the protocol.

This code module is formed by one function that initializes the sockets with the corresponding arguments in each case, either blocking or not blocking socket type for instance, and another two functions that are used to manage the incoming and the outgoing packets, in the sense of writing and reading from the particular sockets. There are two more functions for the registering and delisting of new addresses in the sockets, which are very useful in this protocol of dynamic assignation of MAC addresses.

Recapitulating the objectives for this implementation, we mentioned that one objective was to make the implementation capable for running in embedded systems and these ones probably will lack of socket tools. Nevertheless, these systems have at least some libraries to manage signals or any kind of interruption handlers to be used for the management of incoming and outgoing packets. This means that with a few particular changes we could have a compatible running code for practically any embedded system.

4.3 Timers

PALMA protocol consists largely of timers and that is why the module which manages all the protocol timers is very important for this project.

From the different ways we could have chosen to control and manage these timers, we first thought of Posix timers[24], but these ones use signals and in order to integrate these timers and the network packets with the rest of the implementation, we have better decided to implement a particular timer list.

We basically have encoded a structure for timers and according to the arrival order we put them on a list. This particular list is formed by all the different timeouts the protocol needs to manage. When a new timer needs to be monitored, it is added to the list subtracting the value of its timeout with the timeout value of the timer in the upper position of the list. In other words, the first position of the list has the least value timeout, that is the timer whose time is the first to come due. The second timer of the list is the second timer to run out, but the timeout is indicated as the subtraction of its real timeout with the timeout of the first timer. For instance if we have 3 timers whose timeouts are 10, 8 and 6 seconds (Figure 4.1), the list will be as follows:

In the first position of the list represented on Figure 4.1 we would have the least time value timer, in this case the timer whose timeout value is 6 seconds. For the second position we would take the subtraction of 8 - 6 seconds and the result is 2 seconds. We would say then that for the second timer in the list, its time to come due is the sum of its value in the list, that is 2 seconds, plus the values of all the upper positions in the list, that is 2 + 6 = 8 seconds. For the last case, we have to add a timer whose timeout value is 10 seconds, we firstly deduct 6 seconds to it and we would have now 4 seconds, which is bigger than the second position of the list, 2 seconds. Then we deduct 2 seconds to 4 and we end up with 2 seconds. There are no more items in the list so, the third position would be 2 seconds.

The reason of doing this is because for the timeouts management, we are going to consider only the first position of the list and this value is going to be given to the function

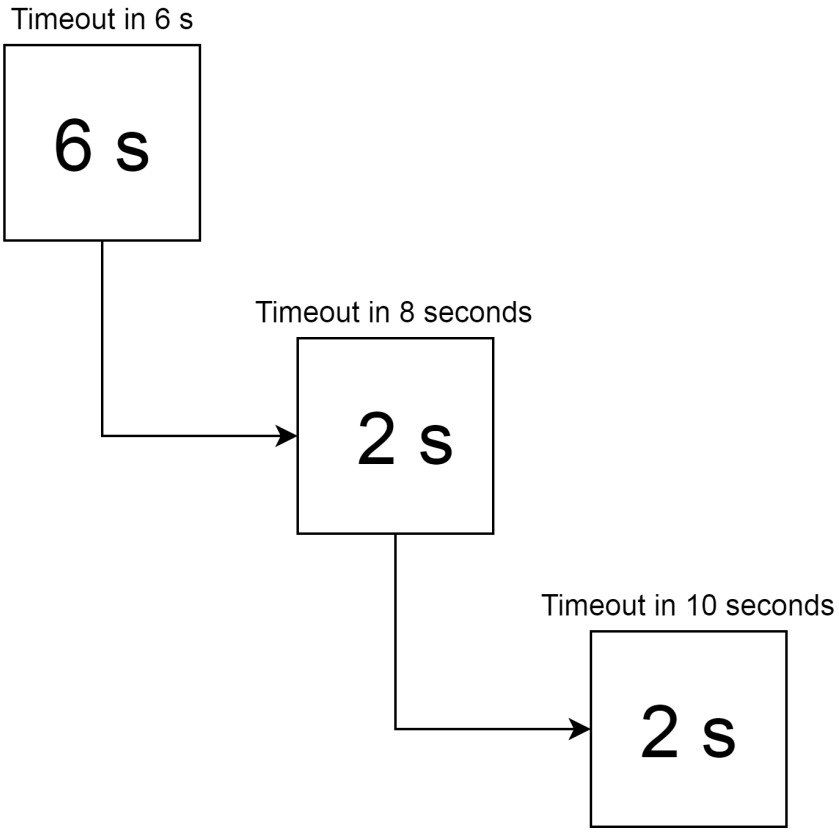


Figure 4.1: Timer List Example

pselect() mentioned in the previous section, the same that manages the network incoming and outgoing packets. If no packets are sent or arrived, pselect() sleeps until the timeout of the given timer value (the first position of the timer list). When this one becomes due, pertinent functions related to this expired timer are called and the next position in the timer list is passed as argument to the function pselect(). We then have leveraged the tool managing the network packets to manage as well all the timers of the protocol, achieving a very efficient functioning.

4.4 States

As we have seen in the draft clients have 5 states: START, DISCOVERY, REQUESTING, DEFENDING, and BOUND. For the implementation we have defined a C++ object for each one of these states a client passes through during the protocol's life. This object includes several timers which are needed in different ways for every state. Furthermore, these state objects virtualize a function for the processing of incoming network packets.

Regarding the server, this one does not have state, that was one of the purposes of the designers of this protocol, make the server stateless in order to be as simple as possible.

4.5 Databases

In earlier versions of PALMA, there was no need of having databases, but in this draft we are implementing, databases play a very useful role. They are hugely necessary to save track of active assignments, both the server based ones and the self-assignments. All servers in the network must have one to track the assignations offered to clients, and clients need a database as well to track the self-assignation set they are interested in. This is because when sending a DISCOVER of the whole set, other clients will send DEFENDS or ANNOUNCES, and all that information stores in the client's database, so that the second DISCOVER could be chosen between the free sets left in this database.

For this purpose, we have gathered all the requirements for this database, which can be resumed below:

- Find rapidly free address sets.
- Search quickly given a certain address.
- Determine whether a particular set is free or assigned.
- Include a timer in each assigned set in order to be liberated when the assignation time expires.

- In the server's case, we must have track of some sort of information of the clients in order to know if a certain client that wants a renewal is been offered and assigned a set within the server's distributable addresses previously.

To cover all the requirements above, we have chosen a tree data structure, concretely a 2-3 tree[25]. This type of trees are characterized by the interesting fact that every children can either have two children nodes and one data element or three children nodes and two data elements. It is actually a specific form of a B tree[26]. For a more familiar view and understanding, Figure 4.2 represents an insertion example of a 2-3 tree.

Going back to the draft implementation, clients will have one of this tree database, whereas servers will need four of them, one for each address type, those are 48-bit unicast, 48-bit multicast, 64-bit unicast and 64-bit multicast. The tree starts with a single root node which contains an address block with all the assignable addresses. In the case of the clients, this block would be the self-assignable block and in the case of the servers it would be the one the administrator has given to them. As addresses are assigned, that block is split, resulting into several nodes in the database. Each of these nodes will have the first address of the containing set and the size of this one. Moreover, to manage the free blocks, we have used a doubly linked list[27]. Every free block in the database is enlisted in this linked list, therefore the search of a free set is faster.

The 2-3 tree self balances perfectly any time a new assignment is realized, or in other words, when new nodes appear, always keeping the same depth for every last node. Consequently to this property, operations of insert, delete and search have a time complex of $O(log(n))$. Furthermore, the doubly linked list stays optimized and disfragmented because free consecutive blocks are spliced every time.

In conclusion, we have chosen this data structure due to its complex but very efficient behavior. This optimization is also on account of the property this structure has that is called self-organized criticality[28].

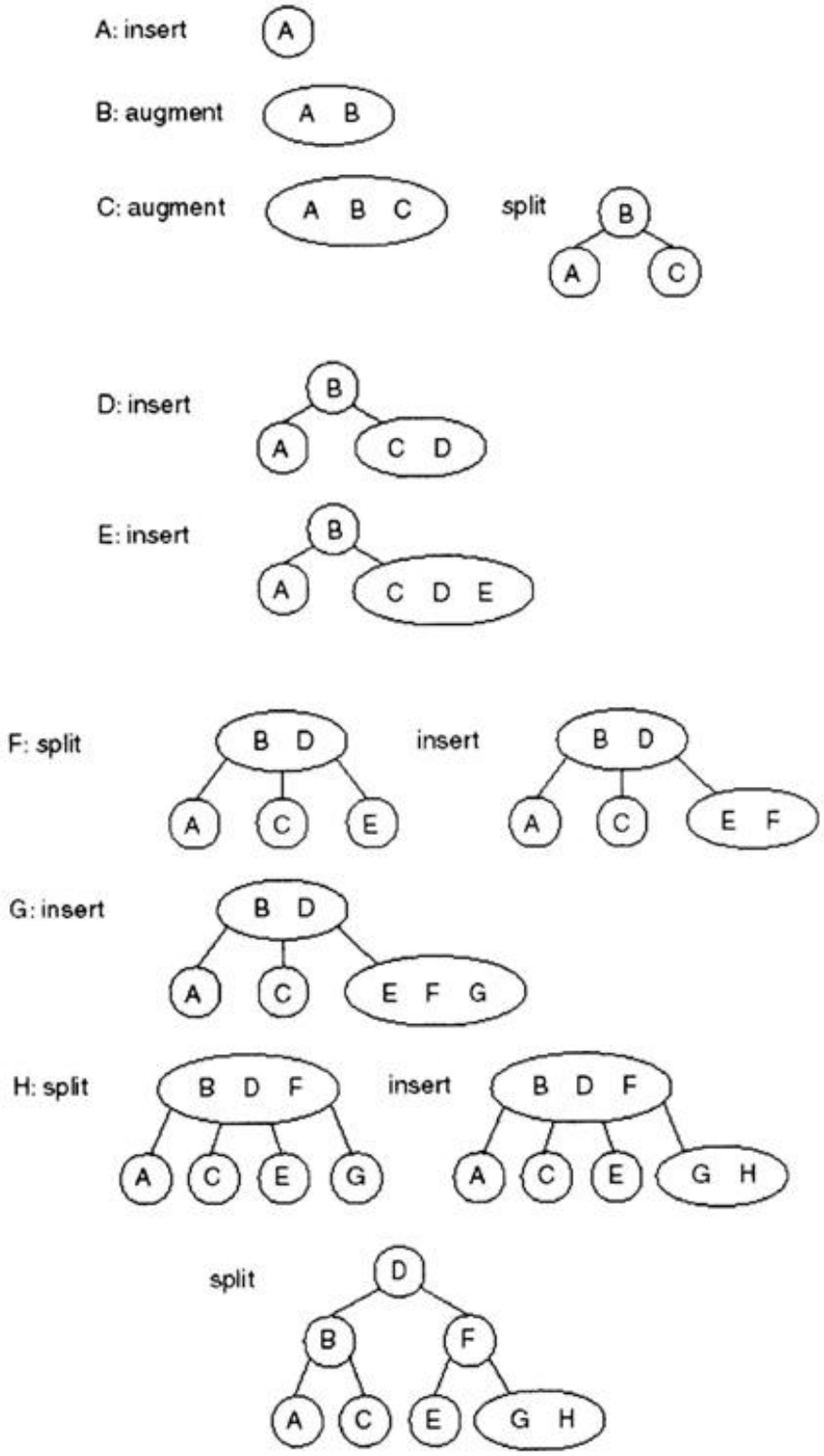


Figure 4.2: Example of 2-3 tree insertion[4]

4.6 Customization

In Chapter 3 we mentioned that all the unknown behaviors or not defined ones were gathered in a configuration file which is going to be checked every time a client or a server in the protocol is launched.

In order to achieve this functionality, we have encoded a function that reads XML files searching the relevant options and interprets them, giving the user executing a particular PALMA client or server, the possibility to manage these behavior options. In listings 4.1 and 4.2 we can find the configuration parameters for a client and a server respectively. Our XML interpreter understands comments and exits the program if a compulsory option is not defined or its value is not correct.

```
<?xml version="1.0" encoding="UTF-8"?>
<ClientConfig>
  <!--InterfaceName id="h1-eth0" /-->
  <!--PreassignedSrcAddress addr="0x01FACE" /-->
  <!--KnownServerAddress addr="0x100ABCDEF001" /-->
  <!--ClaimAddressSet addr="0x0A0000000000" mask="0xff0000000000" /-->
  <MinAssignedAddresses size="1" />
  <MaxAssignedAddresses size="150" />
  <RenewalActive value="true" />
  <RandomAutoAssign value="true" />
  <Verbose value="true" />
  <!--ClientStationId id="" /-->
  <!--VendorParameter id="" /-->
</ClientConfig>
```

Listing 4.1: PALMA Client XML Configuration file

```
<?xml version="1.0" encoding="UTF-8"?>
<ServerConfig>
  <!--InterfaceName id="" /-->
  <SrcAddress addr="0x100ABCDEF001" />
  <UnicastAddressSet addr="0x1ACA00000000" count="100000" />
  <MulticastAddressSet addr="0x1BCB00000000" count="500000" />
  <Unicast64AddressSet addr="0x1ACA000000000000" count="98000" />
  <Multicast64AddressSet addr="0x1BCB000000000000" count="495000" />
  <MaxAssignedUnicast size="1000" />
  <MaxAssignedMulticast size="5000" />
  <MaxAssignedUnicast64 size="1000" />
  <MaxAssignedMulticast64 size="5000" />
  <MaxAssignedDefault size="100" />

  <UnicastLifetime value="60" />
```

```
    <MulticastLifetime value="60" />
    <Unicast64Lifetime value="60" />
    <Multicast64Lifetime value="60" />
    <ReserveLifetime value="2" />

    <RenewalActive value="true" />
    <DefaultMulticast value="false" />
    <Default64bitSet value="false" />
    <AutoassignObjectionActive value="true" />
    <AlternateSetActive value="true" />

    <NetworkId id="SERVER" />
    <VendorParameter id="NOKIA" />
  </ServerConfig>
```

Listing 4.2: PALMA Server XML Configuration file

Some of these parameters are not required to be filled because there are some predefined values for them (for example renewal option or random self-assign among others) or there are not really necessary for the particular client to work properly (the case of Vendor parameter for example), others such as the interface name or the station id can be specified by command line, and that is why it could be not defined in the configuration file 4.1 despite being an obligatory option.

Chapter 5

Tests and Results

5.1 Overview

To guarantee the proper functioning, as in every code implementation project, a set of tests must be made. The results out of these tests have to fulfill all the existent requirements of the draft in which this implementation was based. In this case, as explained in chapter 3, the software behavior must match not the PALMA draft precisely, but this one taking on account the numerous assumptions and clarification we have done. For this to happen, we have elaborated a test validation plan, including as many test cases as different behaviors throughout the states in the code.

Furthermore, a set of performance tests have been developed to achieve a deeper study of the protocol itself, focusing mainly on scalability and MAC assignment convergence time.

5.2 Test Validation Plan

As mentioned before, a test validation plan is needed to make sure that the software works correctly and according to the specification. This one is extensive, and that is why many test cases are designed in order to cover every single functional requirement. This plan consists of a python script (listing 5.1) in which a Mininet custom topology[29] is created and modified based on the wanted number of hosts. This script builds a tree form structure[30] with the total hosts and the switches used to interconnect them. This means that the

switches depend on the number of hosts wanted. More precisely, the number of ports in each switch can be given as a parameter when executing the script, being 64 as default.

In every host of this Mininet topology a PALMA protocol client is going to be executed and in some cases a PALMA server. The script has been slightly modified to be adapted to each test case, varying for example, the rate at which PALMA clients are launched on each host or in particular cases, making the links fail at some points. For a deeper code analysis the script is shown below, listing 5.1:

```
#!/usr/bin/python

from mininet.net import Mininet
from mininet.node import CPULimitedHost
from mininet.topo import Topo
from mininet.log import setLogLevel, info
from mininet.util import custom, pmonitor
from mininet.node import OVSController
from mininet.nodelib import LinuxBridge
from mininet.cli import CLI
from datetime import datetime, timedelta
import random
import signal
import sys
import math
import time

class TimerList:
  def __init__(self):
    self.list = []
    self.ref = datetime.now()

  def refresh(self):
    new_ref = datetime.now()
    if(len(self.list) != 0):
      self.list[0][0] -= (new_ref - self.ref)
    self.ref = new_ref

  def insert(self, time, obj, func):
    self.refresh()
    idx = 0
    t = timedelta(seconds=time)
    for idx in range(len(self.list)):
      if(t < self.list[idx][0]):
        self.list[idx][0] -= t
        self.list.insert(idx,[t, obj, func])
        idx = None
        break
      t -= self.list[idx][0]
```

```
40      if(idx != None):
41        self.list.append([t, obj, func])
42
43    def check(self):
44      self.refresh()
45      res = 1
46      while(len(self.list)):
47        elem = self.list[0]
48        if(elem[0] <= timedelta()):
49          self.list.pop(0)
50          elem[2](elem[1],elem[0])
51        else:
52          res = self.list[0][0].total_seconds()/timedelta(microseconds=1000)
      .total_seconds()
53          break
54      #print("check : %f\n" % (self.list[0][0].total_seconds()/timedelta(
      microseconds=1000).total_seconds()))
55      #print(self.list)
56      return res
57
58  class Test:
59    def __init__(self, n_clients, n_servers=1, n_ports = 64, lambda_in=10,
      lambda_out=1./10, \
60            test_time=-1, output=None):
61      self.popens = {}
62      self.timer_list = TimerList()
63      self.servers = []
64      self.idle_clients = []
65      self.finalize = False
66      self.lambda_in = lambda_in
67      self.lambda_out = lambda_out
68      self.i = 0
69
70      def signal_handler(sig, frame):
71        self.finalize = True
72
73      signal.signal(signal.SIGINT, signal_handler)
74
75      test_topo = Topo()
76
77      def genTree(childs,level):
78        switches = []
79        n_switches = 1
80        n_childs = len(childs)
81        if(n_childs > n_ports):
82          n_switches = int(math.ceil(n_childs/(n_ports-1.)))
83        for s in range(1, n_switches+1):
84          switches.append(test_topo.addSwitch( 's%d_%s' % (level, s)))
85        while(len(childs)):
86          for s in switches:
87            if(len(childs) != 0):
```

```
            test_topo.addLink(childs.pop(0), s)
          else:
            break;
      if(n_switches > 2):
        return(genTree(switches,level+1))
      if(n_switches == 2):
        test_topo.addLink( switches[0], switches[1])
      return

    srvs = []
    clnts = []
    for h in range( 1, n_servers+1 ):
      srvs.append(test_topo.addHost( 'srv%s' % h ))
    for h in range( 1, n_clients+1 ):
      clnts.append(test_topo.addHost( 'h%s' % h ))

    genTree(clnts+srvs, 0)

    self.net = Mininet(topo=test_topo, host=CPULimitedHost, \
               switch=LinuxBridge)

    self.net.start()
    self.net['s0_1'].cmd("sudo tshark -w TC-32.pcap -i s0_1-eth1 -f 'ether
    proto 0x33ff'&")
    self.net['s0_1'].cmd("sudo ifconfig s0_1-eth2 down")
    time.sleep(4)
    for h in range( 1, n_servers+1 ):
      self.servers.append(self.net['srv%s' % h])
    for h in range( 1, n_clients+1 ):
      client = self.net['h%s' % h]
      client.setCPUFrac(.5/n_clients)
      self.idle_clients.append(client)

    self.timer_list.insert(random.expovariate(lambda_in), None, self.
   arrive)
    if(test_time > 0):
      self.timer_list.insert(test_time, None, self.finish)
    if(output != None):
      self.file = open(output,"w")
    else:
      self.file = sys.stdout
    self.file.write("HOST,TIME,CMD,ADDR,COUNT\n")

  def run(self):
    for server in self.servers:
      self.popens[server] = server.popen(\
        "sudo ../server/palma-server -c %s.xml -i %s-eth0" \
          % (server.name, server.name))
    time.sleep(1)
    while(not self.finalize):
      try:
```

```
137         host, line = pmonitor(self.popens, timeoutms=self.timer_list.check
    ()).next()
138         if(host):
139           self.file.write("%s,%s" % (host, line))
140       except StopIteration:
141         pass
142 
143     self.file.close()
144     self.net.stop()
145   '''
146   def run(self):
147     b = True
148     while(not self.finalize):
149       try:
150         host, line = pmonitor(self.popens, timeoutms=self.timer_list.check
    ()).next()
151         if(host):
152           self.file.write("%s,%s" % (host, line))
153         if(b == True):
154           for server in self.servers:
155             time.sleep(2)
156             self.popens[server] = server.popen(\
157               "sudo ../server/palma-server -c %s.xml -i %s-eth0" \
158                 % (server.name, server.name))
159           b = False
160       except StopIteration:
161         pass
162 
163     self.file.close()
164     self.net.stop()
165   '''
166   def arrive(self, obj, t):
167     if(len(self.idle_clients) > 0):
168       new_client = self.idle_clients.pop(0)
169       '''
170       self.popens[new_client] = new_client.popen(\
171         "sudo ../client/palma-client -c client.xml -i %s-eth0 -s %s"\
172           % (new_client.name, new_client.name.upper()))
173       '''
174       self.i += 1
175       self.popens[new_client] = new_client.popen(\
176         "sudo ../client/palma-client -c client%d.xml -i %s-eth0 -s %s"\
177           % (self.i, new_client.name, new_client.name.upper()))
178       '''
179       self.popens[new_client] = new_client.popen(\
180         "sudo ../client/palma-client -c client.xml -i %s-eth0 -s %s -p 0
    x100FACE0000%d"\
181           % (new_client.name, new_client.name.upper(), self.i))
182       '''
183       if(self.lambda_out != 0):
184         self.timer_list.insert(random.expovariate(self.lambda_out),
```

```
    new_client, self.departure)
185
186     if(len(self.idle_clients) > 0):
187       self.timer_list.insert(random.expovariate(self.lambda_in), None,
    self.arrive)
188     else:
189       time.sleep(5)
190       self.reconnect()
191
192   def departure(self, client, t):
193     #if(len(self.idle_clients) == 0):
194       #self.timer_list.insert(random.expovariate(self.lambda_in), None,
    self.arrive)
195     self.popens[client].send_signal(signal.SIGTERM)
196     #self.idle_clients.append(client)
197
198
199   def finish(self, obj, t):
200     self.finalize = True
201
202   def disconnect(self):
203     self.net['s0_1'].cmd("sudo ifconfig s0_1-eth2 down")
204
205   def reconnect(self):
206     self.net['s0_1'].cmd("sudo ifconfig s0_1-eth2 up")
207
208
209 if __name__ == '__main__':
210   setLogLevel( 'info' )
211   Test(2, 0, test_time=120, lambda_in=1./2, lambda_out=0).run()
```

Listing 5.1: Python script for the Validation Plan Tests

Without further delay, test cases forming the test validation plan are presented below. Each one has a brief description of the executed client or server options and its own table where the expected behavior and final result are explained, as well as a Wireshark[31] capture, showing the network packets exchanged, which serves as a proof. In addition, for the best visualization of the packets over the network, we have coded a Wireshark dissector in C[32] language.

Test Case 1

One single client is executed claiming the whole self-assign unicast set without a preassigned source address. This client has been configured with minimum number of addresses to assign

equal to 1 and maximum equal to 100. The test duration is around 640 seconds. Results can be seen in table 5.1 and figure 5.1.

Table 5.1: Validation Plan - Test Case 1

Test Case:	TC - 01
Expected behavior:	Client is expected to issue three DISCOVER messages, with PALMA random source address according to the draft because it lacks of one. After this, it is supposed to self-assign the maximum number of unicast addresses specified, that is 16, and issue ANNOUNCES every 30 seconds for a total of 600 seconds. Then, it should start the protocol again.
Real behavior:	As expected
Result:	Test passed

	Source	Destination	Protocol	Length	Time	Info
1	2a:00:20:9f:50:d2	PALMA-MULTICAST	PALMA	40	0.000000000	DISCOVER: SET:[start=0a:00:00:00:00:00 mask=ff:00:00:00:00:00]
2	2a:00:70:b8:b7:6a	PALMA-MULTICAST	PALMA	40	0.525907514	DISCOVER: SET:[start=0a:00:00:00:00:00 mask=ff:00:00:00:00:00]
3	2a:00:8a:9c:b9:30	PALMA-MULTICAST	PALMA	40	1.038969182	DISCOVER: SET:[start=0a:00:00:00:00:00 mask=ff:00:00:00:00:00]
4	0a:00:00:00:00:06	PALMA-MULTICAST	PALMA	40	1.599058934	ANNOUNCE: SET:[start=0a:00:00:00:00:00, cnt=16] lifetime=600 s
5	0a:00:00:00:00:06	PALMA-MULTICAST	PALMA	40	32.435124341	ANNOUNCE: SET:[start=0a:00:00:00:00:00, cnt=16] lifetime=569 s
6	0a:00:00:00:00:06	PALMA-MULTICAST	PALMA	40	64.229172756	ANNOUNCE: SET:[start=0a:00:00:00:00:00, cnt=16] lifetime=537 s
7	0a:00:00:00:00:06	PALMA-MULTICAST	PALMA	40	94.999567946	ANNOUNCE: SET:[start=0a:00:00:00:00:00, cnt=16] lifetime=507 s
8	0a:00:00:00:00:06	PALMA-MULTICAST	PALMA	40	125.920008672	ANNOUNCE: SET:[start=0a:00:00:00:00:00, cnt=16] lifetime=476 s
9	0a:00:00:00:00:06	PALMA-MULTICAST	PALMA	40	156.104800766	ANNOUNCE: SET:[start=0a:00:00:00:00:00, cnt=16] lifetime=445 s
10	0a:00:00:00:00:06	PALMA-MULTICAST	PALMA	40	186.448174628	ANNOUNCE: SET:[start=0a:00:00:00:00:00, cnt=16] lifetime=415 s
11	0a:00:00:00:00:06	PALMA-MULTICAST	PALMA	40	216.499974627	ANNOUNCE: SET:[start=0a:00:00:00:00:00, cnt=16] lifetime=385 s
12	0a:00:00:00:00:06	PALMA-MULTICAST	PALMA	40	248.228001536	ANNOUNCE: SET:[start=0a:00:00:00:00:00, cnt=16] lifetime=353 s
13	0a:00:00:00:00:06	PALMA-MULTICAST	PALMA	40	278.840381232	ANNOUNCE: SET:[start=0a:00:00:00:00:00, cnt=16] lifetime=323 s
14	0a:00:00:00:00:06	PALMA-MULTICAST	PALMA	40	310.840446004	ANNOUNCE: SET:[start=0a:00:00:00:00:00, cnt=16] lifetime=291 s
15	0a:00:00:00:00:06	PALMA-MULTICAST	PALMA	40	341.000783657	ANNOUNCE: SET:[start=0a:00:00:00:00:00, cnt=16] lifetime=261 s
16	0a:00:00:00:00:06	PALMA-MULTICAST	PALMA	40	372.412265727	ANNOUNCE: SET:[start=0a:00:00:00:00:00, cnt=16] lifetime=229 s
17	0a:00:00:00:00:06	PALMA-MULTICAST	PALMA	40	402.900151097	ANNOUNCE: SET:[start=0a:00:00:00:00:00, cnt=16] lifetime=199 s
18	0a:00:00:00:00:06	PALMA-MULTICAST	PALMA	40	432.932159437	ANNOUNCE: SET:[start=0a:00:00:00:00:00, cnt=16] lifetime=169 s
19	0a:00:00:00:00:06	PALMA-MULTICAST	PALMA	40	464.564038542	ANNOUNCE: SET:[start=0a:00:00:00:00:00, cnt=16] lifetime=137 s
20	0a:00:00:00:00:06	PALMA-MULTICAST	PALMA	40	496.224093135	ANNOUNCE: SET:[start=0a:00:00:00:00:00, cnt=16] lifetime=105 s
21	0a:00:00:00:00:06	PALMA-MULTICAST	PALMA	40	527.563946907	ANNOUNCE: SET:[start=0a:00:00:00:00:00, cnt=16] lifetime=74 s
22	0a:00:00:00:00:06	PALMA-MULTICAST	PALMA	40	559.428479064	ANNOUNCE: SET:[start=0a:00:00:00:00:00, cnt=16] lifetime=42 s
23	0a:00:00:00:00:06	PALMA-MULTICAST	PALMA	40	589.828844869	ANNOUNCE: SET:[start=0a:00:00:00:00:00, cnt=16] lifetime=12 s
24	2a:00:44:b1:59:03	PALMA-MULTICAST	PALMA	40	601.608861592	DISCOVER: SET:[start=0a:00:00:00:00:00 mask=ff:00:00:00:00:00]
25	2a:00:9d:8b:4d:73	PALMA-MULTICAST	PALMA	40	602.172431230	DISCOVER: SET:[start=0a:00:00:00:00:00 mask=ff:00:00:00:00:00]
26	2a:00:a1:a6:3c:f8	PALMA-MULTICAST	PALMA	40	602.763510283	DISCOVER: SET:[start=0a:00:00:00:00:00 mask=ff:00:00:00:00:00]
27	0a:00:00:00:00:0b	PALMA-MULTICAST	PALMA	40	603.315067088	ANNOUNCE: SET:[start=0a:00:00:00:00:00, cnt=16] lifetime=600 s
28	0a:00:00:00:00:0b	PALMA-MULTICAST	PALMA	40	634.372518559	ANNOUNCE: SET:[start=0a:00:00:00:00:00, cnt=16] lifetime=569 s
29	0a:00:00:00:00:0b	PALMA-MULTICAST	PALMA	40	664.416836508	ANNOUNCE: SET:[start=0a:00:00:00:00:00, cnt=16] lifetime=539 s
30	0a:00:00:00:00:0b	PALMA-MULTICAST	PALMA	40	694.460129030	ANNOUNCE: SET:[start=0a:00:00:00:00:00, cnt=16] lifetime=509 s

Figure 5.1: Wireshark - Test Case 1

Test Case 2

One single client claiming the whole self-assign multicast set without a preassigned source address is executed. This client has been configured with minimum number of addresses to assign equal to 1 and maximum equal to 100. The test duration is around 30 seconds. Results can be seen in table 5.2 and figure 5.2.

Table 5.2: Validation Plan - Test Case 2

Test Case:	TC - 02
Expected behavior:	Client is expected to issue DISCOVER messages continuously, with PALMA random source address according to the draft because it lacks of one. It will not self assign any multicast address because it lacks of a preassigned unicast source address.
Real behavior:	As expected
Result:	Test passed

	Source	Destination	Protocol	Length	Time	Info
1	2a:00:8d:e9:d2:3a	PALMA-MULTICAST	PALMA	40	0.000000000	DISCOVER: SET:[start=0b:00:00:00:00:00 mask=ff:00:00:00:00:00]
2	2a:00:5a:7f:11:3e	PALMA-MULTICAST	PALMA	40	0.599950597	DISCOVER: SET:[start=0b:00:00:00:00:00 mask=ff:00:00:00:00:00]
3	2a:00:79:dd:c7:60	PALMA-MULTICAST	PALMA	40	1.198532052	DISCOVER: SET:[start=0b:00:00:00:00:00 mask=ff:00:00:00:00:00]
4	2a:00:36:30:4e:26	PALMA-MULTICAST	PALMA	40	1.751634308	DISCOVER: SET:[start=0b:00:00:00:00:00 mask=ff:00:00:00:00:00]
5	2a:00:6c:56:91:84	PALMA-MULTICAST	PALMA	40	2.325177960	DISCOVER: SET:[start=0b:00:00:00:00:00 mask=ff:00:00:00:00:00]
6	2a:00:54:3a:5b:8f	PALMA-MULTICAST	PALMA	40	2.838476438	DISCOVER: SET:[start=0b:00:00:00:00:00 mask=ff:00:00:00:00:00]
7	2a:00:9b:74:ef:ad	PALMA-MULTICAST	PALMA	40	3.346300996	DISCOVER: SET:[start=0b:00:00:00:00:00 mask=ff:00:00:00:00:00]
8	2a:00:aa:68:89:da	PALMA-MULTICAST	PALMA	40	3.851607709	DISCOVER: SET:[start=0b:00:00:00:00:00 mask=ff:00:00:00:00:00]
9	2a:00:0d:51:ac:e7	PALMA-MULTICAST	PALMA	40	4.432396431	DISCOVER: SET:[start=0b:00:00:00:00:00 mask=ff:00:00:00:00:00]
10	2a:00:bd:04:6f:27	PALMA-MULTICAST	PALMA	40	4.966208325	DISCOVER: SET:[start=0b:00:00:00:00:00 mask=ff:00:00:00:00:00]
11	2a:00:a1:e3:3e:71	PALMA-MULTICAST	PALMA	40	5.500999035	DISCOVER: SET:[start=0b:00:00:00:00:00 mask=ff:00:00:00:00:00]
12	2a:00:63:c2:13:2e	PALMA-MULTICAST	PALMA	40	6.068300969	DISCOVER: SET:[start=0b:00:00:00:00:00 mask=ff:00:00:00:00:00]
13	2a:00:98:d5:7d:78	PALMA-MULTICAST	PALMA	40	6.569270033	DISCOVER: SET:[start=0b:00:00:00:00:00 mask=ff:00:00:00:00:00]
14	2a:00:cc:32:ed:7a	PALMA-MULTICAST	PALMA	40	7.147323144	DISCOVER: SET:[start=0b:00:00:00:00:00 mask=ff:00:00:00:00:00]
15	2a:00:70:52:38:1b	PALMA-MULTICAST	PALMA	40	7.685892704	DISCOVER: SET:[start=0b:00:00:00:00:00 mask=ff:00:00:00:00:00]
16	2a:00:7f:6f:9a:92	PALMA-MULTICAST	PALMA	40	8.199465097	DISCOVER: SET:[start=0b:00:00:00:00:00 mask=ff:00:00:00:00:00]
17	2a:00:c1:06:b7:41	PALMA-MULTICAST	PALMA	40	8.777580407	DISCOVER: SET:[start=0b:00:00:00:00:00 mask=ff:00:00:00:00:00]
18	2a:00:6a:b5:49:16	PALMA-MULTICAST	PALMA	40	9.293149517	DISCOVER: SET:[start=0b:00:00:00:00:00 mask=ff:00:00:00:00:00]
19	2a:00:30:7a:aa:9c	PALMA-MULTICAST	PALMA	40	9.852229590	DISCOVER: SET:[start=0b:00:00:00:00:00 mask=ff:00:00:00:00:00]
20	2a:00:61:f6:16:cb	PALMA-MULTICAST	PALMA	40	10.378797145	DISCOVER: SET:[start=0b:00:00:00:00:00 mask=ff:00:00:00:00:00]
21	2a:00:7a:08:a1:c8	PALMA-MULTICAST	PALMA	40	10.974848948	DISCOVER: SET:[start=0b:00:00:00:00:00 mask=ff:00:00:00:00:00]
22	2a:00:a1:3a:16:43	PALMA-MULTICAST	PALMA	40	11.516965954	DISCOVER: SET:[start=0b:00:00:00:00:00 mask=ff:00:00:00:00:00]
23	2a:00:7f:2c:41:5b	PALMA-MULTICAST	PALMA	40	12.071000336	DISCOVER: SET:[start=0b:00:00:00:00:00 mask=ff:00:00:00:00:00]
24	2a:00:6b:d7:30:72	PALMA-MULTICAST	PALMA	40	12.589066100	DISCOVER: SET:[start=0b:00:00:00:00:00 mask=ff:00:00:00:00:00]
25	2a:00:d7:f4:2c:6d	PALMA-MULTICAST	PALMA	40	13.160162961	DISCOVER: SET:[start=0b:00:00:00:00:00 mask=ff:00:00:00:00:00]
26	2a:00:82:b8:ba:6e	PALMA-MULTICAST	PALMA	40	13.666671678	DISCOVER: SET:[start=0b:00:00:00:00:00 mask=ff:00:00:00:00:00]
27	2a:00:57:d7:16:e2	PALMA-MULTICAST	PALMA	40	14.219756964	DISCOVER: SET:[start=0b:00:00:00:00:00 mask=ff:00:00:00:00:00]
28	2a:00:6a:6b:16:3a	PALMA-MULTICAST	PALMA	40	14.800356973	DISCOVER: SET:[start=0b:00:00:00:00:00 mask=ff:00:00:00:00:00]
29	2a:00:9e:07:12:76	PALMA-MULTICAST	PALMA	40	15.325413863	DISCOVER: SET:[start=0b:00:00:00:00:00 mask=ff:00:00:00:00:00]
30	2a:00:ad:2d:3e:45	PALMA-MULTICAST	PALMA	40	15.882984380	DISCOVER: SET:[start=0b:00:00:00:00:00 mask=ff:00:00:00:00:00]

Figure 5.2: Wireshark - Test Case 2

Test Case 3

One single client claiming the whole self-assign multicast set with a preassigned source address is executed. This client has been configured with minimum number of addresses to assign equal to 1 and maximum equal to 100. The test duration is around 640 seconds. Results can be seen in table 5.3 and figure 5.3.

Table 5.3: Validation Plan - Test Case 3

Test Case:	TC - 03
Expected behavior:	Client is expected to issue three DISCOVER messages, with source address the preassigned unicast specified in the configuration file. After this, it is supposed to self-assign the maximum number of multicast addresses specified, that is 100, and issue ANNOUNCES every 30 seconds for a total of 600 seconds. Then, it should start the protocol again.
Real behavior:	As expected
Result:	Test passed

	Source	Destination	Protocol	Length	Time	Info
1	10:0f:ac:e0:00:01	PALMA-MULTICAST	PALMA	40	0.000000000	DISCOVER: SET:[start=0b:00:00:00:00:00 mask=ff:00:00:00:00:00]
2	10:0f:ac:e0:00:01	PALMA-MULTICAST	PALMA	40	0.515944429	DISCOVER: SET:[start=0b:00:00:00:00:00 mask=ff:00:00:00:00:00]
3	10:0f:ac:e0:00:01	PALMA-MULTICAST	PALMA	40	1.079524804	DISCOVER: SET:[start=0b:00:00:00:00:00 mask=ff:00:00:00:00:00]
4	10:0f:ac:e0:00:01	PALMA-MULTICAST	PALMA	40	1.603622447	ANNOUNCE: SET:[start=0b:00:00:00:00:00, cnt=100] lifetime=600 s
5	10:0f:ac:e0:00:01	PALMA-MULTICAST	PALMA	40	33.406223661	ANNOUNCE: SET:[start=0b:00:00:00:00:00, cnt=100] lifetime=568 s
6	10:0f:ac:e0:00:01	PALMA-MULTICAST	PALMA	40	63.594064051	ANNOUNCE: SET:[start=0b:00:00:00:00:00, cnt=100] lifetime=538 s
7	10:0f:ac:e0:00:01	PALMA-MULTICAST	PALMA	40	94.343468082	ANNOUNCE: SET:[start=0b:00:00:00:00:00, cnt=100] lifetime=507 s
8	10:0f:ac:e0:00:01	PALMA-MULTICAST	PALMA	40	125.389898920	ANNOUNCE: SET:[start=0b:00:00:00:00:00, cnt=100] lifetime=476 s
9	10:0f:ac:e0:00:01	PALMA-MULTICAST	PALMA	40	156.987885213	ANNOUNCE: SET:[start=0b:00:00:00:00:00, cnt=100] lifetime=445 s
10	10:0f:ac:e0:00:01	PALMA-MULTICAST	PALMA	40	187.541277179	ANNOUNCE: SET:[start=0b:00:00:00:00:00, cnt=100] lifetime=414 s
11	10:0f:ac:e0:00:01	PALMA-MULTICAST	PALMA	40	218.559210690	ANNOUNCE: SET:[start=0b:00:00:00:00:00, cnt=100] lifetime=383 s
12	10:0f:ac:e0:00:01	PALMA-MULTICAST	PALMA	40	249.269617980	ANNOUNCE: SET:[start=0b:00:00:00:00:00, cnt=100] lifetime=352 s
13	10:0f:ac:e0:00:01	PALMA-MULTICAST	PALMA	40	280.408579873	ANNOUNCE: SET:[start=0b:00:00:00:00:00, cnt=100] lifetime=321 s
14	10:0f:ac:e0:00:01	PALMA-MULTICAST	PALMA	40	310.509401183	ANNOUNCE: SET:[start=0b:00:00:00:00:00, cnt=100] lifetime=291 s
15	10:0f:ac:e0:00:01	PALMA-MULTICAST	PALMA	40	341.176760431	ANNOUNCE: SET:[start=0b:00:00:00:00:00, cnt=100] lifetime=260 s
16	10:0f:ac:e0:00:01	PALMA-MULTICAST	PALMA	40	371.431147674	ANNOUNCE: SET:[start=0b:00:00:00:00:00, cnt=100] lifetime=230 s
17	10:0f:ac:e0:00:01	PALMA-MULTICAST	PALMA	40	401.800510541	ANNOUNCE: SET:[start=0b:00:00:00:00:00, cnt=100] lifetime=200 s
18	10:0f:ac:e0:00:01	PALMA-MULTICAST	PALMA	40	432.395843737	ANNOUNCE: SET:[start=0b:00:00:00:00:00, cnt=100] lifetime=169 s
19	10:0f:ac:e0:00:01	PALMA-MULTICAST	PALMA	40	463.131266692	ANNOUNCE: SET:[start=0b:00:00:00:00:00, cnt=100] lifetime=138 s
20	10:0f:ac:e0:00:01	PALMA-MULTICAST	PALMA	40	494.974302277	ANNOUNCE: SET:[start=0b:00:00:00:00:00, cnt=100] lifetime=107 s
21	10:0f:ac:e0:00:01	PALMA-MULTICAST	PALMA	40	525.596693958	ANNOUNCE: SET:[start=0b:00:00:00:00:00, cnt=100] lifetime=76 s
22	10:0f:ac:e0:00:01	PALMA-MULTICAST	PALMA	40	557.325694639	ANNOUNCE: SET:[start=0b:00:00:00:00:00, cnt=100] lifetime=44 s
23	10:0f:ac:e0:00:01	PALMA-MULTICAST	PALMA	40	587.885102667	ANNOUNCE: SET:[start=0b:00:00:00:00:00, cnt=100] lifetime=14 s
24	10:0f:ac:e0:00:01	PALMA-MULTICAST	PALMA	40	601.607844726	DISCOVER: SET:[start=0b:00:00:00:00:00 mask=ff:00:00:00:00:00]
25	10:0f:ac:e0:00:01	PALMA-MULTICAST	PALMA	40	602.168901795	DISCOVER: SET:[start=0b:00:00:00:00:00 mask=ff:00:00:00:00:00]
26	10:0f:ac:e0:00:01	PALMA-MULTICAST	PALMA	40	602.691969326	DISCOVER: SET:[start=0b:00:00:00:00:00 mask=ff:00:00:00:00:00]
27	10:0f:ac:e0:00:01	PALMA-MULTICAST	PALMA	40	603.261022883	ANNOUNCE: SET:[start=0b:00:00:00:00:00, cnt=100] lifetime=600 s
28	10:0f:ac:e0:00:01	PALMA-MULTICAST	PALMA	40	633.319863818	ANNOUNCE: SET:[start=0b:00:00:00:00:00, cnt=100] lifetime=570 s
29	10:0f:ac:e0:00:01	PALMA-MULTICAST	PALMA	40	664.878843368	ANNOUNCE: SET:[start=0b:00:00:00:00:00, cnt=100] lifetime=538 s

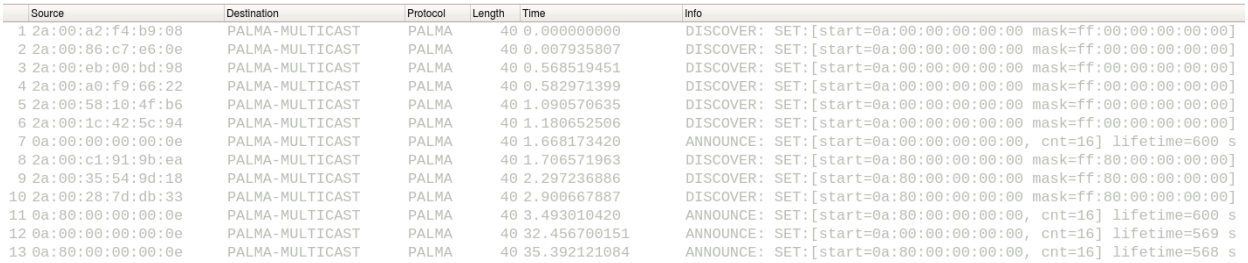
Figure 5.3: Wireshark - Test Case 3

Test Case 4

Two clients claiming the whole self-assign unicast set without a preassigned source address are executed. Both clients have been configured with minimum number of addresses to assign equal to 1 and maximum equal to 100. The test duration is around 60 seconds. Results can be seen in table 5.4 and figure 5.4.

Table 5.4: Validation Plan - Test Case 4

Test Case:	TC - 04
Expected behavior:	Both clients are expected to issue DISCOVER messages of the whole set specified in the configuration file. When any of them self-assign the maximum number of unicast addresses (16) within the discovering set, it will issue an ANNOUNCE and the other client will start the protocol again having registered this assigned set in its database.
Real behavior:	As expected
Result:	Test passed



	Source	Destination	Protocol	Length	Time	Info
1	2a:00:a2:f4:b9:08	PALMA-MULTICAST	PALMA	40	0.000000000	DISCOVER: SET:[start=0a:00:00:00:00:00 mask=ff:00:00:00:00:00]
2	2a:00:86:c7:e6:0e	PALMA-MULTICAST	PALMA	40	0.007935807	DISCOVER: SET:[start=0a:00:00:00:00:00 mask=ff:00:00:00:00:00]
3	2a:00:eb:00:bd:98	PALMA-MULTICAST	PALMA	40	0.568519451	DISCOVER: SET:[start=0a:00:00:00:00:00 mask=ff:00:00:00:00:00]
4	2a:00:a0:f9:66:22	PALMA-MULTICAST	PALMA	40	0.582971399	DISCOVER: SET:[start=0a:00:00:00:00:00 mask=ff:00:00:00:00:00]
5	2a:00:58:10:4f:b6	PALMA-MULTICAST	PALMA	40	1.090570635	DISCOVER: SET:[start=0a:00:00:00:00:00 mask=ff:00:00:00:00:00]
6	2a:00:1c:42:5c:94	PALMA-MULTICAST	PALMA	40	1.180652506	DISCOVER: SET:[start=0a:00:00:00:00:00 mask=ff:00:00:00:00:00]
7	0a:00:00:00:00:0e	PALMA-MULTICAST	PALMA	40	1.668173420	ANNOUNCE: SET:[start=0a:00:00:00:00:00, cnt=16] lifetime=600 s
8	2a:00:c1:91:9b:ea	PALMA-MULTICAST	PALMA	40	1.706571963	DISCOVER: SET:[start=0a:80:00:00:00:00 mask=ff:80:00:00:00:00]
9	2a:00:35:54:9d:18	PALMA-MULTICAST	PALMA	40	2.297236886	DISCOVER: SET:[start=0a:80:00:00:00:00 mask=ff:80:00:00:00:00]
10	2a:00:28:7d:db:33	PALMA-MULTICAST	PALMA	40	2.900667887	DISCOVER: SET:[start=0a:80:00:00:00:00 mask=ff:80:00:00:00:00]
11	0a:80:00:00:00:0e	PALMA-MULTICAST	PALMA	40	3.493010420	ANNOUNCE: SET:[start=0a:80:00:00:00:00, cnt=16] lifetime=600 s
12	0a:00:00:00:00:0e	PALMA-MULTICAST	PALMA	40	32.456700151	ANNOUNCE: SET:[start=0a:00:00:00:00:00, cnt=16] lifetime=569 s
13	0a:80:00:00:00:0e	PALMA-MULTICAST	PALMA	40	35.392121084	ANNOUNCE: SET:[start=0a:80:00:00:00:00, cnt=16] lifetime=568 s

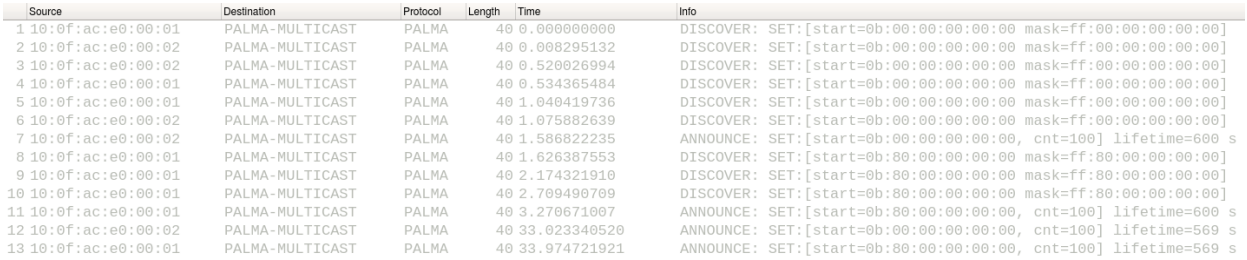
Figure 5.4: Wireshark - Test Case 4

Test Case 5

Two clients claiming the whole self-assign multicast set with a preassigned source address are executed. Both clients have been configured with minimum number of addresses to assign equal to 1 and maximum equal to 100. The test duration is around 60 seconds. Results can be seen in table 5.5 and figure 5.5.

