Escola d'Enginyeria de Telecomunicació i Aeroespacial de Castelldefels
UNIVERSITAT POLITÈCNICA DE CATALUNYA

MASTER THESIS

TITLE: Study of the protocol for home automation Thread.

MASTER DEGREE: Telecommunications Engineering
Master in Science in Telecommunication Engineering & Management.

AUTHOR: Humberto Gonzalez Gonzalez

DIRECTOR: Rafael Vidal Ferré
Lluís Casals Ibáñez

DATE: February, 22nd 2017

Títol: Study of the protocol for home automation Thread.

Autor: Humberto Gonzalez Gonzalez

Director: Rafael Vidal Ferré
Lluís Casals Ibáñez

Data: 22 de febrer del 2017

Resum

L'Internet de les coses (Internet of Things) és el major desafiament i oportunitat actualment a internet. Una aplicació de l'IoT és un sistema d'automatització de la llar. La idea consisteix en dispositius IP encastats connectats a Internet i utilitzant IPv6. L'IETF defineix 6LoWPAN com una tècnica per aplicar IPv6 a IEEE 802.15.4, estàndard de xarxes sense fils de baix consum. Les tecnologies sense fils d'automatització de la llar i productes existents al mercat no compleixen amb els requisits de baixa potència, resiliència, basats en IP, seguretat i ús amigable. Amb l'objectiu d'avançar en aquesta direcció Thread és un protocol de xarxa de malla simplificada basada en IPv6 i desenvolupada per a la comunicació eficient entre dispositius a la llar. Es connecta a Internet i proporciona una interfície senzilla i robusta al núvol. La pila de Thread és un estàndard "royalty-free" però de pagament. Amb l'objectiu de fer-lo àmpliament disponible als desenvolupadors i que comparteixin els seus coneixements tècnics, Nest va alliberar una implementació lliure de Thread anomenada OpenThread al maig de 2016. Així feia disponible la tecnologia utilitzada en els seus productes, tot buscant accelerar el desenvolupament de nous dispositius per a la llar connectada. L'objectiu del present estudi és analitzar aquesta nova tecnologia, centrant-se en la seva implementació OpenThread i fer algunes proves en una plataforma profunditzant en la mesura del possible en la part de routing. L'objectiu de la fase d'implementació és tenir un entorn de treball llest per provar i validar algunes de les funcionalitats de Thread, en especial els paràmetres d'encaminament i els canvis de topologia.

En la fase d'implementació s'ha utilitzant un entorn de proves. Es tracta de la configuració de l'escenari de proves tant a nivell de programari com de maquinari. Això ens ha permès desenvolupar algunes proves com la connectivitat bàsica entre dispositius, el control de la visibilitat dels nodes i la revisió de les taules d'encaminament. Finalment, a través de diferents captures hem analitzat el comportament després d'alguns canvis a l'escenari. La realització d'aquestes proves ha estat molt complexa a causa de la falta d'informació que ens han impedit fer proves més avançades, especialment les relacionades amb l'encaminament, i la comparació amb altres protocols. En canvi hem estat capaços de realitzar algunes proves bàsiques com la verificació de connectivitat, revisar les taules d'encaminament, etc. Amb les funcionalitats implementades actualment amb OpenThread el següent pas podria ser estudiar més a fons l'encaminament així com capes superiors, com ara la posada en funcionament el rol de commissioning utilitzant un router frontera, permetent interactuar des de fora de la xarxa de Thread.

Title: Study of the protocol for home automation Thread.

Author: : Humberto Gonzalez Gonzalez

Director: Rafael Vidal Ferré
Lluís Casals Ibáñez

Date: February, 22nd 2017

Overview

The Internet of Things is the biggest challenge and opportunity for the Internet today. An application of the IoT is a home automation system. The idea consists of IP-enabled embedded devices connected to the Internet using IPv6. The IETF added to this idea by defining 6LoWPAN as a technique to apply IPv6 to IEEE 802.15.4, a low-power wireless network standard. The existing home automation wireless technologies and products in the market do not meet the requirements of low power, resilience, IP-based, security and friendly use. With the goal of advance in this direction Thread is a simplified, IPv6-based mesh networking protocol developed for efficient communication between devices around the home. It connects to the internet and provides simple yet robust interface to the cloud. Thread stack is royalty-free but closed-documentation (payment). With the aim to make broadly available this to developers sharing their know-how, Nest released an open-source implementation of Thread protocol named OpenThread on May 2016 to make the technology used in its products available and accelerate the development of new devices for the connected home.The objective of the present study is to analyze this brand new coming technology, focusing on the released implementation and make some tests in a real platform describing where possible the routing details. The goal of the implementation phase is to have a working bench ready to test and validate some of the Thread functionalities specially the routing parameters and changes of topology.

The implementation phase has been done using a hardware testbed. We address the configuration of the test scenario on both hardware and software levels. This allow us to develop some tests such as the basic connectivity between devices, checking the visibility of the nodes and the revision of the routing tables. Finally, through different captures, we will analyze the behavior after some changes of scenario. Performing these tests was very complex due to the lack of information that prevent us of making more advanced tests, especially those related to the routing and comparison with other protocols. Instead, we were able to perform some basic tests such as check connectivity, see routing tables, etc. With the current OpenThread implemented functionalities the next step could be study more thoroughly the part of the routing, as well as study upper layers such as commissioning roles in a Border Router, allowing to interact from outside the Thread network.

INDEX

LIST OF FIGURES

INTRODUCTION

The Internet of Things (IoT) is the biggest challenge and opportunity for the Internet today. An interesting example of application of the IoT is a home automation system. Using a home automation process in a household environment, we can give additional functionality through the integration of sensors and actuators to normally non-automated systems like lighting, heating, air conditioning and appliances. Recently, home automation systems have been challenged with the need for high interoperability between home devices and for accessing the system from different end points.

The idea consists of IP-enabled embedded devices connected to the Internet using IPv6. The IETF added to this idea by defining 6LoWPAN as a technique to apply IPv6 to IEEE 802.15.4, a low-power wireless network standard, which adds the potential for transparent end-to-end communication, control and monitoring of home automation devices from anywhere on the globe. The use of 6LoWPAN technology also helps lowers expense and decreases complexity of home automation architecture.

The existing home automation wireless technologies and products in the market such as Z-Wave [1] or EnOcean [2] do not meet the requirements of low power, resilience, IP-based, security and friendly use. In that context, Thread comes to the market. Thread aims to build a technology that uses and combines the best of the current systems and creates a networking protocol that can help to develop new products that accomplishes this. Actually a version of Thread is shipping in Nest [3] products **(Fig 1.1)**.

Fig. 0.1 Nest Products: Thermostat, Protect and Cam. [3]

Thread [4] is a simplified, IPv6-based mesh networking protocol developed for efficient communication between devices around the home. It connects to the Internet and provides simple yet robust interface to the cloud. Thread stack is a standard for reliable, cost-effective, low-power, wireless D2D (device-to-device) mesh communication. It is designed specifically for Connected Home applications where IP-based networking is desired and a variety of application layers can be used on the stack. The main advantage is that it can run on existing 802.15.4 wireless SoCs using the 2.4 GHz ISM unlicensed band. Thread is simple to install, highly secure, and scalable to hundreds of devices.

Thread, like most other new IoT standards is centered around a written specification. With the aim to make broadly available this to developers sharing their know-how, Nest released OpenThread [5] on May 2016 to make the technology used in Nest products available and accelerate the development of new devices for the connected home.

The objective of the present study is to analyze this brand new coming technology, focusing on the released implementation and make some tests in a real platform describing where possible the routing details. The first chapter describes the Thread stack architecture explaining the protocols and existing technologies that are based on continuing with the Thread stack itself. Moreover, it approaches the different detail levels that provide a general overview of the Thread Architecture. The second chapter describes the OpenThread implementation released by Nest focusing on how some of the layers have been implemented. Is in the third chapter where we describe the hardware test environment deployed to test the main parts of the OpenThread. The fourth chapter is properly the implementation and test in the chosen scenario focusing on the network and routing parts, using captured frames as examples of implementation. The fifth and last chapter show the general conclusions and improvement proposals to the work done in this study.

CHAPTER 1. THREAD

In this chapter we introduce the Thread networking protocol, the beginnings of Thread, the implementation and the technical details needed to understand the following chapters.

Thread is an IPv6-based, closed-documentation (paid membership required for access to specifications) royalty-free networking protocol for Internet of Things (IoT). In July 2014, the "Thread Group" alliance **(Fig.1.1)** was announced by the working group with the companies Nest Labs (a subsidiary of Alphabet/Google), Samsung, ARM Holdings, BigAss Fans, NXP Semiconductors/Freescale, Silicon Labs and Yale in an attempt to have Thread become the industry standard by providing Thread certification for products. Thread is designed to connect products in and around the home into low-power, wireless mesh networks.

Fig. 1.1 Founders of the Thread Group alliance [4].

Thread is an IPv6 networking protocol built on open standards, millions of existing 802.15.4 wireless devices on the market can be easily updated to run Thread. Existing popular application protocols and IoT platforms like Nest Weave and ZigBee can run over Thread networks to deliver interoperable, end-to-end connectivity

Thread uses 6LoWPAN, which in turn uses the IEEE 802.15.4 wireless protocol with mesh communication, as does ZigBee and other systems. Thread however is IP-addressable, with cloud access and AES encryption. It currently supports up to 250 devices in one local network mesh.

Since opening membership in October 2014, the Thread Group has grown to more than 230 members.

At this time there are only 4 products certified **(Fig. 1.2)**.

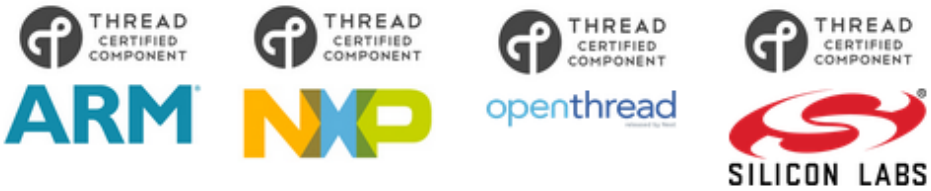

Fig. 1.2 Thread Certified Product Logos [4].

- **ARM mbed OS (NXP FRDM-K64F + Atmel ATZB-RF-233):** ARM mbed OS is an open source embedded operating system designed specifically to facilitate the creation and deployment of commercial, standards-based IoT solutions at scale. mbed OS features full support for Thread to simplify development of secure IoT applications in the home and to ease Thread product certification.

- **NXP Kinetis Thread Stack (KW2xD):** NXP's Kinetis Thread Stack is a complete, robust and scalable certified stack, architected and tested to meet the most demanding product requirements including very low power end nodes, large Thread networks and gateway solutions. The stack is available across multiple NXP microcontrollers and easily connects to host processors to create Thread Border Router solutions.

- **OpenThread (TI CC2538):** OpenThread, released by Nest and supported by Google, is an open-source implementation of the Thread networking protocol. It is a highly portable library that is OS and platform agnostic with a radio abstraction layer that is supported on multiple platforms.

- **Silicon Labs Thread stack (EM35x):** The Silicon Labs Thread stack is a robust implementation of the Thread 1.1 protocols suitable for field deployment and certified on the EM35x platform.

There are over 30 products submitted and awaiting Thread certification. In addition to Nest products, a number of devices – including the OnHub, a router from Google are shipping with Thread-ready radios.

Thread is built upon a foundation of existing standards **(Fig. 1.3)** and in next sections we are going to describe all the main characteristics from a layer level perspective.

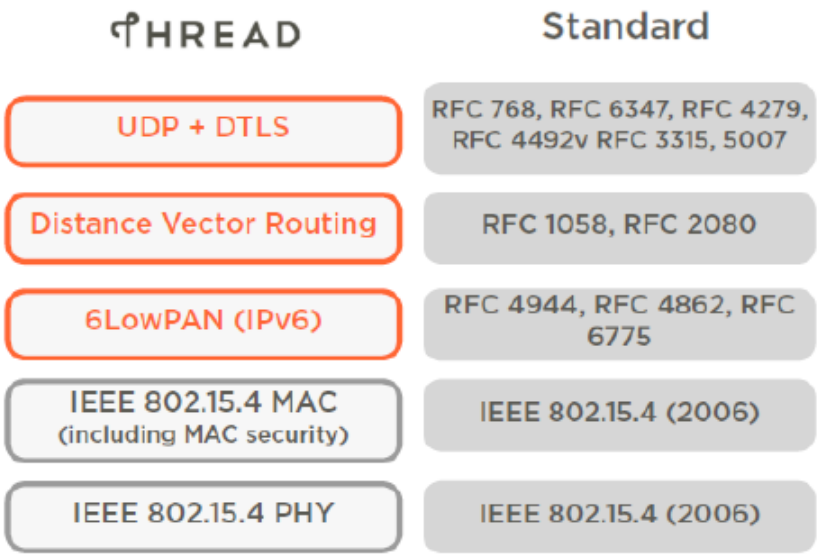

Fig. 1.3 Standards based Thread stack [4].

1.1. Technical Overview

These are the general characteristics of the Thread stack and network:

- **Supports IPv6 addresses and simple IP bridging:** Simple network installation, start up and operation: The simple protocols for forming, joining, and maintaining Thread Networks allow systems to self-configure and fix routing problems as they occur.

- **Secure:** Devices do not join the Thread Network unless authorized and all communications are encrypted and secure. Offers simplified security and commissioning.

- **Small and large networks:** Can support networks of 250 nodes or greater. The network layer is designed to optimize the network operation based on the expected use.

- **Range:** Typical devices in conjunction with mesh networking provide sufficient range to cover a normal home. Spread spectrum technology is used at the physical layer to provide good immunity to interference.

- **No single point of failure:** The stack is designed to provide secure and reliable operations even with the failure or loss of individual devices. Allow the use of multiple gateways.

- **Intended for control and automation**: Supports low latency (less than 100 milliseconds) applications.

- **Low power:** Is optimized for low-power/battery-backed operation. Host devices can typically operate for several years on AA type batteries using suitable duty cycles.

Table 1.1. Overview of Thread Technical specifications.

Specification	**Thread Data**
Design Focus	Home connectivity to IoT
Network Type	Mesh
Network	Thread
Distance	Normally 20-30 meters
Max Nodes Connected	250
Operating Band	2.4GHz (The ISM unlicensed band)
Spread Spectrum	Radio uses direct sequence spread spectrum (DSSS)
Throughput	Radio operates at 250 kbps
Data	Monitoring and control data
Voice Capable	No
Security	Banking-class, public-key cryptography
Modulation	Radio specification is O-QPSK modulation

Thread stack **(Fig.1.4)** only defines the Network and Transport layer but relies on 802.15.4 for the Physical and MAC layer. Applications (like Zigbee, etc) can run above Thread.

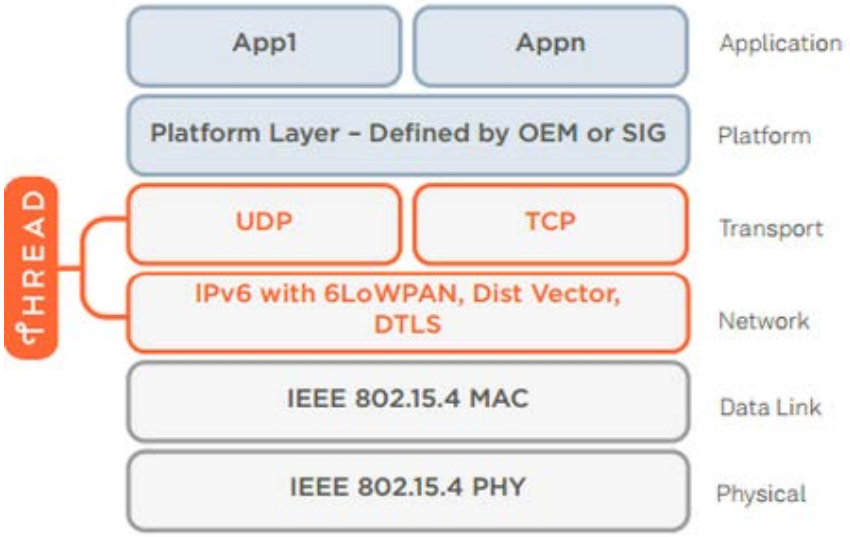

Fig. 1.4 Thread Stack Layer Levels [4].

1.2. Thread Network Architecture

Users communicate with a Thread network **(Fig. 1.5)** from their own device (smartphone, tablet, or computer) via Wi-Fi on their Home Area Network (HAN) or using a cloud-based application.

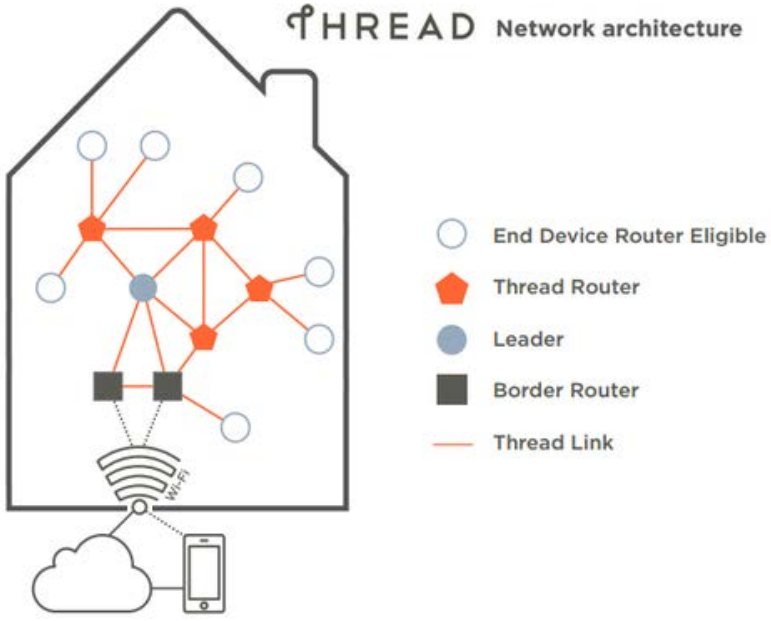

Fig. 1.5 Thread Network.

The following device types are included in a Thread network, starting from the Wi-Fi network:

- **Border Router:** Border Routers (can be more than one) provide connectivity from the IEEE 802.15.4 network to adjacent networks on other physical layers. Provide services for devices within this network, including routing services for off network operations. Border routers function as simplified gateways, handling connections between a Thread network and a non-Thread network and they handle IEEE 802.15.4 and a Wi-Fi (IEEE 802.11) or an Ethernet (IEEE 802.3) connection. This can be separate communication equipment like an access point or a host within the Thread network that incorporates both a Thread and Wi-Fi interface.

- **Leader:** A Leader manages a registry of assigned router IDs and accepts requests from router-eligible end devices (REEDs) to become routers. The Leader decides which should be routers, and the Leader, like all routers in a Thread network, can also have device-end children. The Leader also assigns and manages router addresses using Constrained Application Protocol (CoAP). However, all information contained in the Leader is present in the other Thread Routers. So, if the Leader fails or loses connectivity with the Thread network, another Thread Router is elected, and takes over as Leader without user intervention.

- **Thread Router:** Thread Routers provide routing services to network devices. They are also responsible for handling devices joining the network and providing security; they can function in the leader role and start a Thread network. Always active, they maintain neighbor, child and routing tables and connect with each other so the mesh remains intact. If a router goes down, the remaining Thread routers update their routing information so messages can still be forwarded using existing nodes. Thread Routers are not designed to sleep and can downgrade their functionality and become REEDs. Thread's mesh topology provides a self-healing communication framework

- **Router Eligible End Device**: Routers can be also be downgraded to router-eligible end devices (REEDs) which do not forward messages if conditions require; REEDs and routing algorithms are managed by the Thread network without user intervention. REEDs can become a Thread Router or a Leader, but not necessarily a Border Router that has special properties, such as multiple interfaces. REEDs do not relay messages or provide joining or security services for other devices in the network. The network manages and promotes router-eligible devices to routers if necessary without user interaction.