Table 5.5: Validation Plan - Test Case 5

Test Case:	TC - 05
Expected behavior:	Both clients are expected to issue DISCOVER messages of the whole set specified in the configuration file. When any of them self-assigns the maximum number of unicast addresses (100) within the discovering set, it will issue an ANNOUNCE and the other client will start the protocol again having registered this assigned set in its database.
Real behavior:	As expected
Result:	Test passed



	Source	Destination	Protocol	Length	Time	Info
1	10:0f:ac:e0:00:01	PALMA-MULTICAST	PALMA	40	0.000000000	DISCOVER: SET:[start=0b:00:00:00:00:00 mask=ff:00:00:00:00:00]
2	10:0f:ac:e0:00:02	PALMA-MULTICAST	PALMA	40	0.008295132	DISCOVER: SET:[start=0b:00:00:00:00:00 mask=ff:00:00:00:00:00]
3	10:0f:ac:e0:00:02	PALMA-MULTICAST	PALMA	40	0.520026994	DISCOVER: SET:[start=0b:00:00:00:00:00 mask=ff:00:00:00:00:00]
4	10:0f:ac:e0:00:01	PALMA-MULTICAST	PALMA	40	0.534365484	DISCOVER: SET:[start=0b:00:00:00:00:00 mask=ff:00:00:00:00:00]
5	10:0f:ac:e0:00:01	PALMA-MULTICAST	PALMA	40	1.040419736	DISCOVER: SET:[start=0b:00:00:00:00:00 mask=ff:00:00:00:00:00]
6	10:0f:ac:e0:00:02	PALMA-MULTICAST	PALMA	40	1.075882639	DISCOVER: SET:[start=0b:00:00:00:00:00 mask=ff:00:00:00:00:00]
7	10:0f:ac:e0:00:02	PALMA-MULTICAST	PALMA	40	1.586822235	ANNOUNCE: SET:[start=0b:00:00:00:00:00, cnt=100] lifetime=600 s
8	10:0f:ac:e0:00:01	PALMA-MULTICAST	PALMA	40	1.626387553	DISCOVER: SET:[start=0b:80:00:00:00:00 mask=ff:80:00:00:00:00]
9	10:0f:ac:e0:00:01	PALMA-MULTICAST	PALMA	40	2.174321910	DISCOVER: SET:[start=0b:80:00:00:00:00 mask=ff:80:00:00:00:00]
10	10:0f:ac:e0:00:01	PALMA-MULTICAST	PALMA	40	2.709490709	DISCOVER: SET:[start=0b:80:00:00:00:00 mask=ff:80:00:00:00:00]
11	10:0f:ac:e0:00:01	PALMA-MULTICAST	PALMA	40	3.270671007	ANNOUNCE: SET:[start=0b:80:00:00:00:00, cnt=100] lifetime=600 s
12	10:0f:ac:e0:00:02	PALMA-MULTICAST	PALMA	40	33.023340520	ANNOUNCE: SET:[start=0b:00:00:00:00:00, cnt=100] lifetime=569 s
13	10:0f:ac:e0:00:01	PALMA-MULTICAST	PALMA	40	33.974721921	ANNOUNCE: SET:[start=0b:80:00:00:00:00, cnt=100] lifetime=569 s

Figure 5.5: Wireshark - Test Case 5

Test Case 6

Two clients claiming the whole self-assign unicast set without a preassigned source address are executed. Both clients have been configured with minimum number of addresses to assign equal to 1 and maximum equal to 100. In this case, first we have launched a client and seconds after the other one. The test duration is around 60 seconds. Results can be seen in table 5.6 and figure 5.6.

Table 5.6: Validation Plan - Test Case 6

Test Case:	TC - 06
Expected behavior:	The first client to be executed has self-assigned 16 unicast addresses before the other client starts the protocol. The second client is expected to issue a DISCOVER message and consequently receive a DEFEND message from the first client, specifying the conflicted set. The second client will register this conflicted set in its database and start the protocol again.
Real behavior:	As expected
Result:	Test passed

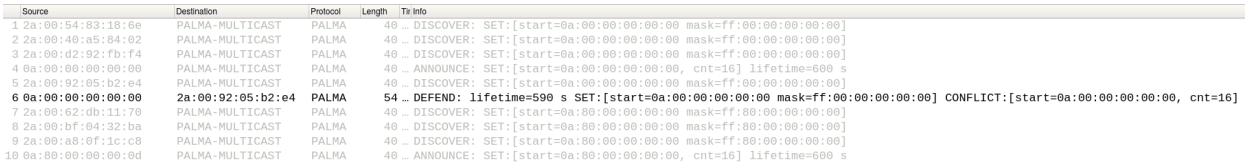


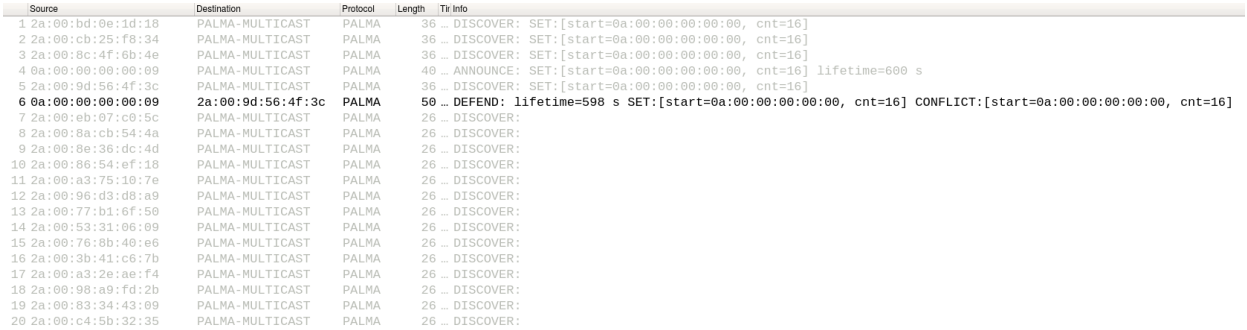
Figure 5.6: Wireshark - Test Case 6

Test Case 7

Two clients claiming the same self-assign unicast set of only 16 addresses without a pre-assigned source address are executed. Both clients have been configured with minimum number of addresses to assign equal to 1 and maximum equal to 16. In this case, first we have launched a client and seconds after the other one. The test duration is around 30 seconds. Results can be seen in table 5.7 and figure 5.7.

Table 5.7: Validation Plan - Test Case 7

Test Case:	TC - 07
Expected behavior:	The first client to be executed has self-assigned 16 unicast addresses before the other client starts the protocol. The second client is expected to issue a DISCOVER message and consequently receive a DEFEND message from the first client, specifying the conflicted set. The second client will send null DISCOVER messages continuously as there are no possible addresses to self-assign left in the specified claiming set of the configuration file, because all of them were self-assigned by the first client.
Real behavior:	As expected
Result:	Test passed



	Source	Destination	Protocol	Length	Tir	Info
1	2a:00:bd:0e:1d:18	PALMA-MULTICAST	PALMA	36	…	DISCOVER: SET:[start=0a:00:00:00:00:00, cnt=16]
2	2a:00:cb:25:f8:34	PALMA-MULTICAST	PALMA	36	…	DISCOVER: SET:[start=0a:00:00:00:00:00, cnt=16]
3	2a:00:8c:4f:6b:4e	PALMA-MULTICAST	PALMA	36	…	DISCOVER: SET:[start=0a:00:00:00:00:00, cnt=16]
4	0a:00:00:00:00:09	PALMA-MULTICAST	PALMA	40	…	ANNOUNCE: SET:[start=0a:00:00:00:00:00, cnt=16] lifetime=600 s
5	2a:00:9d:56:4f:3c	PALMA-MULTICAST	PALMA	36	…	DISCOVER: SET:[start=0a:00:00:00:00:00, cnt=16]
6	0a:00:00:00:00:09	2a:00:9d:56:4f:3c	PALMA	50	…	DEFEND: lifetime=598 s SET:[start=0a:00:00:00:00:00, cnt=16] CONFLICT:[start=0a:00:00:00:00:00, cnt=16]
7	2a:00:eb:07:c0:5c	PALMA-MULTICAST	PALMA	26	…	DISCOVER:
8	2a:00:8a:cb:54:4a	PALMA-MULTICAST	PALMA	26	…	DISCOVER:
9	2a:00:8e:36:dc:4d	PALMA-MULTICAST	PALMA	26	…	DISCOVER:
10	2a:00:86:54:ef:18	PALMA-MULTICAST	PALMA	26	…	DISCOVER:
11	2a:00:a3:75:10:7e	PALMA-MULTICAST	PALMA	26	…	DISCOVER:
12	2a:00:96:d3:d8:a9	PALMA-MULTICAST	PALMA	26	…	DISCOVER:
13	2a:00:77:b1:6f:50	PALMA-MULTICAST	PALMA	26	…	DISCOVER:
14	2a:00:53:31:06:09	PALMA-MULTICAST	PALMA	26	…	DISCOVER:
15	2a:00:76:8b:40:e6	PALMA-MULTICAST	PALMA	26	…	DISCOVER:
16	2a:00:3b:41:c6:7b	PALMA-MULTICAST	PALMA	26	…	DISCOVER:
17	2a:00:a3:2e:ae:f4	PALMA-MULTICAST	PALMA	26	…	DISCOVER:
18	2a:00:98:a9:fd:2b	PALMA-MULTICAST	PALMA	26	…	DISCOVER:
19	2a:00:83:34:43:09	PALMA-MULTICAST	PALMA	26	…	DISCOVER:
20	2a:00:c4:5b:32:35	PALMA-MULTICAST	PALMA	26	…	DISCOVER:

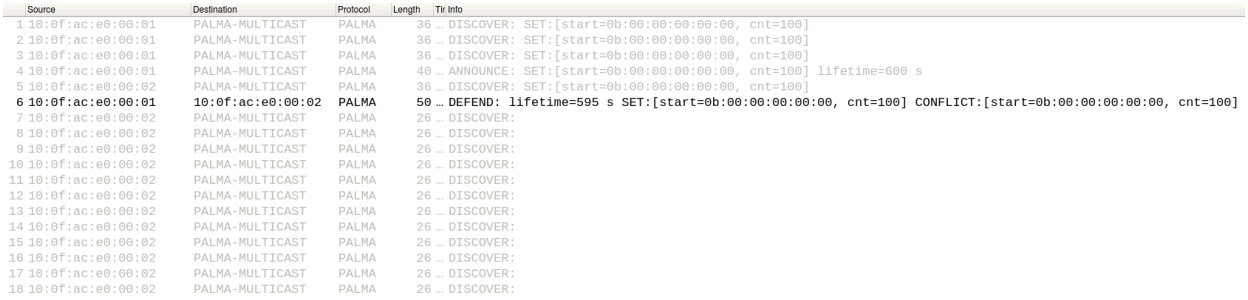
Figure 5.7: Wireshark - Test Case 7

Test Case 8

Two clients claiming the same self-assign multicast set of only 100 addresses with a pre-assigned source address are executed. Both clients have been configured with minimum number of addresses to assign equal to 1 and maximum equal to 100. In this case, first we have launched a client and seconds after the other one. The test duration is around 30 seconds. Results can be seen in table 5.8 and figure 5.8.

Table 5.8: Validation Plan - Test Case 8

Test Case:	TC - 08
Expected behavior:	The first client to be executed has self-assigned 100 multicast addresses before the other client starts the protocol. The second client is expected to issue a DISCOVER message and receive a DEFEND message from the first client, specifying the conflicted set. The second client will send null DISCOVER messages continuously as there are no possible addresses to self-assign left in the specified claiming set of the configuration file, because all of them were self-assigned by the first client.
Real behavior:	As expected
Result:	Test passed



	Source	Destination	Protocol	Length	Tir Info
1	10:0f:ac:e0:00:01	PALMA-MULTICAST	PALMA	36 …	DISCOVER: SET:[start=0b:00:00:00:00:00, cnt=100]
2	10:0f:ac:e0:00:01	PALMA-MULTICAST	PALMA	36 …	DISCOVER: SET:[start=0b:00:00:00:00:00, cnt=100]
3	10:0f:ac:e0:00:01	PALMA-MULTICAST	PALMA	36 …	DISCOVER: SET:[start=0b:00:00:00:00:00, cnt=100]
4	10:0f:ac:e0:00:01	PALMA-MULTICAST	PALMA	40 …	ANNOUNCE: SET:[start=0b:00:00:00:00:00, cnt=100] lifetime=600 s
5	10:0f:ac:e0:00:02	PALMA-MULTICAST	PALMA	36 …	DISCOVER: SET:[start=0b:00:00:00:00:00, cnt=100]
6	10:0f:ac:e0:00:01	10:0f:ac:e0:00:02	PALMA	50 …	DEFEND: lifetime=595 s SET:[start=0b:00:00:00:00:00, cnt=100] CONFLICT:[start=0b:00:00:00:00:00, cnt=100]
7	10:0f:ac:e0:00:02	PALMA-MULTICAST	PALMA	26 …	DISCOVER:
8	10:0f:ac:e0:00:02	PALMA-MULTICAST	PALMA	26 …	DISCOVER:
9	10:0f:ac:e0:00:02	PALMA-MULTICAST	PALMA	26 …	DISCOVER:
10	10:0f:ac:e0:00:02	PALMA-MULTICAST	PALMA	26 …	DISCOVER:
11	10:0f:ac:e0:00:02	PALMA-MULTICAST	PALMA	26 …	DISCOVER:
12	10:0f:ac:e0:00:02	PALMA-MULTICAST	PALMA	26 …	DISCOVER:
13	10:0f:ac:e0:00:02	PALMA-MULTICAST	PALMA	26 …	DISCOVER:
14	10:0f:ac:e0:00:02	PALMA-MULTICAST	PALMA	26 …	DISCOVER:
15	10:0f:ac:e0:00:02	PALMA-MULTICAST	PALMA	26 …	DISCOVER:
16	10:0f:ac:e0:00:02	PALMA-MULTICAST	PALMA	26 …	DISCOVER:
17	10:0f:ac:e0:00:02	PALMA-MULTICAST	PALMA	26 …	DISCOVER:
18	10:0f:ac:e0:00:02	PALMA-MULTICAST	PALMA	26 …	DISCOVER:

Figure 5.8: Wireshark - Test Case 8

Test Case 9

For this test, a server is executed first with the following options:

- Distributable unicast set: 0x1a:ca:00:00:00:00
- Maximum number of addresses to assign: 1000
- Lifetime per assignation: 10 seconds
- Renewal option: enabled

Afterwards a client claiming the whole self-assign unicast set without a preassigned address is executed. This client has been configured with minimum number of addresses to assign equal to 1 and maximum equal to 100. Moreover, the renewal option for the client is also enabled. The test duration is around 30 seconds. Results can be seen in table 5.9 and figure 5.9.

Table 5.9: Validation Plan - Test Case 9

Test Case:	TC - 09
Expected behavior:	Client is expected to issue a DISCOVER message and the server will instantly issue an OFFER message offering the maximum number of unicast addresses specified in its configuration file. When the client receives the OFFER, it will issue a REQUEST message with source address the first one of the offered unicast set. Server will acknowledge this request with an ACK message indicating the assignation life. Before this assignation life expires, client will try to renew the offered address set sending a new REQUEST to the server, and this one will again acknowledge this request with the same assignation life (renewal done), because it has been configured with renewal option enabled.
Real behavior:	As expected
Result:	Test passed

	Source	Destination	Protocol	Length	Time	Info
1	2a:00:e2:3b:2a:8e	PALMA-MULTICAST	PALMA	40	0.000000000	DISCOVER: SET:[start=0a:00:00:00:00:00 mask=ff:00:00:00:00:00]
2	10:0a:bc:de:f0:01	2a:00:e2:3b:2a:8e	PALMA	55	0.000096539	OFFER: lifetime=10 s SET:[start=1a:ca:00:00:00:00, cnt=1000]
3	1a:ca:00:00:00:00	10:0a:bc:de:f0:01	PALMA	36	0.512652714	REQUEST: SET:[start=1a:ca:00:00:00:00, cnt=100]
4	10:0a:bc:de:f0:01	1a:ca:00:00:00:00	PALMA	40	0.512748862	ACK: SET:[start=1a:ca:00:00:00:00, cnt=100] lifetime=10 s
5	1a:ca:00:00:00:00	10:0a:bc:de:f0:01	PALMA	36	9.543005508	REQUEST: SET:[start=1a:ca:00:00:00:00, cnt=100]
6	10:0a:bc:de:f0:01	1a:ca:00:00:00:00	PALMA	40	9.543135990	ACK: SET:[start=1a:ca:00:00:00:00, cnt=100] lifetime=10 s
7	1a:ca:00:00:00:00	10:0a:bc:de:f0:01	PALMA	36	18.558377315	REQUEST: SET:[start=1a:ca:00:00:00:00, cnt=100]
8	10:0a:bc:de:f0:01	1a:ca:00:00:00:00	PALMA	40	18.558446117	ACK: SET:[start=1a:ca:00:00:00:00, cnt=100] lifetime=10 s
9	1a:ca:00:00:00:00	10:0a:bc:de:f0:01	PALMA	36	27.577309657	REQUEST: SET:[start=1a:ca:00:00:00:00, cnt=100]
10	10:0a:bc:de:f0:01	1a:ca:00:00:00:00	PALMA	40	27.577447267	ACK: SET:[start=1a:ca:00:00:00:00, cnt=100] lifetime=10 s

Figure 5.9: Wireshark - Test Case 9

Test Case 10

For this test, a server is executed first with the following options:

- Distributable unicast set: 0x1a:ca:00:00:00:00
- Maximum number of addresses to assign: 1000

- Lifetime per assignation: 10 seconds
- Renewal option: disabled

Afterwards a client claiming the whole self-assign unicast set without a preassigned address is executed. This client has been configured with minimum number of addresses to assign equal to 1 and maximum equal to 100. Moreover, the renewal option for the client is also enabled. The test duration is around 20 seconds. Results can be seen in table 5.10 and figure 5.10.

Table 5.10: Validation Plan - Test Case 10

Test Case:	TC - 10
Expected behavior:	Client is expected to issue a DISCOVER message and the server will instantly issue an OFFER message offering the maximum number of unicast addresses specified in its configuration file. When the client receives the OFFER, it will issue a REQUEST message with source address the first one of the offered unicast set. Server will acknowledge this request with an ACK message indicating the assignation life. Before this assignation life expires client will try to renew the offered address set sending a new REQUEST to the server, but this one will acknowledge this request with the assignation life time left (no renewal done), because it has been configured with renewal option disabled.
Real behavior:	As expected
Result:	Test passed

	Source	Destination	Protocol	Length	Time	Info
1	2a:00:af:3b:2a:46	PALMA-MULTICAST	PALMA	40	0.000000000	DISCOVER: SET:[start=0a:00:00:00:00:00 mask=ff:00:00:00:00:00]
2	10:0a:bc:de:f0:01	2a:00:af:3b:2a:46	PALMA	55	0.000160524	OFFER: lifetime=10 s SET:[start=1a:ca:00:00:00:00, cnt=1000]
3	1a:ca:00:00:00:00	10:0a:bc:de:f0:01	PALMA	36	0.520512061	REQUEST: SET:[start=1a:ca:00:00:00:00, cnt=100]
4	10:0a:bc:de:f0:01	1a:ca:00:00:00:00	PALMA	40	0.520621542	ACK: SET:[start=1a:ca:00:00:00:00, cnt=100] lifetime=10 s
5	1a:ca:00:00:00:00	10:0a:bc:de:f0:01	PALMA	36	9.535197172	REQUEST: SET:[start=1a:ca:00:00:00:00, cnt=100]
6	10:0a:bc:de:f0:01	1a:ca:00:00:00:00	PALMA	40	9.535630473	ACK: SET:[start=1a:ca:00:00:00:00, cnt=100] lifetime=0 s
7	2a:00:97:31:82:67	PALMA-MULTICAST	PALMA	40	9.536236539	DISCOVER: SET:[start=0a:00:00:00:00:00 mask=ff:00:00:00:00:00]
8	10:0a:bc:de:f0:01	2a:00:97:31:82:67	PALMA	55	9.536415594	OFFER: lifetime=10 s SET:[start=1a:ca:00:00:03:e8, cnt=1000]
9	1a:ca:00:00:03:e8	10:0a:bc:de:f0:01	PALMA	36	10.079138901	REQUEST: SET:[start=1a:ca:00:00:03:e8, cnt=100]
10	10:0a:bc:de:f0:01	1a:ca:00:00:03:e8	PALMA	40	10.080014741	ACK: SET:[start=1a:ca:00:00:03:e8, cnt=100] lifetime=10 s
11	1a:ca:00:00:03:e8	10:0a:bc:de:f0:01	PALMA	36	19.099221639	REQUEST: SET:[start=1a:ca:00:00:03:e8, cnt=100]
12	10:0a:bc:de:f0:01	1a:ca:00:00:03:e8	PALMA	40	19.099699645	ACK: SET:[start=1a:ca:00:00:03:e8, cnt=100] lifetime=0 s
13	2a:00:8b:ad:df:1a	PALMA-MULTICAST	PALMA	40	19.100034658	DISCOVER: SET:[start=0a:00:00:00:00:00 mask=ff:00:00:00:00:00]
14	10:0a:bc:de:f0:01	2a:00:8b:ad:df:1a	PALMA	55	19.100292610	OFFER: lifetime=10 s SET:[start=1a:ca:00:00:00:00, cnt=1000]
15	1a:ca:00:00:00:00	10:0a:bc:de:f0:01	PALMA	36	19.681898123	REQUEST: SET:[start=1a:ca:00:00:00:00, cnt=100]
16	10:0a:bc:de:f0:01	1a:ca:00:00:00:00	PALMA	40	19.682144916	ACK: SET:[start=1a:ca:00:00:00:00, cnt=100] lifetime=10 s

Figure 5.10: Wireshark - Test Case 10

Test Case 11

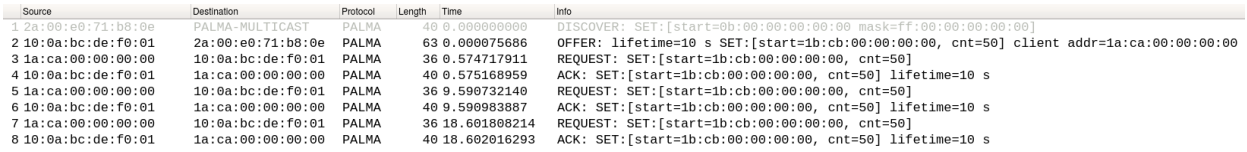
For this test, a server is executed first with the following options:

- Distributable multicast set: 0x1b:cb:00:00:00:00
- Maximum number of addresses to assign: 50
- Lifetime per assignation: 10 seconds
- Renewal option: enabled

Afterwards a client claiming the whole self-assign multicast set with a preassigned address is executed. This client has been configured with minimum number of addresses to assign equal to 1 and maximum equal to 100. Moreover, the renewal option for the client is also enabled. The test duration is around 20 seconds. Results can be seen in table 5.11 and figure 5.11.

Table 5.11: Validation Plan - Test Case 11

Test Case:	TC - 11
Expected behavior:	Client is expected to issue a DISCOVER message and the server will instantly issue an OFFER message offering the maximum number of multicast addresses specified in its configuration file, plus a unicast address for the client to use as source. When the client receives the OFFER, it will issue a REQUEST message with source address the one contained in the client address parameter. Server will acknowledge this request with an ACK message indicating the assignation life. Before this assignation life expires client will try to renew the offered address set sending a new REQUEST to the server, and this one will again acknowledge this request with the same assignation life (renewal done), because it has been configured with renewal option enabled.
Real behavior:	As expected
Result:	Test passed



	Source	Destination	Protocol	Length	Time	Info
1	2a:00:e0:71:b8:0e	PALMA-MULTICAST	PALMA	40	0.000000000	DISCOVER: SET:[start=0b:00:00:00:00:00 mask=ff:00:00:00:00:00]
2	10:0a:bc:de:f0:01	2a:00:e0:71:b8:0e	PALMA	63	0.000075686	OFFER: lifetime=10 s SET:[start=1b:cb:00:00:00:00, cnt=50] client addr=1a:ca:00:00:00:00
3	1a:ca:00:00:00:00	10:0a:bc:de:f0:01	PALMA	36	0.574717911	REQUEST: SET:[start=1b:cb:00:00:00:00, cnt=50]
4	10:0a:bc:de:f0:01	1a:ca:00:00:00:00	PALMA	40	0.575168959	ACK: SET:[start=1b:cb:00:00:00:00, cnt=50] lifetime=10 s
5	1a:ca:00:00:00:00	10:0a:bc:de:f0:01	PALMA	36	9.590732140	REQUEST: SET:[start=1b:cb:00:00:00:00, cnt=50]
6	10:0a:bc:de:f0:01	1a:ca:00:00:00:00	PALMA	40	9.590983887	ACK: SET:[start=1b:cb:00:00:00:00, cnt=50] lifetime=10 s
7	1a:ca:00:00:00:00	10:0a:bc:de:f0:01	PALMA	36	18.601808214	REQUEST: SET:[start=1b:cb:00:00:00:00, cnt=50]
8	10:0a:bc:de:f0:01	1a:ca:00:00:00:00	PALMA	40	18.602016293	ACK: SET:[start=1b:cb:00:00:00:00, cnt=50] lifetime=10 s

Figure 5.11: Wireshark - Test Case 11

Test Case 12

For this test, a server is executed first with the following options:

- Distributable multicast set: 0x1b:cb:00:00:00:00
- Maximum number of addresses to assign: 50

- Lifetime per assignation: 10 seconds
- Renewal option: disabled

Afterwards a client claiming the whole self-assign multicast set with a preassigned address is executed. This client has been configured with minimum number of addresses to assign equal to 1 and maximum equal to 100. Moreover, the renewal option for the client is also enabled. The test duration is around 40 seconds. Results can be seen in table 5.12 and figure 5.12.

Table 5.12: Validation Plan - Test Case 12

Test Case:	TC - 12
Expected behavior:	Client is expected to issue a DISCOVER message and the server will instantly issue an OFFER message offering the maximum number of multicast addresses specified in its configuration file, plus a unicast address for the client to use as source. When the client receives the OFFER, it will issue a REQUEST message with source address the one contained in the client address parameter. Server will acknowledge this request with an ACK message indicating the given assignation life time. Before this assignation life expires client will try to renew the offered address set sending a new REQUEST to the server, but this one will acknowledge this request with the assignation life time left (no renewal done), because it has been configured with renewal option disabled.
Real behavior:	As expected
Result:	Test passed

	Source	Destination	Protocol	Length	Time	Info
1	2a:00:c4:f3:82:34	PALMA-MULTICAST	PALMA	40	0.000000000	DISCOVER: SET:[start=0b:00:00:00:00:00 mask=ff:00:00:00:00:00]
2	10:0a:bc:de:f0:01	2a:00:c4:f3:82:34	PALMA	63	0.000056396	OFFER: lifetime=10 s SET:[start=1b:cb:00:00:00:00, cnt=50] client addr=1a:ca:00:00:00:00
3	1a:ca:00:00:00:00	10:0a:bc:de:f0:01	PALMA	36	0.581354065	REQUEST: SET:[start=1b:cb:00:00:00:00, cnt=50]
4	10:0a:bc:de:f0:01	1a:ca:00:00:00:00	PALMA	40	0.581405008	ACK: SET:[start=1b:cb:00:00:00:00, cnt=50] lifetime=10 s
5	1a:ca:00:00:00:00	10:0a:bc:de:f0:01	PALMA	36	9.601984468	REQUEST: SET:[start=1b:cb:00:00:00:00, cnt=50]
6	10:0a:bc:de:f0:01	1a:ca:00:00:00:00	PALMA	40	9.602044420	ACK: SET:[start=1b:cb:00:00:00:00, cnt=50] lifetime=0 s
7	2a:00:54:8e:f6:1d	PALMA-MULTICAST	PALMA	40	9.602106187	DISCOVER: SET:[start=0b:00:00:00:00:00 mask=ff:00:00:00:00:00]
8	10:0a:bc:de:f0:01	2a:00:54:8e:f6:1d	PALMA	63	9.602190171	OFFER: lifetime=10 s SET:[start=1b:cb:00:00:00:32, cnt=50] client addr=1a:ca:00:00:00:01
9	1a:ca:00:00:00:01	10:0a:bc:de:f0:01	PALMA	36	10.105408797	REQUEST: SET:[start=1b:cb:00:00:00:32, cnt=50]
10	10:0a:bc:de:f0:01	1a:ca:00:00:00:01	PALMA	40	10.105517669	ACK: SET:[start=1b:cb:00:00:00:32, cnt=50] lifetime=10 s
11	1a:ca:00:00:00:01	10:0a:bc:de:f0:01	PALMA	36	19.119134275	REQUEST: SET:[start=1b:cb:00:00:00:32, cnt=50]
12	10:0a:bc:de:f0:01	1a:ca:00:00:00:01	PALMA	40	19.119318480	ACK: SET:[start=1b:cb:00:00:00:32, cnt=50] lifetime=0 s
13	2a:00:75:c0:f1:78	PALMA-MULTICAST	PALMA	40	19.119435432	DISCOVER: SET:[start=0b:00:00:00:00:00 mask=ff:00:00:00:00:00]
14	10:0a:bc:de:f0:01	2a:00:75:c0:f1:78	PALMA	63	19.119513953	OFFER: lifetime=10 s SET:[start=1b:cb:00:00:00:64, cnt=50] client addr=1a:ca:00:00:00:02
15	1a:ca:00:00:00:02	10:0a:bc:de:f0:01	PALMA	36	19.626115591	REQUEST: SET:[start=1b:cb:00:00:00:64, cnt=50]
16	10:0a:bc:de:f0:01	1a:ca:00:00:00:02	PALMA	40	19.626172675	ACK: SET:[start=1b:cb:00:00:00:64, cnt=50] lifetime=10 s
17	1a:ca:00:00:00:02	10:0a:bc:de:f0:01	PALMA	36	28.645512477	REQUEST: SET:[start=1b:cb:00:00:00:64, cnt=50]
18	10:0a:bc:de:f0:01	1a:ca:00:00:00:02	PALMA	40	28.645571553	ACK: SET:[start=1b:cb:00:00:00:64, cnt=50] lifetime=0 s
19	2a:00:39:b8:83:b1	PALMA-MULTICAST	PALMA	40	28.645662760	DISCOVER: SET:[start=0b:00:00:00:00:00 mask=ff:00:00:00:00:00]
20	10:0a:bc:de:f0:01	2a:00:39:b8:83:b1	PALMA	63	28.645708538	OFFER: lifetime=10 s SET:[start=1b:cb:00:00:00:00, cnt=50] client addr=1a:ca:00:00:00:00
21	1a:ca:00:00:00:00	10:0a:bc:de:f0:01	PALMA	36	29.183293941	REQUEST: SET:[start=1b:cb:00:00:00:00, cnt=50]
22	10:0a:bc:de:f0:01	1a:ca:00:00:00:00	PALMA	40	29.183804986	ACK: SET:[start=1b:cb:00:00:00:00, cnt=50] lifetime=10 s
23	1a:ca:00:00:00:00	10:0a:bc:de:f0:01	PALMA	36	38.187817193	REQUEST: SET:[start=1b:cb:00:00:00:00, cnt=50]
24	10:0a:bc:de:f0:01	1a:ca:00:00:00:00	PALMA	40	38.188600993	ACK: SET:[start=1b:cb:00:00:00:00, cnt=50] lifetime=0 s
25	2a:00:9f:91:03:1c	PALMA-MULTICAST	PALMA	40	38.188740810	DISCOVER: SET:[start=0b:00:00:00:00:00 mask=ff:00:00:00:00:00]
26	10:0a:bc:de:f0:01	2a:00:9f:91:03:1c	PALMA	63	38.188821116	OFFER: lifetime=10 s SET:[start=1b:cb:00:00:00:32, cnt=50] client addr=1a:ca:00:00:00:01
27	1a:ca:00:00:00:01	10:0a:bc:de:f0:01	PALMA	36	38.826478546	REQUEST: SET:[start=1b:cb:00:00:00:32, cnt=50]
28	10:0a:bc:de:f0:01	1a:ca:00:00:00:01	PALMA	40	38.826516206	ACK: SET:[start=1b:cb:00:00:00:32, cnt=50] lifetime=10 s

Figure 5.12: Wireshark - Test Case 12

Test Case 13

For this test, a server is executed first with the following options:

- Distributable multicast set: 0x1b:cb:00:00:00:00
- Maximum number of addresses to assign: 50
- Lifetime per assignation: 10 seconds
- Renewal option: enabled

Afterwards a client claiming the whole self-assign multicast set with a preassigned address is executed. This client has been configured with minimum number of addresses to assign equal to 1 and maximum equal to 100. Moreover, the renewal option for the client is disabled. The test duration is around 20 seconds. Results can be seen in table 5.13 and figure 5.13.

Table 5.13: Validation Plan - Test Case 13

Test Case:	TC - 13
Expected behavior:	Client is expected to issue a DISCOVER message and the server will instantly issue an OFFER message offering the maximum number of multicast addresses specified in its configuration file, plus a unicast address for the client to use as source. When the client receives the OFFER, it will issue a REQUEST message with source address the one contained in the client address parameter. Server will acknowledge this request with an ACK message indicating the assignation life. This time the client will not try to do a renewal because it has been configured with renewal option disabled, instead, when assignation life expires, client will restart the protocol and same will happen. Moreover, client will issue a RELEASE message at second 13 because user has exited the program.
Real behavior:	As expected
Result:	Test passed

	Source	Destination	Protocol	Length	Time	Info
1	2a:00:47:f7:e7:3a	PALMA-MULTICAST	PALMA	40	0.000000000	DISCOVER: SET:[start=0b:00:00:00:00:00 mask=ff:00:00:00:00:00]
2	10:0a:bc:de:f0:01	2a:00:47:f7:e7:3a	PALMA	63	0.000165678	OFFER: lifetime=10 s SET:[start=1b:cb:00:00:00:00, cnt=50] client addr=1a:ca:00:00:00:00
3	1a:ca:00:00:00:00	10:0a:bc:de:f0:01	PALMA	36	0.550759191	REQUEST: SET:[start=1b:cb:00:00:00:00, cnt=50]
4	10:0a:bc:de:f0:01	1a:ca:00:00:00:00	PALMA	40	0.550936152	ACK: SET:[start=1b:cb:00:00:00:00, cnt=50] lifetime=10 s
5	2a:00:6a:0f:a0:bb	PALMA-MULTICAST	PALMA	40	10.564329425	DISCOVER: SET:[start=0b:00:00:00:00:00 mask=ff:00:00:00:00:00]
6	10:0a:bc:de:f0:01	2a:00:6a:0f:a0:bb	PALMA	63	10.564523269	OFFER: lifetime=10 s SET:[start=1b:cb:00:00:00:32, cnt=50] client addr=1a:ca:00:00:00:01
7	1a:ca:00:00:00:01	10:0a:bc:de:f0:01	PALMA	36	11.171089158	REQUEST: SET:[start=1b:cb:00:00:00:32, cnt=50]
8	10:0a:bc:de:f0:01	1a:ca:00:00:00:01	PALMA	40	11.171605027	ACK: SET:[start=1b:cb:00:00:00:32, cnt=50] lifetime=10 s
9	1a:ca:00:00:00:01	10:0a:bc:de:f0:01	PALMA	36	13.173017365	RELEASE: SET:[start=1b:cb:00:00:00:32, cnt=50]

Figure 5.13: Wireshark - Test Case 13

Test Case 14

For this test, a server is executed first with the following options:

- Distributable default set: 0x1a:ca:00:00:00:00 (unicast - 48bit)
- Maximum default number of addresses to assign: 2000

- Lifetime per assignation: 10 seconds
- Renewal option: enabled

Afterwards a client claiming a null set without a preassigned address is executed. This client wants unicast addresses and has been configured with minimum number of addresses to assign equal to 1 and maximum equal to 100. Moreover, the renewal option for the client is also enabled. The test duration is around 20 seconds. Results can be seen in table 5.14 and figure 5.14.

Table 5.14: Validation Plan - Test Case 14

Test Case:	TC - 14
Expected behavior:	Client is expected to issue a null DISCOVER message as specified in the configuration file. Server will send an OFFER message with default settings of its configuration file. Client receiving the OFFER will accept it and issue a REQUEST because address set offered matches the address type wanted. Same behavior as TC-09 follows afterwards.
Real behavior:	As expected
Result:	Test passed

	Source	Destination	Protocol	Length	Time	Info
1	2a:00:66:4a:ff:38	PALMA-MULTICAST	PALMA	26	0.000000000	DISCOVER:
2	10:0a:bc:de:f0:01	2a:00:66:4a:ff:38	PALMA	55	0.000095377	OFFER: lifetime=10 s SET:[start=1a:ca:00:00:00:00, cnt=2000]
3	1a:ca:00:00:00:00	10:0a:bc:de:f0:01	PALMA	36	0.535728787	REQUEST: SET:[start=1a:ca:00:00:00:00, cnt=100]
4	10:0a:bc:de:f0:01	1a:ca:00:00:00:00	PALMA	40	0.535778941	ACK: SET:[start=1a:ca:00:00:00:00, cnt=100] lifetime=10 s
5	1a:ca:00:00:00:00	10:0a:bc:de:f0:01	PALMA	36	9.547033662	REQUEST: SET:[start=1a:ca:00:00:00:00, cnt=100]
6	10:0a:bc:de:f0:01	1a:ca:00:00:00:00	PALMA	40	9.547136854	ACK: SET:[start=1a:ca:00:00:00:00, cnt=100] lifetime=10 s

Figure 5.14: Wireshark - Test Case 14

Test Case 15

For this test, a server is executed first with the following options:

- Distributable default set: 0x1a:ca:00:00:00:00:00:00 (unicast - 64-bit)

- Maximum default number of addresses to assign: 2000
- Lifetime per assignation: 10 seconds
- Renewal option: enabled

Afterwards a client claiming a null set without a preassigned address is executed. This client wants multicast addresses and has been configured with minimum number of addresses to assign equal to 1 and maximum equal to 100. Moreover, the renewal option for the client is also enabled. The test duration is around 20 seconds. Results can be seen in table 5.15 and figure 5.15.

Table 5.15: Validation Plan - Test Case 15

Test Case:	TC - 15
Expected behavior:	Client is expected to issue a null DISCOVER message as specified in the configuration file. Server will send an OFFER message with default settings of its configuration file. Client receiving the OFFER will not accept it and issue a null DISCOVER again because address set offered does not match the address type wanted. This behavior will be repeated continuously.
Real behavior:	As expected
Result:	Test passed

	Source	Destination	Protocol	Length	Time	Info
1	2a:00:cf:06:ce:28	PALMA-MULTICAST	PALMA	26	0.000000000	DISCOVER:
2	10:0a:bc:de:f0:01	2a:00:cf:06:ce:28	PALMA	65	0.000183439	OFFER: lifetime=10 s SET:[start=1a:ca:00:00:00:00:00:00, cnt=2000] client addr=1a:ca:00:00:00:00
3	2a:00:df:36:2d:38	PALMA-MULTICAST	PALMA	26	0.512787285	DISCOVER:
4	10:0a:bc:de:f0:01	2a:00:df:36:2d:38	PALMA	65	0.513035370	OFFER: lifetime=10 s SET:[start=1a:ca:00:00:00:00:07:d0, cnt=2000] client addr=1a:ca:00:00:00:01
5	2a:00:67:8e:bb:d6	PALMA-MULTICAST	PALMA	26	1.065638135	DISCOVER:
6	10:0a:bc:de:f0:01	2a:00:67:8e:bb:d6	PALMA	65	1.065749836	OFFER: lifetime=10 s SET:[start=1a:ca:00:00:00:00:0f:a0, cnt=2000] client addr=1a:ca:00:00:00:02
7	2a:00:a7:1e:42:49	PALMA-MULTICAST	PALMA	26	1.642948798	DISCOVER:
8	10:0a:bc:de:f0:01	2a:00:a7:1e:42:49	PALMA	65	1.643120000	OFFER: lifetime=10 s SET:[start=1a:ca:00:00:00:00:17:70, cnt=2000] client addr=1a:ca:00:00:00:03
9	2a:00:d3:dc:fa:44	PALMA-MULTICAST	PALMA	26	2.211327339	DISCOVER:
10	10:0a:bc:de:f0:01	2a:00:d3:dc:fa:44	PALMA	65	2.211535985	OFFER: lifetime=10 s SET:[start=1a:ca:00:00:00:00:1f:40, cnt=2000] client addr=1a:ca:00:00:00:04
11	2a:00:88:ea:a6:9a	PALMA-MULTICAST	PALMA	26	2.810560689	DISCOVER:
12	10:0a:bc:de:f0:01	2a:00:88:ea:a6:9a	PALMA	65	2.810766309	OFFER: lifetime=10 s SET:[start=1a:ca:00:00:00:00:00:00, cnt=2000] client addr=1a:ca:00:00:00:00

Figure 5.15: Wireshark - Test Case 15

Test Case 16

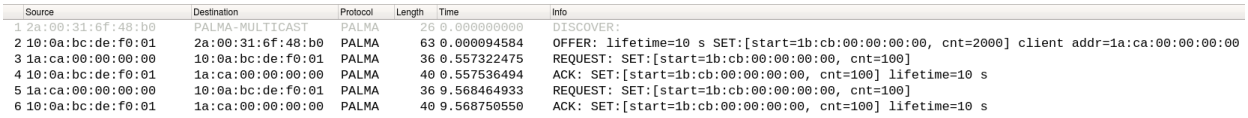
For this test, a server is executed first with the following options:

- Distributable default set: 0x1b:cb:00:00:00:00 (multicast - 48-bit)
- Maximum default number of addresses to assign: 2000
- Lifetime per assignation: 10 seconds
- Renewal option: enabled

Afterwards a client claiming a null set without a preassigned address is executed. This client wants multicast addresses and has been configured with minimum number of addresses to assign equal to 1 and maximum equal to 100. Moreover, the renewal option for the client is also enabled. The test duration is around 20 seconds. Results can be seen in table 5.16 and figure 5.16.

Table 5.16: Validation Plan - Test Case 16

Test Case:	TC - 16
Expected behavior:	Client is expected to issue a null DISCOVER message as specified in the configuration file. Server will send an OFFER message with default settings of its configuration file, and a unicast address for the client to use it as source address. Client receiving the OFFER will accept it and issue a REQUEST because address set offered matches the address type wanted. Same behavior as TC-11 follows afterwards.
Real behavior:	As expected
Result:	Test passed



	Source	Destination	Protocol	Length	Time	Info
1	2a:00:31:6f:48:b0	PALMA-MULTICAST	PALMA	26	0.000000000	DISCOVER:
2	10:0a:bc:de:f0:01	2a:00:31:6f:48:b0	PALMA	63	0.000094584	OFFER: lifetime=10 s SET:[start=1b:cb:00:00:00:00, cnt=2000] client addr=1a:ca:00:00:00:00
3	1a:ca:00:00:00:00	10:0a:bc:de:f0:01	PALMA	36	0.557322475	REQUEST: SET:[start=1b:cb:00:00:00:00, cnt=100]
4	10:0a:bc:de:f0:01	1a:ca:00:00:00:00	PALMA	40	0.557536494	ACK: SET:[start=1b:cb:00:00:00:00, cnt=100] lifetime=10 s
5	1a:ca:00:00:00:00	10:0a:bc:de:f0:01	PALMA	36	9.568464933	REQUEST: SET:[start=1b:cb:00:00:00:00, cnt=100]
6	10:0a:bc:de:f0:01	1a:ca:00:00:00:00	PALMA	40	9.568750550	ACK: SET:[start=1b:cb:00:00:00:00, cnt=100] lifetime=10 s

Figure 5.16: Wireshark - Test Case 16

Test Case 17

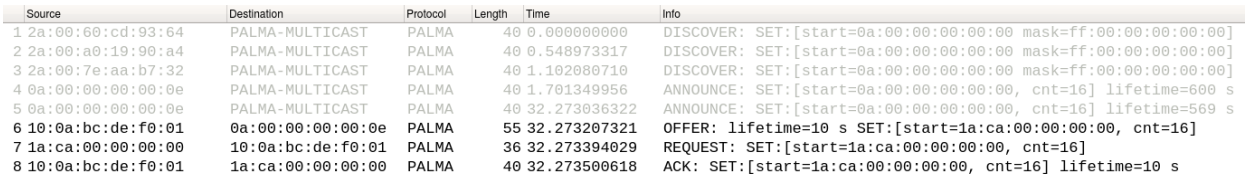
A client claiming the whole self-assign unicast set without a preassigned address is executed in first place. This client has been configured with minimum number of addresses to assign equal to 1 and maximum equal to 100. A server is executed after 10 seconds with the following configuration:

- Distributable default set: 0x1a:ca:00:00:00:00 (unicast - 48-bit)
- Maximum number of addresses to assign: 2000
- Lifetime per assignation: 10 seconds
- Self assignment objection: enabled

The test duration is around 40 seconds. Results can be seen in table 5.17 and figure 5.17.