- **End Devices:** End devices that are not router-eligible can be either full end devices (FEDs) or minimal end devices (MEDs). MEDs do not need to explicitly synchronize with their parent to communicate. Sleepy end devices communicate only through their Thread Router parent and cannot relay messages for other devices.

Host devices are the individual IP-enabled functional equipment, like a light bulb, fan or thermostat, and are endpoints in the network. A device may also be referred to as a sleepy node or sleepy child. The sleepy terminology indicates that the devices spends most of the time in sleep mode, with low duty cycles as required for low power operation. Devices communicate only through the parent router and do not forward messages for other devices. Since devices cannot receive data while in a sleep mode, the parent holds their messages until the device wakes to either poll for data or to send data. A typical send cycle for a device might be:

- Wake from sleep mode.
- Perform any required startup and radio initialization.
- Go into receive mode and check if clear to transmit.
- Go into transmit mode.
- Transmit data.
- Get acknowledgment as applicable.
- Sleep.

1.3. The Thread Advantage

The Thread mesh networks will be built on top of IEEE 802.15.4 hardware, which is already on the market in some Nest devices and is what ZigBee, a previously competing wireless protocol for home automation, used as well. However, ZigBee will also be built on top of Thread at the application layer, so in the future it should become more of a complementary product than a competitor.

Thread uses the 6LoWPAN protocol, on top of which it builds the mesh network. That means the Thread networks are IP-based, and the devices can also connect directly to the Internet, not just to each other. This is an advantage the protocol has over other wireless protocols such as Bluetooth and ZigBee. It's also likely the main reason why ZigBee will be built on top of Thread in the future.

Compared to competitors, a Thread mesh network is also more resilient and can extend the range of a home network of smart devices. If one node fails to connect to the network, other devices in the network will still be able to connect to each other. This is the advantage of a mesh network over more centralized approaches. Bluetooth, for instance, has a typical effective range of 50 meters, and the devices can only connect in pairs of two. This limits the usefulness of Bluetooth in smart homes because it makes managing multiple smart devices much more difficult. On the other hand, a Thread mesh network requires no maintenance after the initial setup for the new devices that join it.

Over 250 devices can be connected to each other in a Thread mesh network as long as any two of those devices have a reasonable range between them. Therefore, a Thread network could easily cover a large house or property without any signal loss.

Where two Thread nodes are too far away from each other, the range can be extended with "Thread routers."
The Thread networks will have a bandwidth of about 250 Kbps, which isn't enough to transfer large files between the devices, but it can still enable the type of communications sensors can have with each other. The low bandwidth is a compromise that had to be made to keep the Thread-enabled devices low-power and last not days, but years, on small batteries. The latency is less than 100ms for typical interactions.

A risk of IoT is to make hacking exponentially more common, once many people begin buying insecure smart devices for their homes. That's why Thread comes with built-in security that's enabled by default and mandatory for all devices. Users will have to authorize any Thread-enabled devices before they are allowed on their home networks. To communicate with each other, the devices will also have to recognize each other's' MAC addresses, which should make it harder for other unauthorized devices to access the network. The communications between authorized devices will be encrypted with DTLS (Datagram Transport Layer Security), an encryption protocol designed to prevent tampering and message forgery.

1.4. Physical Layer and Data Link Layer- IEEE 802.15.4

Thread stack PHY and MAC layer use the IEEE 802.15.4-2006 version of the specification. The IEEE 802.15.4 standard was developed by the 802.15.4 Task Group within the IEEE and defines the physical layer (PHY) and medium access control (MAC) layer specifications for low data rate wireless personal area networks (LRWPANS). Such networks are typically limited to a personal operating space of up to 10 meters and involve little or no infrastructure. The standard provides for low complexity, low power consumption, low data rate wireless connectivity among a wide range of inexpensive devices. Among others, wireless sensor networks seem to be a suitable application scenario for 802.15.4 networks.

Designed with low power in mind, this wireless communication protocol is suitable for applications usually involving a large number of nodes, most of which can function on battery power for many years. The main identifying feature of IEEE 802.15.4 among WPANs is the importance of achieving extremely low manufacturing and operation costs and technological simplicity, without sacrificing flexibility or generality.

One of the characteristics derived from the need for low power and limiting the BER (Bit Error Rate) is enforcing smaller sized packets to be sent over-the-air. These can be up to a maximum of 127 bytes at the PHY layer.

The MAC layer payload can be as low as 88 bytes, depending on the security options and addressing type as illustrated in next Figure **(Fig. 1.6)**

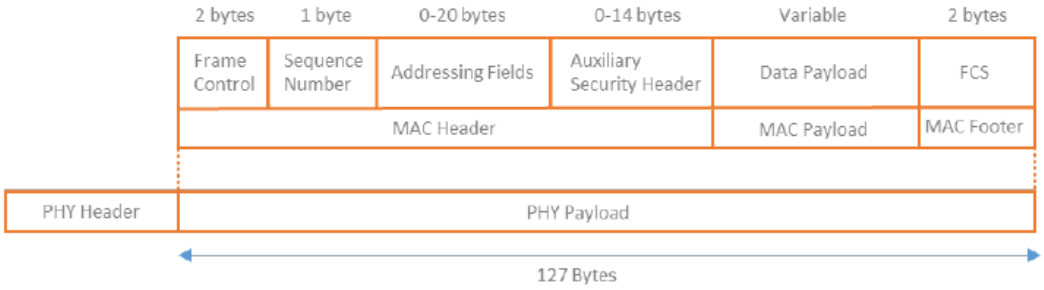

Fig. 1.6 PHY and MAC Frame details [12].

1.4.1. Physical Link Layer

IEEE 802.15.4 PHY provides the interface between the MAC and the physical radio channel. The transceiver hardware and its firmware are accessed over the PHY interfaces. Management functions are consolidated in the PHY Management Entity (PLME), an entity that provides the necessary layer management functions. The PLME is also responsible for the management of important parameters, objects and other manageable information of the PHY. These information are concentrated in the PAN Information Base (PIB). Important PIB attributes for the Frequency Band used by Thread (2400 - 2483.5MHz) are:

- Chip Rate 1600 kchip/s.
- Modulation O-QPSK. DSSS (Direct Sequence Spread Spectrum)
- Bit Rate 250 kbit/s.
- Symbol Rate 62.5 ksymbol/s.
- Channels: 16

The PHY also supports important functions like Energy Detection, Link Quality Indicator or Clear Channel Assessment. Each packet, or PHY protocol data unit (PPDU), contains a preamble, a start of packet delimiter, a packet length, and a payload field, or PHY service data unit. The 32-bit preamble is designed for acquisition of symbol and chip timing.

The IEEE 802.15.4 payload length can vary from 2 to 127 bytes and PHY Header is composed by 4 Bytes Preamble + 1 Byte Start Packet Delimiter + 1 Byte for Length Field.

There are also parameters that provides the receiver of a physical frame:

- **RSSI (Radio Signal Strength Indication**): Indicate the level of the frame received signal.

- **LQI (Link Quality Indication)**: Indicates the corresponding signal quality the frame received. Value is calculated from the intensity of received signal and the number of errors of the received frame. According IEEE 802.15.4 specification, the minimum and maximum values of LQI are 0x00 to 0xFF. The LQI provides information useful to the upper layers, for example to route.

1.4.2. Data Link Layer

The IEEE 802.15.4 MAC layer is used for basic message handling and congestion control. This MAC layer includes a CSMA (Carrier Sense Multiple Access) mechanism for devices to listen for a clear channel, as well as a link layer to handle retries and acknowledgement of messages for reliable communications between adjacent devices. MAC layer encryption and integrity protection is used on messages based on keys established and configured by the higher layers of the software stack. The network layer builds on these underlying mechanisms to provide reliable end-to-end communications in the network.

IEEE 802.15.4 MAC layer supports two modes:

- **Non Beacon Mode:** In this mode, the data is transmitted using unslotted CSMA/CA algorithm. This mode does not use the superframe structure. The merits are scalability and self-organization. However it cannot guarantee the timely delivery of data.

- **Beacon Mode:** In this mode the PAN coordinator generates beacons periodically to synchronize the nodes associated with it. The superframe structure is confined within two beacons

1.5. Network Layer

1.5.1. IPv6

Internet Protocol version 6 (IPv6) is the most recent version of the Internet Protocol (IP). Provides an identification and location system for computers on networks and routes traffic across the Internet. All devices have IPv6 address plus short address on HAN, DHCPv6 used for router address assignment. The main advantage of IPv6 is Home Network can directly address devices through Border Routers and Cloud Services can address devices of local devices on HAN or off network using normal IP addressing.

In IPv6 addressing architecture the devices support one or more ULA (Unique Local Address) or GUA (Global Unicast Address) addresses based on their available resources.
The high-order bits of an IPv6 address specify the network and the rest specify particular addresses in that network. Thus, all the addresses in one network have the same first N bits. Those first N bits are called the "prefix". The "/64" indicates that this is an address with a 64-bit prefix. The device starting the network picks a /64 prefix that is then used throughout the network.

Devices in the Thread stack support IPv6 addressing architecture. Devices configure one or more ULAs (Unique Local Address) or GUA (Global Unicast Address) addresses.

The device starting the network picks a /64 prefix that is then used throughout the Thread Network. The prefix is a Locally Assigned Global ID, often known as a ULA prefix and can be referred to as the mesh local ULA prefix. The Thread Network may also have one or more Border Routers that each may or may not have a prefix that can then be used to generate additional GUAs. The device in the Thread Network uses its Extended MAC address to derive its interface identifier and from this configures a link local IPv6 address with the well-known local prefix FE80::0/64.

The devices also support appropriate multicast addresses. This includes link-local all node multicast, link-local all-router multicast, and realm-local multicast.

Each device joining the Thread Network is assigned a 16-bit short address. For Routers, this address is assigned using the high bits in the address field with the lower bits set to 0, indicating a Router address. Children are then allocated a 16-bit short address using their Parent's high bits and the appropriate lower bits for their address. This allows any other device in the Thread Network to understand the Child's routing location simply by using the high bits of its address field.

1.5.2. 6LowPAN

The 6LoWPAN group has defined encapsulation and header compression mechanisms that allow IPv6 packets to be sent and received over IEEE 802.15.4 based networks.

6LoWPAN stands for "IPv6 Over Low Power Wireless Personal Networks". It is designed specifically to handle the limitations when sending and receiving IPv6 packets over IEEE 802.15.4 links. In doing so, it has to accommodate for the IEEE 802.15.4 maximum frame size that can be sent over-the-air. In Ethernet links, a packet with the size of the IPv6 MTU (1280 bytes) can be easily sent as one frame over the link. In the case of IEEE 802.15.4, 6LoWPAN acts as an adaptation layer between the IPv6 networking layer and the IEEE 802.15.4 link layer. It solves the issue of transmitting an IPv6 MTU by fragmenting the IPv6 packet at the sender and reassembling it at the receiver. 6LoWPAN also provides a compression mechanism that reduces the IPv6 headers sizes sent over-the-air and thus reduces transmission overhead. The fewer bits are sent over-the-air, the less energy is consumed by the device. Thread makes full use of these mechanisms to efficiently transmit packets over the IEEE 802.15.4 network.

Another important feature of the 6LoWPAN layer is the ability to provide link layer packet forwarding. It provides a very efficient and low overhead mechanism for forwarding multi hop packets in a mesh network.

Thread uses IP layer routing with link layer packet forwarding. It makes use of the 6LoWPAN link layer forwarding capabilities to forward the packet by not having to send it up to the network layer. Thread makes full use of the ability of the MAC layer to provide addressing based on short addresses (16-bit length) to further reduce the information bits needed to be sent over-the-air to provide efficient packet forwarding. This saves processing cycles and improves power consumption at the same time while still using an IP based routing protocol.

Looking at the OSI (Open Systems Interconnection) model one can notice that the MAC is considered to be Layer 2 and the IPv6 Network is considered to be Layer 3. Being an adaptation layer, 6LoWPAN sits between these two and provides the necessary mechanisms and interfaces for them to interconnect. To sum up, the 6LoWPAN adaptation layer provides the following:

- **IPv6 packet encapsulation:** To accomplish the functionalities described above, the 6LoWPAN layer takes the IPv6 packets, wraps them using encapsulation headers, and then subsequently sends them over-the-air using the IEEE 802.15.4 MAC and PHY layers.

- **IPv6 packet fragmentation and reassembly:** To meet the IPv6-required MTU of at least 1280 bytes with the IEEE 802.15.4 layer offering at most 102 bytes of payload per frame, a fragmentation mechanism below the IP layer is specified using an optional Fragmentation Header before the actual IPv6 header. A fragmented packet is carried in frames containing the fragmentation header. A typical application payload size in an IEEE 802.15.4 packet using UDP (User Datagram Protocol) and DTLS (Datagram Transport Layer Security) is 63 bytes. 6LoWPAN provides a fragmentation and reassembly mechanisms to adapt IPv6 datagrams to these smaller IEEE 802.15.4 payloads. IPv6 packets that do not fit are split into fragments and sent over the air via 802.15.4. Not all fragments may be received in the correct order. However, 6LoWPAN only requires that all fragments are received and will reorder fragments during reassembly if needed.

- **IPv6 header compression:** To minimize the overhead of sending IPv6 messages in IEEE 802.15.4 frames, 6LoWPAN provides stateless compression mechanisms for both IPv6 and transport headers that take advantage of cross-layer redundancies between protocols such as source and destination addressing, payload length, traffic class and flow labels. . Thread utilizes IPHC (Improved Header Compression) and NHC (Next Header Compression). IPHC is used to compress the IPv6 header. NHC is used to compress the UDP header. Even though 81 octets are left in an IEEE 802.15.4 frame for IPv6, the IPv6 header alone is 40 octets long, leaving 41 octets for upper layers. In case UDP is used, which has a header of 8 octets, only 33 octets can be used for application data. Furthermore, mechanisms for compressing the IPv6 header from 40 down to 2 bytes and the UDP header from 8 down to 4 bytes, in the ideal case, are specified. To distinguish between a compressed and uncompressed header, a 1-byte dispatch value is prepended before the header.

- **Link layer packet forwarding:** Layer Two Forwarding: Thread uses IP routing to forward packets. The IP routing table is maintained with each destination and the next hop to it. The 6LoWPAN mesh header is used to do link level next hop forwarding based on the IP routing table information. Thread uses 6 LoWPAN mesh headers for next hop forwarding

  Support for mesh networking is provided by the optional Mesh Addressing and Broadcast Headers. The former carries the Originator and Final Destination link-layer addresses while the latter contains a sequence number used to detect duplicated frames. Both are carried at the beginning of the IEEE 802.15.4 payload.

6LoWPAN packets are constructed on the same principle as IPv6 packets and contain stacked headers for each added functionality. Each 6LoWPAN header is preceded by a dispatch value that identifies the type of header **(Fig. 1.7)**.

Fig. 1.7 General Format 6LoWPAN Packet [12].

Thread uses the following types of 6LoWPAN headers:

- Mesh Header (used for link layer forwarding).
- Fragmentation Header (used for fragmenting the IPv6 packet into several 6LoWPAN packets).
- Header Compression Header (used for IPv6 headers compression).

The 6LoWPAN specification mandates that if more than one header is present they must appear in the order mentioned above.

The following are examples of 6LoWPAN packets sent over the air:

In the following figure **(Fig. 1.8)**, the 6LoWPAN payload is composed of the compressed IPv6 header and the rest of the IPv6 payload

Fig. 1.8 6LoWPAN Packet with compressed IPv6 header + payload[12].

In the following figure **(Fig. 1.9)**, the 6LoWPAN payload contains the IPv6 header and part of the IPv6 payload

Fig. 1.9 6LoWPAN Packet with Mesh + Fragment + Compression headers [12].

The rest of the payload will be transmitted in subsequent packets per the format in the following figure **(Fig. 1.10)**.

Fig. 1.10 6LoWPAN Subsequent Fragment[12].

1.5.3. Distance Vector Router (RIP & RIPng)

Routing information protocol (RIP) began life as one of the earliest efforts in the field of dynamic routing protocols back in the 1970. Later in the 1990 the RIP version 2 enhance RIP with the original version becoming known as RIP version 1. Also in the mid-1990, the process of defining IPv6 was drawing toward completion at least for the original IPv6 standards the RIPv3 or RIPng.

The Routing Information Protocol (RIP), which is a distance vector based algorithm, is one of the first routing protocols implemented on TCP/IP. Information is sent through the network using UDP. Each router that uses this protocol has limited knowledge of the network around it. This simple protocol uses a hop count mechanism to find an optimal path for packet routing. A maximum number of 16 hops a reemployed to avoid routing loops. However, this parameter limits the size of the networks that this protocol can support. The popularity of this protocol is largely due to its simplicity and its easy configurability. However, its disadvantages include slow convergence times, and its scalability limitations. Therefore, this protocol works best for small scaled networks. Simple distance vector algorithm:

- Provides next-hop information about all router nodes.
- Highly compressed protocol format: one byte per destination.
- No reactive route discovery by devices.
- Child ID encodes parent router ID. Route is known when address is known.
- Point to point routes always available to every router.

RIPng uses hop count as a routing metric. RIPng is intended to allow routers to exchange information for computing routes through an IPv6-based network. Figure **(Fig. 1.11)** shows the message format. Adding IPv6 prefix and prefix length headers. RIPng is based on RIPv2 and has a maximum hop count of 15; uses split horizon, poison reverse, and other loop avoidance mechanisms, but is intended for IPv6. Route tag and prefix length for Next Hop is all 0. Metric will have 0xFF and Next Hop address must be local.

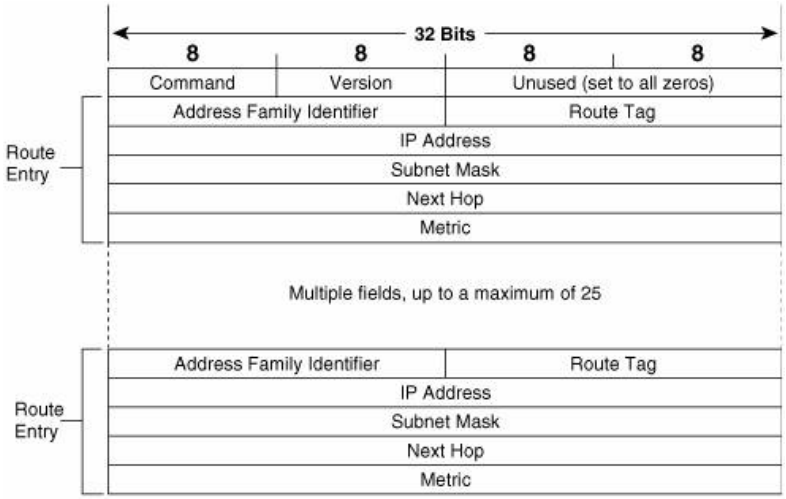

Fig. 1.11 RIPng Message Format.

It is of distance vector protocol and uses the Bellman Ford algorithm to calculate the best path in a network. It still uses multicast to send its updates but uses FF02::9 for the transport address. It´s counterpart IPv4 multicast address is 244.0.0.9. RIPng is a UDP-based protocol and communicates through UDP port 521 known as the RIPng port.