Table 5.17: Validation Plan - Test Case 17

Test Case:	TC - 17
Expected behavior:	Client has self-assigned 16 unicast addresses before server has been executed. When server listening to PALMA multicast address receives an ANNOUNCE, it issues an OFFER to the that particular client, offering the same number of addresses as the ANNOUNCE message announces and the client accepts this OFFER by sending a REQUEST to the server (with source address the first address of the offered unicast set), which ends the server based assignation by issuing an ACK.
Real behavior:	As expected
Result:	Test passed



	Source	Destination	Protocol	Length	Time	Info
1	2a:00:60:cd:93:64	PALMA-MULTICAST	PALMA	40	0.000000000	DISCOVER: SET:[start=0a:00:00:00:00:00 mask=ff:00:00:00:00:00]
2	2a:00:a0:19:90:a4	PALMA-MULTICAST	PALMA	40	0.548973317	DISCOVER: SET:[start=0a:00:00:00:00:00 mask=ff:00:00:00:00:00]
3	2a:00:7e:aa:b7:32	PALMA-MULTICAST	PALMA	40	1.102080710	DISCOVER: SET:[start=0a:00:00:00:00:00 mask=ff:00:00:00:00:00]
4	0a:00:00:00:00:0e	PALMA-MULTICAST	PALMA	40	1.701349956	ANNOUNCE: SET:[start=0a:00:00:00:00:00, cnt=16] lifetime=600 s
5	0a:00:00:00:00:0e	PALMA-MULTICAST	PALMA	40	32.273036322	ANNOUNCE: SET:[start=0a:00:00:00:00:00, cnt=16] lifetime=569 s
6	10:0a:bc:de:f0:01	0a:00:00:00:00:0e	PALMA	55	32.273207321	OFFER: lifetime=10 s SET:[start=1a:ca:00:00:00:00, cnt=16]
7	1a:ca:00:00:00:00	10:0a:bc:de:f0:01	PALMA	36	32.273394029	REQUEST: SET:[start=1a:ca:00:00:00:00, cnt=16]
8	10:0a:bc:de:f0:01	1a:ca:00:00:00:00	PALMA	40	32.273500618	ACK: SET:[start=1a:ca:00:00:00:00, cnt=16] lifetime=10 s

Figure 5.17: Wireshark - Test Case 17

Test Case 18

A client claiming the whole self-assign unicast set with a preassigned address is executed in first place. This client has been configured with minimum number of addresses to assign equal to 1 and maximum equal to 100. A server is executed after 10 seconds with the following configuration:

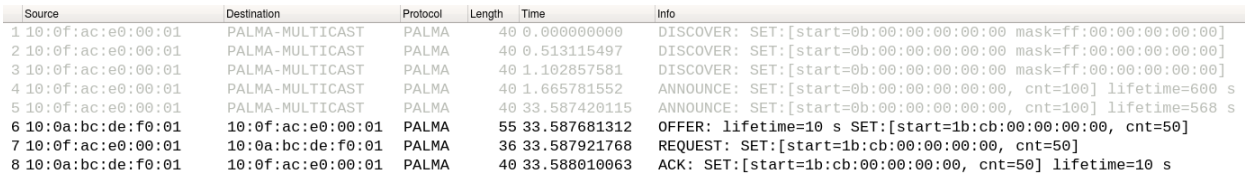
- Distributable default set: 0x1b:cb:00:00:00:00 (multicast - 48-bit)
- Maximum number of addresses to assign: 50
- Lifetime per assignation: 10 seconds

- Self assignment objection: enabled

The test duration is around 40 seconds. Results can be seen in table 5.18 and figure 5.18.

Table 5.18: Validation Plan - Test Case 18

Test Case:	TC - 18
Expected behavior:	Client has self-assigned 100 multicast addresses before server has been executed. When server listening to PALMA multicast address receives an ANNOUNCE, it issues an OFFER to the that particular client, offering the maximum number of addresses specified in its configuration file. The client accepts this OFFER by sending a REQUEST to the server, which ends the server-based assignation by issuing an ACK.
Real behavior:	As expected
Result:	Test passed



	Source	Destination	Protocol	Length	Time	Info
1	10:0f:ac:e0:00:01	PALMA-MULTICAST	PALMA	40	0.000000000	DISCOVER: SET:[start=0b:00:00:00:00:00 mask=ff:00:00:00:00:00]
2	10:0f:ac:e0:00:01	PALMA-MULTICAST	PALMA	40	0.513115497	DISCOVER: SET:[start=0b:00:00:00:00:00 mask=ff:00:00:00:00:00]
3	10:0f:ac:e0:00:01	PALMA-MULTICAST	PALMA	40	1.102857581	DISCOVER: SET:[start=0b:00:00:00:00:00 mask=ff:00:00:00:00:00]
4	10:0f:ac:e0:00:01	PALMA-MULTICAST	PALMA	40	1.665781552	ANNOUNCE: SET:[start=0b:00:00:00:00:00, cnt=100] lifetime=600 s
5	10:0f:ac:e0:00:01	PALMA-MULTICAST	PALMA	40	33.587420115	ANNOUNCE: SET:[start=0b:00:00:00:00:00, cnt=100] lifetime=568 s
6	10:0a:bc:de:f0:01	10:0f:ac:e0:00:01	PALMA	55	33.587681312	OFFER: lifetime=10 s SET:[start=1b:cb:00:00:00:00, cnt=50]
7	10:0f:ac:e0:00:01	10:0a:bc:de:f0:01	PALMA	36	33.587921768	REQUEST: SET:[start=1b:cb:00:00:00:00, cnt=50]
8	10:0a:bc:de:f0:01	10:0f:ac:e0:00:01	PALMA	40	33.588010063	ACK: SET:[start=1b:cb:00:00:00:00, cnt=50] lifetime=10 s

Figure 5.18: Wireshark - Test Case 18

Test Case 19

For this test, a server is executed in first place with the following configuration:

- Distributable set: 0x1a:ca:00:00:00:00 (unicast - 48-bit)
- Maximum number of addresses to assign: 50
- Lifetime per assignation: 10 seconds

- Self assignment objection: enabled

Secondly a client claiming the whole self-assign unicast set without a preassigned address is executed. This client has been configured with minimum number of addresses to assign equal to 60 and maximum equal to 100. The test duration is around 40 seconds. Results can be seen in table 5.19 and figure 5.19.

Table 5.19: Validation Plan - Test Case 19

Test Case:	TC - 19
Expected behavior:	Client is expected to issue a DISCOVER message, which is going to be replied by the server with an OFFER message offering the maximum number of addresses specified in the server's configuration file. The client will not accept this offer because the minimum number of assignable addresses specified in its configuration is bigger than the offered ones. Client will continue the self-assignation procedure ignoring the OFFER messages of the server. Even so, the server will continue issuing OFFER messages to the client when receiving ANNOUNCE messages because the self-assignment objection option is enabled.
Real behavior:	As expected
Result:	Test passed

	Source	Destination	Protocol	Length	Time	Info
1	2a:00:9c:28:72:ac	PALMA-MULTICAST	PALMA	40	0.000000000	DISCOVER: SET:[start=0a:00:00:00:00:00 mask=ff:00:00:00:00:00]
2	10:0a:bc:de:f0:01	2a:00:9c:28:72:ac	PALMA	55	0.000126069	OFFER: lifetime=10 s SET:[start=1a:ca:00:00:00:00, cnt=50]
3	2a:00:8b:1c:8e:8c	PALMA-MULTICAST	PALMA	40	0.512820233	DISCOVER: SET:[start=0a:00:00:00:00:00 mask=ff:00:00:00:00:00]
4	10:0a:bc:de:f0:01	2a:00:8b:1c:8e:8c	PALMA	55	0.512951302	OFFER: lifetime=10 s SET:[start=1a:ca:00:00:00:32, cnt=50]
5	2a:00:25:3d:dc:3a	PALMA-MULTICAST	PALMA	40	1.068107714	DISCOVER: SET:[start=0a:00:00:00:00:00 mask=ff:00:00:00:00:00]
6	10:0a:bc:de:f0:01	2a:00:25:3d:dc:3a	PALMA	55	1.068185335	OFFER: lifetime=10 s SET:[start=1a:ca:00:00:00:64, cnt=50]
7	0a:00:00:00:00:08	PALMA-MULTICAST	PALMA	40	1.643069030	ANNOUNCE: SET:[start=0a:00:00:00:00:00, cnt=16] lifetime=600 s
8	10:0a:bc:de:f0:01	0a:00:00:00:00:08	PALMA	55	1.643189042	OFFER: lifetime=10 s SET:[start=1a:ca:00:00:00:96, cnt=16]
9	0a:00:00:00:00:08	PALMA-MULTICAST	PALMA	40	32.879077021	ANNOUNCE: SET:[start=0a:00:00:00:00:00, cnt=16] lifetime=569 s
10	10:0a:bc:de:f0:01	0a:00:00:00:00:08	PALMA	55	32.879194968	OFFER: lifetime=10 s SET:[start=1a:ca:00:00:00:00, cnt=16]

Figure 5.19: Wireshark - Test Case 19

Test Case 20

For this test, a server is executed in first place with the following configuration:

- Distributable set: 0x1b:cb:00:00:00:00 (multicast - 48-bit)
- Maximum number of addresses to assign: 50
- Lifetime per assignation: 10 seconds
- Self assignment objection: disabled

Secondly a client claiming the whole self-assign unicast set with a preassigned source address is executed. This client has been configured with minimum number of addresses to assign equal to 60 and maximum equal to 100. The test duration is around 40 seconds. Results can be seen in table 5.20 and figure 5.20.

Table 5.20: Validation Plan - Test Case 20

Test Case:	TC - 20
Expected behavior:	Client is expected to issue a DISCOVER message, which is going to be replied by the server with an OFFER message offering the maximum number of addresses specified in the server's configuration file. The client will not accept this offer because the minimum number of assignable addresses specified in its configuration is bigger than the offered ones. Client will continue the self-assignation procedure ignoring the OFFER messages of the server. In any case, the server will only offer address sets in reply to DISCOVER messages and not ANNOUNCE messages, because the self-assignment objection option is disabled.
Real behavior:	As expected
Result:	Test passed

	Source	Destination	Protocol	Length	Time	Info
1	10:0f:ac:e0:00:01	PALMA-MULTICAST	PALMA	40	0.000000000	DISCOVER: SET:[start=0b:00:00:00:00:00 mask=ff:00:00:00:00:00]
2	10:0a:bc:de:f0:01	10:0f:ac:e0:00:01	PALMA	55	0.000061705	OFFER: lifetime=10 s SET:[start=1b:cb:00:00:00:00, cnt=50]
3	10:0f:ac:e0:00:01	PALMA-MULTICAST	PALMA	40	0.542924516	DISCOVER: SET:[start=0b:00:00:00:00:00 mask=ff:00:00:00:00:00]
4	10:0a:bc:de:f0:01	10:0f:ac:e0:00:01	PALMA	55	0.543102396	OFFER: lifetime=10 s SET:[start=1b:cb:00:00:00:32, cnt=50]
5	10:0f:ac:e0:00:01	PALMA-MULTICAST	PALMA	40	1.132246348	DISCOVER: SET:[start=0b:00:00:00:00:00 mask=ff:00:00:00:00:00]
6	10:0a:bc:de:f0:01	10:0f:ac:e0:00:01	PALMA	55	1.132370006	OFFER: lifetime=10 s SET:[start=1b:cb:00:00:00:64, cnt=50]
7	10:0f:ac:e0:00:01	PALMA-MULTICAST	PALMA	40	1.730229803	ANNOUNCE: SET:[start=0b:00:00:00:00:00, cnt=100] lifetime=600 s
8	10:0f:ac:e0:00:01	PALMA-MULTICAST	PALMA	40	32.497944134	ANNOUNCE: SET:[start=0b:00:00:00:00:00, cnt=100] lifetime=569 s

Figure 5.20: Wireshark - Test Case 20

Test Case 21

For this test, a server is executed in first place with the following configuration:

- Distributable set: 0x1a:ca:00:00:00:00 (unicast - 48-bit)
- Maximum number of addresses to assign: 1000
- Lifetime per assignation: 10 seconds
- Alternate set option: enabled
- Renewal option: disabled

Secondly a client with the server's address already known and a preassigned source address is executed. This one is requesting 100 addresses of the self-assign unicast set and has been configured with minimum number of addresses to assign equal to 1 and maximum equal to 100. Moreover, the renewal option for the client is enabled. The test duration is around 30 seconds. Results can be seen in table 5.21 and figure 5.21.

Table 5.21: Validation Plan - Test Case 21

Test Case:	TC - 21
Expected behavior:	Client is expected to issue a REQUEST message to the server's address indicated in the configuration. It will request the maximum configured addresses starting with the address specified in its configuration. Even though the requested set does not belong to the administratively corresponding block of this server, it will offer an alternate set of the maximum number of addresses configured in the server's configuration for that address type in an ACK message, because the alternate set option is enabled. Moreover, the server will not accept renewals due to the disabled renewal option.
Real behavior:	As expected
Result:	Test passed

	Source	Destination	Protocol	Length	Time	Info
1	10:0f:ac:e0:00:01	10:0a:bc:de:f0:01	PALMA	36	0.000000000	REQUEST: SET:[start=0a:00:00:00:00:00, cnt=100]
2	10:0a:bc:de:f0:01	10:0f:ac:e0:00:01	PALMA	40	0.000255261	ACK: SET:[start=1a:ca:00:00:00:00, cnt=100] lifetime=10 s
3	10:0f:ac:e0:00:01	10:0a:bc:de:f0:01	PALMA	36	8.093932730	REQUEST: SET:[start=1a:ca:00:00:00:00, cnt=100]
4	10:0a:bc:de:f0:01	10:0f:ac:e0:00:01	PALMA	40	8.094331879	ACK: SET:[start=1a:ca:00:00:00:00, cnt=100] lifetime=0 s
5	10:0f:ac:e0:00:01	10:0a:bc:de:f0:01	PALMA	36	8.094563391	REQUEST: SET:[start=0a:00:00:00:00:00, cnt=100]
6	10:0a:bc:de:f0:01	10:0f:ac:e0:00:01	PALMA	40	8.094779376	ACK: SET:[start=1a:ca:00:00:00:64, cnt=100] lifetime=10 s
7	10:0f:ac:e0:00:01	10:0a:bc:de:f0:01	PALMA	36	17.104442310	REQUEST: SET:[start=1a:ca:00:00:00:64, cnt=100]
8	10:0a:bc:de:f0:01	10:0f:ac:e0:00:01	PALMA	40	17.104610148	ACK: SET:[start=1a:ca:00:00:00:64, cnt=100] lifetime=0 s
9	10:0f:ac:e0:00:01	10:0a:bc:de:f0:01	PALMA	36	17.104739755	REQUEST: SET:[start=0a:00:00:00:00:00, cnt=100]
10	10:0a:bc:de:f0:01	10:0f:ac:e0:00:01	PALMA	40	17.104935230	ACK: SET:[start=1a:ca:00:00:00:c8, cnt=100] lifetime=10 s
11	10:0f:ac:e0:00:01	10:0a:bc:de:f0:01	PALMA	36	26.152476095	REQUEST: SET:[start=1a:ca:00:00:00:c8, cnt=100]
12	10:0a:bc:de:f0:01	10:0f:ac:e0:00:01	PALMA	40	26.152676773	ACK: SET:[start=1a:ca:00:00:00:c8, cnt=100] lifetime=0 s
13	10:0f:ac:e0:00:01	10:0a:bc:de:f0:01	PALMA	36	26.152879451	REQUEST: SET:[start=0a:00:00:00:00:00, cnt=100]
14	10:0a:bc:de:f0:01	10:0f:ac:e0:00:01	PALMA	40	26.153018582	ACK: SET:[start=1a:ca:00:00:00:00, cnt=100] lifetime=10 s

Figure 5.21: Wireshark - Test Case 21

Test Case 22

For this test, a server is executed in first place with the following configuration:

- Distributable set: 0x1b:cb:00:00:00:00 (multicast - 48-bit)
- Maximum number of addresses to assign: 1000

- Lifetime per assignation: 10 seconds
- Alternate set option: disabled
- Renewal option: enabled

Secondly a client with the server's address already known and a preassigned source address is executed. This one is requesting 100 addresses of the server's distributable set and has been configured with minimum number of addresses to assign equal to 1 and maximum equal to 100. Moreover, the renewal option for the client is enabled. The test duration is around 30 seconds. Results can be seen in table 5.22 and figure 5.22.

Table 5.22: Validation Plan - Test Case 22

Test Case:	TC - 22
Expected behavior:	Client is expected to issue a REQUEST message to the server's address indicated in the configuration file. It will request the maximum configured addresses starting with the address specified in its configuration. The requested set belongs to the administratively corresponding block of this server and it contains less address than the maximum number the server is allowed to distribute per client, so it will acknowledge the requested set with an ACK message. Moreover, the server will accept renewals due to the enabled renewal option.
Real behavior:	As expected
Result:	Test passed

	Source	Destination	Protocol	Length	Time	Info
1	10:0f:ac:e0:00:01	10:0a:bc:de:f0:01	PALMA	36	0.000000000	REQUEST: SET:[start=1b:cb:00:00:00:00, cnt=100]
2	10:0a:bc:de:f0:01	10:0f:ac:e0:00:01	PALMA	40	0.000103893	ACK: SET:[start=1b:cb:00:00:00:00, cnt=100] lifetime=10 s
3	10:0f:ac:e0:00:01	10:0a:bc:de:f0:01	PALMA	36	9.007293199	REQUEST: SET:[start=1b:cb:00:00:00:00, cnt=100]
4	10:0a:bc:de:f0:01	10:0f:ac:e0:00:01	PALMA	40	9.007877723	ACK: SET:[start=1b:cb:00:00:00:00, cnt=100] lifetime=10 s
5	10:0f:ac:e0:00:01	10:0a:bc:de:f0:01	PALMA	36	18.010432120	REQUEST: SET:[start=1b:cb:00:00:00:00, cnt=100]
6	10:0a:bc:de:f0:01	10:0f:ac:e0:00:01	PALMA	40	18.010562968	ACK: SET:[start=1b:cb:00:00:00:00, cnt=100] lifetime=10 s
7	10:0f:ac:e0:00:01	10:0a:bc:de:f0:01	PALMA	36	27.027102848	REQUEST: SET:[start=1b:cb:00:00:00:00, cnt=100]
8	10:0a:bc:de:f0:01	10:0f:ac:e0:00:01	PALMA	40	27.027433301	ACK: SET:[start=1b:cb:00:00:00:00, cnt=100] lifetime=10 s

Figure 5.22: Wireshark - Test Case 22

Test Case 23

For this test, a server is executed in first place with the following configuration:

- Distributable set: 0x1a:ca:00:00:00:00 (unicast - 48-bit)
- Maximum number of addresses to assign: 1000
- Lifetime per assignation: 10 seconds
- Alternate set option: disabled
- Renewal option: disabled

Secondly a client with the server's address already known and a preassigned source address is executed. This one is requesting 100 addresses of the self-assign unicast set and has been configured with minimum number of addresses to assign equal to 1 and maximum equal to 100. Moreover, the renewal option for the client is enabled. The test duration is around 100 milliseconds. Results can be seen in table 5.23 and figure 5.23.

Table 5.23: Validation Plan - Test Case 23

Test Case:	TC - 23
Expected behavior:	Client is expected to issue a REQUEST message to the server's address indicated in the configuration. It will request the maximum configured addresses starting with the address specified in its configuration. The requested set does not belong to the administratively corresponding block of this server, so it will issue an ACK error message saying that the requested set is administratively disallowed, because the alternate set option is not enabled. Afterwards, the client will restart the protocol and the same will happen in a continuous loop.
Real behavior:	As expected
Result:	Test passed

	Source	Destination	Protocol	Length	Time	Info
1	10:0f:ac:e0:00:01	10:0a:bc:de:f0:01	PALMA	36	0.000000000	REQUEST: SET:[start=0a:00:00:00:00:00, cnt=100]
2	10:0a:bc:de:f0:01	10:0f:ac:e0:00:01	PALMA	26	0.000164659	ACK: Assignment rejected - requested address administratively disallowed
3	10:0f:ac:e0:00:01	10:0a:bc:de:f0:01	PALMA	36	0.000263179	REQUEST: SET:[start=0a:00:00:00:00:00, cnt=100]
4	10:0a:bc:de:f0:01	10:0f:ac:e0:00:01	PALMA	26	0.000414707	ACK: Assignment rejected - requested address administratively disallowed
5	10:0f:ac:e0:00:01	10:0a:bc:de:f0:01	PALMA	36	0.000603602	REQUEST: SET:[start=0a:00:00:00:00:00, cnt=100]
6	10:0a:bc:de:f0:01	10:0f:ac:e0:00:01	PALMA	26	0.000670648	ACK: Assignment rejected - requested address administratively disallowed
7	10:0f:ac:e0:00:01	10:0a:bc:de:f0:01	PALMA	36	0.000806581	REQUEST: SET:[start=0a:00:00:00:00:00, cnt=100]
8	10:0a:bc:de:f0:01	10:0f:ac:e0:00:01	PALMA	26	0.000929887	ACK: Assignment rejected - requested address administratively disallowed
9	10:0f:ac:e0:00:01	10:0a:bc:de:f0:01	PALMA	36	0.001151332	REQUEST: SET:[start=0a:00:00:00:00:00, cnt=100]
10	10:0a:bc:de:f0:01	10:0f:ac:e0:00:01	PALMA	26	0.001206023	ACK: Assignment rejected - requested address administratively disallowed
11	10:0f:ac:e0:00:01	10:0a:bc:de:f0:01	PALMA	36	0.001257510	REQUEST: SET:[start=0a:00:00:00:00:00, cnt=100]
12	10:0a:bc:de:f0:01	10:0f:ac:e0:00:01	PALMA	26	0.001415121	ACK: Assignment rejected - requested address administratively disallowed
13	10:0f:ac:e0:00:01	10:0a:bc:de:f0:01	PALMA	36	0.001610292	REQUEST: SET:[start=0a:00:00:00:00:00, cnt=100]
14	10:0a:bc:de:f0:01	10:0f:ac:e0:00:01	PALMA	26	0.001735180	ACK: Assignment rejected - requested address administratively disallowed
15	10:0f:ac:e0:00:01	10:0a:bc:de:f0:01	PALMA	36	0.001797493	REQUEST: SET:[start=0a:00:00:00:00:00, cnt=100]
16	10:0a:bc:de:f0:01	10:0f:ac:e0:00:01	PALMA	26	0.001964758	ACK: Assignment rejected - requested address administratively disallowed
17	10:0f:ac:e0:00:01	10:0a:bc:de:f0:01	PALMA	36	0.002022302	REQUEST: SET:[start=0a:00:00:00:00:00, cnt=100]
18	10:0a:bc:de:f0:01	10:0f:ac:e0:00:01	PALMA	26	0.002139788	ACK: Assignment rejected - requested address administratively disallowed
19	10:0f:ac:e0:00:01	10:0a:bc:de:f0:01	PALMA	36	0.002278578	REQUEST: SET:[start=0a:00:00:00:00:00, cnt=100]
20	10:0a:bc:de:f0:01	10:0f:ac:e0:00:01	PALMA	26	0.002378561	ACK: Assignment rejected - requested address administratively disallowed
21	10:0f:ac:e0:00:01	10:0a:bc:de:f0:01	PALMA	36	0.002599635	REQUEST: SET:[start=0a:00:00:00:00:00, cnt=100]
22	10:0a:bc:de:f0:01	10:0f:ac:e0:00:01	PALMA	26	0.002671227	ACK: Assignment rejected - requested address administratively disallowed
23	10:0f:ac:e0:00:01	10:0a:bc:de:f0:01	PALMA	36	0.002831602	REQUEST: SET:[start=0a:00:00:00:00:00, cnt=100]
24	10:0a:bc:de:f0:01	10:0f:ac:e0:00:01	PALMA	26	0.003019779	ACK: Assignment rejected - requested address administratively disallowed
25	10:0f:ac:e0:00:01	10:0a:bc:de:f0:01	PALMA	36	0.003142836	REQUEST: SET:[start=0a:00:00:00:00:00, cnt=100]
26	10:0a:bc:de:f0:01	10:0f:ac:e0:00:01	PALMA	26	0.003197839	ACK: Assignment rejected - requested address administratively disallowed
27	10:0f:ac:e0:00:01	10:0a:bc:de:f0:01	PALMA	36	0.003411426	REQUEST: SET:[start=0a:00:00:00:00:00, cnt=100]
28	10:0a:bc:de:f0:01	10:0f:ac:e0:00:01	PALMA	26	0.003535682	ACK: Assignment rejected - requested address administratively disallowed
29	10:0f:ac:e0:00:01	10:0a:bc:de:f0:01	PALMA	36	0.003981654	REQUEST: SET:[start=0a:00:00:00:00:00, cnt=100]
30	10:0a:bc:de:f0:01	10:0f:ac:e0:00:01	PALMA	26	0.004142934	ACK: Assignment rejected - requested address administratively disallowed

Figure 5.23: Wireshark - Test Case 23

Test Case 24

For this test, a server is executed in first place with the following configuration:

- Distributable set: 0x1b:cb:00:00:00:00 (multicast - 48-bit)
- Maximum number of addresses to assign: 5
- Lifetime per assignation: 10 seconds
- Alternate set option: disabled
- Renewal option: enabled

Secondly a client with the server's address already known and a preassigned source address is executed. This one is requesting 100 addresses of the server's distributable set and has been configured with minimum number of addresses to assign equal to 1 and maximum equal to 100. Moreover, the renewal option for the client is enabled. The test duration is around 100 milliseconds. Results can be seen in table 5.24 and figure 5.24.

Table 5.24: Validation Plan - Test Case 24

Test Case:	TC - 24
Expected behavior:	Client is expected to issue a REQUEST message to the server's address indicated in the configuration file. It will request the maximum configured addresses starting with the address specified in its configuration. The requested set belongs to the administratively corresponding block of this server but it contains many more addresses than the maximum number the server is allowed to distribute per client, so it will it will issue an ACK error message indicating that the requested set is too large, because the alternate set option is not enabled. Afterwards, the client will restart the protocol and the same will happen in a continuous loop.
Real behavior:	As expected
Result:	Test passed

	Source	Destination	Protocol	Length	Time	Info
1	10:0f:ac:e0:00:01	10:0a:bc:de:f0:01	PALMA	36	0.000000000	REQUEST: SET:[start=1b:cb:00:00:00:00, cnt=100]
2	10:0a:bc:de:f0:01	10:0f:ac:e0:00:01	PALMA	26	0.000059722	ACK: Assignment rejected - requested address set too large
3	10:0f:ac:e0:00:01	10:0a:bc:de:f0:01	PALMA	36	0.000084422	REQUEST: SET:[start=1b:cb:00:00:00:00, cnt=100]
4	10:0a:bc:de:f0:01	10:0f:ac:e0:00:01	PALMA	26	0.000095923	ACK: Assignment rejected - requested address set too large
5	10:0f:ac:e0:00:01	10:0a:bc:de:f0:01	PALMA	36	0.000107605	REQUEST: SET:[start=1b:cb:00:00:00:00, cnt=100]
6	10:0a:bc:de:f0:01	10:0f:ac:e0:00:01	PALMA	26	0.000116359	ACK: Assignment rejected - requested address set too large
7	10:0f:ac:e0:00:01	10:0a:bc:de:f0:01	PALMA	36	0.000126785	REQUEST: SET:[start=1b:cb:00:00:00:00, cnt=100]
8	10:0a:bc:de:f0:01	10:0f:ac:e0:00:01	PALMA	26	0.000134991	ACK: Assignment rejected - requested address set too large
9	10:0f:ac:e0:00:01	10:0a:bc:de:f0:01	PALMA	36	0.000145215	REQUEST: SET:[start=1b:cb:00:00:00:00, cnt=100]
10	10:0a:bc:de:f0:01	10:0f:ac:e0:00:01	PALMA	26	0.000153681	ACK: Assignment rejected - requested address set too large
11	10:0f:ac:e0:00:01	10:0a:bc:de:f0:01	PALMA	36	0.000163867	REQUEST: SET:[start=1b:cb:00:00:00:00, cnt=100]
12	10:0a:bc:de:f0:01	10:0f:ac:e0:00:01	PALMA	26	0.000172084	ACK: Assignment rejected - requested address set too large
13	10:0f:ac:e0:00:01	10:0a:bc:de:f0:01	PALMA	36	0.000182382	REQUEST: SET:[start=1b:cb:00:00:00:00, cnt=100]
14	10:0a:bc:de:f0:01	10:0f:ac:e0:00:01	PALMA	26	0.000190609	ACK: Assignment rejected - requested address set too large
15	10:0f:ac:e0:00:01	10:0a:bc:de:f0:01	PALMA	36	0.000200896	REQUEST: SET:[start=1b:cb:00:00:00:00, cnt=100]
16	10:0a:bc:de:f0:01	10:0f:ac:e0:00:01	PALMA	26	0.000209232	ACK: Assignment rejected - requested address set too large
17	10:0f:ac:e0:00:01	10:0a:bc:de:f0:01	PALMA	36	0.000219456	REQUEST: SET:[start=1b:cb:00:00:00:00, cnt=100]
18	10:0a:bc:de:f0:01	10:0f:ac:e0:00:01	PALMA	26	0.000227690	ACK: Assignment rejected - requested address set too large
19	10:0f:ac:e0:00:01	10:0a:bc:de:f0:01	PALMA	36	0.000237987	REQUEST: SET:[start=1b:cb:00:00:00:00, cnt=100]
20	10:0a:bc:de:f0:01	10:0f:ac:e0:00:01	PALMA	26	0.000246281	ACK: Assignment rejected - requested address set too large
21	10:0f:ac:e0:00:01	10:0a:bc:de:f0:01	PALMA	36	0.000256628	REQUEST: SET:[start=1b:cb:00:00:00:00, cnt=100]
22	10:0a:bc:de:f0:01	10:0f:ac:e0:00:01	PALMA	26	0.000264986	ACK: Assignment rejected - requested address set too large
23	10:0f:ac:e0:00:01	10:0a:bc:de:f0:01	PALMA	36	0.000275555	REQUEST: SET:[start=1b:cb:00:00:00:00, cnt=100]
24	10:0a:bc:de:f0:01	10:0f:ac:e0:00:01	PALMA	26	0.000283644	ACK: Assignment rejected - requested address set too large
25	10:0f:ac:e0:00:01	10:0a:bc:de:f0:01	PALMA	36	0.000293833	REQUEST: SET:[start=1b:cb:00:00:00:00, cnt=100]
26	10:0a:bc:de:f0:01	10:0f:ac:e0:00:01	PALMA	26	0.000301896	ACK: Assignment rejected - requested address set too large
27	10:0f:ac:e0:00:01	10:0a:bc:de:f0:01	PALMA	36	0.000312009	REQUEST: SET:[start=1b:cb:00:00:00:00, cnt=100]
28	10:0a:bc:de:f0:01	10:0f:ac:e0:00:01	PALMA	26	0.000320125	ACK: Assignment rejected - requested address set too large
29	10:0f:ac:e0:00:01	10:0a:bc:de:f0:01	PALMA	36	0.000330384	REQUEST: SET:[start=1b:cb:00:00:00:00, cnt=100]
30	10:0a:bc:de:f0:01	10:0f:ac:e0:00:01	PALMA	26	0.000338509	ACK: Assignment rejected - requested address set too large

Figure 5.24: Wireshark - Test Case 24

Test Case 25

In this test, a server is executed in first place with the following configuration:

- Distributable set: 0x1a:ca:00:00:00:00 (unicast - 48-bit)
- Maximum number of addresses to assign: 5
- Lifetime per assignation: 10 seconds
- Alternate set option: enabled

Secondly a client with the server's address already known and a preassigned source address is executed. This one is requesting 0 addresses of the self-assign unicast set and has been configured with minimum number of addresses to assign equal to 10 and maximum equal to 100. The test duration is around 100 milliseconds. Results can be seen in table 5.25 and figure 5.25.

Table 5.25: Validation Plan - Test Case 25

Test Case:	TC - 25
Expected behavior:	Client is expected to issue a null (requesting 0 addresses) REQUEST message to the server's address indicated in the configuration file. The server will offer as many addresses as configured of the requesting address type with an ACK message, due to the enabled alternate set option. Client will not accept this offer inside the ACK message because it contains less addresses than the minimum configured in the client's configuration file, so it will issue a RELEASE message to the server. Afterwards, the client will restart the protocol and the same will happen in a continuous loop.
Real behavior:	As expected
Result:	Test passed

	Source	Destination	Protocol	Length	Time	Info
1	10:0f:ac:e0:00:01	10:0a:bc:de:f0:01	PALMA	36	0.000000000	REQUEST: SET:[start=0a:00:00:00:00:00, cnt=0]
2	10:0a:bc:de:f0:01	10:0f:ac:e0:00:01	PALMA	40	0.000131436	ACK: SET:[start=1a:ca:00:00:00:00, cnt=5] lifetime=10 s
3	10:0f:ac:e0:00:01	10:0a:bc:de:f0:01	PALMA	36	0.000252286	RELEASE: SET:[start=1a:ca:00:00:00:00, cnt=5]
4	10:0f:ac:e0:00:01	10:0a:bc:de:f0:01	PALMA	36	0.000287438	REQUEST: SET:[start=0a:00:00:00:00:00, cnt=0]
5	10:0a:bc:de:f0:01	10:0f:ac:e0:00:01	PALMA	40	0.000340861	ACK: SET:[start=1a:ca:00:00:00:00, cnt=5] lifetime=10 s
6	10:0f:ac:e0:00:01	10:0a:bc:de:f0:01	PALMA	36	0.000416800	RELEASE: SET:[start=1a:ca:00:00:00:00, cnt=5]
7	10:0f:ac:e0:00:01	10:0a:bc:de:f0:01	PALMA	36	0.000446647	REQUEST: SET:[start=0a:00:00:00:00:00, cnt=0]
8	10:0a:bc:de:f0:01	10:0f:ac:e0:00:01	PALMA	40	0.000483312	ACK: SET:[start=1a:ca:00:00:00:00, cnt=5] lifetime=10 s
9	10:0f:ac:e0:00:01	10:0a:bc:de:f0:01	PALMA	36	0.000549384	RELEASE: SET:[start=1a:ca:00:00:00:00, cnt=5]
10	10:0f:ac:e0:00:01	10:0a:bc:de:f0:01	PALMA	36	0.000568712	REQUEST: SET:[start=0a:00:00:00:00:00, cnt=0]
11	10:0a:bc:de:f0:01	10:0f:ac:e0:00:01	PALMA	40	0.000583894	ACK: SET:[start=1a:ca:00:00:00:00, cnt=5] lifetime=10 s
12	10:0f:ac:e0:00:01	10:0a:bc:de:f0:01	PALMA	36	0.000634582	RELEASE: SET:[start=1a:ca:00:00:00:00, cnt=5]
13	10:0f:ac:e0:00:01	10:0a:bc:de:f0:01	PALMA	36	0.000662373	REQUEST: SET:[start=0a:00:00:00:00:00, cnt=0]
14	10:0a:bc:de:f0:01	10:0f:ac:e0:00:01	PALMA	40	0.000712714	ACK: SET:[start=1a:ca:00:00:00:00, cnt=5] lifetime=10 s
15	10:0f:ac:e0:00:01	10:0a:bc:de:f0:01	PALMA	36	0.000782147	RELEASE: SET:[start=1a:ca:00:00:00:00, cnt=5]
16	10:0f:ac:e0:00:01	10:0a:bc:de:f0:01	PALMA	36	0.000809092	REQUEST: SET:[start=0a:00:00:00:00:00, cnt=0]
17	10:0a:bc:de:f0:01	10:0f:ac:e0:00:01	PALMA	40	0.000863519	ACK: SET:[start=1a:ca:00:00:00:00, cnt=5] lifetime=10 s
18	10:0f:ac:e0:00:01	10:0a:bc:de:f0:01	PALMA	36	0.000934196	RELEASE: SET:[start=1a:ca:00:00:00:00, cnt=5]
19	10:0f:ac:e0:00:01	10:0a:bc:de:f0:01	PALMA	36	0.000953359	REQUEST: SET:[start=0a:00:00:00:00:00, cnt=0]
20	10:0a:bc:de:f0:01	10:0f:ac:e0:00:01	PALMA	40	0.001002753	ACK: SET:[start=1a:ca:00:00:00:00, cnt=5] lifetime=10 s
21	10:0f:ac:e0:00:01	10:0a:bc:de:f0:01	PALMA	36	0.001073054	RELEASE: SET:[start=1a:ca:00:00:00:00, cnt=5]
22	10:0f:ac:e0:00:01	10:0a:bc:de:f0:01	PALMA	36	0.001099882	REQUEST: SET:[start=0a:00:00:00:00:00, cnt=0]
23	10:0a:bc:de:f0:01	10:0f:ac:e0:00:01	PALMA	40	0.001152672	ACK: SET:[start=1a:ca:00:00:00:00, cnt=5] lifetime=10 s
24	10:0f:ac:e0:00:01	10:0a:bc:de:f0:01	PALMA	36	0.001249357	RELEASE: SET:[start=1a:ca:00:00:00:00, cnt=5]
25	10:0f:ac:e0:00:01	10:0a:bc:de:f0:01	PALMA	36	0.001283207	REQUEST: SET:[start=0a:00:00:00:00:00, cnt=0]
26	10:0a:bc:de:f0:01	10:0f:ac:e0:00:01	PALMA	40	0.001315557	ACK: SET:[start=1a:ca:00:00:00:00, cnt=5] lifetime=10 s
27	10:0f:ac:e0:00:01	10:0a:bc:de:f0:01	PALMA	36	0.001390924	RELEASE: SET:[start=1a:ca:00:00:00:00, cnt=5]
28	10:0f:ac:e0:00:01	10:0a:bc:de:f0:01	PALMA	36	0.001417757	REQUEST: SET:[start=0a:00:00:00:00:00, cnt=0]
29	10:0a:bc:de:f0:01	10:0f:ac:e0:00:01	PALMA	40	0.001471263	ACK: SET:[start=1a:ca:00:00:00:00, cnt=5] lifetime=10 s
30	10:0f:ac:e0:00:01	10:0a:bc:de:f0:01	PALMA	36	0.001542748	RELEASE: SET:[start=1a:ca:00:00:00:00, cnt=5]

Figure 5.25: Wireshark - Test Case 25

Test Case 26

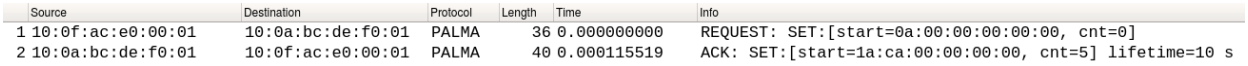
In this test, a server is executed in first place with the following configuration:

- Distributable set: 0x1a:ca:00:00:00:00 (unicast - 48-bit)
- Maximum number of addresses to assign: 5
- Lifetime per assignation: 10 seconds
- Alternate set option: enabled

Secondly a client with the server's address already known and a preassigned source address is executed. This one is requesting 0 addresses of the self-assign unicast set and has been configured with minimum number of addresses to assign equal to 1 and maximum equal to 100. The test duration is around 100 milliseconds. Results can be seen in table 5.26 and figure 5.26.

Table 5.26: Validation Plan - Test Case 26

Test Case:	TC - 26
Expected behavior:	Client is expected to issue a null (requesting 0 addresses) REQUEST message to the server's address indicated in the configuration file. The server will offer as many addresses as configured of the requesting address type with an ACK message, due to the enabled alternate set option. Client will accept this offer inside the ACK message because it contains more addresses than the minimum configured in the client's configuration file. No more messages are expected to be seen.
Real behavior:	As expected
Result:	Test passed



	Source	Destination	Protocol	Length	Time	Info
1	10:0f:ac:e0:00:01	10:0a:bc:de:f0:01	PALMA	36	0.000000000	REQUEST: SET:[start=0a:00:00:00:00:00, cnt=0]
2	10:0a:bc:de:f0:01	10:0f:ac:e0:00:01	PALMA	40	0.000115519	ACK: SET:[start=1a:ca:00:00:00:00, cnt=5] lifetime=10 s

Figure 5.26: Wireshark - Test Case 26

Test Case 27

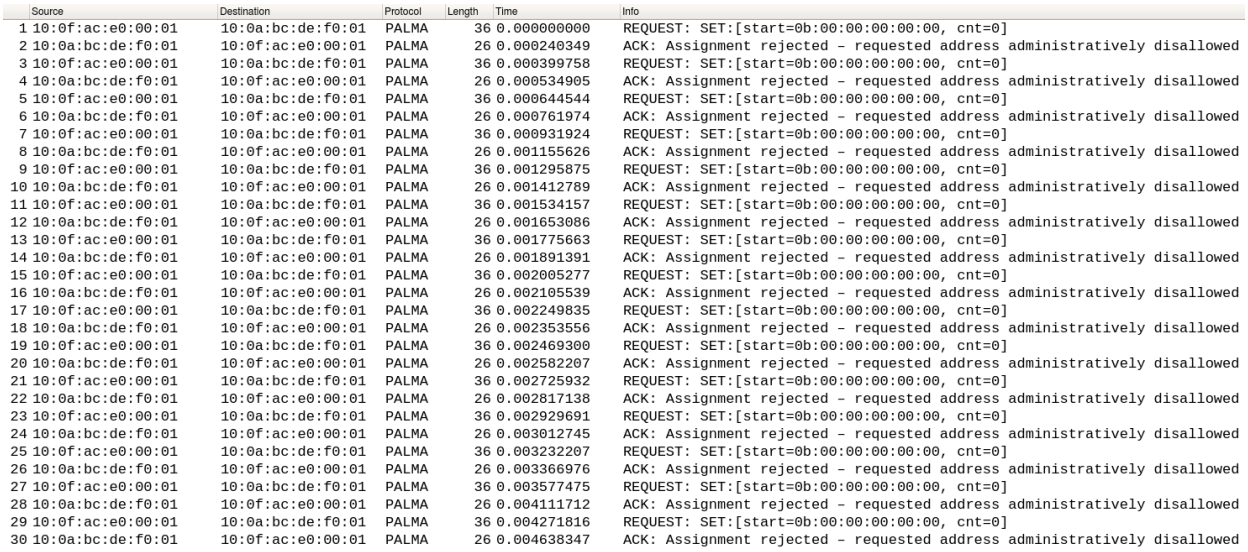
In this test, a server is executed in first place with the following configuration:

- Distributable set: 0x1a:ca:00:00:00:00 (unicast - 48-bit)
- Maximum number of addresses to assign: 5
- Lifetime per assignation: 10 seconds
- Alternate set option: disabled

Secondly a client with the server's address already known and a preassigned source address is executed. This one is requesting 0 addresses of the self-assign unicast set and has been configured with minimum number of addresses to assign equal to 10 and maximum equal to 100. The test duration is around 100 milliseconds. Results can be seen in table 5.27 and figure 5.27.

Table 5.27: Validation Plan - Test Case 27

Test Case:	TC - 27
Expected behavior:	Client is expected to issue a null (requesting 0 addresses) REQUEST message to the server's address indicated in the configuration file. The server does not have the alternate set option enabled so an ACK error message is expected to be sent to the client stating that the requested set is administratively disallowed. Afterwards, the client will restart the protocol and the same will happen in a continuous loop.
Real behavior:	As expected
Result:	Test passed



	Source	Destination	Protocol	Length	Time	Info
1	10:0f:ac:e0:00:01	10:0a:bc:de:f0:01	PALMA	36	0.000000000	REQUEST: SET:[start=0b:00:00:00:00:00, cnt=0]
2	10:0a:bc:de:f0:01	10:0f:ac:e0:00:01	PALMA	26	0.000240349	ACK: Assignment rejected - requested address administratively disallowed
3	10:0f:ac:e0:00:01	10:0a:bc:de:f0:01	PALMA	36	0.000399758	REQUEST: SET:[start=0b:00:00:00:00:00, cnt=0]
4	10:0a:bc:de:f0:01	10:0f:ac:e0:00:01	PALMA	26	0.000534905	ACK: Assignment rejected - requested address administratively disallowed
5	10:0f:ac:e0:00:01	10:0a:bc:de:f0:01	PALMA	36	0.000644544	REQUEST: SET:[start=0b:00:00:00:00:00, cnt=0]
6	10:0a:bc:de:f0:01	10:0f:ac:e0:00:01	PALMA	26	0.000761974	ACK: Assignment rejected - requested address administratively disallowed
7	10:0f:ac:e0:00:01	10:0a:bc:de:f0:01	PALMA	36	0.000931924	REQUEST: SET:[start=0b:00:00:00:00:00, cnt=0]
8	10:0a:bc:de:f0:01	10:0f:ac:e0:00:01	PALMA	26	0.001155626	ACK: Assignment rejected - requested address administratively disallowed
9	10:0f:ac:e0:00:01	10:0a:bc:de:f0:01	PALMA	36	0.001295875	REQUEST: SET:[start=0b:00:00:00:00:00, cnt=0]
10	10:0a:bc:de:f0:01	10:0f:ac:e0:00:01	PALMA	26	0.001412789	ACK: Assignment rejected - requested address administratively disallowed
11	10:0f:ac:e0:00:01	10:0a:bc:de:f0:01	PALMA	36	0.001534157	REQUEST: SET:[start=0b:00:00:00:00:00, cnt=0]
12	10:0a:bc:de:f0:01	10:0f:ac:e0:00:01	PALMA	26	0.001653086	ACK: Assignment rejected - requested address administratively disallowed
13	10:0f:ac:e0:00:01	10:0a:bc:de:f0:01	PALMA	36	0.001775663	REQUEST: SET:[start=0b:00:00:00:00:00, cnt=0]
14	10:0a:bc:de:f0:01	10:0f:ac:e0:00:01	PALMA	26	0.001891391	ACK: Assignment rejected - requested address administratively disallowed
15	10:0f:ac:e0:00:01	10:0a:bc:de:f0:01	PALMA	36	0.002005277	REQUEST: SET:[start=0b:00:00:00:00:00, cnt=0]
16	10:0a:bc:de:f0:01	10:0f:ac:e0:00:01	PALMA	26	0.002105539	ACK: Assignment rejected - requested address administratively disallowed
17	10:0f:ac:e0:00:01	10:0a:bc:de:f0:01	PALMA	36	0.002249835	REQUEST: SET:[start=0b:00:00:00:00:00, cnt=0]
18	10:0a:bc:de:f0:01	10:0f:ac:e0:00:01	PALMA	26	0.002353556	ACK: Assignment rejected - requested address administratively disallowed
19	10:0f:ac:e0:00:01	10:0a:bc:de:f0:01	PALMA	36	0.002469300	REQUEST: SET:[start=0b:00:00:00:00:00, cnt=0]
20	10:0a:bc:de:f0:01	10:0f:ac:e0:00:01	PALMA	26	0.002582207	ACK: Assignment rejected - requested address administratively disallowed
21	10:0f:ac:e0:00:01	10:0a:bc:de:f0:01	PALMA	36	0.002725932	REQUEST: SET:[start=0b:00:00:00:00:00, cnt=0]
22	10:0a:bc:de:f0:01	10:0f:ac:e0:00:01	PALMA	26	0.002817138	ACK: Assignment rejected - requested address administratively disallowed
23	10:0f:ac:e0:00:01	10:0a:bc:de:f0:01	PALMA	36	0.002929691	REQUEST: SET:[start=0b:00:00:00:00:00, cnt=0]
24	10:0a:bc:de:f0:01	10:0f:ac:e0:00:01	PALMA	26	0.003012745	ACK: Assignment rejected - requested address administratively disallowed
25	10:0f:ac:e0:00:01	10:0a:bc:de:f0:01	PALMA	36	0.003232207	REQUEST: SET:[start=0b:00:00:00:00:00, cnt=0]
26	10:0a:bc:de:f0:01	10:0f:ac:e0:00:01	PALMA	26	0.003366976	ACK: Assignment rejected - requested address administratively disallowed
27	10:0f:ac:e0:00:01	10:0a:bc:de:f0:01	PALMA	36	0.003577475	REQUEST: SET:[start=0b:00:00:00:00:00, cnt=0]
28	10:0a:bc:de:f0:01	10:0f:ac:e0:00:01	PALMA	26	0.004111712	ACK: Assignment rejected - requested address administratively disallowed
29	10:0f:ac:e0:00:01	10:0a:bc:de:f0:01	PALMA	36	0.004271816	REQUEST: SET:[start=0b:00:00:00:00:00, cnt=0]
30	10:0a:bc:de:f0:01	10:0f:ac:e0:00:01	PALMA	26	0.004638347	ACK: Assignment rejected - requested address administratively disallowed

Figure 5.27: Wireshark - Test Case 27

Test Case 28

This test is done to see the fault tolerance of the program. For this we have forced the following error:

- Force a client to have a preassigned address from the range of the PALMA random source addresses and issue a REQUEST to a previously known address server. For a deeper understanding see figure 5.28.

The test duration is around 100 milliseconds. Results can be seen in table 5.28 and figure 5.28.

Table 5.28: Validation Plan - Test Case 28

Test Case:	TC - 28
Expected behavior:	Client is expected to issue a REQUEST message to the server's address indicated in the configuration file. The source address of this message is the one the user has configured, in this case one of the ranges of the PALMA random source addresses. Server receiving this request issues instantly an ACK error message stating that the assignment was rejected because of other administrative reasons, as it would happen in any case a client refers to a server with an invalid source address. Afterwards, the client will restart the protocol and the same will happen in a continuous loop.
Real behavior:	As expected
Result:	Test passed

	Source	Destination	Protocol	Length	Time	Info
1	2a:00:00:00:00:01	10:0a:bc:de:f0:01	PALMA	36	0.000000000	REQUEST: SET:[start=0a:00:00:00:00:00, cnt=100]
2	10:0a:bc:de:f0:01	2a:00:00:00:00:01	PALMA	26	0.000071399	ACK: Assignment rejected - other administrative reasons
3	2a:00:00:00:00:01	10:0a:bc:de:f0:01	PALMA	36	0.000128066	REQUEST: SET:[start=0a:00:00:00:00:00, cnt=100]
4	10:0a:bc:de:f0:01	2a:00:00:00:00:01	PALMA	26	0.000151409	ACK: Assignment rejected - other administrative reasons
5	2a:00:00:00:00:01	10:0a:bc:de:f0:01	PALMA	36	0.000180198	REQUEST: SET:[start=0a:00:00:00:00:00, cnt=100]
6	10:0a:bc:de:f0:01	2a:00:00:00:00:01	PALMA	26	0.000198137	ACK: Assignment rejected - other administrative reasons
7	2a:00:00:00:00:01	10:0a:bc:de:f0:01	PALMA	36	0.000223286	REQUEST: SET:[start=0a:00:00:00:00:00, cnt=100]
8	10:0a:bc:de:f0:01	2a:00:00:00:00:01	PALMA	26	0.000240016	ACK: Assignment rejected - other administrative reasons
9	2a:00:00:00:00:01	10:0a:bc:de:f0:01	PALMA	36	0.000264217	REQUEST: SET:[start=0a:00:00:00:00:00, cnt=100]
10	10:0a:bc:de:f0:01	2a:00:00:00:00:01	PALMA	26	0.000280121	ACK: Assignment rejected - other administrative reasons
11	2a:00:00:00:00:01	10:0a:bc:de:f0:01	PALMA	36	0.000303418	REQUEST: SET:[start=0a:00:00:00:00:00, cnt=100]
12	10:0a:bc:de:f0:01	2a:00:00:00:00:01	PALMA	26	0.000319334	ACK: Assignment rejected - other administrative reasons
13	2a:00:00:00:00:01	10:0a:bc:de:f0:01	PALMA	36	0.000385944	REQUEST: SET:[start=0a:00:00:00:00:00, cnt=100]
14	10:0a:bc:de:f0:01	2a:00:00:00:00:01	PALMA	26	0.000414502	ACK: Assignment rejected - other administrative reasons
15	2a:00:00:00:00:01	10:0a:bc:de:f0:01	PALMA	36	0.000442940	REQUEST: SET:[start=0a:00:00:00:00:00, cnt=100]
16	10:0a:bc:de:f0:01	2a:00:00:00:00:01	PALMA	26	0.000462161	ACK: Assignment rejected - other administrative reasons
17	2a:00:00:00:00:01	10:0a:bc:de:f0:01	PALMA	36	0.000487625	REQUEST: SET:[start=0a:00:00:00:00:00, cnt=100]
18	10:0a:bc:de:f0:01	2a:00:00:00:00:01	PALMA	26	0.000504516	ACK: Assignment rejected - other administrative reasons
19	2a:00:00:00:00:01	10:0a:bc:de:f0:01	PALMA	36	0.000528961	REQUEST: SET:[start=0a:00:00:00:00:00, cnt=100]
20	10:0a:bc:de:f0:01	2a:00:00:00:00:01	PALMA	26	0.000545319	ACK: Assignment rejected - other administrative reasons
21	2a:00:00:00:00:01	10:0a:bc:de:f0:01	PALMA	36	0.000570466	REQUEST: SET:[start=0a:00:00:00:00:00, cnt=100]
22	10:0a:bc:de:f0:01	2a:00:00:00:00:01	PALMA	26	0.000586611	ACK: Assignment rejected - other administrative reasons
23	2a:00:00:00:00:01	10:0a:bc:de:f0:01	PALMA	36	0.000610947	REQUEST: SET:[start=0a:00:00:00:00:00, cnt=100]
24	10:0a:bc:de:f0:01	2a:00:00:00:00:01	PALMA	26	0.000627609	ACK: Assignment rejected - other administrative reasons
25	2a:00:00:00:00:01	10:0a:bc:de:f0:01	PALMA	36	0.000652403	REQUEST: SET:[start=0a:00:00:00:00:00, cnt=100]
26	10:0a:bc:de:f0:01	2a:00:00:00:00:01	PALMA	26	0.000668529	ACK: Assignment rejected - other administrative reasons
27	2a:00:00:00:00:01	10:0a:bc:de:f0:01	PALMA	36	0.000692427	REQUEST: SET:[start=0a:00:00:00:00:00, cnt=100]
28	10:0a:bc:de:f0:01	2a:00:00:00:00:01	PALMA	26	0.000708696	ACK: Assignment rejected - other administrative reasons
29	2a:00:00:00:00:01	10:0a:bc:de:f0:01	PALMA	36	0.000733032	REQUEST: SET:[start=0a:00:00:00:00:00, cnt=100]
30	10:0a:bc:de:f0:01	2a:00:00:00:00:01	PALMA	26	0.000749053	ACK: Assignment rejected - other administrative reasons

Figure 5.28: Wireshark - Test Case 28

Test Case 29

This test is done to see the fault tolerance of the program. For this we have forced the following error:

- Make two clients self-assign two overlapped sets by forcing a link (the one of the second client) go down for a certain period of time. For a deeper understanding see figure 5.29 where the network packets corresponding to the second client appear from packet number 6 including this one.
- First client has been configured to claim 200 address set starting from the address 0x0a:00:00:00:00:00 with a minimum number of addresses to assign equal to 5 and maximum equal to 10.
- Second client has been configured to claim 200 address set starting from the address 0x0a:00:00:00:00:05 with a minimum number of addresses to assign equal to 5 and maximum equal to 10.
- Links go up before the second ANNOUNCE of the second client.

The test duration is around 120 seconds. Results can be seen in table 5.29 and figure 5.29.

Table 5.29: Validation Plan - Test Case 29

Test Case:	TC - 29
Expected behavior:	When network links are up again, both clients have self-assigned 16 unicast addresses from the PALMA self-assign unicast claiming set. The second client receives an ANNOUNCE coming from the first client, notices the conflict, registers the announced set in its database and goes directly to START state. The second client cannot resolve the conflict internally because the first address of its self-assigned set is being announced by the other client, therefore no dispensable addresses exist. From this point, more altercation is expected.
Real behavior:	As expected
Result:	Test passed

	Source	Destination	Protocol	Length	Time	Info
1	2a:00:6a:b9:13:72	PALMA-MULTICAST	PALMA	36	0.000000000	DISCOVER: SET:[start=0a:00:00:00:00:00, cnt=200]
2	2a:00:8b:27:0d:d6	PALMA-MULTICAST	PALMA	36	0.503558868	DISCOVER: SET:[start=0a:00:00:00:00:00, cnt=200]
3	2a:00:93:f7:be:d8	PALMA-MULTICAST	PALMA	36	1.033911784	DISCOVER: SET:[start=0a:00:00:00:00:00, cnt=200]
4	0a:00:00:00:00:03	PALMA-MULTICAST	PALMA	40	0.970682194	ANNOUNCE: SET:[start=0a:00:00:00:00:00, cnt=10] lifetime=600 s
5	0a:00:00:00:00:03	PALMA-MULTICAST	PALMA	40	32.233650859	ANNOUNCE: SET:[start=0a:00:00:00:00:00, cnt=10] lifetime=569 s
6	2a:00:60:7e:5c:84	PALMA-MULTICAST	PALMA	36	32.234061008	DISCOVER: SET:[start=0a:00:00:00:00:0a, cnt=195]
7	2a:00:1c:be:1d:da	PALMA-MULTICAST	PALMA	36	32.816921030	DISCOVER: SET:[start=0a:00:00:00:00:0a, cnt=195]
8	2a:00:b7:4c:ba:b5	PALMA-MULTICAST	PALMA	36	33.334434030	DISCOVER: SET:[start=0a:00:00:00:00:0a, cnt=195]
9	0a:00:00:00:00:0f	PALMA-MULTICAST	PALMA	40	33.907408859	ANNOUNCE: SET:[start=0a:00:00:00:00:0a, cnt=10] lifetime=600 s
10	0a:00:00:00:00:03	PALMA-MULTICAST	PALMA	40	63.142422410	ANNOUNCE: SET:[start=0a:00:00:00:00:00, cnt=10] lifetime=538 s
11	0a:00:00:00:00:0f	PALMA-MULTICAST	PALMA	40	64.072071117	ANNOUNCE: SET:[start=0a:00:00:00:00:0a, cnt=10] lifetime=570 s
12	0a:00:00:00:00:03	PALMA-MULTICAST	PALMA	40	94.692861354	ANNOUNCE: SET:[start=0a:00:00:00:00:00, cnt=10] lifetime=506 s
13	0a:00:00:00:00:0f	PALMA-MULTICAST	PALMA	40	95.072468454	ANNOUNCE: SET:[start=0a:00:00:00:00:0a, cnt=10] lifetime=539 s

Figure 5.29: Wireshark - Test Case 29

Test Case 30

This test is done to see the fault tolerance of the program. For this we have forced the following error:

- Make two clients self-assign two overlapped sets by forcing a link (the one of the second client) go down for a certain period of time. For a deeper understanding see figure

5.30 where the network packets corresponding to the second client appear from packet number 6 including this one.

- First client has been configured to claim 200 address set starting from the address 0x0a:00:00:00:00:05 with a minimum number of addresses to assign equal to 5 and maximum equal to 10.
- Second client has been configured to claim 200 address set starting from the address 0x0a:00:00:00:00:00 with a minimum number of addresses to assign equal to 5 and maximum equal to 10.
- Links go up before the second ANNOUNCE of the second client.

The test duration is around 120 seconds. Results can be seen in table 5.30 and figure 5.30.

Table 5.30: Validation Plan - Test Case 30

Test Case:	TC - 30
Expected behavior:	When network links are up again, both clients have self-assigned 16 unicast addresses from the PALMA self-assign unicast claiming set. The second client receives an ANNOUNCE coming from the first client, notices the conflict, registers the announced set in its database and tries to resolve it. The second client can resolve the conflict internally because it can remove all the dispensable addresses from its assigned set and no more conflict exists. This action is expected to be seen in the posterior ANNOUNCE messages of the second client.
Real behavior:	As expected
Result:	Test passed

	Source	Destination	Protocol	Length	Time	Info
1	2a:00:7a:83:e0:f8	PALMA-MULTICAST	PALMA	36	0.000000000	DISCOVER: SET:[start=0a:00:00:00:00:05, cnt=200]
2	2a:00:ad:2b:ba:48	PALMA-MULTICAST	PALMA	36	0.560463040	DISCOVER: SET:[start=0a:00:00:00:00:05, cnt=200]
3	2a:00:58:cb:b8:26	PALMA-MULTICAST	PALMA	36	1.150884334	DISCOVER: SET:[start=0a:00:00:00:00:05, cnt=200]
4	0a:00:00:00:00:0b	PALMA-MULTICAST	PALMA	40	1.739570949	ANNOUNCE: SET:[start=0a:00:00:00:00:05, cnt=10] lifetime=600 s
5	0a:00:00:00:00:0b	PALMA-MULTICAST	PALMA	40	31.825922972	ANNOUNCE: SET:[start=0a:00:00:00:00:05, cnt=10] lifetime=570 s
6	0a:00:00:00:00:03	PALMA-MULTICAST	PALMA	40	33.876957256	ANNOUNCE: SET:[start=0a:00:00:00:00:00, cnt=5] lifetime=569 s
7	0a:00:00:00:00:0b	PALMA-MULTICAST	PALMA	40	62.483726537	ANNOUNCE: SET:[start=0a:00:00:00:00:05, cnt=10] lifetime=539 s
8	0a:00:00:00:00:03	PALMA-MULTICAST	PALMA	40	65.448616926	ANNOUNCE: SET:[start=0a:00:00:00:00:00, cnt=5] lifetime=538 s
9	0a:00:00:00:00:0b	PALMA-MULTICAST	PALMA	40	93.433018800	ANNOUNCE: SET:[start=0a:00:00:00:00:05, cnt=10] lifetime=508 s
10	0a:00:00:00:00:03	PALMA-MULTICAST	PALMA	40	96.578954034	ANNOUNCE: SET:[start=0a:00:00:00:00:00, cnt=5] lifetime=507 s

Figure 5.30: Wireshark - Test Case 30

Test Case 31

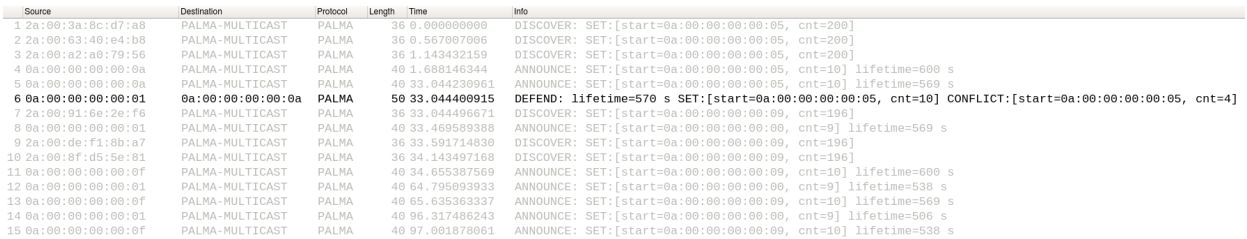
This test is done to see the fault tolerance of the program. For this we have forced the following error:

- Make two clients self-assign two overlapped sets by forcing a link (the one of the second client) go down for a certain period of time. For a deeper understanding see figure 5.30 where the network packets corresponding to the second client appear from packet number 6 including this one.
- First client has been configured to claim 200 address set starting from the address 0x0a:00:00:00:00:05 with a minimum number of addresses to assign equal to 5 and maximum equal to 10.
- Second client has been configured to claim 200 address set starting from the address 0x0a:00:00:00:00:00 with a minimum number of addresses to assign equal to 9 and maximum equal to 10.
- Links go up before the second ANNOUNCE of the second client.

The test duration is around 120 seconds. Results can be seen in table 5.31 and figure 5.31.

Table 5.31: Validation Plan - Test Case 31

Test Case:	TC - 31
Expected behavior:	When network links are up again, both clients have self-assigned 16 unicast addresses from the PALMA self-assign unicast claiming set. The second client receives an ANNOUNCE coming from the first client, notices the conflict, registers the announced set in its database and tries to resolve it. The second client can resolve the conflict internally because it can remove all the dispensable addresses from its assigned set but conflict still exists, so it must issue a DEFEND message defending the non-dispensable addresses of its assigned set. The first client, goes to START state when it receives the DEFEND message, and from this point no more altercation is expected.
Real behavior:	As expected
Result:	Test passed



	Source	Destination	Protocol	Length	Time	Info
1	2a:00:3a:8c:d7:a8	PALMA-MULTICAST	PALMA	36	0.000000000	DISCOVER: SET:[start=0a:00:00:00:00:05, cnt=200]
2	2a:00:63:40:e4:b8	PALMA-MULTICAST	PALMA	36	0.567007006	DISCOVER: SET:[start=0a:00:00:00:00:05, cnt=200]
3	2a:00:a2:a0:79:56	PALMA-MULTICAST	PALMA	36	1.143432159	DISCOVER: SET:[start=0a:00:00:00:00:05, cnt=200]
4	0a:00:00:00:00:0a	PALMA-MULTICAST	PALMA	40	1.688146344	ANNOUNCE: SET:[start=0a:00:00:00:00:05, cnt=10] lifetime=600 s
5	0a:00:00:00:00:0a	PALMA-MULTICAST	PALMA	40	33.044230961	ANNOUNCE: SET:[start=0a:00:00:00:00:05, cnt=10] lifetime=569 s
6	0a:00:00:00:00:01	0a:00:00:00:00:0a	PALMA	50	33.044400915	DEFEND: lifetime=570 s SET:[start=0a:00:00:00:00:05, cnt=10] CONFLICT:[start=0a:00:00:00:00:05, cnt=4]
7	2a:00:91:6e:2e:f6	PALMA-MULTICAST	PALMA	36	33.044496671	DISCOVER: SET:[start=0a:00:00:00:00:09, cnt=196]
8	0a:00:00:00:00:01	PALMA-MULTICAST	PALMA	40	33.469589388	ANNOUNCE: SET:[start=0a:00:00:00:00:00, cnt=9] lifetime=569 s
9	2a:00:de:f1:8b:a7	PALMA-MULTICAST	PALMA	36	33.591714830	DISCOVER: SET:[start=0a:00:00:00:00:09, cnt=196]
10	2a:00:8f:d5:5e:81	PALMA-MULTICAST	PALMA	36	34.143497168	DISCOVER: SET:[start=0a:00:00:00:00:09, cnt=196]
11	0a:00:00:00:00:0f	PALMA-MULTICAST	PALMA	40	34.655387569	ANNOUNCE: SET:[start=0a:00:00:00:00:09, cnt=10] lifetime=600 s
12	0a:00:00:00:00:01	PALMA-MULTICAST	PALMA	40	64.795093933	ANNOUNCE: SET:[start=0a:00:00:00:00:00, cnt=9] lifetime=538 s
13	0a:00:00:00:00:0f	PALMA-MULTICAST	PALMA	40	65.635363337	ANNOUNCE: SET:[start=0a:00:00:00:00:09, cnt=10] lifetime=569 s
14	0a:00:00:00:00:01	PALMA-MULTICAST	PALMA	40	96.317486243	ANNOUNCE: SET:[start=0a:00:00:00:00:00, cnt=9] lifetime=506 s
15	0a:00:00:00:00:0f	PALMA-MULTICAST	PALMA	40	97.001878061	ANNOUNCE: SET:[start=0a:00:00:00:00:09, cnt=10] lifetime=538 s

Figure 5.31: Wireshark - Test Case 31

Test Case 32

This test is done to see the fault tolerance of the program. For this we have forced the following error:

- Make two clients self-assign two overlapped sets by forcing a link (the one of the second client) go down for a certain period of time. For a deeper understanding see figure

5.30 where the network packets corresponding to the second client appear from packet number 6 including this one.

- First client has been configured to claim 200 address set starting from the address 0x0a:00:00:00:00:05 with a minimum number of addresses to assign equal to 5 and maximum equal to 10.
- Second client has been configured to claim 200 address set starting from the address 0x0a:00:00:00:00:00 with a minimum number of addresses to assign equal to 10 and maximum equal to 10.
- Links go up before the second ANNOUNCE of the second client.

The test duration is around 120 seconds. Results can be seen in table 5.32 and figure 5.32.

Table 5.32: Validation Plan - Test Case 32

Test Case:	TC - 32
Expected behavior:	When network links are up again, both clients have self-assigned 16 unicast addresses from the PALMA self-assign unicast claiming set. The second client receives an ANNOUNCE coming from the first client, notices the total conflict, registers the announced set in its database and goes directly to START state. The second client cannot resolve the conflict internally because the conflict is total. From this point, more altercation is expected.
Real behavior:	As expected
Result:	Test passed

	Source	Destination	Protocol	Length	Time	Info
1	2a:00:bc:14:e3:ce	PALMA-MULTICAST	PALMA	36	0.000000000	DISCOVER: SET:[start=0a:00:00:00:00:05, cnt=200]
2	2a:00:a9:3a:93:96	PALMA-MULTICAST	PALMA	36	0.596308497	DISCOVER: SET:[start=0a:00:00:00:00:05, cnt=200]
3	2a:00:80:b7:14:4c	PALMA-MULTICAST	PALMA	36	1.147842909	DISCOVER: SET:[start=0a:00:00:00:00:05, cnt=200]
4	0a:00:00:00:00:05	PALMA-MULTICAST	PALMA	40	1.677966175	ANNOUNCE: SET:[start=0a:00:00:00:00:05, cnt=10] lifetime=600 s
5	0a:00:00:00:00:05	PALMA-MULTICAST	PALMA	40	31.940624919	ANNOUNCE: SET:[start=0a:00:00:00:00:05, cnt=10] lifetime=570 s
6	2a:00:2e:72:23:cc	PALMA-MULTICAST	PALMA	36	31.940797868	DISCOVER: SET:[start=0a:00:00:00:00:0f, cnt=185]
7	2a:00:0c:87:e8:77	PALMA-MULTICAST	PALMA	36	32.523456906	DISCOVER: SET:[start=0a:00:00:00:00:0f, cnt=185]
8	2a:00:13:3b:2f:b5	PALMA-MULTICAST	PALMA	36	33.099839882	DISCOVER: SET:[start=0a:00:00:00:00:0f, cnt=185]
9	0a:00:00:00:00:14	PALMA-MULTICAST	PALMA	40	33.677225966	ANNOUNCE: SET:[start=0a:00:00:00:00:0f, cnt=10] lifetime=600 s
10	0a:00:00:00:00:05	PALMA-MULTICAST	PALMA	40	62.661799254	ANNOUNCE: SET:[start=0a:00:00:00:00:05, cnt=10] lifetime=539 s
11	0a:00:00:00:00:14	PALMA-MULTICAST	PALMA	40	65.358967565	ANNOUNCE: SET:[start=0a:00:00:00:00:0f, cnt=10] lifetime=568 s
12	0a:00:00:00:00:05	PALMA-MULTICAST	PALMA	40	92.770092757	ANNOUNCE: SET:[start=0a:00:00:00:00:05, cnt=10] lifetime=509 s
13	0a:00:00:00:00:14	PALMA-MULTICAST	PALMA	40	96.994856614	ANNOUNCE: SET:[start=0a:00:00:00:00:0f, cnt=10] lifetime=537 s

Figure 5.32: Wireshark - Test Case 32

5.3 Performance Analysis

In this section we are presenting the performance analysis we have done focusing on the scalability the system offers and assignments convergence time, bearing in mind we are virtualizing a whole network with Mininet software. Tests made can be divided into self-assignment and server based assignment performances, according to the PALMA mode we were testing on each case. For this complete analysis we have decided to create a Mininet topology of a single switch in order to avoid link transport delays and focus on the code efficiency.

It is important to keep in mind that all these tests have been performed on a virtual machine with limited processing power and in a virtualized topology, a fact that significantly affects the results.

5.3.1 Self-assignment performance

Regarding self-assignment performance, we have decided to execute 100 clients simultaneously to test the convergence time. Each client will discover the whole defined PALMA self-assignment unicast set and adopt only 1 address. Moreover, each client will be adopting the assigned set for 320 seconds, but for this experiment we have enclosed test's time for 300 seconds in order to analyze only a client's life cycle from the START state to the DEFENDING state without going back to RESTART, and hence without renovations.

The purpose of this test is to analyze the total time needed for all clients to adopt

an address set. Figure 5.33 shows the time distribution of the self-assignments, with a convergence time of 140 seconds approximately. The distribution appears to be uniform, and the reason for this is the following:

- Whenever a client issues 3 DISCOVER messages, it adopts as many addresses as the maximum configured, in this case 1, starting from the first address of the discovering set, which is supposed to be the whole PALMA self-assignment unicast set. Then it announces the adopted set and all the other clients discovering the same set have to restart the protocol registering this assigned set in their corresponding databases. Even without any collision, time for each assignation is at least the timeout of 3 DISCOVER messages, which each one of these is of 0.5 seconds with a variable interval of 0.1 second. That means that the minimum time for a client to adopt an address set is of 3*0.4 = 1.2 seconds and the mean would be 1.5 seconds. And, in addition, each time a client adopts an address set, the rest of the simultaneous clients have to restart. For this reason, the distribution is likely to be uniform with a foreseeable convergence time of 1.5 times the total number of simultaneous clients.

In our case, we have obtained a total convergence time of about 140 seconds, but the explanation of this fact is a little more complex. We can say that as the DISCOVER timeout varies and we need 3 DISCOVER timeouts to adopt an address set, the first two DISCOVER timeouts are approximately of 0.5 seconds (by mean) each, and the last one of the minimum timeout possible, 0.4 seconds. This gives us a total time of 1.4 seconds to get an address set assigned. As we have multiple simultaneous clients trying to get an address set assigned, timeouts synchronize and we end up having a total convergence time of 1.4 times the total number of clients. See this behavior in Figure 5.33.

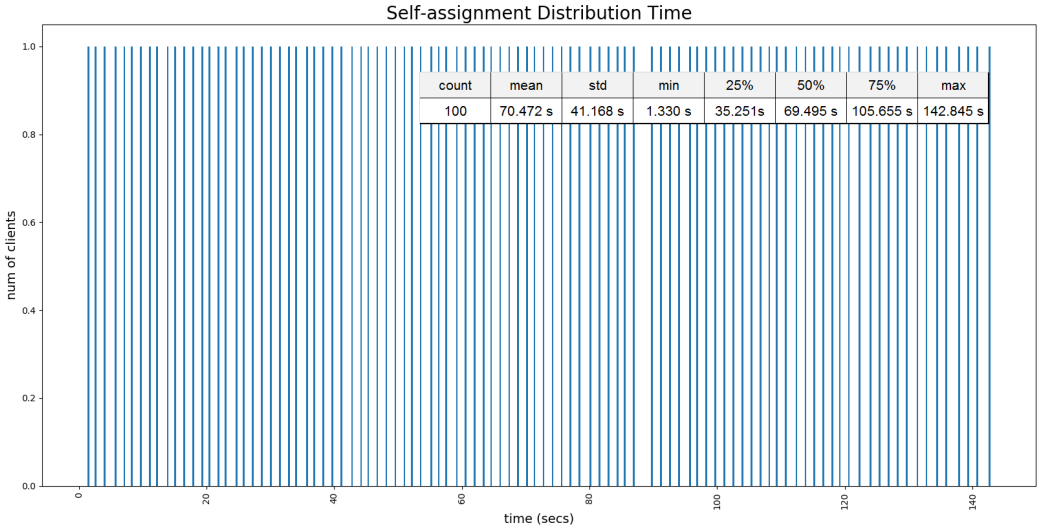


Figure 5.33: PALMA self-assignment distribution time

One more related case is the one of the renovations in the self-assignment PALMA mode. We have studied the behavior clients have when the adopted address set assignment lifetime expires. They should restart the protocol but this time their corresponding databases are full of useful information about the rest of assignations in their network. In this test we have executed 100 simultaneous clients and wait until they get to do 10 renovations each, discarding the initial procedure in which the database is empty. Results can be seen in Figure 5.34. As expected, we managed to get 1000 renovations, resulted from 10 renovations for each client.

The surprising datum in Figure 5.34 is the mean time to get an address assigned, 3.3 seconds, which much faster compared to the uniform distribution of the previous Figure 5.33. This has sense just by looking at the graph and realizing the more attractive results given by the exponential distribution. Furthermore, in spite of the maximum value obtained in this test for a renovation to take place was of ()47.3 seconds), we have to remark that the 75% of the renovations took place under the mean time. This fact proves the improvement that databases suppose to PALMA protocol's functioning.

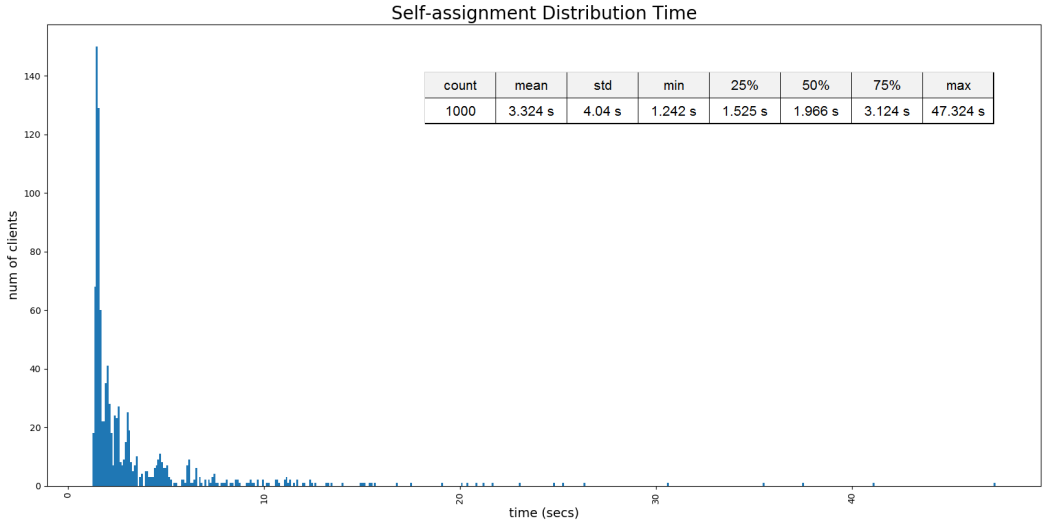


Figure 5.34: PALMA self-assignment renovations distribution time

To conclude the self-assignment experimental results, we have tested a functionality we thought at first it would improve the protocol's efficiency. Recapitulating the client's optional parameters in the configuration file we can find one called "RandomAutoAssign" which can be enabled or disabled as decided by user. If this option is disabled, client adopts an address set starting by the first address of the claiming discovery set, as mentioned before. However, if user decides to enable this option, this would change the client's behavior and this one would adopt a random set within the claiming discovery set.

At first, we thought it would be useful this new behavior if multiple clients want to adopt simultaneously an address set, and in order to prove it, we run several tests with the same characteristics of the one commented in Figure 5.33 at the beginning of this section. In each test we have configured the clients with a different claiming set, varying this one from a total space of 1000 addresses to only 100. We wanted to see the different convergence times as a function of the claiming space occupation. For instance, we have tested the convergence time having 100 simultaneous clients with the same discovery address set and adopting each one of them a single address from it. Unfortunately, results show that convergence time

does not depend on how big or small the claiming discovery set is as long as every client can have assigned the wanted number of addresses.

Moreover, the systematical assignment mode and the random assignment mode have turned out to be practically equal in terms of efficiency, as proved in the graphs of Figure 5.35 for systematical assignments and Figure 5.36 for random assignments.

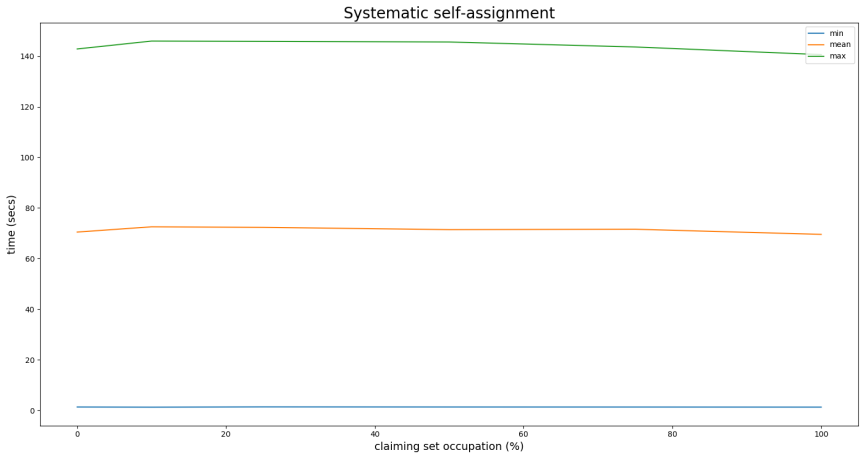


Figure 5.35: PALMA systematic self-assignment time depending on the claiming occupation percentage

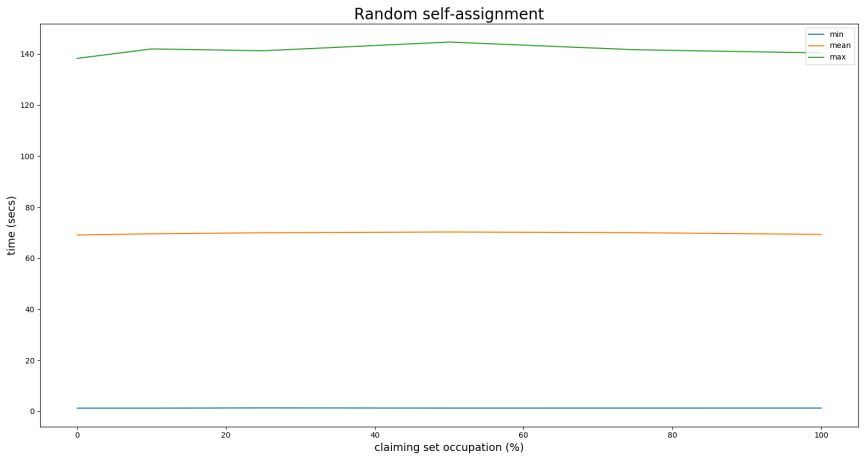


Figure 5.36: PALMA random self-assignment time depending on the claiming occupation percentage

5.3.2 Server based assignment performance

In order to test the PALMA server based mode, we have decided to execute a series of tests, starting with 1 server and 10 simultaneous clients, and increasing the number of clients until the system collapses.

We have studied the time a particular PALMA server takes in order to respond to a REQUEST, which means, process the REQUEST message, look for a free or previously reserved (with a OFFER message) address set in the server's database and assign to the client sending the REQUEST message this address set by issuing the corresponding ACK messages. We can suppose this time is very similar to the one the server would take to respond DISCOVER messages and offer the clients free address sets from the server's database. This time increases depending on the number of simultaneous clients requesting to be assigned an address set. The result out of this study can be seen in Figure 5.37, where y-axis represents the total time passed since the REQUEST message was sent until the client received the corresponding ACK message and the x-axis the number of simultaneous clients asking for an address set to get assigned. This graph 5.37 shows that the server responds pretty well until the system saturates (with 900 simultaneous clients) and some server ACK messages cannot reach the intended clients on time (before 500 milliseconds), which means those clients will need to repeat the REQUEST message.

In addition, to obtain a better visualization, we have represented in Figure 5.38 the distribution of the time a server takes to respond to a REQUEST message given 500 simultaneous requesting clients. The mean time stays under 1 millisecond, which shows how fast and efficient our implemented server is.

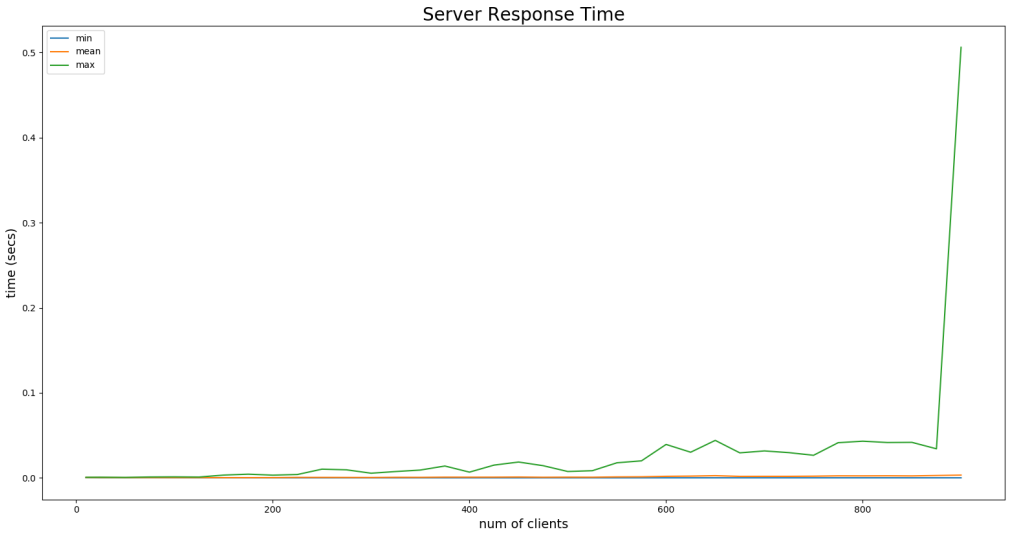


Figure 5.37: PALMA Server response time (to REQUEST messages) depending on the number of simultaneous requesting clients

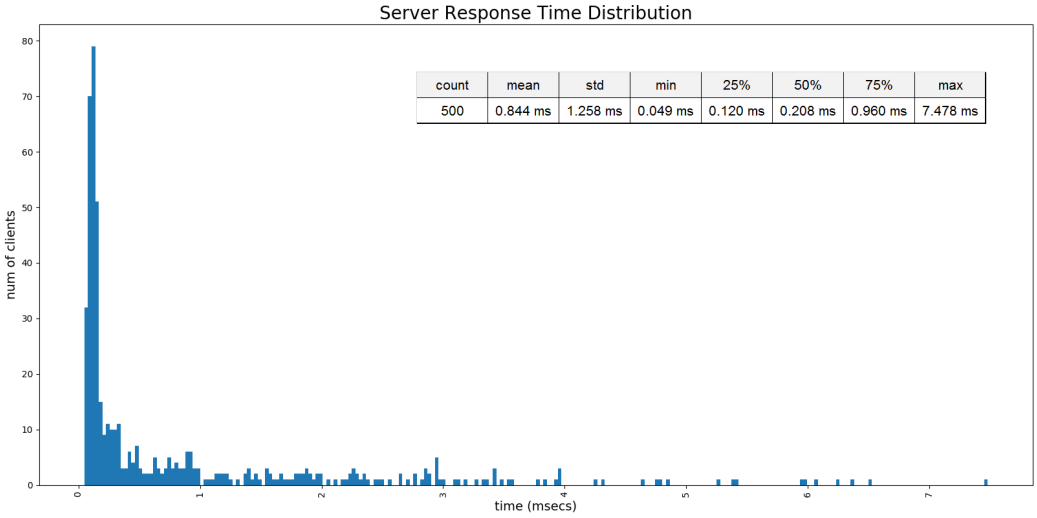


Figure 5.38: PALMA Server response time distribution (to REQUEST messages) given 500 simultaneous requesting clients

Having studied the time a server takes to respond to REQUEST messages, it is interesting to see what happens when all those server assignations are required to renew. For this

purpose, we have done a similar experiment than the previous one, but this time measuring the time a server takes to do renewals, which means, process the pertinent REQUEST message with the renewal flag, search the address set previously assigned to that client and verify the information. Once the server checks that the incoming client requesting the renewal is in fact requesting the previously assigned address set, renewal is done and client in informed.

Figure 5.39 represents server response time to carry out renewals as a function of the total number of simultaneous clients requesting so. We can see the systems starts to collapse with 600 simultaneous clients requesting to be done a renewal, 300 clients less than when the server just give away free assignations (Figure 5.38). This results are not surprising given the fact that it is faster to distribute free address sets when the tree in the server's database is absolutely empty, with no nodes or very few of them, than when it is completely formed. Hence, when the server tries to verify the pertinent renewal REQUEST messages, it has to search along the database's tree structure, which is already full of nodes and the time to do so saturates faster than when the tree is practically empty.

Nevertheless, if we take a look to Figure 5.40, we can realize that the mean time the server takes to do renewals is under 1 millisecond. This fact proves again the high efficiency our implemented server has.

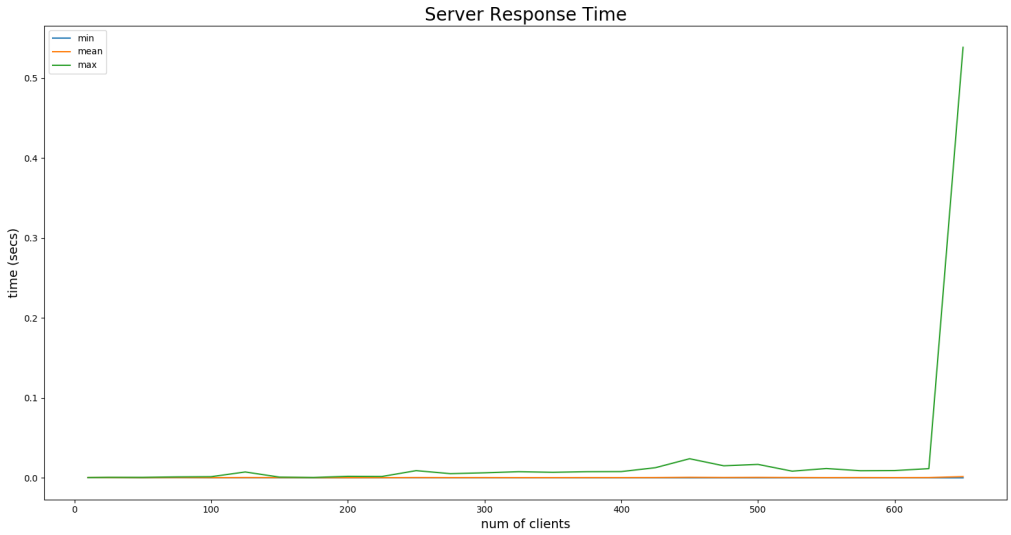


Figure 5.39: PALMA Server response time (to renewals) depending on the number of simultaneous requesting clients

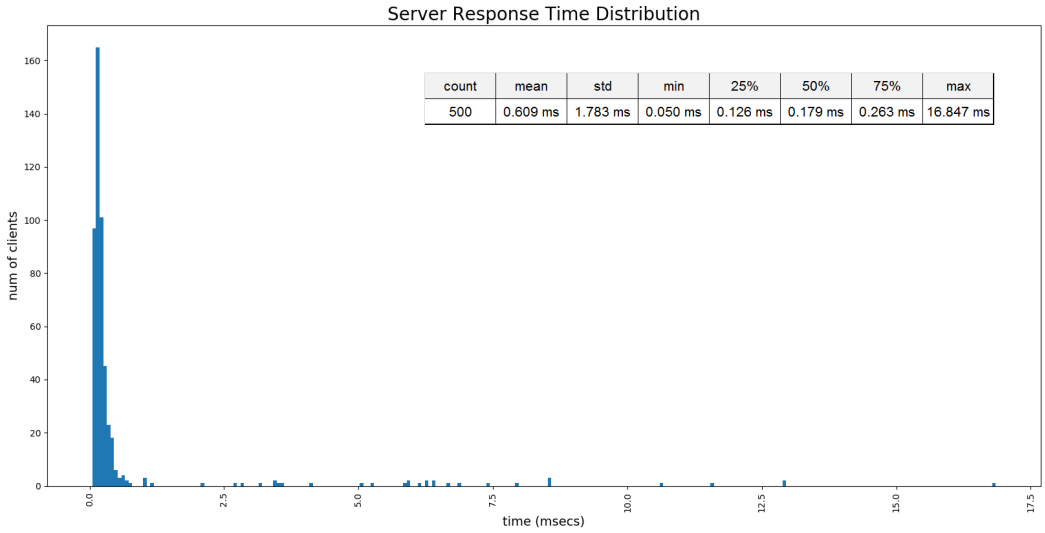


Figure 5.40: PALMA Server response time distribution (to renewals) given 500 simultaneous requesting clients

Chapter 6

Conclusions and Future Development

6.1 Conclusions

In this section we are going to carry out a retrospective of the main objectives that we set for this project and evaluate the work done in respect to them.

The achieved objectives through this thesis development process are listed below:

- Firstly, we have removed all the inconsistencies found in the PALMA last version draft we were supposed to implement. Both the inconsistencies and the contradictions have been replaced by pertinent clarifications and self-assumptions, in order to make the a much more clear and functional specification.

- Secondly, we have achieved to implement code structure that reflects clearly the class architecture we have followed, with a very efficient timer module and a complex, original database structure that guarantees a high velocity searching and inserting. In addition, all the classes are perfectly integrated with each other in a clear way, thus achieving a clear and compact code.

- Moreover, we have submitted the code for functional validation, in order to verify the proper working of all the specified functionalities in the draft. Results out of this validation plan have corroborated the expected behavior given the exception mentioned in the chapter about assumptions to the PALMA specification.

- Finally, we have developed more tests with the purpose of studying the implementation's performance. After obtaining and analyzing the results out of these, the code's efficiency has been confirmed by the high scalability and the low convergence time achieved, given the poor executing conditions of the testing environment.

In conclusion, all this project's goals have been successfully accomplished, including the main one of elaborating a code as lightweight, portable and efficient as possible which can serve as reference for future PALMA developments. In addition, due to the results analysis, we have also gathered some information useful for possible future works in relation to PALMA.

6.2 Future Works

When finishing projects consisting in the implementation of a certain protocol it is important to analyze the results and not only verify the expected behaviors are real but also consider if it exits any other ways of obtaining better results by changing some issues in the protocol's specification.

In this case, server-based mode functioning has nothing to refute, and the performance tests have corroborated this fact. Regarding the self-assignment mode, this one could be improved by modifying some aspects of the PALMA draft's specification[8]. We have seen that, despite the mean convergence time for 100 simultaneous clients to adopt an address set is low when each of the clients has enough information in its particular database, this time is very high at the beginning, when none of them knows anything about the addresses assigned in the network. Fortunately, this could be solved in future implementations by making the clients claim a reduced set and not the whole PALMA self-assignment address block. Moreover, clients should initialize the database with the whole self-assignment set of the type of addresses they are looking for and only claim the maximum number of addresses they are configured to assign to theirselves. These actions must be done listening to the PALMA Multicast Address in order to be able to register any ANNOUNCE or DEFEND messages

sent through the network and learn from this information. With this new behavior, the convergence time we mentioned before will be reduced drastically, so we strongly recommend that changes to be done in the future versions of PALMA.

Furthermore, one more future work it would be interesting to carry out is the slight modification of the written code in order to make it compatible with embedded and bare metal systems, which should not be difficult given the low library dependence and lightweight our code is.

PALMA is a relatively new protocol which is, in fact, in development and for this reason many versions are yet to come, each one surely improving the performance and getting better results than the previous one. Still, this implemented version is undoubtedly a success despite needing some improvements.

Chapter 7

Regulatory Framework

Regulations involved in this project are presented herein. Given the nature of this work, which consists on the implementation of a new protocol, the most relevant regulatory frame refers to the specification itself. PALMA protocol complies with the following normative:

- All projects belonging to the IEEE 802 group must comply with the IEEE 802(R) standards published by the IEEE for frame-based data networks. In this case, PALMA specification mentions the "*802-2014 - IEEE Standard for Local and Metropolitan Area Networks: Overview and Architecture*" where reference models for the IEEE 802 standards are described and the relationship of these ones with the higher layer protocols. Moreover, this standard serves as the foundation for the family of IEEE 802 standards published by IEEE for local area networks (LANs), metropolitan area networks (MANs), personal area networks (PANs), and regional area networks (RANs).[33]

- The Protocol for Assignment of Local and Multicast Addresses (PALMA), its not other than the one which makes possible the dynamic assignation of MAC addresses even to devices that do not have one. For this purpose, PALMA working team has obviously based his specification on the "*802c-2017 - IEEE Standard for Local and Metropolitan Area Networks:Overview and Architecture–Amendment 2: Local Medium Access Control (MAC) Address Usage*", where a MAC address space structure known as SLAP (Structured Local Address Plan) is provided in order to allow the coexistence

of multiple administrations. This amendment recommends, in addition, the use of a range of local MAC addresses for IEEE 802(R) protocols.[34]

- Bridges are not mainly mentioned in the PALMA specification but are needed for a proper working, as they allow the compatible interconnection of information technology equipment attached to separate individual LANs. The "*802.1Q-2018 - IEEE Standard for Local and Metropolitan Area Network–Bridges and Bridged Networks*" explains how the MAC service is supported by Bridge networks and the operations of these bridges, including protocols, algorithms and management. Furthermore, this standard specifies the bridges that interconnect individual LANs, of course supporting the IEEE 802 MAC service.[35]

Chapter 8

Socio-Economical Background

8.1 Budget Analysis

This section compiles both material and personnel costs for the development of this project.