Next Figure **(Fig.1.12)** shows an IPv6 Rip internal table example from Cisco Router.

```
R1#show ipv6 rip RIPng database
RIP process "RIPng", local RIB
 1011:11:11:11::/64, metric 2
    Serial2/0/FE80::A8BB:CCFF:FE00:200, expires in 176 secs
 2001:150:2:2::2/128, metric 2, installed
    Serial2/0/FE80::A8BB:CCFF:FE00:200, expires in 176 secs
 2011::1/128, metric 2, installed
    Serial2/0/FE80::A8BB:CCFF:FE00:200, expires in 176 secs
 2020::1/128, metric 2, installed
    Serial2/0/FE80::A8BB:CCFF:FE00:200, expires in 176 secs
```

Fig. 1.12 IPv6 Rip internal table example from Cisco Router.

Mesh link establishment (MLE) is a protocol for establishing and configuring secure radio links in IEEE 802.15.4 radio mesh networks. MLE extends IEEE 802.15.4 for use in multihop mesh networks by adding three capabilities:

- Dynamically configuring and securing radio connections between neighboring devices.
- Enabling network-wide changes to shared radio parameters.
- Allowing the determination of radio link quality prior to configuration.

MLE operates below the routing layer, insulating it from the details of configuring, securing, and maintaining individual radio links within a larger mesh network.

As described before Thread Routing implementation is based on RIPng. RIP calculates the best route to a destination based solely on how many hops it is to the destination network, RIP tends to be inefficient in network using more than one LAN protocol. This is because RIP prefers paths with the shortest hop count. The path with the shortest hop count might be over the slowest link in the network. This protocol sends using multicast its routing table every 30 seconds.

A packet can contain up to 25 destinations, and the unit measure uses hop count (number of jumps), maximum is 15 routers.

Router Selection:

- Limit of 32 active routers to reduce bandwidth and RAM consumption.
- 64 router addresses to allow timing out and reassignment.
- No neighbor selection required.
- Routers select automatically from router eligible end devices (REED).
- REED behave as end devices, but listen to routing messages.
- As number of routers increases, routers can elect to become REED.
- If REED notes need to become a router it will petition leader.

Leader decision maker in network, chosen autonomously:

- Assigns router ID‟s
- Assigns 6LoWPAN contexts
- Collates border router information
- Assembled network data is distributed using MLE advertisements
- All routers store the network data, only the leader can make changes to it

No single point of failure. Recovery from leader failure or disconnected topology by self-election of new leader and network fragments automatically elect a new leader, and if reconnected the leader returns to being a router

1.5.4. DTLS

The Datagram Transport Layer Security (DTLS) communications protocol provides communications security for datagram protocols. DTLS allows datagram-based applications to communicate in a way that is designed to prevent eavesdropping, tampering, or message forgery. The DTLS protocol is based on the stream-oriented Transport Layer Security (TLS) protocol and is intended to provide similar security guarantees. The DTLS protocol datagram preserves the semantics of the underlying transport and the application does not suffer from the delays associated with stream protocols, but has to deal with

packet reordering, loss of datagram and data larger than the size of a datagram network packet.

There are two main areas that unreliability creates problems for TLS:

- The traffic encryption layer does not allow individual packets to be decrypted, there are two inter-record dependencies:
    - Cryptographic context is chained between records.
    - A Message Authentication Code (MAC) that includes a sequence number provides anti-replay and message reordering protection, but the sequence numbers are implicit in the records.

- The handshake layer breaks if messages are lost because it depends on them being transmitted reliably for these two reasons:
    - The handshake is a lockstep cryptographic handshake requiring messages to be transmitted and received in a defined order, causing a problem with potential reordering and message loss.
    - Fragmentation can be a problem because the handshake messages are potentially larger than any given datagram.

The first problem caused by the inter-packet dependencies can be solved by using a method employed in the Secure Internet Protocol (IPsec) by adding explicit state to each individual record.

To solve the issue of packet loss DTLS employs a simple retransmission timer. Figure **(Fig. 1.12)** below illustrates the basic concept. The client is expecting to see the *HelloVerifyRequest* message from the server. If the timer expires then the client knows that either the *ClientHello* or the *HelloVerifyRequest* was lost and retransmits.

Reordering is solved by giving each handshake message a specific sequence number used to determine if it has received the next message in the sequence. If the message is the next one then the peer processes it, if it is not the next one then it queues it up for future handling when message’s individual sequence number is reached.

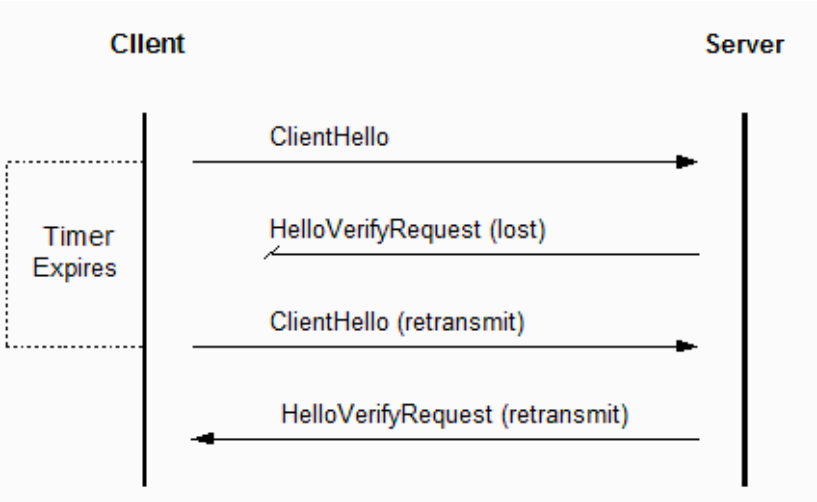

Fig. 1.13 DTLS Timer Basic Concept [6].

Handshake messages can be quite large (2^{24} – 1 bytes) and UDP datagrams are usually limited to less than 1500 bytes. DTLS compensates for this by allowing each handshake message to be fragmented over several UDP datagrams. Each handshake message contains a fragment offset and a fragment length allowing the recipient to reassemble the bytes into the complete message when all packets are received.

Optionally DTLS supports replay detection by maintaining a bitmap window of received records. Records that are too old to fit in the window and those that have been previously received are discarded. This is the same technique used by IPsec AH/ESP.

1.6. Transport Layer (UDP & TCP)

User Datagram Protocol (UDP) is defined to make available a datagram mode of packet-switched computer communication in the environment of an interconnected set of computer networks. This protocol assumes that the Internet Protocol (IP) is used as the underlying protocol.

This protocol provides a procedure for application programs to send messages to other programs with a minimum of protocol mechanism. The protocol is transaction oriented, and delivery and duplicate protection are not guaranteed. Applications requiring ordered reliable delivery of streams of data should use the Transmission Control Protocol (TCP)

The Thread stack supports UDP (User Datagram Protocol) as defined in RFC 768 for messaging between devices. TCP is an optional implementation and is impractical with unreliable networks and can be fails on sleepy platforms.

1.7. Application Layer

A standard definition of an application layer is an “abstraction layer that specifies the shared protocols and interface methods used by hosts in a communications network”. Put more simply, an application layer is the “language of devices,” for example, how a switch talks to a light bulb. The Thread specification defines standard methods for forming and joining a network (called commissioning) and custom applications are responsible for interoperability. Thread does, however, provide these basic application services:

- **UDP messaging:** UDP offers a way to send messages using a 16-bit port number and an IPv6 address. UDP is a simpler protocol than TCP and has less connection overhead (for example, UDP does not implement keep-alive messages). As a result, UDP enables a faster, higher throughput of messages and reduces the overall power budget of an application. UDP also has a smaller code space than TCP, which leaves more available flash on the chip for custom applications.
- **Multicast messaging:** Thread provides the ability to broadcast messages, that is, sending the same message to multiple nodes on a Thread network. Multicast allows a built-in way to talk to neighbor nodes, routers, and an entire Thread network with standard IPv6 addresses.

- **Application layers using IP services:** Thread allows the use of application layers such as UDP and CoAP (Constrained Application Protocol) to allow devices to communicate interactively over the Internet. Non-IP application layers will require some adaptation to work on Thread.

Silicon Labs has developed two sample applications–client and server–that demonstrate basic interoperability features of the Thread network and how to build a simple client and server example, including a sleepy end device.

CHAPTER 2. OPENTHREAD

Nest Labs, Inc. (acquired by Google in the beginning of 2014) released OpenThread in May 2016, OpenThread is an open source implementation based on the draft Thread 1.0 specification of the Thread networking protocol. With OpenThread, Nest wants to make the technology used in Nest products more broadly available to accelerate the development of products for the connected home. The idea is as more silicon providers adopt Thread, manufacturers will have the option of using a proven networking technology rather than creating their own, and consumers will have a growing selection of secure and reliable connected products to choose from.

Along with Nest, ARM, Atmel, Dialog Semiconductor, Qualcomm Technologies, Inc. and Texas Instruments Incorporated are contributing to the ongoing development of OpenThread **(Fig.2.1)**. In addition, OpenThread can run on Thread-capable radios and corresponding development kits from silicon providers like NXP Semiconductors and Silicon Labs.

Fig. 2.1 OpenThread supporters [5].

OpenThread implements all Thread networking layers including:

- IEEE 802.15.4 with MAC security.
- IPv6 and 6LoWPAN.
- Mesh Link Establishment and Mesh Routing.
- Key management.
- Definitions in code of specific roles in Thread including:
  - Leader.
  - Router.
  - End Device.
  - The Border router.
- UDP packet compression.
- A CoAP implementation.

OpenThread is highly portable: OS and platform agnostic with a radio abstraction layer. Is written mostly in C++. The implementation depends on a platform layer, basically a Hardware Abstraction Layer (HAL), and if that layer is implemented, it can potentially run on most microcontroller or 802.15.4 SoCs (essentially microcontrollers with an integrated 802.15.4 radio) with the advantage of small memory footprint.

All code are available at Github repository and can be run in a variety of Software platforms and SoCs development boards including:

- Dialog DA15000
- Nordic Semiconductor nRF52840
- Texas Instruments CC2538 & CC2650
- Zolertia RE-Mote
- Windows 10
- RIOT
- POSIX Emulation

Next sections explain some of the code snippets about implemented functions **(Fig. 2.2)**.