8.1.1 Personnel Costs

In order to make a good estimation of the personnel costs, we firstly classify the work roles of the participants and consequently, use data available at the National Institute of Statistics (INE) in order to assign each role its salary. The tutor and author of this project are considered as Postdoctoral Researcher Director and Research Assistant respectively. Following the INE statistics, we assign the research assistant a tentative salary of 34.790,1€ (2017 median salary for "Technicians, scientific professionals and intellectual in health and teaching"), and the project director a salary of 55.478,0€ (2017 median salary for Directors and managers).[36]

Table 8.1: Approximate total personnel costs

Working Labor	**Annual Cost**	**Cost per hour**	**Working hours**	**Total cost**
Research Assistant	34 790.10€	19.41€	720 hours	13 975.20€
Project Director	55 478.00€	30.96€	90 hours	2 786.40€
			Total	**16 761.60€**

The estimation of personnel costs is about 16 761.40€.

8.1.2 Equipment costs

Regarding the equipment costs, those are minimum compared with the personnel ones, given the naturalness of this project, which consists mainly on the elaboration of C++ code. Still, we have considered the author's personal computer as the only equipment and approximate its deprecation cost. Moreover, we have disregarded the costs associated with the testing procedures.

In the case of software, no licenses were needed because only free software tools were used in order to achieve all the required goals.

"Ley 27/2014, de 27 de noviembre, del Impuesto sobre Sociedades"[37] specifies the following calculation formula for Depreciation:

$$Amortization\,Cost = \frac{Use\,time}{Expected\,lifespan} \cdot Item\,cost \tag{8.1}$$

where amortization costs are calculated from a simple linear approximation, relating the *Use time* of the device with its approximated *Expected Lifespan*.

Table 8.2 shows total costs related to the equipment, with a total value of 98.0€.

Table 8.2: Approximate equipment usage costs

Item	Cost	Use Time	Expected Lifespan	Final Amortization
MSI GE72 2QD	980.0€	0.5 year	5 years	98.0€
			Total	**98.0€**

8.2 Socio-Economical Impact

At the beginning of the thesis, specifically in the motivation section, the scarcity future problem of MAC addresses was commented. The benefits of this project are not only achieve a good implementation of the PALMA protocol, but make this one a reality. This protocol will make a difference when we lack of MAC addresses in the near future, because no changes in the internet layer 2 functioning will be necessary. This means that due to the success of this protocol, new designs of protocols for layer 2 are not going to take place, in contrast to what happened with IPv6[38] for layer 3, when we run out of IPv4 addresses[7].

The avoidance of having to encounter the MAC addresses problem with other much more expensive ideas, will suppose a huge economical impact. Furthermore, this impact could be also social as we are allowing the manufacturing of new devices, such as mobile phones, IoT gadgets, laptops and many others, in spite of the lack of the unique MAC addresses they needed before.

One important aspect of this project is the resulted code, which due to its efficiency, lightweight and very low dependency on external libraries is capable to be inserted and work properly on embedded or even bare metal systems. As a result, we can have this implementation running on practically all systems from a simple sensor to a complex mainframe.

Bibliography

[1] Carlos J. Bernardos and Alain Mourad. SLAP quadrant selection option for DHCPv6. Internet-Draft draft-ietf-dhc-slap-quadrant-10, Internet Engineering Task Force, 2020. Work in Progress.

[2] Bernie Volz, Tomek Mrugalski, and Carlos J. Bernardos. Link-Layer Addresses Assignment Mechanism for DHCPv6. Internet-Draft draft-ietf-dhc-mac-assign-09, Internet Engineering Task Force, 2020. Work in Progress.

[3] Ieee standard for a transport protocol for time-sensitive applications in bridged local area networks. *IEEE Std 1722-2016 (Revision of IEEE Std 1722-2011)*, pages 1–233, 2016.

[4] Jorge Luis Orejel. Balanced multiway trees. 10 2014.

[5] B Carpenter, Jon Crowcroft, and Yakov Rekhter. Ipv4 address behaviour today. Technical report, February 1997. RFC 2101, 1997.

[6] Brian P Crow, Indra Widjaja, Jeong Geun Kim, and Prescott Sakai. Investigation of the ieee 802.11 medium access control (mac) sublayer functions. In *Proceedings of INFOCOM'97*, volume 1, pages 126–133. IEEE, 1997.

[7] Philipp Richter, Mark Allman, Randy Bush, and Vern Paxson. A primer on ipv4 scarcity. *ACM SIGCOMM Computer Communication Review*, 45(2):21–31, 2015.

[8] LAN/MAN Standards Committee of the IEEE Computer Society. Draft standard for local and metropolitan area networks: Multicast and local address assignment. *IEEE Std 802.1Q-2018 (Revision of IEEE Std 802.1Q-2014)*, pages 1–65, 2020.

[9] J Jeong et al. Ipv6 host configuration of dns server information approaches. Technical report, RFC 4339, February, 2006.

[10] Nick Moore et al. Optimistic duplicate address detection (dad) for ipv6. Technical report, RFC 4429, April, 2006.

[11] Ieee standard for layer 2 transport protocol for time sensitive applications in a bridged local area network. *IEEE Std 1722-2011*, pages 1–65, 2011.

[12] Chris Tomlinson, R. LAVENDER, and DENNIS KAFURA. Implementing communication protocols using object-oriented techniques. 01 1992.

[13] Chandramohan Thekkath, Thu Nguyen, Evelyn Moy, and Edward Lazowska. Implementing network protocols at user level. *IEEE/ACM Trans. Netw.*, 1:554–565, 10 1993.

[14] Paul J Deitel and Harvey M Deitel. *Java: como programar*. Pearson educacion, 2008.

[15] Harvey M Deitel and Paul J Deitel. *C++ como programar*. Prentice Hall, 1999.

[16] Egon Börger and Wolfram Schulte. Defining the java virtual machine as platform for provably correct java compilation. In *International Symposium on Mathematical Foundations of Computer Science*, pages 17–35. Springer, 1998.

[17] Richard Fox. *Unix and Linux*, pages 459–499. 08 2020.

[18] Jo Hoey. *Program Analysis with a Debugger: GDB*, pages 21–33. 10 2019.

[19] Zuo Xiang and Patrick Seeling. *Mininet: an instant virtual network on your computer*, pages 219–230. 01 2021.

[20] J. But. A c++ socket library for linux. 27, 06 2002.

[21] Christian Nagel, Ajit Mungale, Vinod Kumar, Nauman Laghari, Andrew Krowczyk, Tim Parker, S. Sivakumar, and Alexandru Serban. *Raw Socket Programming*, pages 163–187. 01 2004.

[22] Donald Lewine. *POSIX programmers guide*. " O'Reilly Media, Inc.", 1991.

[23] W Richard Stevens and Stephen A Rago. *Advanced programming in the UNIX environment*. Addison-Wesley, 2008.

[24] G. Anzinger. High resolution posix timers. 08 2020.

[25] Donald Ervin Knuth. *The art of computer programming*, volume 3. Pearson Education, 1997.

[26] Douglas Comer. Ubiquitous b-tree. *ACM Computing Surveys (CSUR)*, 11(2):121–137, 1979.

[27] Paul F Dietz. Maintaining order in a linked list. In *Proceedings of the fourteenth annual ACM symposium on Theory of computing*, pages 122–127, 1982.

[28] J. Gabarró and Llorenç Roselló. Self-organization and evolution on large computer data structures. 01 1999.

[29] Chandan Pal, S Veena, Ram P Rustagi, and KNB Murthy. Implementation of simplified custom topology framework in mininet. In *2014 Asia-Pacific Conference on Computer Aided System Engineering (APCASE)*, pages 48–53. IEEE, 2014.

[30] Andre Szczepanek, Denis R Beaudoin, and Iain Robertson. Modular interconnection of network switches, February 10 2004. US Patent 6,690,668.

[31] Laura Chappell. *Wireshark network analysis*. PODBOOKS. COM, LLC, 2012.

[32] Paul M Embree, Bruce Kimble, and James F Bartram. C language algorithms for digital signal processing, 1991.

[33] Ieee standard for local and metropolitan area networks: Overview and architecture. *IEEE Std 802-2014 (Revision to IEEE Std 802-2001)*, pages 1–74, 2014.

[34] Ieee standard for local and metropolitan area networks:overview and architecture–amendment 2: Local medium access control (mac) address usage. *IEEE Std 802c-2017 (Amendment to IEEE Std 802-2014 as amended by IEEE Std 802d-2017)*, pages 1–26, 2017.

[35] Ieee standard for local and metropolitan area network–bridges and bridged networks. *IEEE Std 802.1Q-2018 (Revision of IEEE Std 802.1Q-2014)*, pages 1–1993, 2018.

[36] Salarios, ingresos, cohesión social. https://www.ine.es/jaxiT3/Tabla.htm?t=10882, 2020. Last access: 1/9/2020.

[37] Ley 27/2014, de 27 de noviembre, del impuesto sobre sociedades. https://www.boe.es/eli/es/l/2014/11/27/27/con, 2014. Last access: 1/9/2020.

[38] Silvia Hagen. *IPv6 essentials*. " O'Reilly Media, Inc.", 2006.

Appendix A

Appendix A: Wireshark Dissector

A.1 Capture Examples

A.1.1 ANNOUNCE message

```
▸ Frame 13: 40 bytes on wire (320 bits), 40 bytes captured (320 bits) on interface s0_1-eth1, id 0
▾ Ethernet II, Src: 0a:00:00:00:00:14 (0a:00:00:00:00:14), Dst: PALMA-MULTICAST (01:80:c2:ab:cd:ef)
  ▸ Destination: PALMA-MULTICAST (01:80:c2:ab:cd:ef)
  ▸ Source: 0a:00:00:00:00:14 (0a:00:00:00:00:14)
    Type: Unknown (0x33ff)
▾ IEEE PALMA Protocol
    Subtype: PALMA CLIENT (0x00)
    000. .... = Version: 0x0
    ...0 0111 = Message Type: ANNOUNCE (0x07)
  ▾ Control Word: 0x0182, ELI bit, MAC provided bit, STATION ID bit
      .... .... .... ...0 = AAI bit: False
      .... .... .... ..1. = ELI bit: True
      .... .... .... .0.. = SAI bit: False
      .... .... ...0 .... = MAC64 bit: False
      .... .... ..0. .... = MULTICAST bit: False
      .... .... .0.. .... = SERVER bit: False
      .... .... 1... .... = MAC provided bit: True
      .... ...1 .... .... = STATION ID bit: True
      .... ..0. .... .... = NETWORK ID bit: False
      .... .0.. .... .... = Codefield bit: False
      .... 0... .... .... = Specific Vendor bit: False
      ...0 .... .... .... = Renewal bit: False
    Token: 0x8221
    0000 .... = Status: Field not used (0x0)
    .... 0000 0001 1010 = Length: 26
  ▾ PARAMETER 0: MAC Address Set
      Type: MAC Address Set (0x02)
      Length: 10
      Address: 0a:00:00:00:00:0f (0a:00:00:00:00:0f)
      Count: 10
  ▾ PARAMETER 1: Lifetime
      Type: Lifetime (0x04)
      Length: 4
      Lifetime: 537
  ▾ PARAMETER 2: Station_ID
      Type: Station_ID (0x01)
      Length: 4
      Station Id: H2
```

A.1.2 DEFEND message

```
▸ Frame 6: 50 bytes on wire (400 bits), 50 bytes captured (400 bits) on interface s0_1-eth1, id 0
▸ Ethernet II, Src: 0a:00:00:00:00:01 (0a:00:00:00:00:01), Dst: 0a:00:00:00:00:0a (0a:00:00:00:00:0a)
▾ IEEE PALMA Protocol
    Subtype: PALMA CLIENT (0x00)
    000. .... = Version: 0x0
    ...0 0110 = Message Type: DEFEND (0x06)
  ▾ Control Word: 0x0182, ELI bit, MAC provided bit, STATION ID bit
      .... .... .... ...0 = AAI bit: False
      .... .... .... ..1. = ELI bit: True
      .... .... .... .0.. = SAI bit: False
      .... .... ...0 .... = MAC64 bit: False
      .... .... ..0. .... = MULTICAST bit: False
      .... .... .0.. .... = SERVER bit: False
      .... .... 1... .... = MAC provided bit: True
      .... ...1 .... .... = STATION ID bit: True
      .... ..0. .... .... = NETWORK ID bit: False
      .... .0.. .... .... = Codefield bit: False
      .... 0... .... .... = Specific Vendor bit: False
      ...0 .... .... .... = Renewal bit: False
    Token: 0x7367
    0000 .... = Status: Field not used (0x0)
    .... 0000 0010 0100 = Length: 36
  ▾ PARAMETER 0: Station_ID
      Type: Station_ID (0x01)
      Length: 4
      Station Id: H1
  ▾ PARAMETER 1: Lifetime
      Type: Lifetime (0x04)
      Length: 4
      Lifetime: 570
  ▾ PARAMETER 2: MAC Address Set
      Type: MAC Address Set (0x02)
      Length: 10
      Address: 0a:00:00:00:00:05 (0a:00:00:00:00:05)
      Count: 10
  ▾ PARAMETER 3: MAC Address Set (IN CONFLICT)
      Type: MAC Address Set (0x02)
      Length: 10
      Conflict Address: 0a:00:00:00:00:05 (0a:00:00:00:00:05)
      Count: 4
```

A.1.3 REQUEST message

```
▸ Frame 3: 36 bytes on wire (288 bits), 36 bytes captured (288 bits) on interface s0_1-eth1, id 0
▾ Ethernet II, Src: 10:0f:ac:e0:00:01 (10:0f:ac:e0:00:01), Dst: 10:0a:bc:de:f0:01 (10:0a:bc:de:f0:01)
  ▸ Destination: 10:0a:bc:de:f0:01 (10:0a:bc:de:f0:01)
  ▸ Source: 10:0f:ac:e0:00:01 (10:0f:ac:e0:00:01)
    Type: Unknown (0x33ff)
▾ IEEE PALMA Protocol
    Subtype: PALMA CLIENT (0x00)
    000. .... = Version: 0x0
    ...0 0011 = Message Type: REQUEST (0x03)
  ▾ Control Word: 0x1182, ELI bit, MAC provided bit, STATION ID bit, Renewal bit
      .... .... .... ...0 = AAI bit: False
      .... .... .... ..1. = ELI bit: True
      .... .... .... .0.. = SAI bit: False
      .... .... ...0 .... = MAC64 bit: False
      .... .... ..0. .... = MULTICAST bit: False
      .... .... .0.. .... = SERVER bit: False
      .... .... 1... .... = MAC provided bit: True
      .... ...1 .... .... = STATION ID bit: True
      .... ..0. .... .... = NETWORK ID bit: False
      .... .0.. .... .... = Codefield bit: False
      .... 0... .... .... = Specific Vendor bit: False
      ...1 .... .... .... = Renewal bit: True
    Token: 0x1f92
    0000 .... = Status: Field not used (0x0)
    .... 0000 0001 0110 = Length: 22
  ▾ PARAMETER 0: MAC Address Set
      Type: MAC Address Set (0x02)
      Length: 10
      Address: 1a:ca:00:00:00:00 (1a:ca:00:00:00:00)
      Count: 100
  ▾ PARAMETER 1: Station_ID
      Type: Station_ID (0x01)
      Length: 4
      Station Id: H1
```

A.1.4 OFFER message

```
▸ Frame 2: 55 bytes on wire (440 bits), 55 bytes captured (440 bits) on interface s0_1-eth1, id 0
▸ Ethernet II, Src: 10:0a:bc:de:f0:01 (10:0a:bc:de:f0:01), Dst: 2a:00:af:3b:2a:46 (2a:00:af:3b:2a:46)
▾ IEEE PALMA Protocol
    Subtype: PALMA CLIENT (0x00)
    000. .... = Version: 0x0
    ...0 0010 = Message Type: OFFER (0x02)
  ▸ Control Word: 0x0bc2, ELI bit, SERVER bit, MAC provided bit, STATION ID bit, NETWORK ID bit, Specific Vendor bit
    Token: 0x5386
    0000 .... = Status: Field not used (0x0)
    .... 0000 0010 1001 = Length: 41
  ▾ PARAMETER 0: Lifetime
      Type: Lifetime (0x04)
      Length: 4
      Lifetime: 10
  ▾ PARAMETER 1: MAC Address Set
      Type: MAC Address Set (0x02)
      Length: 10
      Address: 1a:ca:00:00:00:00 (1a:ca:00:00:00:00)
      Count: 1000
  ▾ PARAMETER 2: Station_ID
      Type: Station_ID (0x01)
      Length: 4
      Station Id: H1
  ▾ PARAMETER 3: Network_ID
      Type: Network_ID (0x03)
      Length: 8
      Network Id: SERVER
  ▾ PARAMETER 4: Vendor Specific
      Type: Vendor Specific (0x06)
      Length: 7
      Vendor: NOKIA
```

A.1.5 ACK message

```
▸ Frame 4: 40 bytes on wire (320 bits), 40 bytes captured (320 bits) on interface s0_1-eth1, id 0
▸ Ethernet II, Src: 10:0a:bc:de:f0:01 (10:0a:bc:de:f0:01), Dst: 1a:ca:00:00:00:00 (1a:ca:00:00:00:00)
▾ IEEE PALMA Protocol
    Subtype: PALMA CLIENT (0x00)
    000. .... = Version: 0x0
    ...0 0100 = Message Type: ACK (0x04)
  ▸ Control Word: 0x05c2, ELI bit, SERVER bit, MAC provided bit, STATION ID bit, Codefield bit
    Token: 0x5386
    0001 .... = Status: Proposed assignment agreed (0x1)
    .... 0000 0001 1010 = Length: 26
  ▾ PARAMETER 0: Station_ID
      Type: Station_ID (0x01)
      Length: 4
      Station Id: H1
  ▾ PARAMETER 1: MAC Address Set
      Type: MAC Address Set (0x02)
      Length: 10
      Address: 1a:ca:00:00:00:00 (1a:ca:00:00:00:00)
      Count: 100
  ▾ PARAMETER 2: Lifetime
      Type: Lifetime (0x04)
      Length: 4
      Lifetime: 10
```

A.2 Dissector Code

```
/* packet-palma.c
 * Routines for PALMA Address Allocation Protocol defined by ...
 * Copyright 2020, Miguel Gonzalez Saiz <100346858@alumnos.uc3m.es>
 *
 * $Id$
 *
 * Wireshark - Network traffic analyzer
 * By Gerald Combs <gerald@wireshark.org>
 * Copyright 1998 Gerald Combs
 *
 * This program is free software; you can redistribute it and/or
 * modify it under the terms of the GNU General Public License
 * as published by the Free Software Foundation; either version 2
 * of the License, or (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA
   02110-1301 USA.
 *
 */

#include "config.h"

#include <epan/packet.h>
#include <epan/etypes.h>
#include <epan/to_str.h>

#define PALMA_ETHERTYPE   0x33ff

/* PALMA Field Offsets */
#define PALMA_SUBTYPE_OFFSET        0
#define PALMA_VERSION_OFFSET        1
#define PALMA_MSG_TYPE_OFFSET       1
#define PALMA_CONTROL_OFFSET        2
#define PALMA_TOKEN_OFFSET          4
#define PALMA_STATUS_OFFSET         6
#define PALMA_LENGTH_OFFSET         6
#define PALMA_PARS_OFFSET         8

/* Bit Field Masks */
#define PALMA_VERSION_MASK          0xe0
#define PALMA_MSG_TYPE_MASK         0x1f
#define PALMA_STATUS_MASK         0xf0
```

```
#define PALMA_LENGTH_MASK         0x0fff

/* PALMA message_type */

#define PALMA_MSG_TYPE_DISCOVER       0x01
#define PALMA_MSG_TYPE_OFFER        0x02
#define PALMA_MSG_TYPE_REQUEST        0x03
#define PALMA_MSG_TYPE_ACK          0x04
#define PALMA_MSG_TYPE_RELEASE        0x05
#define PALMA_MSG_TYPE_DEFEND       0x06
#define PALMA_MSG_TYPE_RANNOUNCE      0x07

#define MAX_PARS              6
#define INITIALIZE_PARS_VARS        {-1, -1, -1, -1, -1, -1}

/* PALMA par_type */
#define PALMA_PAR_TYPE_STATION_ID   0x01
#define PALMA_PAR_TYPE_MAC_ADDR_SET   0x02
#define PALMA_PAR_TYPE_NETWORK_ID   0x03
#define PALMA_PAR_TYPE_LIFETIME     0x04
#define PALMA_PAR_TYPE_CLIENT_ADDR    0x05
#define PALMA_PAR_TYPE_VENDOR     0x06

/* PALMA Subtype */
#define PALMA_CLIENT          0x00
#define PALMA_SERVER          0x01

/* PALMA Control word bits */
#define PALMA_AAI_CW_BIT      (1<<0)
#define PALMA_ELI_CW_BIT      (1<<1)
#define PALMA_SAI_CW_BIT      (1<<2)
#define PALMA_MAC64_CW_BIT      (1<<4)
#define PALMA_MCAST_CW_BIT      (1<<5)
#define PALMA_SERVER_CW_BIT     (1<<6)
#define PALMA_SETPROV_CW_BIT    (1<<7)
#define PALMA_STATIONID_CW_BIT    (1<<8)
#define PALMA_NETWORKID_CW_BIT    (1<<9)
#define PALMA_CODEFIELD_CW_BIT    (1<<10)
#define PALMA_VENDOR_CW_BIT     (1<<11)
#define PALMA_RENEWAL_CW_BIT    (1<<12)

/* PALMA status */
#define PALMA_STATUS_FIELD_UNUSED   0
#define PALMA_STATUS_ASSIGN_OK      1
#define PALMA_STATUS_ALTERNATE_SET    2
#define PALMA_STATUS_FAIL_CONFLICT    3
#define PALMA_STATUS_FAIL_DISALLOWED  4
#define PALMA_STATUS_FAIL_TOO_LARGE   5
#define PALMA_STATUS_FAIL_OTHER     6

static const value_string palma_msg_type_vals [] = {
```

```
  {PALMA_MSG_TYPE_DISCOVER,   "DISCOVER"},
  {PALMA_MSG_TYPE_OFFER,      "OFFER"},
  {PALMA_MSG_TYPE_REQUEST,    "REQUEST"},
  {PALMA_MSG_TYPE_ACK,      "ACK"},
  {PALMA_MSG_TYPE_RELEASE,    "RELEASE"},
  {PALMA_MSG_TYPE_DEFEND,     "DEFEND"},
  {PALMA_MSG_TYPE_RANNOUNCE,    "ANNOUNCE"},
    {0,               NULL}
};

static const value_string palma_par_type_vals [] = {
    {PALMA_PAR_TYPE_STATION_ID,     "Station_ID"},
    {PALMA_PAR_TYPE_MAC_ADDR_SET,   "MAC Address Set"},
    {PALMA_PAR_TYPE_NETWORK_ID,     "Network_ID"},
    {PALMA_PAR_TYPE_LIFETIME,     "Lifetime"},
    {PALMA_PAR_TYPE_CLIENT_ADDR,    "Client Address"},
    {PALMA_PAR_TYPE_VENDOR,       "Vendor Specific"},
    {0,             NULL}
};

static const value_string palma_subtype_vals [] = {
    {PALMA_CLIENT,    "PALMA CLIENT"},
    {PALMA_SERVER,    "PALMA SERVER"},
    {0,             NULL}
};

static const value_string palma_status_vals [] = {
    {PALMA_STATUS_FIELD_UNUSED,   "Field not used"},
    {PALMA_STATUS_ASSIGN_OK,    "Proposed assignment agreed"},
    {PALMA_STATUS_ALTERNATE_SET,  "Alternate addres set provided"},
    {PALMA_STATUS_FAIL_CONFLICT,  "Assignment rejected - address conflict"
   },
    {PALMA_STATUS_FAIL_DISALLOWED,  "Assignment rejected - requested
   address administratively disallowed"},
    {PALMA_STATUS_FAIL_TOO_LARGE, "Assignment rejected - requested address
    set too large"},
    {PALMA_STATUS_FAIL_OTHER,   "Assignment rejected - other
   administrative reasons"},
    {0,               NULL}
};

/*****************************************************/
/* Initialize the protocol and registered fields     */
/*****************************************************/
static int proto_palma = -1;
//static int proto_palma_pars = -1;

/* PALMA PDU */
static int hf_palma_subtype = -1;
static int hf_palma_version = -1;
static int hf_palma_message_type = -1;
```

```
static int hf_palma_control = -1;
static int hf_palma_token = -1;
static int hf_palma_status = -1;
static int hf_palma_length = -1;

static int hf_palma_aai_cw_bit = -1;
static int hf_palma_eli_cw_bit = -1;
static int hf_palma_sai_cw_bit = -1;
static int hf_palma_mac64_cw_bit = -1;
static int hf_palma_mcast_cw_bit = -1;
static int hf_palma_server_cw_bit = -1;
static int hf_palma_mac_provided_cw_bit = -1;
static int hf_palma_stationid_cw_bit = -1;
static int hf_palma_networkid_cw_bit = -1;
static int hf_palma_codefield_cw_bit = -1;
static int hf_palma_vendor_cw_bit = -1;
static int hf_palma_renewal_cw_bit = -1;

static int hf_palma_par_type[MAX_PARS] = INITIALIZE_PARS_VARS;
static int hf_palma_par_len[MAX_PARS] = INITIALIZE_PARS_VARS;

static int hf_palma_station_id = -1;
static int hf_palma_48_bit_address[2] = {-1, -1};
static int hf_palma_64_bit_address[2] = {-1, -1};
static int hf_palma_set_len[2] = {-1, -1};
static int hf_palma_48_bit_mask[2] = {-1, -1};
static int hf_palma_64_bit_mask[2] = {-1, -1};
static int hf_palma_network_id = -1;
static int hf_palma_lifetime = -1;
static int hf_palma_client_address = -1;
static int hf_palma_vendor = -1;

static const int* bits[] = {
  &hf_palma_aai_cw_bit,
  &hf_palma_eli_cw_bit,
  &hf_palma_sai_cw_bit,
  &hf_palma_mac64_cw_bit,
  &hf_palma_mcast_cw_bit,
  &hf_palma_server_cw_bit,
  &hf_palma_mac_provided_cw_bit,
  &hf_palma_stationid_cw_bit,
  &hf_palma_networkid_cw_bit,
  &hf_palma_codefield_cw_bit,
  &hf_palma_vendor_cw_bit,
  &hf_palma_renewal_cw_bit,
  NULL
};

/* Initialize the subtree pointers */
static int ett_palma = -1;
static int ett_control_bits = -1;
```

```
198 static int ett_par[MAX_PARS] = INITIALIZE_PARS_VARS;
199
200 static int
201 dissect_palma(tvbuff_t *tvb, packet_info *pinfo, proto_tree *tree, void *
    data)
202 {
203   gint offset = PALMA_PARS_OFFSET;
204   guint16 msglen = (tvb_get_guint16(tvb, PALMA_LENGTH_OFFSET,
     ENC_BIG_ENDIAN) & PALMA_LENGTH_MASK) -
205       PALMA_PARS_OFFSET;
206
207     guint8      palma_msg_type;
208     proto_item *palma_item     = NULL;
209     proto_tree *palma_tree     = NULL;
210
211   gint addr_set_index = 0;
212
213     col_set_str(pinfo->cinfo, COL_PROTOCOL, "PALMA");
214     col_clear(pinfo->cinfo, COL_INFO);
215
216     /* The palma msg type will be handy in a moment */
217     palma_msg_type = tvb_get_guint8(tvb, PALMA_MSG_TYPE_OFFSET);
218     palma_msg_type &= PALMA_MSG_TYPE_MASK;
219
220     /* Display the name of the packet type in the info column. */
221     col_add_fstr(pinfo->cinfo, COL_INFO, "%s:",
222                  val_to_str(palma_msg_type, palma_msg_type_vals,
223                             "Unknown Type(0x%02x)"));
224
225   /*Ack status variables */
226   guint8 ack_status = tvb_get_guint8(tvb, PALMA_STATUS_OFFSET);
227     ack_status &= PALMA_STATUS_MASK;
228   ack_status >>= 4;
229     /* Now, we'll add pars data to the info column as appropriate */
230
231     palma_item = proto_tree_add_item(tree, proto_palma, tvb, 0, -1, ENC_NA
    );
232     palma_tree = proto_item_add_subtree(palma_item, ett_palma);
233
234     proto_tree_add_item(palma_tree, hf_palma_subtype,   tvb,
235     PALMA_SUBTYPE_OFFSET,     1, ENC_BIG_ENDIAN);
236     proto_tree_add_item(palma_tree, hf_palma_version,   tvb,
237     PALMA_VERSION_OFFSET,     1, ENC_BIG_ENDIAN);
238     proto_tree_add_item(palma_tree, hf_palma_message_type,  tvb,
239     PALMA_MSG_TYPE_OFFSET,      1, ENC_BIG_ENDIAN);
240   proto_tree_add_bitmask(palma_tree,            tvb,
241     PALMA_CONTROL_OFFSET, hf_palma_control, ett_control_bits,
242     bits, ENC_BIG_ENDIAN);
243     proto_tree_add_item(palma_tree, hf_palma_token,   tvb,
244     PALMA_TOKEN_OFFSET,     2, ENC_BIG_ENDIAN);
245     proto_tree_add_item(palma_tree, hf_palma_status,    tvb,
```

```
246     PALMA_STATUS_OFFSET,      1, ENC_BIG_ENDIAN);
247     proto_tree_add_item(palma_tree, hf_palma_length,    tvb,
248     PALMA_LENGTH_OFFSET,      2, ENC_BIG_ENDIAN);
249
250   if(palma_msg_type == PALMA_MSG_TYPE_ACK && ack_status >
      PALMA_STATUS_ALTERNATE_SET)
251   {
252     col_append_fstr(pinfo->cinfo, COL_INFO, " %s",
253               val_to_str(ack_status, palma_status_vals,
254                          "Unknown Code(0x%02x)"));
255   }
256   for(int npars=0; npars<MAX_PARS && msglen; npars++)
257   {
258     guint8 partype = tvb_get_guint8(tvb, offset);
259     guint8 parlen = tvb_get_guint8(tvb, offset+1);
260     proto_item *ti = NULL;
261     proto_tree *subtree = proto_tree_add_subtree_format(palma_tree, tvb,
      0, -1,
262       ett_par[npars], &ti,
263       "PARAMETER %d: %s%s", npars, val_to_str(partype, palma_par_type_vals
      ,"(%s)"),
264       partype == PALMA_PAR_TYPE_MAC_ADDR_SET && addr_set_index ? " (IN
      CONFLICT)" : "");
265     proto_tree_add_item(subtree, hf_palma_par_type[npars],tvb,
266       offset++, 1, ENC_BIG_ENDIAN);
267     proto_tree_add_item(subtree, hf_palma_par_len[npars], tvb,
268       offset++, 1, ENC_BIG_ENDIAN);
269     switch(partype)
270     {
271       case PALMA_PAR_TYPE_STATION_ID:
272         proto_tree_add_item(subtree, hf_palma_station_id, tvb, offset,
      parlen - 2, ENC_ASCII);
273         /*col_append_fstr(pinfo->cinfo, COL_INFO, " Station_id=%s",
274                     tvb_ether_to_str(tvb, offset),
275                     tvb_get_ntohs(tvb, PALMA_REQ_COUNT_OFFSET));*/
276         offset += parlen - 2;
277         break;
278       case PALMA_PAR_TYPE_MAC_ADDR_SET:
279         if(parlen == 10 || parlen == 14)
280         {
281           proto_tree_add_item(subtree, hf_palma_48_bit_address[
      addr_set_index],
282             tvb, offset, 6, ENC_BIG_ENDIAN);
283           col_append_fstr(pinfo->cinfo, COL_INFO, " %s:[start=%s",
284             addr_set_index ? "CONFLICT" : "SET",
285             tvb_ether_to_str(tvb, offset));
286           offset += 6;
287         }
288         else if(parlen == 12 || parlen == 18)
289         {
290           proto_tree_add_item(subtree, hf_palma_64_bit_address[
```

```
addr_set_index],
291           tvb, offset, 8, ENC_BIG_ENDIAN);
292         col_append_fstr(pinfo->cinfo, COL_INFO, " %s:[start=%s",
293           addr_set_index ? "CONFLICT" : "SET",
294           tvb_eui64_to_str(tvb, offset));
295         offset += 8;
296       }
297       else
298       {
299         col_append_fstr(pinfo->cinfo, COL_INFO, " BAD SET");
300         offset += parlen - 2;
301       }
302       if(parlen == 10 || parlen == 12)
303       {
304         proto_tree_add_item(subtree, hf_palma_set_len[addr_set_index],
305           tvb, offset, 2, ENC_BIG_ENDIAN);
306         col_append_fstr(pinfo->cinfo, COL_INFO, ", cnt=%d]",
tvb_get_ntohs(tvb, offset));
307         offset += 2;
308       }
309       else if(parlen == 14)
310       {
311         proto_tree_add_item(subtree, hf_palma_48_bit_mask[addr_set_index
],
312           tvb, offset, 6, ENC_BIG_ENDIAN);
313         col_append_fstr(pinfo->cinfo, COL_INFO, " mask=%s]",
314           tvb_ether_to_str(tvb, offset));
315         offset += 6;
316       }
317       else if(parlen == 18)
318       {
319         proto_tree_add_item(subtree, hf_palma_64_bit_mask[addr_set_index
],
320           tvb, offset, 8, ENC_BIG_ENDIAN);
321         col_append_fstr(pinfo->cinfo, COL_INFO, " mask=%s]",
322           tvb_eui64_to_str(tvb, offset));
323         offset += 8;
324       }
325       addr_set_index++;
326       break;
327     case PALMA_PAR_TYPE_NETWORK_ID:
328       proto_tree_add_item(subtree, hf_palma_network_id, tvb, offset,
parlen - 2, ENC_ASCII);
329       offset += parlen - 2;
330       break;
331     case PALMA_PAR_TYPE_LIFETIME:
332       proto_tree_add_item(subtree, hf_palma_lifetime, tvb, offset, 2,
ENC_BIG_ENDIAN);
333       col_append_fstr(pinfo->cinfo, COL_INFO, " lifetime=%d s",
tvb_get_ntohs(tvb, offset));
334       offset += 2;
```

```
335         break;
336       case PALMA_PAR_TYPE_CLIENT_ADDR:
337         proto_tree_add_item(subtree, hf_palma_client_address, tvb, offset,
    6, ENC_BIG_ENDIAN);
338         col_append_fstr(pinfo->cinfo, COL_INFO, " client addr=%s",
   tvb_ether_to_str(tvb, offset));
339         offset += 6;
340         break;
341       case PALMA_PAR_TYPE_VENDOR:
342         proto_tree_add_item(subtree, hf_palma_vendor, tvb, offset, parlen
   - 2, ENC_ASCII);
343         offset += parlen - 2;
344         break;
345     }
346     msglen -= parlen;
347   }
348   return tvb_captured_length(tvb);
349 } /* end dissect_palma() */
350
351 /* Register the protocol with Wireshark */
352 void
353 proto_register_palma(void)
354 {
355     static hf_register_info hf[] = {
356         { &hf_palma_subtype,
357             { "Subtype", "palma.subtype",
358                 FT_UINT8, BASE_HEX,
359                 VALS(palma_subtype_vals), 0x00,
360                 NULL, HFILL }},
361
362         { &hf_palma_message_type,
363             { "Message Type", "palma.message_type",
364                 FT_UINT8, BASE_HEX,
365                 VALS(palma_msg_type_vals), PALMA_MSG_TYPE_MASK,
366                 NULL, HFILL }},
367
368         { &hf_palma_version,
369             { "Version", "palma.version",
370                 FT_UINT8, BASE_HEX,
371                 NULL, PALMA_VERSION_MASK,
372                 NULL, HFILL }},
373
374         { &hf_palma_control,
375             { "Control Word", "palma.control",
376                 FT_UINT16, BASE_HEX,
377                 NULL, 0,
378                 NULL, HFILL }},
379
380         { &hf_palma_aai_cw_bit,
381             { "AAI bit", "palma.control.aai",
382             FT_BOOLEAN, 16,
```

```
            NULL, PALMA_AAI_CW_BIT,
            NULL, HFILL }},

        { &hf_palma_eli_cw_bit,
            { "ELI bit", "palma.control.eli",
            FT_BOOLEAN, 16,
            NULL, PALMA_ELI_CW_BIT,
            NULL, HFILL }},

        { &hf_palma_sai_cw_bit,
            { "SAI bit", "palma.control.sai",
            FT_BOOLEAN, 16,
            NULL, PALMA_SAI_CW_BIT,
            NULL, HFILL }},

        { &hf_palma_mac64_cw_bit,
            { "MAC64 bit", "palma.control.mac64",
            FT_BOOLEAN, 16,
            NULL, PALMA_MAC64_CW_BIT,
            NULL, HFILL }},

        { &hf_palma_mcast_cw_bit,
            { "MULTICAST bit", "palma.control.mcast",
            FT_BOOLEAN, 16,
            NULL, PALMA_MCAST_CW_BIT,
            NULL, HFILL }},

        { &hf_palma_server_cw_bit,
            { "SERVER bit", "palma.control.server",
            FT_BOOLEAN, 16,
            NULL, PALMA_SERVER_CW_BIT,
            NULL, HFILL }},

        { &hf_palma_mac_provided_cw_bit,
            { "MAC provided bit", "palma.control.macprov",
            FT_BOOLEAN, 16,
            NULL, PALMA_SETPROV_CW_BIT,
            NULL, HFILL }},

        { &hf_palma_stationid_cw_bit,
            { "STATION ID bit", "palma.control.stationid",
            FT_BOOLEAN, 16,
            NULL, PALMA_STATIONID_CW_BIT,
            NULL, HFILL }},

        { &hf_palma_networkid_cw_bit,
            { "NETWORK ID bit", "palma.control.networkid",
            FT_BOOLEAN, 16,
            NULL, PALMA_NETWORKID_CW_BIT,
            NULL, HFILL }},

```

```
434         { &hf_palma_codefield_cw_bit,
435             { "Codefield bit", "palma.control.codefield",
436             FT_BOOLEAN, 16,
437             NULL, PALMA_CODEFIELD_CW_BIT,
438             NULL, HFILL }},
439
440         { &hf_palma_vendor_cw_bit,
441             { "Specific Vendor bit", "palma.control.vendor",
442             FT_BOOLEAN, 16,
443             NULL, PALMA_VENDOR_CW_BIT,
444             NULL, HFILL }},
445
446         { &hf_palma_renewal_cw_bit,
447             { "Renewal bit", "palma.control.renewal",
448             FT_BOOLEAN, 16,
449             NULL, PALMA_RENEWAL_CW_BIT,
450             NULL, HFILL }},
451
452         { &hf_palma_token,
453             { "Token", "palma.token",
454                 FT_UINT16, BASE_HEX,
455                 NULL, 0x00,
456                 NULL, HFILL }},
457
458         { &hf_palma_status,
459             { "Status", "palma.status",
460                 FT_UINT8, BASE_HEX,
461                 VALS(palma_status_vals), PALMA_STATUS_MASK,
462                 NULL, HFILL }},
463
464         { &hf_palma_length,
465             { "Length", "palma.length",
466                 FT_UINT16, BASE_DEC,
467                 NULL, PALMA_LENGTH_MASK,
468                 NULL, HFILL }},
469
470     /* PAR */
471         { &hf_palma_par_type[0],
472             { "Type", "palma.par_type_0",
473                 FT_UINT8, BASE_HEX,
474                 VALS(palma_par_type_vals), 0x00,
475                 NULL, HFILL }},
476         { &hf_palma_par_type[1],
477             { "Type", "palma.par_type_1",
478                 FT_UINT8, BASE_HEX,
479                 VALS(palma_par_type_vals), 0x00,
480                 NULL, HFILL }},
481         { &hf_palma_par_type[2],
482             { "Type", "palma.par_type_2",
483                 FT_UINT8, BASE_HEX,
484                 VALS(palma_par_type_vals), 0x00,
```

```
485                 NULL, HFILL }},
486         { &hf_palma_par_type[3],
487             { "Type", "palma.par_type_3",
488                 FT_UINT8, BASE_HEX,
489                 VALS(palma_par_type_vals), 0x00,
490                 NULL, HFILL }},
491         { &hf_palma_par_type[4],
492             { "Type", "palma.par_type_4",
493                 FT_UINT8, BASE_HEX,
494                 VALS(palma_par_type_vals), 0x00,
495                 NULL, HFILL }},
496 
497         { &hf_palma_par_type[5],
498             { "Type", "palma.par_type_5",
499                 FT_UINT8, BASE_HEX,
500                 VALS(palma_par_type_vals), 0x00,
501                 NULL, HFILL }},
502 
503         { &hf_palma_par_len[0],
504             { "Length", "palma.par_len_0",
505                 FT_UINT8, BASE_DEC,
506                 NULL, 0x00,
507                 NULL, HFILL }},
508         { &hf_palma_par_len[1],
509             { "Length", "palma.par_len_1",
510                 FT_UINT8, BASE_DEC,
511                 NULL, 0x00,
512                 NULL, HFILL }},
513         { &hf_palma_par_len[2],
514             { "Length", "palma.par_len_2",
515                 FT_UINT8, BASE_DEC,
516                 NULL, 0x00,
517                 NULL, HFILL }},
518         { &hf_palma_par_len[3],
519             { "Length", "palma.par_len_3",
520                 FT_UINT8, BASE_DEC,
521                 NULL, 0x00,
522                 NULL, HFILL }},
523         { &hf_palma_par_len[4],
524             { "Length", "palma.par_len_4",
525                 FT_UINT8, BASE_DEC,
526                 NULL, 0x00,
527                 NULL, HFILL }},
528         { &hf_palma_par_len[5],
529             { "Length", "palma.par_len_5",
530                 FT_UINT8, BASE_DEC,
531                 NULL, 0x00,
532                 NULL, HFILL }},
533 
534         { &hf_palma_station_id,
535             { "Station Id", "palma.station_id",
```

```
                FT_STRING, BASE_NONE,
                NULL, 0x00,
                NULL, HFILL }},

        { &hf_palma_48_bit_address[0],
            { "Address", "palma.addr_48",
                FT_ETHER, BASE_NONE,
                NULL, 0x00,
                NULL, HFILL }},

        { &hf_palma_64_bit_address[0],
            { "Address", "palma.addr_64",
                FT_UINT64, BASE_HEX,
                NULL, 0x00,
                NULL, HFILL }},

        { &hf_palma_set_len[0],
            { "Count", "palma.addr_len",
                FT_UINT16, BASE_DEC,
                NULL, 0x00,
                NULL, HFILL }},

        { &hf_palma_48_bit_mask[0],
            { "Mask", "palma.addr_mask_48",
                FT_ETHER, BASE_NONE,
                NULL, 0x00,
                NULL, HFILL }},

        { &hf_palma_64_bit_mask[0],
            { "Mask", "palma.addr_mask_64",
                FT_UINT64, BASE_HEX,
                NULL, 0x00,
                NULL, HFILL }},

        { &hf_palma_48_bit_address[1],
            { "Conflict Address", "palma.confict_addr_48",
                FT_ETHER, BASE_NONE,
                NULL, 0x00,
                NULL, HFILL }},

        { &hf_palma_64_bit_address[1],
            { "Conflict Address", "palma.confict_addr_64",
                FT_UINT64, BASE_HEX,
                NULL, 0x00,
                NULL, HFILL }},

        { &hf_palma_set_len[1],
            { "Count", "palma.confict_addr_len",
                FT_UINT16, BASE_DEC,
                NULL, 0x00,
                NULL, HFILL }},
```

```
587
588         { &hf_palma_48_bit_mask[1],
589             { "Mask", "palma.confict_addr_mask_48",
590                 FT_ETHER, BASE_NONE,
591                 NULL, 0x00,
592                 NULL, HFILL }},
593
594         { &hf_palma_64_bit_mask[1],
595             { "Mask", "palma.confict_addr_mask_64",
596                 FT_UINT64, BASE_HEX,
597                 NULL, 0x00,
598                 NULL, HFILL }},
599
600         { &hf_palma_network_id,
601             { "Network Id", "palma.network_id",
602                 FT_STRING, BASE_NONE,
603                 NULL, 0x00,
604                 NULL, HFILL }},
605
606         { &hf_palma_lifetime,
607             { "Lifetime", "palma.lifetime",
608                 FT_UINT16, BASE_DEC,
609                 NULL, 0x00,
610                 NULL, HFILL }},
611
612         { &hf_palma_client_address,
613             { "Client Address", "palma.client_addr",
614                 FT_ETHER, BASE_NONE,
615                 NULL, 0x00,
616                 NULL, HFILL }},
617
618     { &hf_palma_vendor,
619             { "Vendor", "palma.vendor",
620                 FT_STRING, BASE_NONE,
621                 NULL, 0x00,
622                 NULL, HFILL }},
623
624     }; /* end of static hf_register_info hf[] = */
625
626     /* Setup protocol subtree array */
627     static gint *ett[] = { &ett_palma,
628             &ett_control_bits,
629             &ett_par[0],
630             &ett_par[1],
631             &ett_par[2],
632             &ett_par[3],
633             &ett_par[4],
634             &ett_par[5],
635             };
636
637     /* Register the protocol name and description */
```

```
638     proto_palma = proto_register_protocol (
639         "IEEE PALMA Protocol", /* name */
640         "PALMA", /* short name */
641         "palma" /* abbrev */
642         );
643 
644     /* Required function calls to register the header fields and subtrees
        used */
645     proto_register_field_array(proto_palma, hf, array_length(hf));
646     proto_register_subtree_array(ett, array_length(ett));
647 
648 } /* end proto_register_palma() */
649 
650 void
651 proto_reg_handoff_palma(void)
652 {
653     dissector_handle_t palma_handle;
654 
655     palma_handle = create_dissector_handle(dissect_palma, proto_palma);
656     dissector_add_uint("ethertype", PALMA_ETHERTYPE, palma_handle);
657 }
```

Appendix B

Appendix B: PALMA Code

B.1 Common Modules

B.1.1 Network Management

```
#ifndef NETITF_H
#define NETITF_H

#include <linux/if_packet.h>
#include <stdint.h>
#include "eventloop.h"
#include "packet.h"

class Palma;

class NetItf : public EventSource
{
  int m_ifidx;
  Palma *m_protocol;

public:
  NetItf(Palma *protocol);
  ~NetItf();
  void init(uint8_t *ifname);
  int onInput();
  void netsend(Packet *pkt);
  void fillMreq(packet_mreq& mreq, uint64_t addr, bool multicast);
  void addAddr(uint64_t addr, bool multicast = false);
  void delAddr(uint64_t addr, bool multicast = false);
};

#endif
```

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
```

```
#include <sys/socket.h>
#include <net/ethernet.h>
#include <arpa/inet.h>
#include <sys/ioctl.h>
#include <net/if.h>
#include <memory.h>
#include <unistd.h>

#include "netitf.h"
#include "details.h"
#include "palma.h"
#include "packet.h"

NetItf::NetItf(Palma *protocol) : m_protocol(protocol){}

void NetItf::init(uint8_t *ifname)
{
  int res;
  ifreq ifbuf = {0};
  sockaddr_ll saddr = {0};

  m_fd = socket(PF_PACKET, SOCK_RAW, htons(PALMA_TYPE));
  if(m_fd < 0)
  {
    perror("Opening socket");
    exit(1);
  }
  strncpy(ifbuf.ifr_name, (char *)ifname, IFNAMSIZ);
  res = ioctl(m_fd, SIOCGIFINDEX, &ifbuf);
  if(res < 0)
  {
    perror("Getting interface index");
    exit(1);
  }
  m_ifidx = ifbuf.ifr_ifindex;
  saddr.sll_family = AF_PACKET;
  saddr.sll_ifindex = m_ifidx;
  saddr.sll_halen = ETH_ALEN;

  res = bind(m_fd, (sockaddr*)&saddr, sizeof(saddr));
  if(res != 0)
  {
    perror("Binding datagram socket");
    exit(1);
  }
}

NetItf::~NetItf()
{
  close(m_fd);
}
```

```
55
56 int NetItf::onInput()
57 {
58   uint8_t rcvbuf[MAX_PKT_SIZE+1];
59   int rcvlen;
60
61   while ((rcvlen = recv(m_fd, rcvbuf, MAX_PKT_SIZE+1, MSG_DONTWAIT)) >= 0)
62   {
63     Packet pkt;                     //solo para depurar
64     /*
65     printf("\nRecibidos %d bytes\n",rcvlen);
66     printf("\nDATA->");
67     for(int i=0; i<rcvlen; i++)
68       printf("%02x ",rcvbuf[i]);
69     printf("\n");
70     */
71     if(pkt.parse(rcvbuf, rcvlen) == 0 && pkt.check())
72       m_protocol->handlePacket(&pkt);
73   }
74   if (errno != EAGAIN && errno != EWOULDBLOCK)
75   {
76     perror("Reading from network socket");
77     exit(1);
78   }
79 }
80
81 void NetItf::netsend(Packet *pkt)
82 {
83   uint8_t sndbuf[MAX_PKT_SIZE];
84
85   uint16_t len = pkt->toBuffer(sndbuf);
86   int res = send(m_fd, sndbuf, len, 0);
87   if(res  < 0)
88   {
89     perror("Sending packet");
90     exit(1);
91   }
92   /*
93   printf("\nEnviados %d bytes->\n",res);
94   for(int i=0; i<res; i++)
95     printf("%02x ",sndbuf[i]);
96   printf("\n");
97   */
98 }
99
100 void NetItf::fillMreq(packet_mreq& mreq, uint64_t addr, bool multicast)
101 {
102   addr = htobe64(addr);
103   mreq.mr_ifindex = m_ifidx;
104   mreq.mr_type = multicast ? PACKET_MR_MULTICAST : PACKET_MR_UNICAST;
105   mreq.mr_alen = 6;
```

```
106   memcpy(mreq.mr_address, &addr, mreq.mr_alen);
107 }
108
109 void NetItf::addAddr(uint64_t addr, bool multicast)
110 {
111   packet_mreq mreq = {0};
112   fillMreq(mreq, addr, multicast);
113
114   int res = setsockopt(m_fd, SOL_PACKET, PACKET_ADD_MEMBERSHIP, &mreq,
       sizeof(mreq));
115   if(res  < 0)
116   {
117     perror("Adding address to filter");
118     exit(1);
119   }
120 }
121
122 void NetItf::delAddr(uint64_t addr, bool multicast)
123 {
124   packet_mreq mreq = {0};
125   fillMreq(mreq, addr, multicast);
126
127   int res = setsockopt(m_fd, SOL_PACKET, PACKET_DROP_MEMBERSHIP, &mreq,
       sizeof(mreq));
128   if(res  < 0)
129   {
130     perror("Dropping address to filter");
131     exit(1);
132   }
133 }
```

B.1.2 MAC Address Set

```
#ifndef SET_H
#define SET_H

#include <stdint.h>

enum class SetType : uint8_t
{
  AUTO = 0,
  ADDR = 1,
  MASK = 2,
};

enum class SetSize : uint8_t
{
  AUTO = 0,
  SIZE48 = 6,
  SIZE64 = 8,
};

/* Same bit structure as control word */

#define AAI    (1<<0)
#define ELI    (1<<1)
#define SAI    (1<<2)
#define SZ64   (1<<4)
#define MCST   (1<<5)



class AddrSet
{
public:
  uint64_t m_addr;
  uint64_t m_val;
  SetType m_type;
  SetSize m_size;

  AddrSet(uint64_t addr=0, uint64_t count=1, SetSize size = SetSize::AUTO,
    SetType type = SetType::AUTO);
  uint8_t slapBits();
  uint8_t slapType();
  bool isELI();
  bool isSAI();
  bool isAAI();
  bool isLocal();
  bool isMulticast();
  bool isSize48();
  bool isSize64();
  uint8_t addrLen();
```

```
  uint64_t getAlignedMask();
  void alignToMask(SetType type);
  bool checkConflict(const AddrSet *Set1, const AddrSet *Set2);
  uint64_t getFirstAddr();
  uint64_t getLastAddr();
  uint64_t getMask();
  SetType getType();
  uint64_t getSize();
  uint64_t getAlignedSize();
  void setSize(uint64_t new_size);
};

class RandomAddrSet : public AddrSet
{
public:
  RandomAddrSet(const AddrSet& addr, uint64_t count=1);
};

#endif
```

```
#include <stdlib.h>
#include "addrset.h"

#define Z_BIT (1<<3)
#define Y_BIT (1<<2)
#define X_BIT (1<<1)
#define M_BIT (1<<0)

AddrSet::AddrSet(uint64_t addr, uint64_t val, SetSize size, SetType type)
{
  m_addr = addr;
  m_val = val;
  m_type = type != SetType::AUTO ? type : SetType::ADDR;
  m_size = size != SetSize::AUTO ? size : (addr > 0xffffffffffff) ?
   SetSize::SIZE64 : SetSize::SIZE48;
  if (m_type == SetType::MASK)
    m_addr &= m_val;
  if (m_size == SetSize::SIZE48)
  {
    m_addr &= 0x0000FFFFFFFFFFFF;
    if (m_type == SetType::MASK)
      m_val |= 0xFFFF000000000000;
  }

}

uint8_t AddrSet::slapBits()
{
  switch(m_size)
  {
    case SetSize::SIZE48:
      return (uint8_t)(m_addr >> 40) & 0xF;
```

```
    case SetSize::SIZE64:
      return (uint8_t)(m_addr >> 56) & 0xF;
  }
  return 0;
}

uint8_t AddrSet::slapType()
{
  uint8_t res = isELI() ? ELI : isSAI() ? SAI : isAAI() ? AAI : 0;
  res |= isSize64() ? SZ64 : 0;
  res |= isMulticast() ? MCST : 0;
}

bool AddrSet::isELI()
{
  return ((slapBits() & (Y_BIT|Z_BIT|X_BIT)) == (Z_BIT|X_BIT));
}

bool AddrSet::isSAI()
{
  return ((slapBits() & (Y_BIT|Z_BIT|X_BIT)) == (Y_BIT|Z_BIT|X_BIT));
}

bool AddrSet::isAAI()
{
  return ((slapBits() & (Y_BIT|Z_BIT|X_BIT)) == X_BIT);
}

bool AddrSet::isLocal()
{
  return ((slapBits() & X_BIT) == X_BIT);
}

bool AddrSet::isMulticast()
{
  return ((slapBits() & M_BIT) == M_BIT);
}

bool AddrSet::isSize48()
{
  return (m_size == SetSize::SIZE48);
}

bool AddrSet::isSize64()
{
  return (m_size == SetSize::SIZE64);
}

uint8_t AddrSet::addrLen()
{
  return (uint8_t) m_size;
```

```
}


static void convertToMask(uint64_t &addr, uint64_t &val)
{
  uint64_t aux = addr + val - 1;
  val = 0xFFFFFFFFFFFFFFFF;
  aux ^= ~addr;
  for (int i=0; i<64 && (aux & (1L<<63)); i++)
  {
    aux <<= 1;
    val >>= 1;
  }
  val = ~val;
  addr &= val;
}

static void convertToAddr(uint64_t &val)
{
  uint64_t aux = ~val;
  uint64_t count = 1;
  for (int i=0; i<64 && (aux & 1); i++)
  {
    aux >>= 1;
    count <<= 1;
  }
  val = count;
}

uint64_t AddrSet::getAlignedMask()
{
  uint64_t mask, first, last, new_mask;
  mask = 0xffffffffffffffff;
  first = getFirstAddr();
  last = getLastAddr();
  while(first >= getFirstAddr() && last <= getLastAddr())
  {
    new_mask = mask;
    mask <<= 1;
    first = (first + ~mask) & mask;
    last = first + ~mask;
  }
  return new_mask;
}

void AddrSet::alignToMask(SetType type)
{
  if(m_type == SetType::MASK)
  {
    if(type == SetType::ADDR)
    {
```

```
      convertToAddr(m_val);
      m_type = type;
    }
    return;
  }
  m_val = getAlignedMask();
  m_addr = (m_addr + ~m_val) & m_val;
  if(type == SetType::MASK)
    m_type = type;
  else
    m_val = ~m_val + 1;
}


bool AddrSet::checkConflict(const AddrSet *set1, const AddrSet *set2)
{
  if(set1 == NULL || set2 == NULL)
    return false;
  uint64_t addr1 = set1->m_addr;
  uint64_t val1 = set1->m_val;
  uint64_t addr2 = set2->m_addr;
  uint64_t val2 = set2->m_val;
  uint64_t aux;
  if(set1->m_size != set2->m_size)
    return false;
  if(set1->m_type == SetType::MASK)
    convertToAddr(val1);
  if(set2->m_type == SetType::MASK)
    convertToAddr(val2);
  if(val1 == 0 || val2 == 0)
    return false;
  m_type = SetType::ADDR;
  m_size = set1->m_size;
  m_addr = (addr1 > addr2) ? addr1 : addr2;
  m_val = (addr1 + val1 < addr2 + val2) ? addr1 + val1 : addr2 + val2;
  if(m_addr >= m_val)
    return false;
  m_val -= m_addr;
  return true;
}

uint64_t AddrSet::getFirstAddr()
{
  return m_addr;
}

uint64_t AddrSet::getLastAddr()
{
  return m_addr + getSize() - 1;
}

```

```
185 uint64_t AddrSet::getMask()
186 {
187   if(m_type == SetType::MASK)
188     return m_val;
189   else
190     return ~(getSize() - 1);
191 }
192
193 SetType AddrSet::getType()
194 {
195   return m_type;
196 }
197
198 uint64_t AddrSet::getSize()
199 {
200   uint64_t count = m_val;
201   if(m_type == SetType::MASK)
202     convertToAddr(count);
203   return count;
204 }
205
206 uint64_t AddrSet::getAlignedSize()
207 {
208   if(m_type == SetType::MASK)
209     return getSize();
210   return  (~getAlignedMask() + 1);
211 }
212
213 void AddrSet::setSize(uint64_t new_size)
214 {
215   if(m_type == SetType::MASK)
216     m_type = SetType::ADDR;
217   m_val = new_size;
218 }
219
220 RandomAddrSet::RandomAddrSet(const AddrSet& set, uint64_t val)
221 {
222   uint64_t count = ((AddrSet)set).getSize();
223   if(val > count)
224     val = count;
225   m_type = SetType::ADDR;
226   m_size = set.m_size;
227   uint64_t random = (lrand48() ^ ((uint64_t)lrand48() << 32));
228   if(val > 0xffff)
229   {
230     m_val = val;
231     uint64_t mask = getAlignedMask();
232     AddrSet pool = set;
233     pool.alignToMask(SetType::MASK);
234     random &= (mask ^ pool.getMask());
235     m_addr = pool.getFirstAddr() + random;
```

```
236     m_type = SetType::MASK;
237     m_val = mask;
238   }
239   else
240   {
241     m_addr = ((AddrSet)set).getFirstAddr() + (count <= val ? 0 : (random %
      (count - val)));
242     m_val = val;
243   }
244 }
```

B.1.3 PALMA Packet

```
#ifndef PACKET_H
#define PACKET_H

#include <stdint.h>
#include "addrset.h"

#define ETH_HDR_SIZE  (2*6 + 2)
#define HEADER_SIZE   8
#define MIN_PKT_SIZE  (ETH_HDR_SIZE + HEADER_SIZE)
#define MAX_PKT_SIZE  (MIN_PKT_SIZE+4+18+2*255+10+255) /* Fix part +
    Lifetime + MacAddrSet +  2 x id + ClientAddr + Vendor */
#define MAX_PAR   6

enum class MsgType : uint8_t
{
  DISCOVER    = 1,
  OFFER     = 2,
  REQUEST     = 3,
  ACK       = 4,
  RELEASE     = 5,
  DEFEND      = 6,
  ANNOUNCE    = 7,
};

enum class ParType : uint8_t
{
  STATION_ID    = 1,
  MAC_ADDR_SET  = 2,
  NETWORK_ID    = 3,
  LIFETIME      = 4,
  CLIENT_ADDR   = 5,
  VENDOR      = 6,
  MAX_PARTYPE   = 7
};

enum class StatusCode : uint8_t
{
  NO_CODE     = 0,
  ASSIGN_OK   = 1,
  ALTERNATE_SET = 2,
  FAIL_CONFLICT = 3,
  FAIL_DISALLOWED = 4,
  FAIL_TOO_LARGE  = 5,
  FAIL_OTHER    = 6,
};

class PacketPar
{
public:
```

```
49    ParType m_par_id;
50    uint8_t m_length;
51
52    PacketPar(ParType par_id, uint8_t len);
53    virtual ~PacketPar() {};
54    virtual int toBuffer(uint8_t *&data);
55    virtual PacketPar* clone() = 0;
56  };
57
58  class IdPar : public PacketPar
59  {
60  public:
61    uint8_t *m_id;
62
63    IdPar(ParType par_id, uint8_t *data, uint8_t len);
64    ~IdPar();
65    int toBuffer(uint8_t *&data);
66    PacketPar* clone();
67  };
68
69  class MacSetPar : public PacketPar
70  {
71  public:
72    AddrSet m_set;
73
74    MacSetPar(uint8_t *data, uint8_t len);
75    MacSetPar(AddrSet &set);
76    ~MacSetPar() {}
77    int toBuffer(uint8_t *&data);
78    uint8_t length(AddrSet &set);
79    PacketPar* clone();
80  };
81
82
83  class LifetimePar : public PacketPar
84  {
85  public:
86    uint16_t m_lifetime;
87
88    LifetimePar(uint8_t *data, uint8_t len);
89    LifetimePar(uint16_t lifetime);
90    ~LifetimePar() {}
91    int toBuffer(uint8_t *&data);
92    PacketPar* clone();
93  };
94
95  class ClientAddrPar : public PacketPar
96  {
97  public:
98    AddrSet m_set;
99
```

```
100   ClientAddrPar(uint8_t *data, uint8_t len);
101   ClientAddrPar(AddrSet &set);
102   ~ClientAddrPar() {}
103   int toBuffer(uint8_t *&data);
104   PacketPar* clone();
105 };
106 
107 class VendorPar : public PacketPar
108 {
109 public:
110   uint8_t *m_var;
111 
112   VendorPar(uint8_t *data, uint8_t var_len);
113   ~VendorPar();
114   int toBuffer(uint8_t *&data);
115   PacketPar* clone();
116 };
117 
118 class Packet
119 {
120   uint64_t m_DA;
121   uint64_t m_SA;
122   uint8_t m_version;
123   MsgType m_message_type;
124   uint16_t m_control_word;
125   uint16_t m_token;
126   StatusCode m_status;
127   uint16_t m_length;
128   uint8_t m_num_par;
129   PacketPar *m_par[MAX_PAR];
130 
131 public:
132 
133   Packet();
134   Packet(Packet *pkt);
135   Packet(MsgType type, uint64_t DA, uint64_t SA, uint16_t token,
       StatusCode status = StatusCode::NO_CODE);
136   ~Packet();
137   int addPar(PacketPar *par);
138   int addIdPar(ParType par_id, uint8_t *id);
139   int addMacSetPar(AddrSet *set, bool update_cw = true);
140   int addLifetimePar(uint16_t lifetime);
141   int addVendorPar(uint8_t *var, uint8_t var_len);
142   int addClientAddrPar(AddrSet *set);
143   int parsePar(uint8_t *&data, int &len, PacketPar *&Par);
144   static uint64_t get64(uint8_t *&p, uint8_t nbytes = 8);
145   static void set64(uint64_t val, uint8_t *&p,  uint8_t nbytes = 8);
146   static uint16_t get16(uint8_t *&p);
147   static void set16(uint16_t val, uint8_t *&p);
148   int parse(uint8_t *data, int len);
149   bool check();
```

```
  int toBuffer(uint8_t *data);
  void setSA(uint64_t addr);
  void setRenewal();
  bool getRenewal();
  MsgType getType();
  uint64_t getDA();
  uint64_t getSA();
  uint16_t getToken();
  StatusCode getStatus();
  AddrSet *getSet(bool first_instance = true);
  uint64_t getClientAddr();
  uint16_t getLifetime();
  uint8_t *getStationId();
  uint8_t *getNetworkId();
  uint8_t *getVendorVar();
};

#endif
```

```
#include <string.h>
#include <endian.h>
#include "packet.h"
#include "details.h"

#include <stdio.h>

/* Same bit structure as slap_type */

#define SLAP_CW     0x37
#define AAI_CW      (1<<0)
#define ELI_CW      (1<<1)
#define SAI_CW      (1<<2)
#define MAC64_CW    (1<<4)
#define MCAST_CW    (1<<5)

#define SERVER_CW   (1<<6)
#define SETPROV_CW    (1<<7)
#define STATIONID_CW  (1<<8)
#define NETWORKID_CW  (1<<9)
#define CODEFIELD_CW  (1<<10)
#define VENDOR_CW   (1<<11)
#define RENEWAL_CW    (1<<12)


PacketPar::PacketPar(ParType par_id, uint8_t len) : m_par_id(par_id),
    m_length(len) {}

int PacketPar::toBuffer(uint8_t *&data)
{
  *data++ = (uint8_t) m_par_id;
  *data++ = m_length + 2;
  return 2;
```

```
33 }
34
35 IdPar::IdPar(ParType par_id, uint8_t *data, uint8_t len) : PacketPar(
       par_id, len) //Constructor para sacar el Parameter Station/Network
                                                    del paquete
36 {
37   m_id = new uint8_t[len+1];
38   memcpy(m_id,data,len);
39   m_id[len] = 0;
40 }
41
42 IdPar::~IdPar()
43 {
44   delete m_id;
45 }
46
47 int IdPar::toBuffer(uint8_t *&data)
48 {
49   int len = PacketPar::toBuffer(data);
50   memcpy(data, m_id, m_length);
51   data += m_length;
52   return len + m_length;
53 }
54
55 PacketPar* IdPar::clone()
56 {
57   return new IdPar(m_par_id, m_id, m_length);
58 }
59
60 MacSetPar::MacSetPar(uint8_t *data, uint8_t len) : PacketPar(ParType::
       MAC_ADDR_SET, len)
61 {
62   uint64_t addr;
63
64   if (len < 12)
65   {
66     uint16_t count;
67     len -= 2;
68     addr = Packet::get64(data, len);
69     count = Packet::get16(data);
70     m_set = AddrSet(addr, count, (len == 6) ? SetSize::SIZE48 : SetSize::
      SIZE64, SetType::ADDR);
71   }
72   else
73   {
74     uint64_t mask;
75     len /= 2;
76     addr = Packet::get64(data, len);
77     mask = Packet::get64(data, len);
78     m_set = AddrSet(addr, mask, (len == 6) ? SetSize::SIZE48 : SetSize::
      SIZE64, SetType::MASK);
```

```
  }
}

MacSetPar::MacSetPar(AddrSet &set) : m_set(set), PacketPar(ParType::
    MAC_ADDR_SET, length(set)) {}

int MacSetPar::toBuffer(uint8_t *&data)
{
  int len = PacketPar::toBuffer(data);
  Packet::set64(m_set.getFirstAddr(), data, m_set.addrLen());
  if (m_set.getType() == SetType::ADDR)
    Packet::set16(m_set.getSize(), data);
  else
    Packet::set64(m_set.getMask(), data, m_set.addrLen());
  return len + m_length;
}

uint8_t MacSetPar::length(AddrSet &set)
{
  uint8_t len = set.addrLen();
  if (set.m_type == SetType::ADDR)
    return len + 2;
  else
    return len * 2;
}

PacketPar* MacSetPar::clone()
{
  return new MacSetPar(m_set);
}

LifetimePar::LifetimePar(uint8_t *data, uint8_t len) : PacketPar(ParType::
    LIFETIME, len)
{
  m_lifetime = Packet::get16(data);
}

LifetimePar::LifetimePar(uint16_t lifetime) :
        PacketPar(ParType::LIFETIME, 2), m_lifetime(lifetime) {}

int LifetimePar::toBuffer(uint8_t *&data)
{
  int len = PacketPar::toBuffer(data);
  Packet::set16(m_lifetime, data);
  return len + m_length;
}

PacketPar* LifetimePar::clone()
{
  return new LifetimePar(m_lifetime);
}
```

```
128
129 ClientAddrPar::ClientAddrPar(uint8_t *data, uint8_t len) : PacketPar(
        ParType::CLIENT_ADDR, len)
130 {
131   uint64_t addr = Packet::get64(data, len);
132   m_set = AddrSet(addr, 1, (len == 6) ? SetSize::SIZE48 : SetSize::SIZE64,
        SetType::ADDR);
133 }
134
135 ClientAddrPar::ClientAddrPar(AddrSet &set) : m_set(set), PacketPar(ParType
        ::CLIENT_ADDR, set.addrLen()) {}
136
137 int ClientAddrPar::toBuffer(uint8_t *&data)
138 {
139   int len = PacketPar::toBuffer(data);
140   Packet::set64(m_set.m_addr, data, m_set.addrLen());
141   return len + m_length;
142 }
143
144 PacketPar* ClientAddrPar::clone()
145 {
146   return new ClientAddrPar(m_set);
147 }
148
149 VendorPar::VendorPar(uint8_t *data, uint8_t var_len) : PacketPar(ParType::
        VENDOR, var_len)
150 {
151   m_var = new uint8_t[var_len];
152   memcpy(m_var,data,var_len);
153 }
154
155 VendorPar::~VendorPar()
156 {
157   delete m_var;
158 }
159
160 int VendorPar::toBuffer(uint8_t *&data)
161 {
162   int len = PacketPar::toBuffer(data);
163   memcpy(data, m_var, m_length);
164   data += m_length;
165   return len + m_length;
166 }
167
168 PacketPar* VendorPar::clone()
169 {
170   return new VendorPar(m_var, m_length);
171 }
172
173 Packet::Packet()
174 {
```

```
175   for(m_num_par = 0; m_num_par < MAX_PAR; m_num_par++)
176     m_par[m_num_par] = NULL;
177 }
178
179 Packet::Packet(Packet *pkt)
180 {
181   *this = *pkt;
182   for(int num_par = 0; num_par < m_num_par; num_par++)
183     m_par[num_par] = pkt->m_par[num_par]->clone();
184 }
185
186 Packet::Packet(MsgType type, uint64_t DA, uint64_t SA, uint16_t token,
      StatusCode status)
187 {
188   m_DA = DA;
189   m_SA = SA;
190   m_version = 0;
191   m_message_type = type;
192   m_control_word = (type == MsgType::OFFER ||
193             type == MsgType::ACK) ? SERVER_CW : 0;
194   m_control_word |= ((uint8_t)status != 0) ? CODEFIELD_CW : 0;
195   m_token = token;
196   m_status = status;
197   m_length = HEADER_SIZE;
198   for(m_num_par = 0; m_num_par < MAX_PAR; m_num_par++)
199     m_par[m_num_par] = NULL;
200   m_num_par = 0;
201 }
202
203 Packet::~Packet()
204 {
205   for(int m_num_par = 0; m_num_par < MAX_PAR; m_num_par++)
206     delete m_par[m_num_par];
207 }
208
209 int Packet::addPar(PacketPar *par)
210 {
211   if(m_num_par < MAX_PAR)
212   {
213     m_par[m_num_par++] = par;
214     m_length += (par->m_length +2);
215     return 0;
216   }
217   return -1;
218 }
219
220 int Packet::addIdPar(ParType par_id, uint8_t *id)
221 {
222   m_control_word |= (par_id == ParType::STATION_ID) ? STATIONID_CW :
      NETWORKID_CW;
223   return addPar(new IdPar(par_id, id, strlen((char *)id)));
```

```
}

int Packet::addMacSetPar(AddrSet *set, bool update_cw)
{
  if (update_cw)
    m_control_word |= (SETPROV_CW | set->slapType());
  return addPar(new MacSetPar(*set));
}

int Packet::addLifetimePar(uint16_t lifetime)
{
  return addPar(new LifetimePar(lifetime));
}

int Packet::addClientAddrPar(AddrSet *set)
{
  return addPar(new ClientAddrPar(*set));
}

int Packet::addVendorPar(uint8_t *var, uint8_t var_len)
{
  m_control_word |= VENDOR_CW;
  return addPar(new VendorPar(var, var_len));
}

int Packet::parsePar(uint8_t *&data, int &len, PacketPar *&par)
{
  ParType par_id = (ParType)*data++;
  uint8_t par_len = *data++;
  if(len < par_len)
    return -1;
  par_len -= 2;
  switch(par_id)
  {
    case ParType::STATION_ID :
    case ParType::NETWORK_ID :
      if (par_len <= 1)
        return -1;
      par = new IdPar(par_id, data, par_len);
      break;
    case ParType::MAC_ADDR_SET :
      if (par_len != 8 && par_len != 10 && par_len != 12 && par_len != 16)
        return -1;
      par = new MacSetPar(data, par_len);
      break;
    case ParType::CLIENT_ADDR :
      if (par_len != 6 && par_len != 8)
        return -1;
      par = new ClientAddrPar(data, par_len);
      break;
    case ParType::LIFETIME :
```

```
275       if (par_len != 2)
276         return -1;
277       par = new LifetimePar(data, par_len);
278       break;
279     case ParType::VENDOR :
280       if (par_len <= 1)
281         return -1;
282       par = new VendorPar(data, par_len);
283       break;
284     default:
285       return -1;
286   }
287   data += par_len;
288   len -= par_len + 2;
289   return 0;
290 }
291
292 uint64_t Packet::get64(uint8_t *&p, uint8_t nbytes)
293 {
294   uint64_t tmp64;
295
296   memcpy(&tmp64, p, nbytes);
297   p += nbytes;
298   return(be64toh(tmp64) >> 8*(8-nbytes));
299 }
300
301 void Packet::set64(uint64_t val, uint8_t *&p, uint8_t nbytes)
302 {
303   uint64_t tmp64;
304
305   tmp64 = htobe64(val << 8*(8-nbytes));
306   memcpy(p, &tmp64, nbytes);
307   p += nbytes;
308 }
309
310 uint16_t Packet::get16(uint8_t *&p)
311 {
312   uint64_t tmp16;
313
314   memcpy(&tmp16, p, 2);
315   p += 2;
316   return(be16toh(tmp16));
317 }
318
319 void Packet::set16(uint16_t val, uint8_t *&p)
320 {
321   uint16_t tmp16;
322
323   tmp16 = htobe16(val);
324   memcpy(p, &tmp16, 2);
325   p += 2;
```

```
}

int Packet::parse(uint8_t *data, int len)
{
  if(len < MIN_PKT_SIZE || len > MAX_PKT_SIZE)
    return -1;

  m_DA = get64(data, 6);
  m_SA = get64(data, 6);

  if(get16(data) != PALMA_TYPE || *data++ != PALMA_SUBTYPE)
    return -1;

  m_version = (*data & 0xE0) >> 5;
  m_message_type = (MsgType)(*data++ & 0x1F);
  m_control_word = get16(data);
  m_token = get16(data);
  m_status = (StatusCode) ((*data & 0xF0) >> 4);
  m_length = ((uint16_t)(*data++ & 0x0F)) << 8;
  m_length |= (*data++);

  len -= MIN_PKT_SIZE;
  if(len != m_length - HEADER_SIZE)
    return -1;

  for(m_num_par = 0 ; m_num_par < MAX_PAR && len >= 2; m_num_par++)
  {
    int res = parsePar(data, len, m_par[m_num_par]);
    if(res < 0)
      return res;
  }

  if(len != 0)
    return -1;
  return 0;
}

bool Packet::check()
{
  int par_cnt[(uint8_t) ParType::MAX_PARTYPE] = {0};
  for(int num_par = 0 ; num_par < m_num_par; num_par++)
  {
    uint16_t val;
    par_cnt[(uint8_t)m_par[num_par]->m_par_id]++;
    switch(m_par[num_par]->m_par_id)
    {
      case ParType::STATION_ID:
        if(!(m_control_word & STATIONID_CW))
          return false;
        break;
      case ParType::MAC_ADDR_SET:
```

```
377         val = SETPROV_CW | ((MacSetPar *)(m_par[num_par]))->m_set.slapType
    ();
378         if((m_control_word & (SETPROV_CW | SLAP_CW)) != val)
379           return false;
380         break;
381       case ParType::NETWORK_ID:
382         if(!(m_control_word & NETWORKID_CW))
383           return false;
384         break;
385       case ParType::VENDOR:
386         if(!(m_control_word & VENDOR_CW))
387           return false;
388         break;
389     }
390   }
391   if(par_cnt[(uint8_t)ParType::MAC_ADDR_SET] > 2
392     || par_cnt[(uint8_t)ParType::MAC_ADDR_SET] == 2 && m_message_type !=
    MsgType::DEFEND)
393     return false;
394 
395   for(int i = 0; i < (uint8_t)ParType::MAX_PARTYPE; i++)
396   {
397     if(i != (uint8_t)ParType::MAC_ADDR_SET && par_cnt[i] > 1)
398       return false;
399   }
400   switch(m_message_type)
401   {
402     case MsgType::DISCOVER:
403       if((m_control_word & (SERVER_CW | NETWORKID_CW | CODEFIELD_CW |
    RENEWAL_CW))
404           || par_cnt[(uint8_t)ParType::LIFETIME] > 0)
405         return false;
406       break;
407     case MsgType::OFFER:
408       if((m_control_word & (CODEFIELD_CW | RENEWAL_CW))
409           || par_cnt[(uint8_t)ParType::LIFETIME] != 1
410           || (m_control_word & (SERVER_CW | SETPROV_CW) != (SERVER_CW |
    SETPROV_CW)))
411         return false;
412       break;
413     case MsgType::REQUEST:
414       if((m_control_word & (SERVER_CW | NETWORKID_CW | CODEFIELD_CW))
415           || par_cnt[(uint8_t)ParType::LIFETIME] > 0
416           || par_cnt[(uint8_t)ParType::CLIENT_ADDR] > 0
417           || (m_control_word & (SETPROV_CW) != SETPROV_CW))
418         return false;
419       break;
420     case MsgType::ACK:
421       if((m_control_word & (RENEWAL_CW))
422           || (m_control_word & (SERVER_CW | CODEFIELD_CW) != (SERVER_CW |
    CODEFIELD_CW))
```

```
423           || par_cnt[(uint8_t)ParType::CLIENT_ADDR] > 0)
424         return false;
425       break;
426     case MsgType::RELEASE:
427       if((m_control_word & (SERVER_CW | NETWORKID_CW | CODEFIELD_CW |
    RENEWAL_CW))
428           || par_cnt[(uint8_t)ParType::LIFETIME] > 0
429           || par_cnt[(uint8_t)ParType::CLIENT_ADDR] > 0
430           || (m_control_word & (SETPROV_CW) != SETPROV_CW))
431         return false;
432       break;
433     case MsgType::DEFEND:
434       if((m_control_word & (SERVER_CW | NETWORKID_CW | CODEFIELD_CW |
    RENEWAL_CW))
435           || par_cnt[(uint8_t)ParType::LIFETIME] != 1
436           || par_cnt[(uint8_t)ParType::CLIENT_ADDR] > 0
437           || (m_control_word & (SETPROV_CW) != SETPROV_CW)
438           || par_cnt[(uint8_t)ParType::MAC_ADDR_SET] != 2)
439         return false;
440       break;
441     case MsgType::ANNOUNCE:
442       if((m_control_word & (SERVER_CW | NETWORKID_CW | CODEFIELD_CW |
    RENEWAL_CW))
443           || par_cnt[(uint8_t)ParType::LIFETIME] != 1
444           || par_cnt[(uint8_t)ParType::CLIENT_ADDR] > 0
445           || (m_control_word & (SETPROV_CW) != SETPROV_CW))
446         return false;
447       break;
448   }
449   return true;
450 }
451
452
453 int Packet::toBuffer(uint8_t *data)
454 {
455   set64(m_DA, data, 6);
456   set64(m_SA, data, 6);
457   set16(PALMA_TYPE, data);
458   *data++ = PALMA_SUBTYPE;
459   *data++ = (m_version << 5) | (uint8_t) m_message_type;
460   set16(m_control_word, data);
461   set16(m_token, data);
462   *data++ = ((uint8_t) m_status << 4) | (m_length >> 8);
463   *data++ = m_length & 0xFF;
464
465   int len = MIN_PKT_SIZE;
466
467   for(int m_num_par = 0 ; m_num_par < MAX_PAR && m_par[m_num_par] != NULL;
    m_num_par++)
468     len += m_par[m_num_par]->toBuffer(data);
469
```

```
470   return len;
471 }
472
473 void Packet::setSA(uint64_t addr)
474 {
475   m_SA = addr;
476 }
477
478 void Packet::setRenewal()
479 {
480   m_control_word |= RENEWAL_CW;
481 }
482
483 bool Packet::getRenewal()
484 {
485   return m_control_word & RENEWAL_CW;
486 }
487
488 MsgType Packet::getType()
489 {
490   return m_message_type;
491 }
492
493 uint64_t Packet::getDA()
494 {
495   return m_DA;
496 }
497
498 uint64_t Packet::getSA()
499 {
500   return m_SA;
501 }
502
503 StatusCode Packet::getStatus()
504 {
505   return m_status;
506 }
507
508 uint16_t Packet::getToken()
509 {
510   return m_token;
511 }
512
513 AddrSet *Packet::getSet(bool first_instance)
514 {
515   if(m_control_word & SETPROV_CW)
516   {
517     for(int i = 0; i < m_num_par; i++)
518     {
519       if(m_par[i] != NULL && (m_par[i]->m_par_id == ParType::MAC_ADDR_SET)
    )
```

```
520         if(first_instance)
521           return &(((MacSetPar *)(m_par[i]))->m_set);
522         else
523           first_instance = true;
524     }
525   }
526   return NULL;
527 }
528
529 uint64_t Packet::getClientAddr()
530 {
531   for(int i = 0; i < m_num_par; i++)
532   {
533     if(m_par[i] != NULL && m_par[i]->m_par_id == ParType::CLIENT_ADDR)
534       return ((ClientAddrPar *)(m_par[i]))->m_set.m_addr;
535   }
536   return 0;
537 }
538
539 uint16_t Packet::getLifetime()
540 {
541   for(int i = 0; i < m_num_par; i++)
542   {
543     if(m_par[i] != NULL && m_par[i]->m_par_id == ParType::LIFETIME)
544       return ((LifetimePar *)(m_par[i]))->m_lifetime;
545   }
546   return 0;
547 }
548
549 uint8_t *Packet::getStationId()
550 {
551   if(m_control_word & STATIONID_CW)
552   {
553     for(int i = 0; i < m_num_par; i++)
554     {
555       if(m_par[i] != NULL && m_par[i]->m_par_id == ParType::STATION_ID)
556         return ((IdPar *)(m_par[i]))->m_id;
557     }
558   }
559   return NULL;
560 }
561
562 uint8_t *Packet::getNetworkId()
563 {
564   if(m_control_word & NETWORKID_CW)
565   {
566     for(int i = 0; i < m_num_par; i++)
567     {
568       if(m_par[i] != NULL && m_par[i]->m_par_id == ParType::NETWORK_ID)
569         return ((IdPar *)(m_par[i]))->m_id;
570     }
```

```
571   }
572   return NULL;
573 }
574
575 uint8_t *Packet::getVendorVar()
576 {
577   if(m_control_word & VENDOR_CW)
578   {
579     for(int i = 0; i < m_num_par; i++)
580     {
581       if(m_par[i] != NULL && m_par[i]->m_par_id == ParType::VENDOR)
582         return ((VendorPar *)(m_par[i]))->m_var;
583     }
584   }
585   return NULL;
586 }
```

B.1.4 Timers

```
#ifndef TIMER_H
#define TIMER_H

#include <time.h>

class Time : public timespec
{
  void normalize();
public:
  Time();
  double elapsed(Time const &t);
  void add(double d);
  double get();
  void set(double d);
};

class Timer
{
public:
  double m_duration;
  Timer *m_next;
  bool active;

  Timer(double t = 0);
  void set(double t);
  virtual void timeout() {}
};

class TimerList
{
public:
  Time m_ref;
  Timer m_first;

  TimerList();

  void refresh();
  double read(Timer *timer);
  void add(Timer *newtimer);
  void del(Timer *timer);
  Time* check(Time *t);
};

#endif
```

```
#include <stdio.h>
#include <stdlib.h>
#include "timer.h"
#define NSPERS  (1000000000UL)

```

```
Time::Time()
{
  clock_gettime(CLOCK_MONOTONIC, this);
}

void Time::normalize()
{
  time_t s = tv_nsec/NSPERS;

  if(s)
  {
    tv_sec += s;
    tv_nsec = tv_nsec%NSPERS;
  }
  if(tv_sec < 0 && tv_nsec > 0)
  {
    tv_sec++;
    tv_nsec -= NSPERS;
  }
  else if(tv_sec > 0 && tv_nsec < 0)
  {
    tv_sec--;
    tv_nsec += NSPERS;
  }
}

double Time::elapsed(Time const &t)
{
  return (tv_nsec - t.tv_nsec)*1e-9 + tv_sec - t.tv_sec;
}

void Time::add(double d)
{
  time_t sec = (time_t)d;
  tv_sec += sec;
  tv_nsec += (long)((d - sec) * 1e9);
  normalize();
}

double Time::get()
{
  return tv_sec + tv_nsec * 1e-9;
}

void Time::set(double d)
{
  time_t sec = (time_t)d;
  tv_sec = sec;
  tv_nsec = (long)((d - sec) * 1e9);
}

```

```
Timer::Timer(double t) : m_duration(t), m_next(NULL), active(false) {}

void Timer::set(double t)
{
  m_duration = t;
}

TimerList::TimerList() : m_first()
{
  refresh();
}

void TimerList::refresh()
{
  Time now;
  if(m_first.m_next != NULL)
    m_first.m_next->m_duration -= now.elapsed(m_ref);
  m_ref = now;
}

double TimerList::read(Timer *timer)
{
  if(!timer->active)
    return 0.;

  double time = 0.;
  refresh();

  for(Timer *p = m_first.m_next; p != timer && p != NULL; p = p->m_next)
  {
    time += p->m_duration;
  }
  time += timer->m_duration;
  return time;
}

void TimerList::add(Timer *newtimer)
{
  refresh();

  newtimer->active = true;
  for(Timer *p = &m_first;; p = p->m_next)
  {
    if(p->m_next == NULL)
    {
      p->m_next = newtimer;
      newtimer->m_next = NULL;
      return;
    }
    else if(newtimer->m_duration < p->m_next->m_duration)
    {
```

```
      p->m_next->m_duration -= newtimer->m_duration;
      newtimer->m_next = p->m_next;
      p->m_next = newtimer;
      return;
    }
    else
      newtimer->m_duration -= p->m_next->m_duration;
  }
}

void TimerList::del(Timer *timer)
{
  Timer *p;
  double left = 0.;

  if(!timer->active)
    return;
  refresh();
  for(p = &m_first; p->m_next != timer; p = p->m_next)
  {
    if(p->m_next == NULL) return;
    left += p->m_next->m_duration;
  }
  if(timer->m_next != NULL)
    timer->m_next->m_duration += timer->m_duration;
  p->m_next = timer->m_next;
  timer->active = false;
  timer->m_duration += left;
}

Time* TimerList::check(Time *t)
{
  if(m_first.m_next == NULL)
    return NULL;
  refresh();

  while(m_first.m_next != NULL && m_first.m_next->m_duration <= 0)
  {
    Timer *p = m_first.m_next;
    p->active = false;
    if(p->m_next != NULL)
      p->m_next->m_duration += p->m_duration;
    m_first.m_next = m_first.m_next->m_next;
    p->timeout();
    refresh();
  }
  if(m_first.m_next != NULL)
    t->set(m_first.m_next->m_duration);
  return t;
}
```

B.1.5 Event Loop

```
#ifndef EVENTLOOP_H
#define EVENTLOOP_H

#include "timer.h"
#include <sys/select.h>

class EventSource
{
public:
  int m_fd;
  EventSource *m_next;

  virtual int onInput() {}
};

class ExitHandler
{
public:
  ExitHandler *m_next = NULL;

  virtual void onExit() {}
};

class EventLoop
{
  fd_set m_readfds;
  int m_nfds;
  EventSource m_first_src;
  TimerList m_timerlist;

  static void doExit(int signum);

public:
  static ExitHandler m_first_hnd;
  static bool m_finalize;

  EventLoop();
  void regSource(EventSource *src);
  void regHandler(ExitHandler *hnd);
  void startTimer(Timer *newtimer, double t = 0.);
  void stopTimer(Timer *timer);
  double readTimer(Timer *timer);
  void unregSource(EventSource *src);
  void unregHandler(ExitHandler *hnd);
  void run();
};

#endif
```

```
#include <stdio.h>
```

```
#include <stdlib.h>
#include <signal.h>
#include "eventloop.h"

bool EventLoop::m_finalize = false;
ExitHandler EventLoop::m_first_hnd;

EventLoop::EventLoop()
{
  FD_ZERO(&m_readfds);
  m_nfds = 0;
  m_first_src.m_next = NULL;

  sigset_t blockset;

  sigemptyset(&blockset);          /* Block SIGINT, SIGTERM */
  sigaddset(&blockset, SIGINT);
  sigaddset(&blockset, SIGTERM);

  sigprocmask(SIG_BLOCK, &blockset, NULL);

  struct sigaction sa;
  sa.sa_handler = doExit;         /* Establish signal handler */
  sa.sa_flags = 0;
  sigemptyset(&sa.sa_mask);
  sigaction(SIGINT, &sa, NULL);
  sigaction(SIGTERM, &sa, NULL);
}

void EventLoop::doExit(int signum)
{
  for(ExitHandler *h = m_first_hnd.m_next; h != NULL; h = h->m_next)
    h->onExit();
  m_finalize = true;
}

void EventLoop::regSource(EventSource *src)
{
  src->m_next = m_first_src.m_next;
  m_first_src.m_next = src;
  FD_SET(src->m_fd, &m_readfds);
  if(m_nfds < src->m_fd) m_nfds = src->m_fd;
}

void EventLoop::regHandler(ExitHandler *hnd)
{
  hnd->m_next = m_first_hnd.m_next;
  m_first_hnd.m_next = hnd;
}

void EventLoop::startTimer(Timer *newtimer, double t)
```

```
{
  if(t != 0.) newtimer->set(t);
  m_timerlist.add(newtimer);
}

void EventLoop::stopTimer(Timer *timer)
{
  m_timerlist.del(timer);
}

double EventLoop::readTimer(Timer *timer)
{
  return m_timerlist.read(timer);
}

void EventLoop::unregSource(EventSource *src)
{
  m_nfds = 0;
  for(EventSource *s = &m_first_src; s != NULL; s = s->m_next)
  {
    if(s->m_next == src)
    {
      s->m_next = s->m_next->m_next;
      FD_CLR(src->m_fd, &m_readfds);
    }
    else if(m_nfds < s->m_next->m_fd) m_nfds = s->m_next->m_fd;
  }
}

void EventLoop::unregHandler(ExitHandler *hnd)
{
  for(ExitHandler *h = &m_first_hnd; h != NULL; h = h->m_next)
    if(h->m_next == hnd)
    {
      h->m_next = h->m_next->m_next;
      break;
    }
}

void EventLoop::run()
{
  fd_set rdfds;
  int n;
  Time timeout;
  sigset_t emptyset;
  sigemptyset(&emptyset);
  while(!m_finalize)
  {
    rdfds = m_readfds;

    n = pselect(m_nfds+1, &rdfds, NULL, NULL,
```

```
104         m_timerlist.check(&timeout), &emptyset);
105     for(EventSource *s = m_first_src.m_next; s != NULL && n > 0; s = s->
    m_next)
106     {
107       if(FD_ISSET(s->m_fd, &rdfds))
108       {
109         s->onInput();
110         n--;
111       }
112     }
113   }
114 }
```

B.1.6 Hashing

```
#ifndef SIPHASH_H
#define SIPHASH_H

#include <inttypes.h>

class SipHash {
private:
    int m_idx;
    uint64_t m_k0, m_k1;
    uint64_t m_v0, m_v1, m_v2, m_v3, m_m;
    uint8_t m_input_len;
public:
    SipHash();
    void begin();
    void update(uint8_t data);
    void update(uint8_t *data);
    void update(uint16_t data);
    void update(uint64_t data);
      uint64_t digest();
};

#endif
```

```
#include <string.h>
#include <stdlib.h>
#include "siphash.h"

#define ROTATE_LEFT(x, b) (unsigned long)(((x) << (b)) | ((x) >> (64 - (b)
    )))

#define COMPRESS                \
  m_v0 += m_v1;                     \
  m_v2 += m_v3;                     \
  m_v1 = ROTATE_LEFT(m_v1, 13);     \
  m_v3 = ROTATE_LEFT(m_v3, 16);     \
  m_v1 ^= m_v0;                     \
  m_v3 ^= m_v2;                     \
  m_v0 = ROTATE_LEFT(m_v0, 32);     \
  m_v2 += m_v1;                     \
  m_v0 += m_v3;                     \
  m_v1 = ROTATE_LEFT(m_v1, 17);     \
  m_v3 = ROTATE_LEFT(m_v3, 21);     \
  m_v1 ^= m_v2;                     \
  m_v3 ^= m_v0;                     \
  m_v2 = ROTATE_LEFT(m_v2, 32);

#define DIGEST_BLOCK            \
  m_v3 ^= m_m;                      \
  do {                          \
    int i;                      \
```

```
    for(i = 0; i < 2; i++){      \
      COMPRESS                   \
    }                            \
  } while (0);                   \
  m_v0 ^= m_m;

SipHash::SipHash()
{
  m_k0 = lrand48() ^ ((uint64_t)lrand48() << 32);
  m_k1 = lrand48() ^ ((uint64_t)lrand48() << 32);
}

void SipHash::begin()
{
  m_v0 = (0x736f6d6570736575 ^ m_k0);
  m_v1 = (0x646f72616e646f6d ^ m_k1);
  m_v2 = (0x6c7967656e657261 ^ m_k0);
  m_v3 = (0x7465646279746573 ^ m_k1);

  m_idx = 0;
  m_input_len = 0;
  m_m = 0;
}

void SipHash::update(uint8_t b)
{
  m_input_len++;
  m_m |= (((uint64_t) b & 0xff) << (m_idx++ * 8));
  if (m_idx >= 8)
  {
    DIGEST_BLOCK
    m_idx = 0;
    m_m = 0;
  }
}

void SipHash::update(uint8_t *data)
{
  while (data != NULL && *data != '\0')
  {
      update(*data++);
  }
}

void SipHash::update(uint16_t data)
{
  update((uint8_t)(data & 0xff));
  update((uint8_t)(data >> 8));
}

void SipHash::update(uint64_t data)
```

```
{
  for(int i = 0; i < 4; i++)
    update((uint16_t) ((data << (8*i)) & 0xffff));
}

uint64_t SipHash::digest()
{
  while (m_idx < 7)
  {
      m_m |= 0 << (m_idx++ * 8);
  }

  m_m |= ((uint64_t) m_input_len) << (m_idx * 8);

  DIGEST_BLOCK

  m_v2 ^= 0xff;

  for(int i = 0; i < 4; i++)
  {
      COMPRESS
  }

  return ((uint64_t) m_v0 ^ m_v1 ^ m_v2 ^ m_v3);
}
```

B.1.7 Databases

```
#ifndef DATABASE_H
#define DATABASE_H
#include "addrset.h"
#include "timer.h"

class Palma;

enum class DbStatus
{
  FREE,
  RESERVED,
  ASSIGNED,
  INVALID
};

class AssignableSet : public AddrSet, public Timer
{
public:
  AssignableSet *m_next_free;
  void *m_ptr;
  uint64_t m_security_id;
  bool m_reserved;