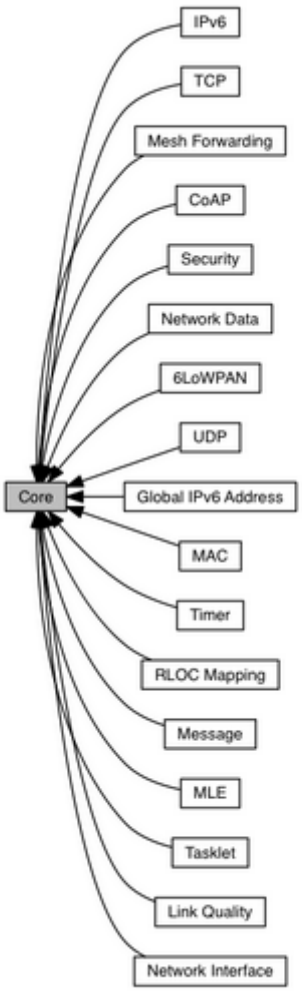

Fig. 2.2 General Overview of OpenThread modules [5].

.

2.1. Network Implementation

IPv6 includes definitions for the IPv6 network layer. IPV6 module **(Fig. 2.3)** defines the ICMPv6 implementation the network interfaces, the multicast protocol and the IPv6 implementation itself.

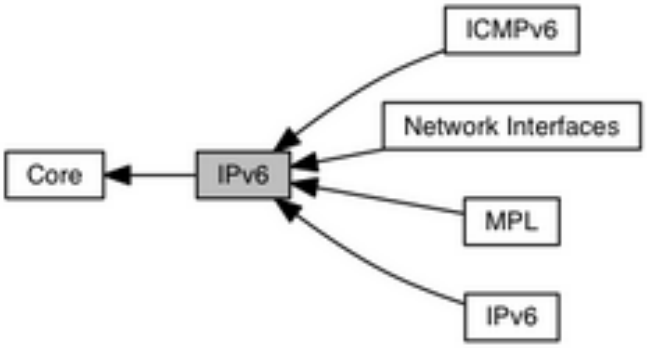

Fig. 2.3 IPv6 module details [5].

2.2. Routing Implementation

OpenThread implement MLE to propagate the Routing table information and RIPng to process information and maintain routing tables.

The implemented MLE module **(Fig. 2.4)** depends on Core and implements MLE functionality required for the Thread Router and Leader roles. The Type Length Value (TLV) module includes definitions for generating and processing MLE TLVs.

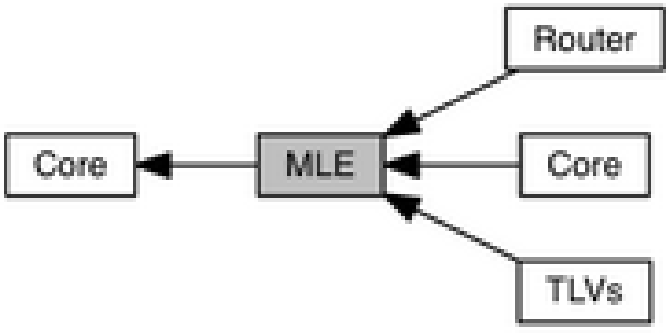

Fig. 2.4 MLE module details [5].

MLE main file is *src/core/thread/mle.cpp* and contains all the functions and references to implement MLE functionalities. The Routing parameters are defined in file *src/core/thread/mle_constants.hpp* **(Fig. 2.5)**.

```
kAdvertiseIntervalMin       = 1,              ///< ADVERTISEMENT_I_MIN (seconds)
kAdvertiseIntervalMax       = 32,             ///< ADVERTISEMENT_I_MAX (seconds)
kFailedRouterTransmissions  = 4,              ///< FAILED_ROUTER_TRANSMISSIONS
kRouterIdReuseDelay         = 100,            ///< ID_REUSE_DELAY (seconds)
kRouterIdSequencePeriod     = 10,             ///< ID_SEQUENCE_PERIOD (seconds)
kMaxNeighborAge             = 100,            ///< MAX_NEIGHBOR_AGE (seconds)
kMaxRouteCost               = 16,             ///< MAX_ROUTE_COST
kMaxRouterId                = 62,             ///< MAX_ROUTER_ID
kInvalidRouterId            = kMaxRouterId + 1, ///< Value indicating incorrect Router Id
kMaxRouters                 = 32,             ///< MAX_ROUTERS
kMinDowngradeNeighbors      = 7,              ///< MIN_DOWNGRADE_NEIGHBORS
kNetworkIdTimeout           = 120,            ///< NETWORK_ID_TIMEOUT (seconds)
kParentRouteToLeaderTimeout = 20,             ///< PARENT_ROUTE_TO_LEADER_TIMEOUT (seconds)
kRouterSelectionJitter      = 120,            ///< ROUTER_SELECTION_JITTER (seconds)
kRouterDowngradeThreshold   = 23,             ///< ROUTER_DOWNGRADE_THRESHOLD (routers)
kRouterUpgradeThreshold     = 16,             ///< ROUTER_UPGRADE_THRESHOLD (routers)
kMaxLeaderToRouterTimeout   = 90,             ///< INFINITE_COST_TIMEOUT (seconds)
kReedAdvertiseInterval      = 570,            ///< REED_ADVERTISEMENT_INTERVAL (seconds)
kReedAdvertiseJitter        = 60,             ///< REED_ADVERTISEMENT_JITTER (seconds)
kMleEndDeviceTimeout        = 240,            ///< MLE_END_DEVICE_TIMEOUT (seconds)
kLeaderWeight               = 64,             ///< Default leader weight for the Thread Network Partition
```

Fig. 2.5 Routing Parameters [5].

The command types are defined in *src/core/thread/mle.hpp***(Fig. 2.6)**.

```
kCommandLinkRequest          = 0,   ///< Link Reject
kCommandLinkAccept           = 1,   ///< Link Accept
kCommandLinkAcceptAndRequest = 2,   ///< Link Accept and Reject
kCommandLinkReject           = 3,   ///< Link Reject
kCommandAdvertisement        = 4,   ///< Advertisement
kCommandUpdate               = 5,   ///< Update
kCommandUpdateRequest        = 6,   ///< Update Request
kCommandDataRequest          = 7,   ///< Data Request
kCommandDataResponse         = 8,   ///< Data Response
kCommandParentRequest        = 9,   ///< Parent Request
kCommandParentResponse       = 10,  ///< Parent Response
kCommandChildIdRequest       = 11,  ///< Child ID Request
kCommandChildIdResponse      = 12,  ///< Child ID Response
kCommandChildUpdateRequest   = 13,  ///< Child Update Request
kCommandChildUpdateResponse  = 14,  ///< Child Update Response
kCommandAnnounce             = 15,  ///< Announce
kCommandDiscoveryRequest     = 16,  ///< Discovery Request
kCommandDiscoveryResponse    = 17,  ///< Discovery Response
```

Fig. 2.6 MLE command types [5].

IPV6 routing tables are implemented in the file *src/core/net/ip6_routes.cpp & src/core/net/ip6_routes.hpp* **(Fig. 2.7)**. and defines the next address scopes.

```
kNodeLocalScope      = 0,  ///< Node-Local scope
kInterfaceLocalScope = 1,  ///< Interface-Local scope
kLinkLocalScope      = 2,  ///< Link-Local scope
kRealmLocalScope     = 3,  ///< Realm-Local scope
kAdminLocalScope     = 4,  ///< Admin-Local scope
kSiteLocalScope      = 5,  ///< Site-Local scope
kOrgLocalScope       = 8,  ///< Organization-Local scope
kGlobalScope         = 14, ///< Global scope
```

Fig. 2.7 MLE scopes [5].

2.3. DTLS Implementation

DTLS class **(Fig. 2.X)** is part of the MeshCoP class and defines all the related messages to implement the DTLS functionalities in OpenThread stack

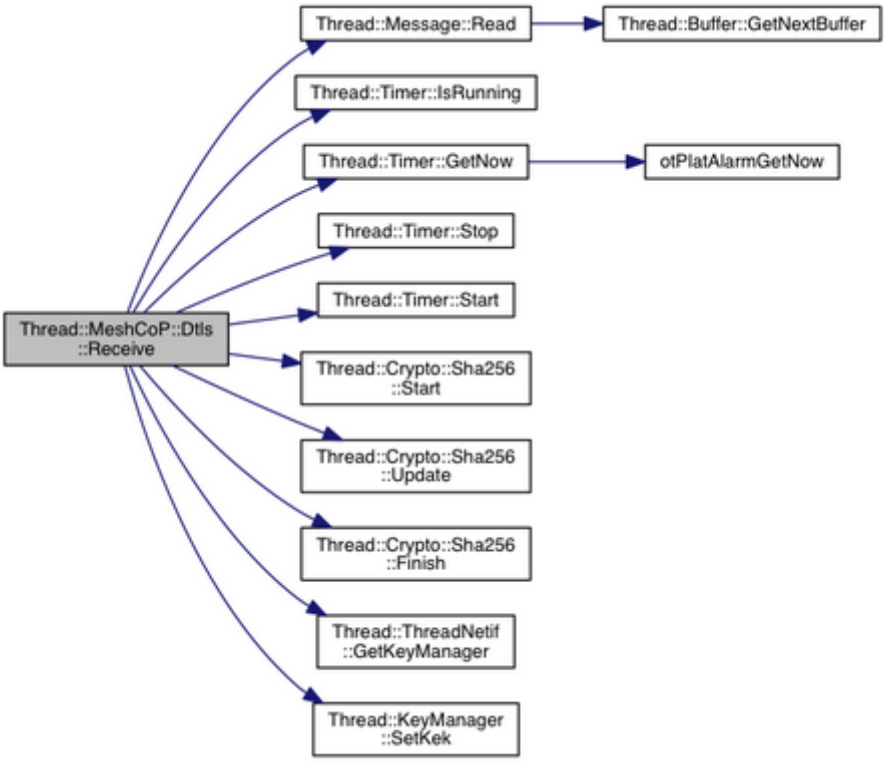

Fig. 2.8 DTLS module details [5].

2.4. Thread Management Protocol Implementation (CoaP)

While the Thread protocol uses CoAP for its control messaging, the CoAP server instances within a Thread network are dedicated to Thread's control messaging (i.e. Thread's CoAP server uses a different port than the default CoAP port).

At the same time, OpenThread does expose a CoAP API that allows application to use the same CoAP implementation to send/receive CoAP messages. If you want to test that out, you will need to set up a CoAP server, add a resource, and implement the appropriate higher-layer logic to send and process CoAP messages. The UDP port for Thread-specific is 61631.

Currently the CoAP server object in OpenThread only implements the subset of the protocol that is necessary for Thread Commissioning. OpenThread CoAP API is disabled by default. To enable that feature, you will need to add `--enable-application-coap` to the `./configure` options.

A border router can be created with a cc2538 running the ncp example and a linux machine running wpantund [15]. Wpantund (Userspace WPAN Network Daemon) is a user-space network interface driver/daemon that provides a native IPv6 network interface to a low-power wireless Network Co-Processor (or NCP). It was written and developed by Nest Labs to make supporting Thread connectivity on Unix-like operating systems more straightforward.

The OpenThread API supports sending and receiving UDP messages. You can use those APIs to send/receive CoAP messages. You could also look into RIOT-OS, which supports CoAP and has a recent PR to integrate OpenThread.

Joiner Entrust CoAP message and it shall be secured on MAC with Key ID Mode 0, using single-use KEK. But it still is a confirmable message, and CoAP may need to retransmit it. CoAP messages are transmitted over secured DTLS connection.

The CoAP Message Types can be either confirmable (CON), non-confirmable (NON). Confirmable messages require an ACK, while non-confirmable messages don’t. If we don’t need reliability, we use NON, for example, a sensor broadcasting data and if we need reliability, we use CON, for example, issuing a GET to a server.

2.5. CLI Commands

The OpenThread CLI exposes configuration and management APIs via a command line interface. Use the CLI to play with OpenThread, which can also be used with additional application code.

Some of the CLI available commands are:

- autostart
  - *autostart* → Show the status of automatically starting Thread.
  - *autostart true* → Automatically start Thread on initialization.
  - *autostart false* → Don't automatically start Thread on initialization.

- blacklist
  - *blacklist* → List the blacklist entries.
  - *blacklist enable* → Enable MAC blacklist filtering.
  - *blacklist disable* → Disable MAC blacklist filtering.
  - *blacklist add <extaddr>* → Add an address to the blacklist.
  - *blacklist remove <extaddr>* → Remove address from the blacklist.
  - *blacklist clear* → Clear all entries from the blacklist.

- channel
  - *channel* → Get the IEEE 802.15.4 Channel value.
  - *channel <channel>* → Set the IEEE 802.15.4 Channel value.

- child
  - *child list* → List attached Child IDs.
  - *child table* → Print table of attached children.

```
> child table
| ID  | RLOC16 | Timeout    | Age        | LQI In | C_VN |R|S|D|N| Extended MAC     |
+-----+--------+------------+------------+--------+------+-+-+-+-+------------------+
|   1 | 0xe001 |        240 |         44 |      3 |  237 |1|1|1|1| d28d7f875888fccb |
|   2 | 0xe002 |        240 |         27 |      3 |  237 |0|1|0|1| e2b3540590b0fd87 |
Done
```

  - *child <id>* → Print diagnostic information for an attached Thread Child. The `<id>` may be a Child ID or an RLOC16.

```
> child 1
Child ID: 1
Rloc: 9c01
Ext Addr: e2b3540590b0fd87
Mode: rsn
Net Data: 184
Timeout: 100
Age: 0
LQI: 3
RSSI: -20
Done
```

- childmax
  - *childmax* → Get the Thread maximum number of allowed children.
  - *childmax <count>* → Set the Thread maximum number of allowed children.

- childtimeout
  - *childtimeout* → Get the Thread Child Timeout value.
  - *childtimeout <timeout>* → Set the Thread Child Timeout value.

- Commissioner
  - *commissioner start <provisioningUrl>* → Start the Commissioner role.
  - *commissioner stop* → Stop the Commissioner role.
  - *commissioner joiner add <hashmacaddr> <psdk>* → Add a Joiner entry.
  - *commissioner joiner remove <hashmacaddr>* → Remove a Joiner entry.
  - *commissioner provisioningurl <provisioningUrl>* → Set the Provisioning URL.
  - *commissioner energy <mask> <count> <period> <scanDuration> <destination>* → Send a MGMT_ED_SCAN message.
  - *commissioner panid <panid> <mask> <destination>* → Send a MGMT_PANID_QUERY message.
  - *commissioner sessionid* →Get current commissioner session id.

- contextreusedelay
  - *contextreusedelay* → Get the CONTEXT_ID_REUSE_DELAY.
  - *contextreusedelay <delay>* → Set the CONTEXT_ID_REUSE_DELAY.

- counter
  - *counter* → Get the supported counter names.
  - *counter <countername>* → Get the counter value.

- delaytimermin
  - *delaytimermin* → Get the minimal delay timer.
  - *delaytimermin <delaytimermin>* →Set the minimal delay.

- discover
  - *discover [channel]* → Perform an MLE Discovery operation.

```
> discover
| J | Network Name     | Extended PAN     | PAN  | MAC Address      | Ch | dBm | LQI |
+---+------------------+------------------+------+------------------+----+-----+-----+
| 0 | OpenThread       | dead00beef00cafe | ffff | f1d92a82c8d8fe43 | 11 | -20 |   0 |
Done
```

- eui64
  - *eui64* → Get the factory-assigned IEEE EUI-64.

- extaddr
  - *extaddr* → Get the IEEE 802.15.4 Extended Address.
  - *extaddr <extaddr>* → Set the IEEE 802.15.4 Extended Address.

- extpanid
  - *extpanid* → Get the Thread Extended PAN ID value.
  - *extpanid <extpanid>* → Set the Thread Extended PAN ID value.

- factoryreset
  - *factoryreset* → Delete all stored settings, and signal a platform reset.

- hashmacaddr
  - *hashmacaddr* → Get the HashMac address.

- ifconfig
  - *ifconfig* →Show the status of the IPv6 interface.
  - *ifconfig up* →Bring up the IPv6 interface.
  - *ifconfig down* → Bring down the IPv6 interface.

- ipaddr
  - *ipaddr* → List all IPv6 addresses assigned to the Thread interface.

```
> ipaddr
fdde:ad00:beef:0:0:ff:fe00:0
fdde:ad00:beef:0:558:f56b:d688:799
fe80:0:0:0:f3d9:2a82:c8d8:fe43
Done
```

  - *ipaddr add <ipaddr>* → Add address to the Thread interface.
  - *ipaddr del <ipaddr>* → Delete address from the Thread interface.

- ipmaddr
  - *ipmaddr* → List all IPv6 multicast addresses.

```
> ipmaddr
ff05:0:0:0:0:0:0:1
ff33:40:fdde:ad00:beef:0:0:1
ff32:40:fdde:ad00:beef:0:0:1
Done
```

  - *ipmaddr add <ipaddr>* → Subscribe the Thread interface to the IPv6 multicast address.
  - *ipmaddr del <ipaddr>* → Unsubscribe the Thread interface to the IPv6 multicast address.

- joiner
  - *joiner start <pskd> <provisioningUrl>* → Start the Joiner role.
  - *joiner stop* → Stop the Joiner role.

- leaderpartitionid
  - *leaderpartitionid* → Get the Thread Leader Partition ID.
  - *leaderpartitionid <partitionid>* → Set the Thread Leader Partition ID.

- linkquality
  - *linkquality <extaddr>*
    → Get the link quality on the link to a given extended address.
  - *linkquality <extaddr> <linkquality>*
    → Set the link quality on the link to a given extended address.

- masterkey
  - *masterkey* →Get the Thread Master Key value.
  - *masterkey <key>* →Set the Thread Master Key value.

- mode
  - *mode* → Get the Thread Device Mode value.
  - *mode [rsdn]* → Set the Thread Device Mode value.

    - r: rx-on-when-idle
    - s: Secure IEEE 802.15.4 data requests
    - d: Full Function Device
    - n: Full Network Data

- netdataregister
  - *netdataregister* →Register local network data with Thread Leader.

- networkidtimeout
  - *networkidtimeout*
    → Get the NETWORK_ID_TIMEOUT used in the Router role.
  - *networkidtimeout <timeout>*
    → Set the NETWORK_ID_TIMEOUT used in the Router role.

- networkname
  - *networkname* →Get the Thread Network Name.
  - *networkname <name>* → Set the Thread Network Name.

- panid
  - *panid* → Get the IEEE 802.15.4 PAN ID value.
  - *panid <panid>* →Set the IEEE 802.15.4 PAN ID value.

- parent
  - *parent* → Get the information for a Thread Router as parent.

- ping
  - *ping <ipaddr> [size] [count] [interval]* → Send an ICMPv6 Echo Request.

- releaserouterid
  - *releaserouterid <routerid>*
    → Release a Router ID that has been allocated by the device in the Leader role.

- reset
  - *reset* → Signal a platform reset.

- rloc16
  - *rloc16* → Get the Thread RLOC16 value.

- route
  - *route remove <prefix>* → Invalidate a prefix in the Network Data.
  - *route add <prefix> [s] [prf]* → Add a valid prefix to the Network Data.
    - s: Stable flag.
    - prf: Default Router Preference, which may be: 'high', 'med', or 'low'.

- router
  - *router list* →List allocated Router IDs.
  - *router <id>* →Print diagnostic information for a Thread Router.
  - *router table* → Print table of routers.

```
> router table
| ID | RLOC16 | Next Hop | Path Cost | LQI In | LQI Out | Age | Extended MAC     |
+----+--------+----------+-----------+--------+---------+-----+------------------+
| 21 | 0x5400 |       21 |         0 |      3 |       3 |   5 | d28d7f875888fccb |
| 56 | 0xe000 |       56 |         0 |      0 |       0 | 182 | f2d92a82c8d8fe43 |
Done
```

- routerrole
  - *routerrole* → Indicates whether the router role is enabled or disabled.
  - *routerrole enable* →Enable the router role.
  - *routerrole disable* →Disable the router role.