  AssignableSet(uint64_t addr=0, uint64_t count=1);
  AssignableSet(AddrSet *set);
  void chain(AssignableSet *set);
  bool unchain(void *db);
  void timeout();
};

class TreeNode
{
public:
  TreeNode *m_parent;
  TreeNode *m_child[3];
  AssignableSet *m_set[2];

  TreeNode();
  ~TreeNode();
  TreeNode *locate(uint64_t addr, int &index);
  AssignableSet *getSet(int index);
  void setChild(int index, TreeNode *&item);
  void add(AssignableSet *set, TreeNode *item);
  int childIndex();
  int redistribute_left(int index);
  int redistribute_right(int index);
  int redistribute_up(int index);
  TreeNode *merge(int index);
  TreeNode *redistribute();
```

```
  TreeNode *del(int index);
};

class SetDatabase
{
public:
  Palma *m_protocol;
  TreeNode *m_root;
  AssignableSet *m_free_list;
  AddrSet m_total_set;

  SetDatabase(Palma *protocol);
  ~SetDatabase();
  void init(AddrSet *set);
  AssignableSet *search(uint64_t addr);
  AssignableSet* splitAndInsert(AssignableSet *set, uint64_t size);
  void joinAndDelete(AssignableSet *set);
  int exclude(AddrSet *set, uint16_t lifetime);
  AssignableSet* getFreeSet(uint64_t min, uint64_t max, bool random =
   false);

  void extract(AssignableSet* &container_set, AddrSet *set);
  AssignableSet* findSet(uint64_t count);
  AssignableSet* reserve(uint64_t count, uint64_t security_id, uint16_t
   lifetime);
  AddrSet* assign(AssignableSet *container_set, AddrSet *set, uint64_t
   security_id, uint16_t lifetime);
  AddrSet* assign(uint64_t count, uint64_t security_id, uint16_t lifetime)
   ;
  void release(AssignableSet *set);
  DbStatus checkStatus(AddrSet *set, uint64_t &security_id, uint16_t &
   lifetime, bool &identical, AssignableSet *&result);
};

#endif
```

```
#include <stdio.h>
#include <inttypes.h>
#include <math.h>
#include "database.h"
#include "palma.h"

#define MIN(a,b) (a < b ? a : b)
#define MAX(a,b) (a > b ? a : b)

AssignableSet::AssignableSet(uint64_t addr, uint64_t count) :
                    AddrSet(addr, count),
                    m_next_free(NULL),
                    m_ptr(NULL) {}

AssignableSet::AssignableSet(AddrSet* set) : AddrSet(set->getFirstAddr(),
   set->getSize()),
```

```
                        m_next_free(NULL),
                        m_ptr(NULL){}

void AssignableSet::chain(AssignableSet *set)
{
  if(set == NULL)
  {
    set = this;
    m_next_free = this;
    m_ptr = this;
  }
  else
  {
    m_next_free = set;
    AssignableSet *prev = (AssignableSet*) set->m_ptr;
    prev->m_next_free = this;
    m_ptr = prev;
    set->m_ptr = this;
  }
}

bool AssignableSet::unchain(void *db)
{
  bool last_set = false;
  if(m_next_free != this)
  {
    ((AssignableSet*) m_ptr)->m_next_free = m_next_free;
    m_next_free->m_ptr = m_ptr;
  }
  else
    last_set = true;
  m_next_free = NULL;
  m_ptr = db;
  return last_set;
}

void AssignableSet::timeout()
{
  SetDatabase *db = (SetDatabase*) m_ptr;
  AssignableSet *prev_set = db->search(getFirstAddr() - 1);
  AssignableSet *next_set = db->search(getLastAddr() + 1);
  chain(db->m_free_list);
  if(db->m_free_list == NULL)
    db->m_free_list = this;
  if(next_set != NULL && next_set->m_next_free != NULL)
    db->joinAndDelete(this);
  if(prev_set != NULL && prev_set->m_next_free != NULL)
    db->joinAndDelete(prev_set);
}

TreeNode::TreeNode()
```

```
67 {
68   m_parent = NULL;
69   m_child[0] = NULL;
70   m_child[1] = NULL;
71   m_child[2] = NULL;
72   m_set[0] = NULL;
73   m_set[1] = NULL;
74 }
75
76
77 TreeNode::~TreeNode()
78 {
79   delete(m_child[0]);
80   delete(m_child[1]);
81   delete(m_child[2]);
82   delete(m_set[0]);
83   delete(m_set[1]);
84 }
85
86 TreeNode *TreeNode::locate(uint64_t addr, int &index)
87 {
88   if(addr >= m_set[0]->getFirstAddr() && addr <= m_set[0]->getLastAddr())
89   {
90     index = 0;
91     return this;
92   }
93   else if(m_set[1] != NULL && addr >= m_set[1]->getFirstAddr() && addr <=
      m_set[1]->getLastAddr())
94   {
95     index = 1;
96     return this;
97   }
98   else if(m_child[0] == NULL)
99   {
100    index = -1;
101    return this;
102  }
103  if(addr < m_set[0]->getFirstAddr())
104    return m_child[0]->locate(addr, index);
105  if(m_set[1] == NULL || addr > m_set[1]->getLastAddr())
106    return m_child[2]->locate(addr, index);
107  return m_child[1]->locate(addr, index);
108 }
109
110 AssignableSet *TreeNode::getSet(int index)
111 {
112   if(index >= 0)
113     return m_set[index];
114   return NULL;
115 }
116
```

```
117 void TreeNode::setChild(int index, TreeNode *&item)
118 {
119   m_child[index] = item;
120   if(item != NULL)
121     item->m_parent = this;
122   item = NULL;
123 }
124 
125 void TreeNode::add(AssignableSet *set, TreeNode *item)
126 {
127   AssignableSet *r = set;
128   if(m_set[1] == NULL)
129   {
130     if(set->getFirstAddr() < m_set[0]->getFirstAddr())
131     {
132       set = m_set[0];
133       m_set[0] = r;
134       setChild(1, item);
135     }
136     else
137     {
138       m_child[1] = m_child[2];
139       setChild(2, item);
140     }
141     m_set[1] = set;
142     return;
143   }
144   TreeNode *sibling = new TreeNode();
145   if(set->getFirstAddr() < m_set[0]->getFirstAddr())
146   {
147     set = m_set[0];
148     m_set[0] = r;
149     sibling->m_set[0] = m_set[1];
150     sibling->setChild(0, m_child[1]);
151     sibling->setChild(2, m_child[2]);
152     setChild(2, item);
153   }
154   else
155   {
156     if(set->getFirstAddr() > m_set[1]->getFirstAddr())
157     {
158       sibling->m_set[0] = set;
159       set = m_set[1];
160       sibling->setChild(0, m_child[2]);
161       sibling->setChild(2, item);
162     }
163     else
164     {
165       sibling->m_set[0] = m_set[1];
166       sibling->setChild(0, item);
167       sibling->setChild(2, m_child[2]);
```

```
168       }
169       m_child[2] = m_child[1];
170       if(item != NULL)
171         item->m_parent = sibling;
172     }
173     m_child[1] = NULL;
174     m_set[1] = NULL;
175     if(m_parent != NULL)
176     {
177       m_parent->add(set, sibling);
178     }
179     else
180     {
181       TreeNode *parent = new TreeNode();
182       parent->m_set[0] = set;
183       parent->m_child[0] = this;
184       parent->m_child[2] = sibling;
185       m_parent = sibling->m_parent = parent;
186     }
187 }
188 
189 int TreeNode::childIndex()
190 {
191   if(!m_parent)
192     return -1;
193   for(int i=0; i<=2; i++)
194   {
195     if(m_parent->m_child[i] == this)
196       return i;
197   }
198   return -1;
199 }
200 
201 int TreeNode::redistribute_left(int index)
202 {
203   if(index <= 0)
204     return 0;
205   int sib_idx = (m_parent->m_child[1] != NULL) ? index-1 : index-2;
206   TreeNode *sibling = m_parent->m_child[sib_idx];
207   int parent_ridx = (m_parent->m_child[1] != NULL) ? index-1 : 0;
208   if(sibling->m_set[1] != NULL)
209   {
210     m_set[0] = m_parent->m_set[parent_ridx];
211     m_parent->m_set[parent_ridx] = sibling->m_set[1];
212     sibling->m_set[1] = NULL;
213     m_child[2] = m_child[0];
214     setChild(0, sibling->m_child[2]);
215     sibling->setChild(2, sibling->m_child[1]);
216     return 1;
217   }
218   return 0;
```

```
}

int TreeNode::redistribute_right(int index)
{
  if(index >= 2)
    return 0;
  int sib_idx = (m_parent->m_child[1] != NULL) ? index+1 : index+2;
  TreeNode *sibling = m_parent->m_child[sib_idx];
  if(sibling->m_set[1] != NULL)
  {
    int parent_ridx = (m_parent->m_child[1] != NULL) ? index : 0;
    m_set[0] = m_parent->m_set[parent_ridx];
    m_parent->m_set[parent_ridx] = sibling->m_set[0];
    sibling->m_set[0] = sibling->m_set[1];
    sibling->m_set[1] = NULL;
    setChild(2, sibling->m_child[0]);
    sibling->setChild(0, sibling->m_child[1]);
    return 1;
  }
  return 0;
}

int TreeNode::redistribute_up(int index)
{
  if(m_parent->m_set[1] != NULL)
  {
    switch(index)
    {
      case 0:
        m_set[0] = m_parent->m_set[0];
        m_set[1] = m_parent->m_child[1]->m_set[0];
        m_parent->m_set[0] = m_parent->m_set[1];
        setChild(1, m_parent->m_child[1]->m_child[0]);
        setChild(2, m_parent->m_child[1]->m_child[2]);
        break;
      case 1:
        m_parent->m_child[0]->m_set[1] = m_parent->m_set[0];
        m_parent->m_set[0] = m_parent->m_set[1];
        m_parent->m_child[0]->setChild(1, m_parent->m_child[0]->m_child
    [2]);
        m_parent->m_child[0]->setChild(2, m_parent->m_child[1]->m_child
    [0]);
        break;
      case 2:
        m_set[0] = m_parent->m_child[1]->m_set[0];
        m_set[1] = m_parent->m_set[1];
        setChild(2, m_child[0]);
        setChild(1, m_parent->m_child[1]->m_child[2]);
        setChild(0, m_parent->m_child[1]->m_child[0]);
        break;
    }
```

```
268     m_parent->m_child[1]->m_set[0] = NULL;
269     m_parent->m_child[1]->m_child[0] = m_parent->m_child[1]->m_child[2] =
    NULL;
270     m_parent->m_set[1] = NULL;
271     TreeNode *item = m_parent->m_child[1];
272     m_parent->m_child[1] = NULL;
273     delete item;
274     return 1;
275   }
276   return 0;
277 }
278 
279 TreeNode *TreeNode::merge(int index)
280 {
281   TreeNode *parent = m_parent;
282   if(index == 0)
283   {
284     m_set[0] = parent->m_set[0];
285     m_set[1] = parent->m_child[2]->m_set[0];
286     parent->m_child[2]->m_set[0] = NULL;
287     setChild(1, parent->m_child[2]->m_child[0]);
288     setChild(2, parent->m_child[2]->m_child[2]);
289   }
290   else
291   {
292     m_parent->m_child[0]->m_set[1] = m_parent->m_set[0];
293     parent->m_child[0]->setChild(1, parent->m_child[0]->m_child[2]);
294     parent->m_child[0]->setChild(2, m_child[0]);
295     m_set[0] = NULL;
296   }
297   delete parent->m_child[2];
298   parent->m_set[0] = NULL;
299   parent->m_child[2] = NULL;
300   if(parent->m_parent != NULL)
301     return(parent->redistribute());
302   TreeNode *new_root = parent->m_child[0];
303   new_root->m_parent = NULL;
304   parent->m_child[0] = NULL;
305   delete parent;
306   return new_root;
307 }
308 
309 TreeNode *TreeNode::redistribute()
310 {
311   int child_index = childIndex();
312   if(redistribute_left(child_index) ||
313     redistribute_right(child_index) ||
314     redistribute_up(child_index))
315     return NULL;
316   return merge(child_index);
317 }
```

```
318
319 TreeNode *TreeNode::del(int index)
320 {
321   AssignableSet *set = m_set[index];
322   TreeNode *new_root = NULL;
323   if(m_child[0] != NULL) //search inorder successor
324   {
325     int idx;
326     TreeNode *successor = locate(m_set[index]->getLastAddr()+1, idx);
327     m_set[index] = successor->m_set[0];
328     successor->m_set[0] = set;
329     return(successor->del(0));
330   }
331   else if(m_set[1] != NULL)
332   {
333     m_set[index] = m_set[1-index];
334     m_set[1] = NULL;
335   }
336   else
337   {
338     new_root = redistribute();
339
340   }
341   delete set;
342   return new_root;
343 }
344
345
346 SetDatabase::SetDatabase(Palma *protocol) : m_protocol(protocol),
347                          m_total_set(){}
348
349 void SetDatabase::init(AddrSet *set)
350 {
351   m_total_set = *set;
352   m_root = new TreeNode();
353   m_root->m_set[0] = new AssignableSet(set);
354   m_root->m_set[0]->m_next_free = m_root->m_set[0];
355   m_root->m_set[0]->m_ptr = m_root->m_set[0];
356   m_free_list = m_root->m_set[0];
357 }
358
359 SetDatabase::~SetDatabase()
360 {
361   delete(m_root);
362 }
363
364 AssignableSet *SetDatabase::search(uint64_t addr)
365 {
366   int index;
367   return m_root->locate(addr, index)->getSet(index);
368 }
```

```
369
370 AssignableSet* SetDatabase::splitAndInsert(AssignableSet* set, uint64_t
    size)
371 {
372   if(set->getSize() <= size)
373     return NULL;
374   AssignableSet *new_set =
375       new AssignableSet(set->getFirstAddr() + set->getSize() - size, size)
    ;
376   set->setSize(set->getSize() - size);
377   new_set->chain(set);
378   int idx;
379   m_root->locate(new_set->getFirstAddr(), idx)->add(new_set, NULL);
380   if(m_root->m_parent)
381     m_root = m_root->m_parent;
382   return new_set;
383 }
384
385 void SetDatabase::joinAndDelete(AssignableSet *set)
386 {
387   int index;
388   TreeNode *node = m_root->locate(set->getLastAddr() + 1, index);
389   AssignableSet *next = node->getSet(index);
390   if(next->m_next_free == NULL)
391     m_protocol->m_event_loop.stopTimer(next);
392   else
393   {
394     next->unchain(NULL);
395     if(m_free_list == next)
396       m_free_list = set;
397   }
398
399   set->setSize(set->getSize() + next->getSize());
400   TreeNode *new_root = node->del(index);
401   if(new_root != NULL)
402     m_root = new_root;
403 }
404
405 int SetDatabase::exclude(AddrSet *recv_set, uint16_t lifetime)
406 {
407   AddrSet set;
408   if(set.checkConflict(&m_total_set, recv_set))
409   {
410     AssignableSet *r = search(set.getFirstAddr());
411     if(r == NULL)
412       return -1;
413     if(r->m_next_free == NULL)
414     {
415       if(r->getFirstAddr() == set.getFirstAddr()
416         && r->getSize() == set.getSize())
417       {
```

```
418         if(fabs(m_protocol->m_event_loop.readTimer(r) - lifetime) > 1.)
419         {
420           m_protocol->m_event_loop.stopTimer(r);
421           m_protocol->m_event_loop.startTimer(r, lifetime);
422         }
423         return 0;
424       }
425       else
426       {
427         m_protocol->m_event_loop.stopTimer(r);
428         r->chain(m_free_list);
429         if(m_free_list == NULL)
430           m_free_list = r;
431       }
432     }
433     while(r->getLastAddr() < set.getLastAddr())
434     {
435       joinAndDelete(r);
436     }
437     extract(r, &set);
438     m_protocol->m_event_loop.startTimer(r, lifetime);
439     return 0;
440   }
441   return -1;
442 }
443
444 AssignableSet* SetDatabase::getFreeSet(uint64_t min, uint64_t max, bool
      random)
445 {
446   AssignableSet *p = m_free_list;
447   AssignableSet *best;
448   uint64_t best_size = 0, size;
449   if(p == NULL)
450     return NULL;
451   do
452   {
453     size = p->getSize() > 0xffff ? MAX(p->getAlignedSize(), 0xffff) : p->
     getSize();
454     size = MIN(size, max);
455     if(best_size < size
456         || (random && best_size == size && (lrand48() % 1)))
457     {
458       best = p;
459       best_size = size;
460     }
461     p = p->m_next_free;
462     if(p == m_free_list)
463       break;
464   } while(best_size < max || random);
465   if(best_size >= min)
466     return best;
```

```
467   else
468     return NULL;
469 }
470 /*
471 AssignableSet* SetDatabase::getFreeSet(uint64_t min, uint64_t max, bool
      random)
472 {
473   AssignableSet *p = m_free_list;
474   AssignableSet *best;
475   uint64_t best_size = 0, size;
476   if(p == NULL)
477     return NULL;
478   do
479   {
480     size = p->getSize() > 0xffff ? MAX(p->getAlignedSize(), 0xffff) : p->
      getSize();
481     if(best_size < size)
482     {
483       best = p;
484       best_size = size;
485     }
486     p = p->m_next_free;
487     if(p == m_free_list)
488       break;
489   } while(best_size < max);
490   if(best_size >= min)
491     return best;
492   else
493     return NULL;
494 } */
495
496 void SetDatabase::extract(AssignableSet* &container_set, AddrSet *set)
497 {
498   uint64_t size = container_set->getLastAddr() - set->getFirstAddr() + 1;
499   if(set->getFirstAddr() > container_set->getFirstAddr())
500   {
501     container_set = splitAndInsert(container_set, size);
502   }
503   size -= set->getSize();
504   if(size)
505   {
506     splitAndInsert(container_set, size);
507   }
508   if(m_free_list == container_set)
509     m_free_list = m_free_list->m_next_free;
510   if(container_set->unchain(this))
511     m_free_list = NULL;
512 }
513
514 AssignableSet* SetDatabase::findSet(uint64_t count)
515 {
```

```
  AssignableSet *free_set = getFreeSet(1, count);
  if(free_set == NULL)
    return NULL;
  AddrSet set = *free_set;
  if(MIN(free_set->getSize(),count) > 0xffff)
    set.alignToMask(SetType::ADDR);
  if(set.getSize() > count)
    set.setSize(count);
  extract(free_set, &set);
  return free_set;
}

AssignableSet* SetDatabase::reserve(uint64_t count, uint64_t security_id,
    uint16_t lifetime)
{
  AssignableSet *free_set = findSet(count);
  if(free_set == NULL)
    return NULL;
  free_set->m_security_id = security_id;
  free_set->m_reserved = true;
  m_protocol->m_event_loop.startTimer(free_set, lifetime);
  return free_set;
}

AddrSet* SetDatabase::assign(AssignableSet *container_set, AddrSet *set,
    uint64_t security_id, uint16_t lifetime)
{
  if(container_set->m_next_free == NULL)
  {
    m_protocol->m_event_loop.stopTimer(container_set);
    container_set->chain(m_free_list);
    if(m_free_list == NULL)
      m_free_list = container_set;
  }
  extract(container_set, set);
  container_set->m_security_id = security_id;
  container_set->m_reserved = false;
  m_protocol->m_event_loop.startTimer(container_set, lifetime + 1);
  return container_set;
}

AddrSet* SetDatabase::assign(uint64_t count, uint64_t security_id,
    uint16_t lifetime)
{
  AssignableSet *free_set = findSet(count);
  free_set->m_security_id = security_id;
  free_set->m_reserved = false;
  m_protocol->m_event_loop.startTimer(free_set, lifetime + 1);
  return free_set;
}

```

```
564 void SetDatabase::release(AssignableSet *set)
565 {
566   m_protocol->m_event_loop.stopTimer(set);
567   set->timeout();
568 }
569
570 DbStatus SetDatabase::checkStatus(AddrSet *set, uint64_t &security_id,
571                   uint16_t &lifetime, bool &identical, AssignableSet *&
    result)
572 {
573   result = search(set->getFirstAddr());
574   if(result != NULL && result->getLastAddr() >= set->getLastAddr())
575   {
576     identical = (result->getSize() == set->getSize());
577     if(result->m_next_free != NULL)
578       return DbStatus::FREE;
579     security_id = result->m_security_id;
580     lifetime = m_protocol->m_event_loop.readTimer(result);
581     if(result->m_reserved)
582       return DbStatus::RESERVED;
583     if(lifetime > 0)
584       lifetime--;
585     return DbStatus::ASSIGNED;
586   }
587   else
588     return DbStatus::INVALID;
589 }
```

B.1.8 Configuration

```
#ifndef CONFIG_H
#define CONFIG_H

#include <stdio.h>
#include <stdlib.h>
#include "addrset.h"

#define MAX_NUM_ATTRIBUTES  (16)
#define MAX_ID_LENGTH (256)
#define MAX_MSG_LENGTH (256)
#define TO_ADDR(ptr) (*(uint64_t*)ptr)
#define TO_SIZE(ptr) (*(uint64_t*)ptr)
#define TO_UINT(ptr) (*(uint16_t*)ptr)
#define TO_ADDRSET_PTR(ptr) ((AddrSet*)ptr)
#define TO_BOOL(ptr) (*(bool*)ptr)
#define TO_STRING(ptr) ((uint8_t*)ptr)

class AttributeList
{
public:
  char m_name[MAX_NUM_ATTRIBUTES][MAX_ID_LENGTH];
  char m_value[MAX_NUM_ATTRIBUTES][MAX_ID_LENGTH];
  int m_length;

  bool getValue(int index, const char *name, uint64_t &val);
  bool getString(int index, const char *name, char *&val);
};


class ConfigElement
{
public:

  virtual bool  init(AttributeList *attr){}
  virtual void* get() {return NULL;}
  virtual void set(void *ptr) {}
  virtual ~ConfigElement(){}
};

class ConfigAddr : public ConfigElement
{
  uint64_t m_addr;
public:

  ConfigAddr(uint64_t addr);
  bool init(AttributeList *attr);
  void* get() {return &m_addr;}
  void set(void *addr) { m_addr = *(uint64_t *)addr; }
};
```

```
50
51 class ConfigSize : public ConfigElement
52 {
53   uint64_t m_size;
54 public:
55
56   ConfigSize(uint64_t size);
57   bool init(AttributeList *attr);
58   void* get() {return &m_size;}
59   void set(void *size) { m_size = *(uint64_t *)size; }
60 };
61
62 class ConfigInt : public ConfigElement
63 {
64   uint16_t m_val;
65 public:
66
67   ConfigInt(uint16_t val);
68   bool init(AttributeList *attr);
69   void* get(){return &m_val;}
70   void set(void *val) { m_val = *(uint16_t *)val; }
71 };
72
73 class ConfigSet : public ConfigElement
74 {
75   AddrSet m_set;
76
77 public:
78
79   ConfigSet(uint64_t addr, uint64_t count);
80   bool init(AttributeList *attr);
81   void* get(){return &m_set;}
82   void set(void *set) { m_set = *(AddrSet *)set; }
83 };
84
85 class ConfigBool : public ConfigElement
86 {
87   bool m_logic_val;
88
89 public:
90
91   ConfigBool(bool logic_val);
92   bool init(AttributeList *attr);
93   void* get(){return &m_logic_val;}
94   void set(void *logic_val) { m_logic_val = *(bool *)logic_val; }
95 };
96
97 class ConfigString : public ConfigElement
98 {
99   char *m_str;
100
```

```
public:

  ConfigString(char *str);
  ~ConfigString();
  bool init(AttributeList *attr);
  void* get(){return m_str;}
  void set(void *str);
};


class Config
{
public:
  ConfigElement **m_list;
  const char **m_array_tags;
  const char *m_root_tag;

  int m_num_line;

  virtual bool check(const char *fname) = 0;
  virtual int getMaxItem() = 0;

  static bool toUint64(char *str, uint64_t &val);
  void skipBlank(FILE *fp);
  void skipUntil(FILE *fp, const char *pat);
  void error(const char *fname, const char *msg);
  void set(int item, void* ptr);
  void* get(int item);
  bool read(const char *fname);
};

#endif
```

```
#include <string.h>
#include <ctype.h>
#include "config.h"


bool AttributeList::getValue(int index, const char *name, uint64_t &val)
{
  if (!strcmp(m_name[index], name))
  {
    return Config::toUint64(m_value[index], val);
  }
  return false;
}

bool AttributeList::getString(int index, const char *name, char *&val)
{
  if (!strcmp(m_name[index], name))
  {
    val = m_value[index];
```

```
    return true;
  }
  return false;
}

ConfigAddr::ConfigAddr(uint64_t addr) : m_addr(addr){}

bool ConfigAddr::init(AttributeList *attr)
{
  return(attr->m_length == 1 && attr->getValue(0, "addr", m_addr));
}

ConfigSize::ConfigSize(uint64_t size) : m_size(size) {}

bool ConfigSize::init(AttributeList *attr)
{
  return (attr->m_length == 1 && attr->getValue(0, "size", m_size));
}

ConfigInt::ConfigInt(uint16_t val) : m_val(val){}

bool ConfigInt::init(AttributeList *attr)
{
  uint64_t val;
  if(attr->m_length != 1 || !attr->getValue(0, "value", val))
    return false;
  m_val = val;
  return true;
}

ConfigSet::ConfigSet(uint64_t addr, uint64_t count) : m_set(addr, count){}

bool ConfigSet::init(AttributeList *attr)
{
  bool addr_ok = false;
  bool val_ok = false;
  uint64_t addr,val;
  SetType type;

  for(int i=0; i < attr->m_length; i++)
  {
    if (!addr_ok && attr->getValue(i, "addr", addr))
      addr_ok = true;
    else if (!val_ok && attr->getValue(i, "count", val))
    {
      type = SetType::ADDR;
      val_ok = true;
    }
    else if (!val_ok && attr->getValue(i, "mask", val))
    {
      type = SetType::MASK;
```

```
      val_ok = true;
    }
    else
      return false;
  }
  m_set = AddrSet(addr, val, SetSize::AUTO, type);
  return true;
}

ConfigBool::ConfigBool(bool logic_val) : m_logic_val(logic_val){}

bool ConfigBool::init(AttributeList *attr)
{
  char *p;
  char *val;
  if(attr->m_length != 1 || !attr->getString(0, "value", val))
    return false;
  else
  {
    for(char *p = val; *p != 0; p++)
      *p = toupper(*p);
    if(!strcmp(val, "TRUE"))
      m_logic_val = 1;
    else if(!strcmp(val, "FALSE"))
      m_logic_val = 0;
    else
      return false;
  }
  return true;
}

ConfigString::ConfigString(char *str)
{
  m_str = NULL;
  if(str)
    m_str = strdup(str);
}

bool ConfigString::init(AttributeList *attr)
{
  char* str;
  if(attr->m_length == 1 && attr->getString(0, "id", str))
  {
    delete(m_str);
    m_str = strdup(str);
  }
  else
    return false;
  return true;
}

```

```
122 void ConfigString::set(void *str)
123 {
124   delete(m_str);
125   m_str = NULL;
126   if(str)
127     m_str = strdup((char *)str);
128 }
129
130 ConfigString::~ConfigString()
131 {
132   delete m_str;
133 }
134
135 bool Config::toUint64(char *str, uint64_t &val)
136 {
137   char *p;
138   val = strtoull(str, &p, 0);
139   if (p != &str[strlen(str)])
140     return false;
141   return true;
142 }
143
144 void Config::skipBlank(FILE *fp)
145 {
146   const char *spaces = "\n\r \t";
147   while(!feof(fp))
148   {
149     int c = fgetc(fp);
150     const char *p = strchr(spaces,c);
151     if(p == NULL)
152     {
153       ungetc(c,fp);
154       break;
155     }
156     else if(p == spaces)
157       m_num_line++;
158   }
159 }
160
161 void Config::skipUntil(FILE *fp, const char *pat)
162 {
163   int cnt=0;
164   do
165   {
166     skipBlank(fp);
167     fscanf(fp, pat,&cnt);
168     if(!cnt)
169       fgetc(fp);
170   }
171   while(!cnt && !feof(fp));
172 }
```

```
173
174 void Config::error(const char *fname, const char *msg)
175 {
176   fprintf(stderr,"%s:%d: %s\n",fname, m_num_line, msg);
177 }
178
179 void Config::set(int item, void* ptr)
180 {
181   m_list[item]->set(ptr);
182 }
183
184 void* Config::get(int item)
185 {
186   return m_list[item]->get();
187 }
188
189 bool Config::read(const char *fname)
190 {
191   FILE *fp = fopen(fname, "r");
192   if(fp == NULL)
193   {
194     perror("Opening configuration file");
195     return false;
196   }
197   char tag[MAX_ID_LENGTH];
198   AttributeList attr;
199
200   bool opened_tag = false;
201   int root_state = -1;
202   const char *last_tag = NULL;
203
204   char error_msg[MAX_MSG_LENGTH];
205   m_num_line = 1;
206
207   while(!feof(fp))
208   {
209     skipBlank(fp);
210     if(!fscanf(fp, "<%255[^> \t\n\r]", tag))
211     {
212       if(root_state != 0)
213         error(fname, "Invalid char: Expected '<'");
214       else
215         error(fname, "Extra chars: Expected EOF");
216       return false;
217     }
218     if(tag[0] == '?')
219       skipUntil(fp, "?>%n");
220     else if(tag[0] == '!')
221       skipUntil(fp, "-->%n");
222     else if(tag[0] == '/')
223     {
```

```
224       if(opened_tag)
225       {
226         if(!strcmp(&tag[1],last_tag))
227           opened_tag = false;
228         else
229         {
230           sprintf(error_msg, "Expected element '%s' close.", last_tag);
231           error(fname, error_msg);
232           return false;
233         }
234       }
235       else if(root_state == 1 && !strcmp(&tag[1],m_root_tag))
236         root_state = 0;
237       else
238       {
239         sprintf(error_msg, "Unexpected element '%s' close.", &tag[1]);
240         error(fname, error_msg);
241         return false;
242       }
243       skipUntil(fp, ">%n");
244     }
245     else if(root_state == -1)
246     {
247       if(!strcmp(tag,m_root_tag))
248       {
249         root_state = 1;
250         skipUntil(fp, ">%n");
251       }
252       else
253       {
254         sprintf(error_msg, "Unexpected root element '%s'", tag);
255         error(fname, error_msg);
256         return false;
257       }
258     }
259     else if(root_state == 1 && !opened_tag)
260     {
261       if(tag[strlen(tag)-1] == '/')
262         tag[strlen(tag)-1] = 0;
263       else
264       {
265         attr.m_length = 0;
266         for(;;)
267         {
268           skipBlank(fp);
269           if(!fscanf(fp, "%255[^>= \t\n\r]", attr.m_name[attr.m_length]))
270           {
271             opened_tag = true;
272             skipUntil(fp, ">%n");
273             break;
274           }
```

```
275           skipBlank(fp);
276           if(attr.m_name[attr.m_length][0] == '/')
277           {
278             opened_tag = false;
279             skipUntil(fp, ">%n");
280             break;
281           }
282           else
283           {
284             skipBlank(fp);
285             if(fgetc(fp) != '=')
286             {
287               error(fname, "Invalid char: Expected '='.");
288               return false;
289             }
290             skipBlank(fp);
291             if(!fscanf(fp, "\"%255[^>\"]\"", attr.m_value[attr.m_length
    ++]))
292             {
293               error(fname,
294                 "Invalid attribute value format:"
295                 " Expected quotation marks enclosed string.");
296               return false;
297             }
298           }
299         }
300       }
301       int i;
302       for (i=0; i<getMaxItem() && strcmp(tag, m_array_tags[i]); i++);
303       if (i < getMaxItem())
304       {
305         if(!m_list[i]->init(&attr))
306         {
307           sprintf(error_msg, "Invalid attributes for tag: '%s'.",tag);
308           error(fname, error_msg);
309           return false;
310         }
311         last_tag = m_array_tags[i];
312       }
313       else
314       {
315         sprintf(error_msg, "Unknown tag: '%s'.",tag);
316         error(fname, error_msg);
317         return false;
318       }
319     }
320     else
321     {
322       error(fname, "Syntax error.");
323       return false;
324     }
```

```
    skipBlank(fp);
  }
  fclose(fp);
  if(root_state == 1)
  {
    error(fname, "Unexpected EOF.");
    return false;
  }
  else if(root_state == -1)
  {
    error(fname, "Empty configuration.");
    return false;
  }
  return check(fname);
}
```

B.1.9 Details and Constants

```
#ifndef DETAILS_H
#define DETAILS_H

#include <stdlib.h>
#include "addrset.h"

#define DISCOVERY_TIMEOUT (.4 + drand48()*.2)
#define DISCOVER_DSC_COUNT 3
#define REQUESTING_TIMEOUT (.4 + drand48()*.2)
#define REQUEST_REQ_COUNT 3
#define ANNOUNCE_TIMEOUT  (28. + drand48()*4.)
#define SELF_ASSIGMENT_LIFETIME 320 //(12.*60.*60.)

extern const uint64_t PALMA_MCAST;
extern const uint16_t PALMA_TYPE;
extern const uint8_t PALMA_SUBTYPE;
extern const AddrSet DISCOVER_SOURCE_ADDR_RANGE;
extern const AddrSet AUTOASSIGN_UNICAST;
extern const AddrSet AUTOASSIGN_MULTICAST;
extern const AddrSet AUTOASSIGN_UNICAST_64;
extern const AddrSet AUTOASSIGN_MULTICAST_64;
extern const int MAX_AUTOASSIGN_UNICAST;
#endif
```

```
#include "details.h"

const uint64_t PALMA_MCAST = 0x0180c2abcdef;
const uint16_t PALMA_TYPE = 0x33ff;
const uint8_t PALMA_SUBTYPE = 0x00;

const AddrSet DISCOVER_SOURCE_ADDR_RANGE(0x2a0000000000L,0xffffffffL + 1);
const AddrSet AUTOASSIGN_UNICAST(0x0a0000000000L,0xffffffffffL + 1);
const AddrSet AUTOASSIGN_MULTICAST(0x0b0000000000L,0xffffffffffL + 1);
const AddrSet AUTOASSIGN_UNICAST_64(0x0a00000000000000L,0xffffffffffffffL
   + 1);
const AddrSet AUTOASSIGN_MULTICAST_64(0x0b00000000000000L,0
   xffffffffffffffL + 1);
const int MAX_AUTOASSIGN_UNICAST = 16;
```

B.2 Client Modules

B.2.1 Palma Client

```
#ifndef PALMA_CLIENT_H
#define PALMA_CLIENT_H

#include "states.h"
#include "config-client.h"
#include "../common/database.h"
#include "../common/palma.h"

class PalmaClient : public Palma
{
public:
  ConfigClient m_config;
  SetDatabase m_db;
  State *m_curstate;
  uint16_t m_token;
  bool m_mcast_on;
  bool m_preassigned_addr;
  uint64_t m_src_addr;
  uint64_t m_server_addr;
  AddrSet m_assigned_set;

  DiscoveryState m_discovery_state;
  RequestingState m_requesting_state;
  BoundState m_bound_state;
  DefendingState m_defending_state;

  PalmaClient();
  void begin();
  void handlePacket(Packet *pkt);
  void restart();
  bool checkStationId(uint8_t *station_id);
  void updateToken();
  uint16_t getToken();
  void onExit();
  void printv(const char *format, ...);
};

#endif
```

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdarg.h>

#include "palma-client.h"

PalmaClient::PalmaClient() :
```

```
                m_db(this),
                m_curstate(NULL),
                m_token((uint16_t)lrand48()),
                m_mcast_on(false),
                m_src_addr(0),
                m_server_addr(0),
                m_preassigned_addr(false),
                m_discovery_state(this),
                m_requesting_state(this),
                m_bound_state(this),
                m_defending_state(this) {}

void PalmaClient::begin()
{
  printv("BEGIN\n");
  m_netitf.init(TO_STRING(m_config.get(ConfigItem::INTERFACE)));
  m_event_loop.regSource(&m_netitf);
  m_event_loop.regHandler(this);
  m_src_addr = TO_ADDR(m_config.get(ConfigItem::PREASSIGNED_ADDR));
  if(m_src_addr)
  {
    m_netitf.addAddr(m_src_addr);
    m_preassigned_addr = true;
  }
  m_server_addr = TO_ADDR(m_config.get(ConfigItem::KNOWN_SERVER_ADDR));
  AddrSet *claim_set = TO_ADDRSET_PTR(m_config.get(ConfigItem::CLAIM_SET))
   ;
  m_db.init(claim_set);
  if(m_server_addr && m_preassigned_addr)
    m_requesting_state.start(m_server_addr, m_src_addr, claim_set);
  else
    m_discovery_state.start();
  m_event_loop.run();
}

void PalmaClient::handlePacket(Packet *pkt)
{
  m_curstate->handlePacket(pkt);
}

void PalmaClient::restart()
{
  printv("RESTARTING\n");
  m_curstate->clean();
  if(!m_preassigned_addr)
  {
    if(m_src_addr != 0)
    {
      m_netitf.delAddr(m_src_addr);
      m_src_addr = 0;
    }
```

```
59    }
60    m_server_addr = 0;
61    m_assigned_set = AddrSet();
62    updateToken();
63    m_server_addr = TO_ADDR(m_config.get(ConfigItem::KNOWN_SERVER_ADDR));
64    AddrSet *claim_set = TO_ADDRSET_PTR(m_config.get(ConfigItem::CLAIM_SET))
       ;
65    if(m_server_addr && m_preassigned_addr)
66      m_requesting_state.start(m_server_addr, m_src_addr, claim_set);
67    else
68      m_discovery_state.start();
69  }
70
71  bool PalmaClient::checkStationId(uint8_t *station_id)
72  {
73    uint8_t *id = TO_STRING(m_config.get(ConfigItem::STATION_ID));
74    if(station_id == NULL ||
75        (id != NULL &&
76        strcmp((const char *)station_id, (const char *)id) == 0))
77      return true;
78    return false;
79  }
80
81  void PalmaClient::updateToken()
82  {
83     m_token++;
84  }
85
86  uint16_t PalmaClient::getToken()
87  {
88    return m_token;
89  }
90
91  void PalmaClient::onExit()
92  {
93    if(m_curstate == &m_bound_state)
94      m_bound_state.sendRelease(true);
95    m_curstate->clean();
96    printv("ENDING\n");
97  }
98
99  void PalmaClient::printv(const char *format, ...)
100 {
101   if(TO_BOOL(m_config.get(ConfigItem::VERBOSE)))
102   {
103     Time now;
104     va_list args;
105       va_start(args, format);
106     printf("%.6f,", now.get());
107     vprintf(format, args);
108     va_end(args);
```

```
109     fflush(stdout);
110   }
111 }
```

B.2.2 States

```
#ifndef STATES_H
#define STATES_H

#include <string.h>
#include "../common/packet.h"
#include "../common/timer.h"

class PalmaClient;

class State
{
public:
  PalmaClient *m_protocol;

  State(PalmaClient *protocol);

  virtual void handlePacket(Packet *pkt) {}
  virtual void clean() {}
  bool validSet(Packet *pkt);
  bool validOffer(Packet *pkt);
};

class DiscoveryState : public State
{
public:
  class DiscoveryTimer : public Timer
  {
  public:
    DiscoveryState *m_state;
    DiscoveryTimer(DiscoveryState *state);
    void timeout();
  };
  DiscoveryTimer m_discovery_timer;
  int m_dsc_count;
  uint64_t m_src_addr;
  Packet *m_offer;
  AddrSet m_discovery_set;
  bool m_change_discovery;

  DiscoveryState(PalmaClient *protocol);
  void start();
  void clean();
  void sendDiscover();
  void handlePacket(Packet *pkt);
  void checkSet(AddrSet *set, uint16_t lifetime);
  uint64_t getSrcAddr();
  void storeOffer(Packet *pkt);
};

```

```
50 class RequestingState : public State
51 {
52 public:
53   class RequestTimer : public Timer
54   {
55   public:
56     RequestingState *m_state;
57     RequestTimer(RequestingState *state);
58     void timeout();
59   };
60   RequestTimer m_request_timer;
61   int m_req_count;
62   uint64_t m_server_addr;
63   uint64_t m_src_addr;
64   Packet *m_pkt;
65 
66   RequestingState(PalmaClient *protocol);
67   void start(uint64_t server_addr, uint64_t src_addr, AddrSet *set, bool
      renewal = false);
68   void clean();
69   void sendRequest();
70   void handlePacket(Packet *pkt);
71   bool checkNetworkId(uint8_t *rcv, uint8_t *snd);
72 };
73 
74 class DefendingState : public State
75 {
76 public:
77   class LeaseLifetimeTimer : public Timer
78   {
79   public:
80     DefendingState *m_state;
81     LeaseLifetimeTimer(DefendingState *state);
82     void timeout();
83   };
84   class AnnouncementTimer : public Timer
85   {
86   public:
87     DefendingState *m_state;
88     AnnouncementTimer(DefendingState *state);
89     void timeout();
90   };
91   LeaseLifetimeTimer m_lease_lifetime_timer;
92   AnnouncementTimer m_announcement_timer;
93 
94   DefendingState(PalmaClient *protocol);
95   void start(AddrSet *set);
96   void clean();
97   void sendAnnounce();
98   void sendDefend(AddrSet *original_set, AddrSet *conflict_set, uint64_t
      DA, uint8_t *station_id = NULL);
```

```
  void handlePacket(Packet *pkt);
  void processConflict(AddrSet* original_set, AddrSet *conflict_set,
   uint64_t DA, uint8_t *station_id = NULL);
};

class BoundState : public State
{
public:
  class LeaseLifetimeTimer : public Timer
  {
  public:
    BoundState *m_state;
    LeaseLifetimeTimer(BoundState *state);
    void timeout();
  };
  LeaseLifetimeTimer m_lease_lifetime_timer;

  BoundState(PalmaClient *protocol);
  void start(uint16_t lifetime, bool acceptable);
  void clean();
  void sendRelease(bool terminate = false);
};

#endif
```

```
#include <cmath>
#include "states.h"
#include "../common/details.h"
#include "palma-client.h"

#define MIN(a,b) ((a < b) ? a : b)

State::State(PalmaClient *protocol) : m_protocol(protocol) {}

static void adjustSet(AddrSet *set, uint64_t max)
{
  uint64_t size = MIN(set->getSize(), max);
  set->setSize(size);
  if(size > 0xffff)
    set->alignToMask(SetType::MASK);
}

DiscoveryState::DiscoveryState(PalmaClient *protocol) : State(protocol),
                            m_discovery_timer(this) {}

DiscoveryState::DiscoveryTimer::DiscoveryTimer(DiscoveryState *state) :
   m_state(state) {}

void DiscoveryState::start()
{
  m_protocol->printv("STARTING\n");
  m_protocol->m_curstate = this;
```

```
27    m_dsc_count = DISCOVER_DSC_COUNT;
28    m_src_addr = 0;
29    uint64_t min = TO_SIZE(m_protocol->m_config.get(ConfigItem::
       MIN_ADDR_CLAIM));
30    uint64_t max = TO_SIZE(m_protocol->m_config.get(ConfigItem::
       MAX_ADDR_CLAIM));
31    bool random = TO_BOOL(m_protocol->m_config.get(ConfigItem::RANDOM_ASSIGN
       ));
32    m_offer = NULL;
33    AddrSet *set = m_protocol->m_db.getFreeSet(min, max, random);
34    if(set != NULL)
35      m_discovery_set = *set;
36    else
37      m_discovery_set.setSize(0);
38    if(m_discovery_set.getSize() > 0xffff)
39      m_discovery_set.alignToMask(SetType::MASK);
40    m_change_discovery = false;
41    if(!m_protocol->m_mcast_on)
42    {
43      m_protocol->m_netitf.addAddr(PALMA_MCAST,true);
44      m_protocol->m_mcast_on = true;
45    }
46    sendDiscover();
47  }
48
49  void DiscoveryState::clean()
50  {
51    if(m_src_addr && !m_protocol->m_preassigned_addr)
52    {
53      m_protocol->m_netitf.delAddr(m_src_addr);
54      m_src_addr = 0;
55    }
56    delete m_offer;
57    m_offer = NULL;
58    m_discovery_set = AddrSet();
59    m_change_discovery = false;
60    m_protocol->m_event_loop.stopTimer(&m_discovery_timer);
61  }
62
63  void DiscoveryState::sendDiscover()
64  {
65    Packet pkt(MsgType::DISCOVER, PALMA_MCAST, getSrcAddr(), m_protocol->
       getToken());
66    if(m_discovery_set.getSize() > 0)
67      pkt.addMacSetPar(&m_discovery_set);
68    if(TO_STRING(m_protocol->m_config.get(ConfigItem::STATION_ID)) != NULL)
69      pkt.addIdPar(ParType::STATION_ID, TO_STRING(m_protocol->m_config.get(
       ConfigItem::STATION_ID)));
70    uint8_t *vendor_specific = TO_STRING(m_protocol->m_config.get(ConfigItem
       ::VENDOR));
71    if(vendor_specific != NULL)
```

```
72      pkt.addVendorPar(vendor_specific, strlen((const char *)vendor_specific
        ));
73    m_protocol->m_netitf.netsend(&pkt);
74    m_protocol->m_event_loop.startTimer(&m_discovery_timer,
        DISCOVERY_TIMEOUT);
75  }
76
77  void DiscoveryState::handlePacket(Packet *pkt)
78  {
79    if((pkt->getDA() == PALMA_MCAST && pkt->getType() != MsgType::ANNOUNCE))
80      return;
81    else if((pkt->getDA() != m_src_addr
82          || !m_protocol->checkStationId(pkt->getStationId()))
83          && pkt->getType() != MsgType::ANNOUNCE)
84      return;
85    switch(pkt->getType())
86    {
87      case MsgType::OFFER:
88        if(pkt->getToken() == m_protocol->getToken())
89          storeOffer(pkt);
90        break;
91      case MsgType::ANNOUNCE:
92        checkSet(pkt->getSet(), pkt->getLifetime());
93        break;
94      case MsgType::DEFEND:
95        checkSet(pkt->getSet(false), pkt->getLifetime());
96        break;
97    }
98  }
99
100 void DiscoveryState::checkSet(AddrSet *set, uint16_t lifetime)
101 {
102   AddrSet conflict_set;
103   m_protocol->m_db.exclude(set, lifetime);
104   if(conflict_set.checkConflict(set, &m_discovery_set))
105     m_change_discovery = true;
106 }
107
108 uint64_t DiscoveryState::getSrcAddr()
109 {
110   if(m_protocol->m_preassigned_addr)
111   {
112     m_src_addr = m_protocol->m_src_addr;
113   }
114   else
115   {
116     RandomAddrSet src_addr(DISCOVER_SOURCE_ADDR_RANGE);
117     if (m_src_addr != 0)
118       m_protocol->m_netitf.delAddr(m_src_addr);
119     m_src_addr = src_addr.getFirstAddr();
120     m_protocol->m_netitf.addAddr(m_src_addr);
```

```
121   }
122   return m_src_addr;
123 }
124 
125 bool State::validSet(Packet *pkt)
126 {
127   return (pkt->getSet()->getSize() >= TO_SIZE(m_protocol->m_config.get(
      ConfigItem::MIN_ADDR_CLAIM))
128       && (TO_ADDRSET_PTR(m_protocol->m_config.get(ConfigItem::CLAIM_SET))
      ->isMulticast() == pkt->getSet()->isMulticast())
129       && (TO_ADDRSET_PTR(m_protocol->m_config.get(ConfigItem::CLAIM_SET))
      ->isSize48() == pkt->getSet()->isSize48()));
130 }
131 
132 bool State::validOffer(Packet *pkt)
133 {
134   return (validSet(pkt) && (!TO_ADDRSET_PTR(m_protocol->m_config.get(
      ConfigItem::CLAIM_SET))->isMulticast()
135                 || m_protocol->m_preassigned_addr || pkt->getClientAddr())
      );
136 }
137 
138 void DiscoveryState::storeOffer(Packet *pkt)
139 {
140   uint pkt_set_size = MIN(pkt->getSet()->getSize(), TO_SIZE(m_protocol->
      m_config.get(ConfigItem::MAX_ADDR_CLAIM)));
141 
142   if(validOffer(pkt) && (m_offer == NULL
143                 ||(m_offer->getSet()->getSize() < pkt_set_size)
144                 ||(m_offer->getSet()->getSize() == pkt_set_size
145                   && m_offer->getLifetime() < pkt->getLifetime())))
146   {
147     delete m_offer;
148     m_offer = new Packet(pkt);
149   }
150   return;
151 }
152 
153 void DiscoveryState::DiscoveryTimer::timeout()
154 {
155   AddrSet aux_set;
156   if(m_state->m_offer != NULL)
157   {
158     uint64_t server_addr = m_state->m_offer->getSA();
159     uint64_t src_addr = m_state->m_src_addr;
160     AddrSet offered_set(*m_state->m_offer->getSet());
161     if(!m_state->m_protocol->m_preassigned_addr)
162     {
163       src_addr = m_state->m_offer->getClientAddr();
164       if(!src_addr && !offered_set.isMulticast())
165         src_addr = offered_set.getFirstAddr();
```

```
166     }
167     m_state->clean();
168     m_state->m_protocol->m_requesting_state.start(server_addr, src_addr, &
        offered_set);
169   }
170   else if(m_state->m_change_discovery == true)
171     m_state->m_protocol->restart();
172   else if(--(m_state->m_dsc_count) > 0)
173     m_state->sendDiscover();
174   else if(aux_set.checkConflict(&m_state->m_discovery_set, &
        AUTOASSIGN_UNICAST)
175         || ((aux_set.checkConflict(&m_state->m_discovery_set, &
        AUTOASSIGN_MULTICAST)
176             || aux_set.checkConflict(&m_state->m_discovery_set, &
        AUTOASSIGN_UNICAST_64)
177             || aux_set.checkConflict(&m_state->m_discovery_set, &
        AUTOASSIGN_MULTICAST_64))
178           && m_state->m_protocol->m_preassigned_addr))
179   {
180     AddrSet assigned_set = m_state->m_discovery_set;
181     uint64_t size = TO_SIZE(m_state->m_protocol->m_config.get(ConfigItem::
        MAX_ADDR_CLAIM));
182     if(!assigned_set.isMulticast())
183       size = MIN(size, MAX_AUTOASSIGN_UNICAST);
184     if(TO_BOOL(m_state->m_protocol->m_config.get(ConfigItem::RANDOM_ASSIGN
        )))
185       assigned_set = RandomAddrSet(assigned_set, size);
186     else
187       adjustSet(&assigned_set, size);
188     m_state->clean();
189     m_state->m_protocol->m_defending_state.start(&assigned_set);
190   }
191   else
192     m_state->m_protocol->restart();
193 }
194
195
196 RequestingState::RequestingState(PalmaClient *protocol) : State(protocol),
197                                m_pkt(NULL),
198                                m_request_timer(this) {}
199
200 RequestingState::RequestTimer::RequestTimer(RequestingState *state) :
        m_state(state) {}
201
202 void RequestingState::start(uint64_t server_addr, uint64_t src_addr,
        AddrSet *set, bool renewal)
203 {
204   m_protocol->printv("SERVER_REQUESTING,0x%lx,0x%lx\n",
205       set->getFirstAddr(),
206       set->getSize());
207   m_protocol->m_curstate = this;
```

```
208   if(m_protocol->m_mcast_on)
209   {
210     m_protocol->m_netitf.delAddr(PALMA_MCAST,true);
211     m_protocol->m_mcast_on = false;
212   }
213   m_req_count = REQUEST_REQ_COUNT;
214   m_server_addr = server_addr;
215   m_src_addr = src_addr;
216   m_pkt = new Packet(MsgType::REQUEST, server_addr, src_addr, m_protocol->
      getToken());
217   adjustSet(set, TO_SIZE(m_protocol->m_config.get(ConfigItem::
      MAX_ADDR_CLAIM)));
218   m_pkt->addMacSetPar(set);
219   if(TO_STRING(m_protocol->m_config.get(ConfigItem::STATION_ID)) != NULL)
220     m_pkt->addIdPar(ParType::STATION_ID, TO_STRING(m_protocol->m_config.
      get(ConfigItem::STATION_ID)));
221   if(renewal)
222     m_pkt->setRenewal();
223   else if(!m_protocol->m_preassigned_addr)
224     m_protocol->m_netitf.addAddr(src_addr);
225   sendRequest();
226 }
227
228 void RequestingState::clean()
229 {
230   m_server_addr = 0;
231   if(m_src_addr != 0)
232   {
233     m_protocol->m_netitf.delAddr(m_src_addr);
234     m_src_addr = 0;
235   }
236   if(m_pkt)
237   {
238     delete m_pkt;
239     m_pkt = NULL;
240   }
241   m_protocol->m_event_loop.stopTimer(&m_request_timer);
242 }
243
244 void RequestingState::sendRequest()
245 {
246   m_protocol->m_netitf.netsend(m_pkt);
247   m_protocol->m_event_loop.startTimer(&m_request_timer, REQUESTING_TIMEOUT
      );
248 }
249
250 void RequestingState::handlePacket(Packet *pkt)
251 {
252   if(pkt->getType() == MsgType::ACK &&
253       pkt->getDA() == m_src_addr &&
254       pkt->getSA() == m_server_addr &&
```

```
255       pkt->getToken() == m_protocol->getToken() &&
256       m_protocol->checkStationId(pkt->getStationId()))
257   {
258     uint16_t lifetime = pkt->getLifetime();
259     if((pkt->getStatus() == StatusCode::ASSIGN_OK
260         || pkt->getStatus() == StatusCode::ALTERNATE_SET)
261           && lifetime > 0)
262     {
263       m_protocol->m_src_addr = m_src_addr;
264       m_src_addr = 0;
265       m_protocol->m_server_addr = m_server_addr;
266       m_protocol->m_assigned_set = *pkt->getSet();
267       clean();
268       m_protocol->m_bound_state.start(lifetime, validSet(pkt));
269     }
270     else
271     {
272       m_protocol->restart();
273     }
274   }
275 }
276
277 bool RequestingState::checkNetworkId(uint8_t *rcv, uint8_t *snd)
278 {
279   if(rcv == NULL ||
280       (snd != NULL && strcmp((const char *)rcv, (const char *)snd) == 0))
281     return true;
282   return false;
283 }
284
285 void RequestingState::RequestTimer::timeout()
286 {
287   if(--(m_state->m_req_count) > 0)
288   {
289     m_state->sendRequest();
290   }
291   else
292   {
293     m_state->m_protocol->restart();
294   }
295 }
296
297
298 DefendingState::DefendingState(PalmaClient *protocol) : State(protocol),
299                           m_lease_lifetime_timer(this),
300                           m_announcement_timer(this) {}
301
302 DefendingState::LeaseLifetimeTimer::LeaseLifetimeTimer(DefendingState *
    state) : m_state(state) {}
303
304 DefendingState::AnnouncementTimer::AnnouncementTimer(DefendingState *state
```

```
    ) : m_state(state) {}
305
306
307 void DefendingState::start(AddrSet *set)
308 {
309   m_protocol->m_curstate = this;
310   if(!m_protocol->m_preassigned_addr)
311   {
312     RandomAddrSet src_addr(*set);
313     m_protocol->m_src_addr = src_addr.getFirstAddr();
314     m_protocol->m_netitf.addAddr(m_protocol->m_src_addr);
315   }
316   m_protocol->m_assigned_set = *set;
317   m_protocol->printv("AUTO_ASSIGNED,0x%lx,0x%lx\n",
318       m_protocol->m_assigned_set.getFirstAddr(),
319       m_protocol->m_assigned_set.getSize());
320   m_protocol->m_event_loop.startTimer(&m_lease_lifetime_timer,
     SELF_ASSIGMENT_LIFETIME);
321   sendAnnounce();
322 }
323
324 void DefendingState::clean()
325 {
326   if(!m_protocol->m_preassigned_addr)
327   {
328     m_protocol->m_netitf.delAddr(m_protocol->m_src_addr);
329     m_protocol->m_src_addr = 0;
330   }
331   m_protocol->m_assigned_set = AddrSet();
332   m_protocol->m_event_loop.stopTimer(&m_announcement_timer);
333   m_protocol->m_event_loop.stopTimer(&m_lease_lifetime_timer);
334 }
335
336 void DefendingState::sendAnnounce()
337 {
338   Packet pkt(MsgType::ANNOUNCE, PALMA_MCAST, m_protocol->m_src_addr,
     m_protocol->getToken());
339   pkt.addMacSetPar(&m_protocol->m_assigned_set);
340   pkt.addLifetimePar((uint16_t)(m_protocol->m_event_loop.readTimer(&
     m_lease_lifetime_timer) + 0.5));
341   if(TO_STRING(m_protocol->m_config.get(ConfigItem::STATION_ID)) != NULL)
342     pkt.addIdPar(ParType::STATION_ID, TO_STRING(m_protocol->m_config.get(
     ConfigItem::STATION_ID)));
343   m_protocol->m_netitf.netsend(&pkt);
344   m_protocol->m_event_loop.startTimer(&m_announcement_timer,
     ANNOUNCE_TIMEOUT);
345 }
346
347 void DefendingState::sendDefend(AddrSet* original_set, AddrSet *
    conflict_set, uint64_t DA, uint8_t *station_id)
348 {
```

```
349   Packet pkt(MsgType::DEFEND, DA, m_protocol->m_src_addr, m_protocol->
       getToken());
350   if(station_id != NULL)
351     pkt.addIdPar(ParType::STATION_ID, station_id);
352   pkt.addLifetimePar((uint16_t)(m_protocol->m_event_loop.readTimer(&
       m_lease_lifetime_timer) + 0.5));
353   pkt.addMacSetPar(original_set);
354   pkt.addMacSetPar(conflict_set, false);
355   m_protocol->m_netitf.netsend(&pkt);
356 }
357
358 void DefendingState::handlePacket(Packet *pkt)
359 {
360   AddrSet conflict_set;
361   if(pkt->getDA() == PALMA_MCAST)
362   {
363     if(pkt->getType() == MsgType::ANNOUNCE)
364     {
365       m_protocol->m_db.exclude(pkt->getSet(), pkt->getLifetime());
366       if(conflict_set.checkConflict(pkt->getSet(),&m_protocol->
       m_assigned_set))
367         processConflict(pkt->getSet(), &conflict_set, pkt->getSA(), pkt->
       getStationId());
368     }
369     else if(pkt->getType() == MsgType::DISCOVER &&
370         conflict_set.checkConflict(pkt->getSet(), &m_protocol->
       m_assigned_set))
371     {
372       sendDefend(pkt->getSet(), &conflict_set, pkt->getSA(), pkt->
       getStationId());
373     }
374   }
375   else if(pkt->getDA() == m_protocol->m_src_addr &&
376       m_protocol->checkStationId(pkt->getStationId()))
377   {
378     if(pkt->getType() == MsgType::DEFEND)
379     {
380       m_protocol->m_db.exclude(pkt->getSet(false), pkt->getLifetime());
381       if(conflict_set.checkConflict(pkt->getSet(false),&m_protocol->
       m_assigned_set))
382       {
383         processConflict(pkt->getSet(false), &conflict_set, pkt->getSA());
384       }
385
386     }
387     else if(pkt->getType() == MsgType::OFFER && pkt->getToken() ==
       m_protocol->getToken())
388     {
389       if(validOffer(pkt))
390       {
391         AddrSet offered_set(*pkt->getSet());
```

```
392         uint64_t server_addr = pkt->getSA();
393         uint64_t src_addr = m_protocol->m_src_addr;
394         if(!m_protocol->m_preassigned_addr)
395         {
396           m_protocol->m_src_addr = 0;
397           src_addr = pkt->getClientAddr();
398           if(!src_addr && !offered_set.isMulticast())
399             src_addr = offered_set.getFirstAddr();
400         }
401         clean();
402         m_protocol->m_requesting_state.start(server_addr, src_addr, &
    offered_set);
403       }
404     }
405   }
406 }
407
408 void DefendingState::processConflict(AddrSet *original_set, AddrSet *
    conflict_set, uint64_t DA, uint8_t *station_id)
409 {
410   uint64_t min_assigned = TO_SIZE(m_protocol->m_config.get(ConfigItem::
    MIN_ADDR_CLAIM));
411   uint64_t max_assigned = TO_SIZE(m_protocol->m_config.get(ConfigItem::
    MAX_ADDR_CLAIM));
412   int64_t left_count = conflict_set->getFirstAddr() - m_protocol->
    m_assigned_set.getFirstAddr();
413   bool set_changed = false;
414   bool send_defend = false;
415   if(left_count >= min_assigned)
416   {
417     adjustSet(&m_protocol->m_assigned_set, left_count);
418     set_changed = true;
419   }
420   else if(min_assigned < m_protocol->m_assigned_set.getSize() &&
    left_count > 0)
421   {
422     adjustSet(&m_protocol->m_assigned_set, min_assigned);
423     set_changed = true;
424     conflict_set->checkConflict(original_set, &m_protocol->m_assigned_set)
    ;
425     send_defend = true;
426   }
427   else
428   {
429     m_protocol->restart();
430   }
431   if(set_changed)
432   {
433     if(!m_protocol->m_preassigned_addr)
434     {
435       RandomAddrSet src_addr(m_protocol->m_assigned_set);
```

```
436       m_protocol->m_netitf.addAddr(m_protocol->m_src_addr);
437       m_protocol->m_src_addr = src_addr.getFirstAddr();
438       m_protocol->m_netitf.addAddr(m_protocol->m_src_addr);
439     }
440   }
441   if(send_defend)
442     sendDefend(original_set, conflict_set, DA, station_id);
443 }
444
445 void DefendingState::AnnouncementTimer::timeout()
446 {
447   m_state->sendAnnounce();
448 }
449
450 void DefendingState::LeaseLifetimeTimer::timeout()
451 {
452   m_state->m_protocol->restart();
453 }
454
455 BoundState::BoundState(PalmaClient *protocol) : State(protocol),
456                         m_lease_lifetime_timer(this) {}
457
458 BoundState::LeaseLifetimeTimer::LeaseLifetimeTimer(BoundState *state) :
    m_state(state) {}
459
460 void BoundState::start(uint16_t lifetime, bool acceptable)
461 {
462   m_protocol->printv("SERVER_ASSIGNED,0x%lx,0x%lx\n",
463       m_protocol->m_assigned_set.getFirstAddr(),
464       m_protocol->m_assigned_set.getSize());
465   m_protocol->m_curstate = this;
466   if(acceptable)
467   {
468     if(TO_BOOL(m_protocol->m_config.get(ConfigItem::RENEWAL)) && lifetime
    > 1)
469       lifetime -= 1;
470     m_protocol->m_event_loop.startTimer(&m_lease_lifetime_timer, (double)
    lifetime);
471   }
472   else
473     sendRelease();
474 }
475
476 void BoundState::clean()
477 {
478   m_protocol->m_event_loop.stopTimer(&m_lease_lifetime_timer);
479 }
480
481 void BoundState::sendRelease(bool terminate)
482 {
483   Packet pkt(MsgType::RELEASE, m_protocol->m_server_addr, m_protocol->
```

```
    m_src_addr, m_protocol->getToken());
  pkt.addMacSetPar(&m_protocol->m_assigned_set);
  if(TO_STRING(m_protocol->m_config.get(ConfigItem::STATION_ID)) != NULL)
    pkt.addIdPar(ParType::STATION_ID, TO_STRING(m_protocol->m_config.get(
    ConfigItem::STATION_ID)));
  m_protocol->m_netitf.netsend(&pkt);
  m_protocol->printv("SERVER_RELEASED,0x%lx,0x%lx\n",
      m_protocol->m_assigned_set.getFirstAddr(),
      m_protocol->m_assigned_set.getSize());
  if(!terminate)
    m_protocol->restart();
}

void BoundState::LeaseLifetimeTimer::timeout()
{
  if(TO_BOOL(m_state->m_protocol->m_config.get(ConfigItem::RENEWAL)))
  {
    m_state->clean();
    m_state->m_protocol->printv("SERVER_RENEWAL,0x%lx,0x%lx\n",
      m_state->m_protocol->m_assigned_set.getFirstAddr(),
      m_state->m_protocol->m_assigned_set.getSize());
    m_state->m_protocol->m_requesting_state.start(m_state->m_protocol->
    m_server_addr, m_state->m_protocol->m_src_addr, &m_state->m_protocol->
    m_assigned_set, true);
  }
  else
    m_state->m_protocol->restart();
}
```

B.2.3 Configuration Parameters

```
#ifndef CONFIG_CLIENT_H
#define CONFIG_CLIENT_H

#include "../common/config.h"

enum ConfigItem
{
  INTERFACE,
  PREASSIGNED_ADDR,
  CLAIM_SET,
  MIN_ADDR_CLAIM,
  MAX_ADDR_CLAIM,
  KNOWN_SERVER_ADDR,
  RENEWAL,
  RANDOM_ASSIGN,
  STATION_ID,
  VENDOR,
  VERBOSE,
  MAX_CONFIG_ITEM,
};


class ConfigClient : public Config
{
public:
  ConfigClient();
  ~ConfigClient();
  int getMaxItem() {return MAX_CONFIG_ITEM;}
  bool check(const char *fname);
};

#endif
```

```
#include <string.h>
#include <ctype.h>
#include <stdio.h>
#include <stdlib.h>
#include "../common/addrset.h"
#include "config-client.h"

ConfigClient::ConfigClient()
{
  m_list = new ConfigElement *[ConfigItem::MAX_CONFIG_ITEM]
  {
    new ConfigString(NULL),
    new ConfigAddr(0),
    new ConfigSet(0x0a0000000000L, (0xffffffffffL + 1)),
    new ConfigSize(1),
    new ConfigSize(16),
    new ConfigAddr(0),
```

```
    new ConfigBool(false),
    new ConfigBool(true),
    new ConfigString(NULL),
    new ConfigString(NULL),
    new ConfigBool(false),
  };
  m_root_tag = "ClientConfig";
  m_array_tags = new const char*[ConfigItem::MAX_CONFIG_ITEM]
  {
    "InterfaceName",
    "PreassignedSrcAddress",
    "ClaimAddressSet",
    "MinAssignedAddresses",
    "MaxAssignedAddresses",
    "KnownServerAddress",
    "RenewalActive",
    "RandomAutoAssign",
    "ClientStationId",
    "VendorParameter",
    "Verbose"
  };
}

ConfigClient::~ConfigClient()
{
  for(int i=0; i<ConfigItem::MAX_CONFIG_ITEM; i++)
    delete m_list[i];
  delete m_list;
  delete m_array_tags;
}

bool ConfigClient::check(const char *fname)
{
  if(TO_STRING(get(ConfigItem::INTERFACE)) == NULL)
  {
    fprintf(stderr, "%s: Missing interface name\n", fname);
    return false;
  }
  if((TO_ADDRSET_PTR(get(ConfigItem::CLAIM_SET))->getSize() > 0
    && TO_ADDRSET_PTR(get(ConfigItem::CLAIM_SET))->getSize() < TO_SIZE(get
   (ConfigItem::MAX_ADDR_CLAIM)))
      || TO_SIZE(get(ConfigItem::MIN_ADDR_CLAIM)) > TO_SIZE(get(ConfigItem
   ::MAX_ADDR_CLAIM)))
  {
    fprintf(stderr, "%s: Incoherent sizes\n", fname);
    return false;
  }
  if(!TO_ADDR(get(ConfigItem::PREASSIGNED_ADDR)) && TO_ADDR(get(ConfigItem
   ::KNOWN_SERVER_ADDR)))
  {
    fprintf(stderr, "%s: PreassignedSrcAddress is required for
```

```
    KnownServerAddress\n", fname);
66      return false;
67    }
68    return true;
69  }
```

B.2.4 Main file

```
#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>
#include <unistd.h>

#include "palma-client.h"
#include "../common/packet.h"
#include "config-client.h"


int main(int argc, char *argv[])
{
  // Procesa opciones
  int c;
  char *itfname = NULL;
  char *confname = NULL;
  char *station_id = NULL;
  uint64_t preassigned_addr = 0;

  while ((c = getopt (argc, argv, "c:i:s:p:")) != -1)
  {
    switch (c)
    {
      case 'c':
        confname = optarg;
        break;
      case 'i':
        itfname = optarg;
        break;
      case 's':
        station_id = optarg;
        break;
      case 'p':
          if(!Config::toUint64(optarg, preassigned_addr))
        {
          fprintf(stderr,"Invalid address value.\n");
          exit(1);
        }
        break;
      case '?':
        fprintf(stderr,"Invalid option.\n");
        fprintf(stderr, "Uso:%s [-c <config filename>][-i <interface name
   >][-s <station id>][-p <preassigned address>]\n", argv[0]);
        exit(1);
      default:
        abort();
    }
  }
  if(optind != argc)
```

```
49    {
50      fprintf(stderr,"Invalid arguments.\n");
51      fprintf(stderr, "Uso:%s [-c <config filename>][-i <interface name>][-s
        <station id>][-p <preassigned address>]\n", argv[0]);
52      exit(1);
53    }
54
55    PalmaClient::initRandom();
56    PalmaClient client;
57    client.m_config.set(ConfigItem::INTERFACE, itfname);
58    client.m_config.set(ConfigItem::STATION_ID, station_id);
59    client.m_config.set(ConfigItem::PREASSIGNED_ADDR, &preassigned_addr);
60
61    if(confname && !client.m_config.read(confname))
62    {
63      fprintf(stderr, "Invalid configuration in: %s\n", confname);
64      exit(1);
65    }
66    client.begin();
67  }
```

B.3 Server Modules

B.3.1 Palma Server

```
#ifndef PALMA_SERVER_H
#define PALMA_SERVER_H

#include "../common/netitf.h"
#include "../common/eventloop.h"
#include "config-server.h"
#include "../common/database.h"
#include "../common/siphash.h"
#include "../common/palma.h"

class PalmaServer : public Palma
{
public:
  ConfigServer m_config;
  SetDatabase m_db_unicast;
  SetDatabase m_db_multicast;
  SetDatabase m_db_unicast_64;
  SetDatabase m_db_multicast_64;
  SipHash m_hash;
  uint64_t m_src_addr;

  PalmaServer();
  void begin();
  void handlePacket(Packet *pkt);
  bool defineSet(bool isMulticast, bool isSize64, SetDatabase *&db,
          uint64_t *max_addr = NULL, uint16_t *lifetime = NULL, bool *
    send_client_addr = NULL);
  void processClaim(Packet *pkt);
  void sendOffer(uint64_t dest_addr, uint16_t token, AddrSet *offer_set,
            uint16_t lifetime, uint8_t *station_id = NULL, AddrSet *
    client_addr = NULL);
  void processRequest(Packet *pkt);
  void sendAck(uint64_t dest_addr, uint16_t token, uint8_t *station_id,
            StatusCode status, AddrSet *set = NULL, uint16_t lifetime = 0)
    ;
  void processRelease(Packet *pkt);
  uint64_t getSecurityId(uint16_t token, uint8_t *station_id, uint64_t
    src_addr = 0);
  void onExit();
};

#endif
```

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

```

```
#include "palma-server.h"
#include "../common/details.h"

#define MIN(a,b) ((a < b) ? a : b)

PalmaServer::PalmaServer() :  m_db_unicast(this),
                m_db_multicast(this),
                m_db_unicast_64(this),
                m_db_multicast_64(this),
                m_src_addr(0) {}

void PalmaServer::begin()
{
  m_netitf.init(TO_STRING(m_config.get(ConfigItem::INTERFACE)));
  m_event_loop.regSource(&m_netitf);
  m_event_loop.regHandler(this);
  m_src_addr = TO_ADDR(m_config.get(ConfigItem::SRC_ADDR));
  m_netitf.addAddr(m_src_addr);

  AddrSet *unicast_set = TO_ADDRSET_PTR(m_config.get(ConfigItem::
   UNICAST_SET));
  AddrSet *multicast_set = TO_ADDRSET_PTR(m_config.get(ConfigItem::
   MULTICAST_SET));
  AddrSet *unicast_64_set = TO_ADDRSET_PTR(m_config.get(ConfigItem::
   UNICAST_64_SET));
  AddrSet *multicast_64_set = TO_ADDRSET_PTR(m_config.get(ConfigItem::
   MULTICAST_64_SET));
  m_db_unicast.init(unicast_set);
  m_db_multicast.init(multicast_set);
  m_db_unicast_64.init(unicast_64_set);
  m_db_multicast_64.init(multicast_64_set);
  m_event_loop.run();
}

void PalmaServer::handlePacket(Packet *pkt)
{
  if(pkt->getDA() == PALMA_MCAST)
  {
    switch(pkt->getType())
    {
      case MsgType::DISCOVER:
        processClaim(pkt);
        break;
      case MsgType::ANNOUNCE:
        if(TO_BOOL(m_config.get(ConfigItem::AUTOASSIGN_OBJECTION)))
          processClaim(pkt);
        break;
    }
  }
  else if(pkt->getDA() == m_src_addr)
  {
```

```
52      switch(pkt->getType())
53      {
54        case MsgType::REQUEST:
55          processRequest(pkt);
56          break;
57        case MsgType::RELEASE:
58          processRelease(pkt);
59          break;
60      }
61    }
62  }
63
64  bool PalmaServer::defineSet(bool isMulticast, bool isSize64, SetDatabase
      *&db, uint64_t *max_addr, uint16_t *lifetime, bool *send_client_addr)
65  {
66    uint64_t max_addr_unicast = TO_SIZE(m_config.get(ConfigItem::
      MAX_ADDR_UNICAST));
67    uint64_t max_addr_multicast = TO_SIZE(m_config.get(ConfigItem::
      MAX_ADDR_MULTICAST));
68    uint64_t max_addr_unicast_64 = TO_SIZE(m_config.get(ConfigItem::
      MAX_ADDR_UNICAST_64));
69    uint64_t max_addr_multicast_64 = TO_SIZE(m_config.get(ConfigItem::
      MAX_ADDR_MULTICAST_64));
70    if(send_client_addr != NULL)
71      *send_client_addr = true;
72
73    if(isMulticast)
74      {
75        if(!isSize64)
76        {
77          if(max_addr_multicast <= 0)
78            return false;
79          db = &m_db_multicast;
80          if(max_addr != NULL)
81            *max_addr = max_addr_multicast;
82          if(lifetime != NULL)
83            *lifetime = TO_UINT(m_config.get(ConfigItem::MULTICAST_LIFETIME)
      );
84        }
85        else
86        {
87          if(max_addr_multicast_64 <= 0)
88            return false;
89          db = &m_db_multicast_64;
90          if(max_addr != NULL)
91            *max_addr = max_addr_multicast_64;
92          if(lifetime != NULL)
93            *lifetime = TO_UINT(m_config.get(ConfigItem::
      MULTICAST_64_LIFETIME));
94        }
95      }
```

```
96      else
97      {
98        if(isSize64)
99        {
100         if(max_addr_unicast_64 <= 0)
101           return false;
102         db = &m_db_unicast_64;
103         if(max_addr != NULL)
104           *max_addr = max_addr_unicast_64;
105         if(lifetime != NULL)
106           *lifetime = TO_UINT(m_config.get(ConfigItem::UNICAST_64_LIFETIME
    ));
107       }
108       else
109       {
110         if(max_addr_unicast <= 0)
111           return false;
112         db = &m_db_unicast;
113         if(max_addr != NULL)
114           *max_addr = max_addr_unicast;
115         if(lifetime != NULL)
116           *lifetime = TO_UINT(m_config.get(ConfigItem::UNICAST_LIFETIME));

117         if(send_client_addr != NULL)
118           *send_client_addr = false;
119       }
120
121     }
122   return true;
123 }
124
125 void PalmaServer::processClaim(Packet *pkt)
126 {
127   uint64_t src_addr = pkt->getSA();
128   uint8_t *station_id = pkt->getStationId();
129   uint16_t token = pkt->getToken();
130   AddrSet *claimed_set = pkt->getSet();
131   uint64_t security_id = getSecurityId(token, station_id);
132   bool isMulticast;
133   bool isSize64;
134   uint64_t max_addr_offer;
135   AddrSet *offer_set;
136   SetDatabase *db;
137   uint16_t lifetime;
138   bool send_client_addr;
139   uint64_t max_addr;
140
141   max_addr_offer = TO_SIZE(m_config.get(ConfigItem::DEFAULT_ADDR_OFFER));
142   if(claimed_set == NULL)
143   {
144     isSize64 = TO_BOOL(m_config.get(ConfigItem::DEFAULT_64));
```

```
145     isMulticast = TO_BOOL(m_config.get(ConfigItem::DEFAULT_MULTICAST));
146   }
147   else
148   {
149     isSize64 =  claimed_set->isSize64();
150     isMulticast = claimed_set->isMulticast();
151     if(claimed_set->getSize() > 0)
152       max_addr_offer = claimed_set->getSize();
153   }
154 
155   if(!defineSet(isMulticast, isSize64, db, &max_addr, &lifetime, &
      send_client_addr))
156     return;
157 
158   AddrSet src_addr_set(src_addr);
159   AddrSet check_set;
160   AddrSet *client_addr = NULL;
161 
162   max_addr_offer = MIN(max_addr, max_addr_offer);
163   offer_set = db->reserve(max_addr_offer, security_id, TO_UINT(m_config.
      get(ConfigItem::RESERVE_LIFETIME)));
164   if(offer_set != NULL)
165   {
166     if(send_client_addr && check_set.checkConflict(&src_addr_set, &
      DISCOVER_SOURCE_ADDR_RANGE))
167     {
168       client_addr = m_db_unicast.reserve(1, security_id, TO_UINT(m_config.
      get(ConfigItem::RESERVE_LIFETIME)));
169       if(client_addr == NULL)
170       {
171         db->release((AssignableSet*)offer_set);
172         return;
173       }
174     }
175     sendOffer(src_addr, token, offer_set, lifetime, station_id,
      client_addr);
176   }
177 }
178 
179 
180 void PalmaServer::sendOffer(uint64_t dest_addr, uint16_t token, AddrSet *
      offer_set,
181         uint16_t lifetime, uint8_t *station_id, AddrSet *client_addr)
182 {
183   uint8_t *id;
184 
185   Packet pkt(MsgType::OFFER, dest_addr, m_src_addr, token);
186   pkt.addLifetimePar(lifetime);
187   if(offer_set->getSize() > 0xffff)
188     offer_set->alignToMask(SetType::MASK);
189   pkt.addMacSetPar(offer_set);
```

```
190   if(station_id != NULL)
191     pkt.addIdPar(ParType::STATION_ID, station_id);
192   id = TO_STRING(m_config.get(ConfigItem::NETWORK_ID));
193   if(id != NULL)
194     pkt.addIdPar(ParType::NETWORK_ID, id);
195   if(client_addr != NULL)
196     pkt.addClientAddrPar(client_addr);
197   id = TO_STRING(m_config.get(ConfigItem::VENDOR));
198   if(id != NULL)
199     pkt.addVendorPar(id, strlen((const char *)id));
200   m_netitf.netsend(&pkt);
201 }
202
203 void PalmaServer::processRequest(Packet *pkt)
204 {
205   uint64_t security_id;
206   AddrSet *requested_set = pkt->getSet();
207   uint8_t * station_id = pkt->getStationId();
208   uint16_t token = pkt->getToken();
209   uint64_t src_addr = pkt->getSA();
210   uint64_t reserved_security_id = getSecurityId(token, station_id);
211   uint64_t assigned_security_id = getSecurityId(token, station_id,
       src_addr);
212
213   bool isSize64 =  requested_set->isSize64();
214   bool isMulticast = requested_set->isMulticast();
215   uint64_t max_addr;
216   uint16_t lifetime;
217   SetDatabase *db;
218
219   AddrSet src_addr_set(src_addr);
220   AddrSet check_set;
221
222   if(!defineSet(isMulticast, isSize64, db, &max_addr, &lifetime)
223     || check_set.checkConflict(&src_addr_set, &AUTOASSIGN_UNICAST)
224     || check_set.checkConflict(&src_addr_set, &DISCOVER_SOURCE_ADDR_RANGE)
       )
225   {
226     sendAck(src_addr, token, station_id, StatusCode::FAIL_OTHER);
227     return;
228   }
229
230   DbStatus db_status;
231   uint16_t left_lifetime;
232   bool identical;
233   AssignableSet *src_assign_set = NULL;
234   AssignableSet *result;
235
236   if(check_set.checkConflict(&src_addr_set, TO_ADDRSET_PTR(m_config.get(
       ConfigItem::UNICAST_SET))))
237   {
```

```
    db_status = m_db_unicast.checkStatus(&src_addr_set, security_id,
   left_lifetime, identical, result);
    if((db_status == DbStatus::RESERVED && security_id ==
   reserved_security_id)
       || (db_status == DbStatus::ASSIGNED && security_id ==
   assigned_security_id))
    {
      if(!check_set.checkConflict(&src_addr_set, requested_set))
        src_assign_set = result;
    }
    else
    {
      sendAck(src_addr, token, station_id, StatusCode::FAIL_OTHER);
      return;
    }
  }
  StatusCode ack_status = StatusCode::ASSIGN_OK;
  uint64_t size = requested_set->getSize();
  if(size == 0)
  {
    if(TO_BOOL(m_config.get(ConfigItem::ENABLE_ALTERNATE_SET)))
    {
      requested_set->setSize(TO_SIZE(m_config.get(ConfigItem::
   DEFAULT_ADDR_OFFER)));
      ack_status = StatusCode::ALTERNATE_SET;
    }
    else
    {
      sendAck(src_addr, token, station_id, StatusCode::FAIL_DISALLOWED);
      return;
    }
  }
  else if(size > max_addr)
  {
    if(TO_BOOL(m_config.get(ConfigItem::ENABLE_ALTERNATE_SET)))
    {
      requested_set->setSize(max_addr);
      ack_status = StatusCode::ALTERNATE_SET;
    }
    else
    {
      sendAck(src_addr, token, station_id, StatusCode::FAIL_TOO_LARGE);
      return;
    }

  }

  db_status = db->checkStatus(requested_set, security_id, left_lifetime,
   identical, result);

  if (db_status == DbStatus::FREE
```

```
284       || ((db_status == DbStatus::RESERVED) && (security_id ==
    reserved_security_id))
285       || (((db_status == DbStatus::ASSIGNED) && (security_id ==
    assigned_security_id))
286           && (pkt->getRenewal() && TO_BOOL(m_config.get(ConfigItem::
    ACCEPT_RENEWAL)))))
287   {
288     if(src_assign_set != NULL)
289       m_db_unicast.assign(src_assign_set, &src_addr_set,
    assigned_security_id, lifetime);
290     db->assign(result, requested_set, assigned_security_id, lifetime);
291     sendAck(src_addr, token, station_id, ack_status, requested_set,
    lifetime);
292   }
293
294   else if(db_status == DbStatus::ASSIGNED && security_id ==
    assigned_security_id)
295     sendAck(src_addr, token, station_id, ack_status, requested_set,
    left_lifetime);
296
297   else if(TO_BOOL(m_config.get(ConfigItem::ENABLE_ALTERNATE_SET)))
298   {
299     AddrSet *set = db->assign(requested_set->getSize(),
    assigned_security_id, lifetime);
300     if(set == NULL)
301       sendAck(src_addr, token, station_id, StatusCode::FAIL_CONFLICT);
302     else
303     {
304       if(src_assign_set != NULL)
305         m_db_unicast.assign(src_assign_set, &src_addr_set,
    assigned_security_id, lifetime);
306       sendAck(src_addr, token, station_id, StatusCode::ALTERNATE_SET, set,
     lifetime);
307     }
308   }
309   else if((db_status == DbStatus::ASSIGNED && security_id !=
    assigned_security_id)
310         || (db_status == DbStatus::RESERVED && security_id !=
    reserved_security_id))
311     sendAck(src_addr, token, station_id, StatusCode::FAIL_CONFLICT);
312   else
313     sendAck(src_addr, token, station_id, StatusCode::FAIL_DISALLOWED);

314 }
315
316 void PalmaServer::sendAck(uint64_t dest_addr, uint16_t token, uint8_t *
    station_id, StatusCode status, AddrSet *set, uint16_t lifetime)
317 {
318   Packet pkt(MsgType::ACK, dest_addr, m_src_addr, token, status);
319   if(station_id != NULL)
320     pkt.addIdPar(ParType::STATION_ID, station_id);
```

```
321   if(set != NULL)
322   {
323     pkt.addMacSetPar(set);
324     pkt.addLifetimePar(lifetime);
325   }
326   m_netitf.netsend(&pkt);
327 }
328
329 void PalmaServer::processRelease(Packet *pkt)
330 {
331
332   AddrSet *released_set = pkt->getSet();
333   uint8_t * station_id = pkt->getStationId();
334   uint16_t token = pkt->getToken();
335   uint64_t src_addr = pkt->getSA();
336   uint64_t security_id;
337   uint16_t left_lifetime;
338   uint64_t assigned_security_id = getSecurityId(token, station_id,
       src_addr);
339   AddrSet src_addr_set(src_addr);
340   SetDatabase *db;
341   bool isMulticast = released_set->isMulticast();
342   bool isSize64 = released_set->isSize64();
343
344   if(!defineSet(isMulticast, isSize64, db))
345     return;
346
347   bool identical;
348   AssignableSet *result;
349   if(db->checkStatus(released_set, security_id, left_lifetime, identical,
       result) == DbStatus::ASSIGNED
350     && security_id == assigned_security_id && identical)
351   {
352     db->release(result);
353     if(m_db_unicast.checkStatus(&src_addr_set, security_id, left_lifetime,
       identical, result) == DbStatus::ASSIGNED
354       && security_id == assigned_security_id && identical)
355       m_db_unicast.release(result);
356   }
357 }
358
359 uint64_t PalmaServer::getSecurityId(uint16_t token, uint8_t *station_id,
       uint64_t src_addr)
360 {
361   m_hash.begin();
362   if(src_addr != 0)
363     m_hash.update(src_addr);
364   m_hash.update(token);
365   m_hash.update(station_id);
366   return m_hash.digest();
367 }
```

```
368
369 void PalmaServer::onExit()
370 {
371   printf("ENDING\n");
372 }
```

B.3.2 Main file

```
#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>
#include <unistd.h>

#include "palma-server.h"
#include "../common/packet.h"
#include "config-server.h"


int main(int argc, char *argv[])
{
  // Procesa opciones
  int c;
  char *itfname = NULL;
  char *confname = NULL;

  while ((c = getopt (argc, argv, "c:i:")) != -1)
  {
    switch (c)
    {
      case 'c':
        confname = optarg;
        break;
      case 'i':
        itfname = optarg;
        break;
      case '?':
        fprintf(stderr,"Invalid option.\n");
        fprintf(stderr,"Uso:%s [-c <config filename>][-i <interface name
   >]\n", argv[0]);
        exit(1);
      default:
        abort();
    }
  }
  if(optind != argc)
  {
    fprintf(stderr,"Invalid arguments.\n");
    fprintf(stderr,"Uso:%s [-c <config filename>][-i <interface name>]\n",
    argv[0]);
    exit(1);
  }

  PalmaServer::initRandom();
  PalmaServer server;
  server.m_config.set(ConfigItem::INTERFACE, itfname);

  if(confname && !server.m_config.read(confname))
```

```
  {
    fprintf(stderr, "Invalid configuration in: %s\n", confname);
    exit(1);
  }

  server.begin();
}
```

B.3.3 Configuration Parameters

```
#ifndef CONFIG_CLIENT_H
#define CONFIG_CLIENT_H

#include "../common/config.h"

enum ConfigItem
{
  INTERFACE,
  SRC_ADDR,
  UNICAST_SET,
  MULTICAST_SET,
  UNICAST_64_SET,
  MULTICAST_64_SET,
  MAX_ADDR_UNICAST,
  MAX_ADDR_MULTICAST,
  MAX_ADDR_UNICAST_64,
  MAX_ADDR_MULTICAST_64,
  DEFAULT_ADDR_OFFER,
  UNICAST_LIFETIME,
  MULTICAST_LIFETIME,
  UNICAST_64_LIFETIME,
  MULTICAST_64_LIFETIME,
  RESERVE_LIFETIME,
  ACCEPT_RENEWAL,
  DEFAULT_MULTICAST,
  DEFAULT_64,
  AUTOASSIGN_OBJECTION,
  ENABLE_ALTERNATE_SET,
  NETWORK_ID,
  VENDOR,
  MAX_CONFIG_ITEM,
};


class ConfigServer : public Config
{
public:
  ConfigServer();
  ~ConfigServer();
  int getMaxItem() {return MAX_CONFIG_ITEM;}
  bool check(const char *fname);
};

#endif
```

```
#include <string.h>
#include <ctype.h>
#include <stdio.h>
#include <stdlib.h>
#include "../common/addrset.h"
```

```
#include "config-server.h"

ConfigServer::ConfigServer()
{
  m_list = new ConfigElement *[ConfigItem::MAX_CONFIG_ITEM]
  {
    new ConfigString(NULL),
    new ConfigAddr(0),
    new ConfigSet(0x000000000000, 0),
    new ConfigSet(0x000000000000, 0),
    new ConfigSet(0x0000000000000000, 0),
    new ConfigSet(0x0000000000000000, 0),
    new ConfigSize(1),
    new ConfigSize(0),
    new ConfigSize(0),
    new ConfigSize(0),
    new ConfigSize(1),
    new ConfigInt(60),
    new ConfigInt(60),
    new ConfigInt(60),
    new ConfigInt(60),
    new ConfigInt(3),
    new ConfigBool(false),
    new ConfigBool(false),
    new ConfigBool(false),
    new ConfigBool(true),
    new ConfigBool(true),
    new ConfigString(NULL),
    new ConfigString(NULL),
  };
  m_root_tag = "ServerConfig";
  m_array_tags = new const char*[ConfigItem::MAX_CONFIG_ITEM]
  {
    "InterfaceName",
    "SrcAddress",
    "UnicastAddressSet",
    "MulticastAddressSet",
    "Unicast64AddressSet",
    "Multicast64AddressSet",
    "MaxAssignedUnicast",
    "MaxAssignedMulticast",
    "MaxAssignedUnicast64",
    "MaxAssignedMulticast64",
    "MaxAssignedDefault",
    "UnicastLifetime",
    "MulticastLifetime",
    "Unicast64Lifetime",
    "Multicast64Lifetime",
    "ReserveLifetime",
    "RenewalActive",
    "DefaultMulticast",
```

```
57      "Default64bitSet",
58      "AutoassignObjectionActive",
59      "AlternateSetActive",
60      "NetworkId",
61      "VendorParameter",
62    };
63  }
64
65  ConfigServer::~ConfigServer()
66  {
67    for(int i=0; i<ConfigItem::MAX_CONFIG_ITEM; i++)
68      delete m_list[i];
69    delete m_list;
70    delete m_array_tags;
71  }
72
73  bool ConfigServer::check(const char *fname)
74  {
75    if(TO_STRING(get(ConfigItem::INTERFACE)) == NULL)
76    {
77      fprintf(stderr, "%s: Missing interface name\n", fname);
78      return false;
79    }
80    if(!TO_ADDR(get(ConfigItem::SRC_ADDR)))
81    {
82      fprintf(stderr, "%s: SrcAddress is required\n", fname);
83      return false;
84    }
85    if(TO_ADDRSET_PTR(get(ConfigItem::UNICAST_SET))->getSize() <= 0)
86    {
87      fprintf(stderr, "%s: Missing Unicast set for distribution\n", fname);
88      return false;
89    }
90    if(TO_ADDRSET_PTR(get(ConfigItem::UNICAST_SET))->getSize() < TO_UINT(get
      (ConfigItem::MAX_ADDR_UNICAST)))
91    {
92      fprintf(stderr, "%s: Incoherent size of MaxAssignedUnicast\n", fname);
93      return false;
94    }
95    if(TO_ADDRSET_PTR(get(ConfigItem::MULTICAST_SET))->getSize() < TO_UINT(
      get(ConfigItem::MAX_ADDR_MULTICAST)))
96    {
97      fprintf(stderr, "%s: Incoherent size of MaxAssignedMulticast\n", fname
      );
98      return false;
99    }
100   if(TO_ADDRSET_PTR(get(ConfigItem::UNICAST_64_SET))->getSize() < TO_UINT(
      get(ConfigItem::MAX_ADDR_UNICAST_64)))
101   {
102     fprintf(stderr, "%s: Incoherent size of MaxAssignedUnicast64\n", fname
      );
```

```
    return false;
  }
  if(TO_ADDRSET_PTR(get(ConfigItem::MULTICAST_64_SET))->getSize() <
   TO_UINT(get(ConfigItem::MAX_ADDR_MULTICAST_64)))
  {
    fprintf(stderr, "%s: Incoherent size of MaxAssignedMulticast64\n",
   fname);
    return false;
  }
  if(TO_BOOL(get(ConfigItem::DEFAULT_MULTICAST)))
  {
    if(TO_BOOL(get(ConfigItem::DEFAULT_64)))
    {
      if(TO_ADDRSET_PTR(get(ConfigItem::MULTICAST_64_SET))->getSize() <=
   0)
      {
        fprintf(stderr, "%s: Missing Multicast-64 set for distribution\n",
    fname);
        return false;
      }
      if(TO_ADDRSET_PTR(get(ConfigItem::MULTICAST_64_SET))->getSize() <
   TO_UINT(get(ConfigItem::DEFAULT_ADDR_OFFER)))
      {
        fprintf(stderr, "%s: Incoherent size of MaxAssignedDefault\n",
   fname);
        return false;
      }
    }
    else
    {
      if(TO_ADDRSET_PTR(get(ConfigItem::MULTICAST_SET))->getSize() <= 0)
      {
        fprintf(stderr, "%s: Missing Multicast set for distribution\n",
   fname);
        return false;
      }
      if(TO_ADDRSET_PTR(get(ConfigItem::MULTICAST_SET))->getSize() <
   TO_UINT(get(ConfigItem::DEFAULT_ADDR_OFFER)))
      {
        fprintf(stderr, "%s: Incoherent size of MaxAssignedDefault\n",
   fname);
        return false;
      }
    }
  }
  else
  {
    if(TO_BOOL(get(ConfigItem::DEFAULT_64)))
    {
      if(TO_ADDRSET_PTR(get(ConfigItem::UNICAST_64_SET))->getSize() <= 0)
      {
```

```
145         fprintf(stderr, "%s: Missing Unicast-64 set for distribution\n",
    fname);
146         return false;
147       }
148       if(TO_ADDRSET_PTR(get(ConfigItem::UNICAST_64_SET))->getSize() <
    TO_UINT(get(ConfigItem::DEFAULT_ADDR_OFFER)))
149       {
150         fprintf(stderr, "%s: Incoherent size of MaxAssignedDefault\n",
    fname);
151         return false;
152       }
153     }
154     else
155     {
156       if(TO_ADDRSET_PTR(get(ConfigItem::UNICAST_SET))->getSize() < TO_UINT
    (get(ConfigItem::DEFAULT_ADDR_OFFER)))
157       {
158         fprintf(stderr, "%s: Incoherent size of MaxAssignedDefault\n",
    fname);
159         return false;
160       }
161     }
162   }
163   return true;
164 }
```