- routerupgradethreshold
  - *routerupgradethreshold*
    → Get the ROUTER_UPGRADE_THRESHOLD value.
  - *routerupgradethreshold <threshold>*
    → Set the ROUTER_UPGRADE_THRESHOLD value.

- scan
  - *scan [channel]* → Perform an IEEE 802.15.4 Active Scan.

```
> scan
| J | Network Name     | Extended PAN     | PAN  | MAC Address      | Ch | dBm | LQI |
+---+------------------+------------------+------+------------------+----+-----+-----+
| 0 | OpenThread       | dead00beef00cafe | ffff | f1d92a82c8d8fe43 | 11 | -20 |   0 |
Done
```

- singleton
  - *singleton* → Return true when there are no other nodes in the network, otherwise return false.

- state
  - *state* → Return state of current state.
  - *state <mode>* → Try to switch to State:
    - detached
    - child
    - router
    - leader.

- thread
  - *thread start* → Enable Thread protocol operation and attach to a Thread network.
  - *thread stop* → Disable Thread protocol operation and detach from a Thread network.

- version
  - *version* →Print the build version information.

- whitelist
  - *whitelist* → List the whitelist entries.
  - *withelist enable* → Enable MAC withelist filtering.
  - *withelist disable* → Disable MAC withelist filtering.
  - *withelist add <extaddr>* → Add an address to the withelist.
  - *withelist remove <extaddr>* → Remove address from the withelist.
  - *withelist clear* → Clear all entries from the withelist.

CHAPTER 3. HARDWARE TEST ENVIRONMENT

The goal of the implementation phase is to have a working bench ready to test and validate some of the Thread functionalities specially the routing parameters and changes when propagation or node presence varies. The implementation phase has been done using the hardware testbed described below.

3.1. OpenMote Development System

OpenMote development Kit **(Fig 3.1)** is based on the TI CC2538 System on Chip (SoC), which combines an ARM Cortex-M3 with an IEEE 802.15.4 transceiver in one chip. The board follows the XBee form factor for easier extensibility, which is used to connect the core board to either the OpenBattery or Open-Base extension boards. It can run both as a battery-powered sensor board and as a border router, depending on what extension board it is attached to, e.g OpenBattery or OpenBase.

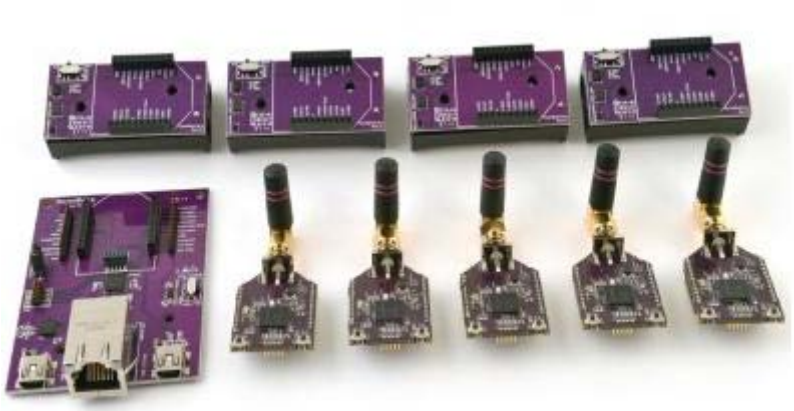

Fig. 3.1 OpenMote development Kit [6].

Furthermore, the board has limited support but ongoing development for RIOT and also full support for freeRTOS.

3.1.1. TI CC2538

TI CC2538 is a power wireless MCU System-On-Chip (SoC) for high performance IoT applications. The chip combine an ARM Cortex-M3 based MCU, providing up to 32KB on-chip RAM, and up to 512KB on-chip flash together with an IEEE 802.15.4 radio. The tiny shaped chip is able to run the most up-to-date network stacks with high-level security and robustness applications. The 32 GPIO ports and serial peripherals enables the connection between the chip and test board. There is also a micro-USB port on the board, which could be connected to external power source.

The SoC allows efficient authentication and encryption process, while minimizing the workload for the MCU. Furthermore, three sets of low-power modes with retention enables the quick sleep and recharge for periodic tasks, leveraging the performance and power consumption. TI has provided a comprehensive driver library and a series of debugging tools, which guarantees the smooth development of CC2538. The chip is also equipped with state of the art IoT technologies and solutions such as ZigBee and 6LoWPAN.

3.1.2. OpenMote - CC2538

The OpenMote-CC2538 **(Fig. 3.2)** is the core of the OpenMote hardware Ecosystem. It is the brain of the platform, and the element a developer programs.

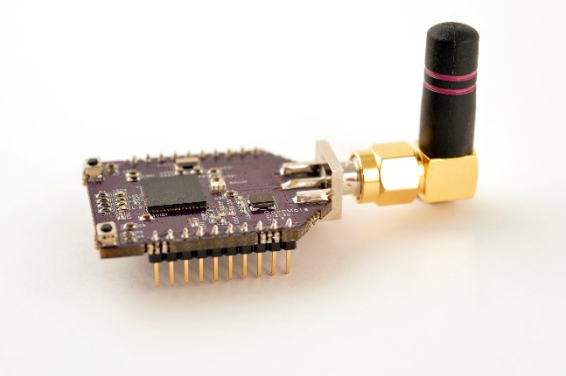

Fig. 3.2 OpenMote module with antena [6].

The OpenMote-CC2538 includes the following hardware:

- **CC2538:** It is a SoC (System on Chip) from Texas Instruments with a 32-bit Cortex-M3 microcontroller and a CC2520-like radio transceiver. The radio operates at the 2.4 GHz band and is fully compatible with the IEEE 802.15.4-2006 standard.

- **TPS62730:** It is a step-down DC/DC converter from Texas Instruments with two operation modes, regulated and bypass. In bypass mode the TPS62730 directly connects the input voltage from the battery (typically 3 V) to the whole system. In regulated mode the TPS62730 regulates the input voltage (typically 3 V) down to 2.1 V. The benefit of such approach is that the whole efficiency of the system can be improved under both low and high load conditions, that is, either when the system is sleeping or when the radio is transmitting or receiving.

- **ABM8G:** It is a 32 MHz crystal from Abracon Corporation used to clock the microcontroller and the radio transceiver. The part is rated at 30ppm (parts per million) from -20 ºC to +70 ºC.
- **ABS07:** It is a 32.768 kHz crystal from Abracon Corporation used to clock the microcontroller RTC (Real Time Clock). The part is rated at 10ppm (parts per millon) from -40 ºC to +85 ºC.

- **LEDs:** It includes 4 LEDs (colors: red, green, yellow and orange) from Rohm Semiconductor for debugging purposes.

- **Buttons:** It includes 2 buttons from Omron, one is used to reset the board and the other is connected to a GPIO line, thus enabling to wake up the microcontroller from sleep modes through an interrupt.

- **Antenna connector:** The antenna connector enables to connect an external antenna to the board.

- **XBee layout:** The OpenMote is fully compliant with the XBee form factor, meaning that it can be easily interfaced with a computer or board using the XBee connector.

3.1.3. OpenBase

The OpenBase **(Fig. 3.3)** is a skin for the OpenMote-CC2538 which offers all the interfaces needed for efficient firmware development. It features a socket for the OpenMote-CC2538, a 10-pin JTAG connector for in-circuit debugging of the OpenMote-CC2538, a circuit to monitor the current draw of the OpenMote-CC2538, pins to interface the OpenMote-CC2538 to external devices, a USB connector to re-program and debug the OpenMote-CC2538, and a 10/100 Mbps Ethernet connector to connect the OpenMote-CC2538 directly to a LAN.

Fig. 3.3 OpenBase Module [6].

This wealth of interfaces means that the OpenBase can serve several purposes. Through the JTAG interface, it can be used during code development to place breakpoints and inspect variables. Through the USB interface, it can be used to reprogram the OpenMote-CC2538 with pre-compiled binary images, and receive status information from that firmware over a serial interface.
Through the 10/100 Mbps Ethernet interface, the OpenMote-CC2538 can be connected to the Internet without requiring a computer.

3.1.4. OpenBattery

The OpenBattery **(Fig. 3.4)** is a skin for the OpenMote-CC2538 which provides power and basic sensing capabilities. It is composed of a battery holder for 2 AAA batteries, a socket for the OpenMote-CC2538, an on/offswitch, and three sensors: a temperature/humidity sensor (SHT21), a 3-axis accelerometer (ADXL346) and a light sensor (MAX44009). All sensors are interfaced with the OpenMote-CC2538 using an I2C bus. The temperature sensor can be used in a wide set of applications, including network synchronization. The 3-axis accelerometer can be used for dynamic or static motion detection. The light sensor can be used for a wide range of applications, from presence detection to touch-less switching.

Fig. 3.4 OpenBattery Module [6].

3.2. SEGGER J-Link Debug Probe

SEGGER J-Link edu is a JTAG **(Fig. 3.5)** Debug probe emulator designed for ARM cores. It connects via USB to a PC running Linux or Microsoft Windows 2000 or later. Has a built-in 20-pin JTAG connector, which is compatible with the standard 20-pin connector defined by ARM.

Fig. 3.5 Segger J-link edu Device [7].

Features:

- Direct download into flash memory of most popular microcontrollers supported.
- Full-speed USB 2.0 interface.
- Serial Wire Debug supported.
- Serial Wire Viewer supported.
- Download speed up to 1 MBytes/second.
- Debug interface (JTAG/SWD/...) speed up to 15 MHz.
- RDI interface available, which allows using J-Link with RDI compliant software.

3.3. TI CC2531EMK Sniffer Tool

The CC2531EMK kit provides one CC2531 USB Dongle and documentation to support a PC interface to 802.15.4 / ZigBee applications. The dongle can be plugged directly into your PC and can be used as an IEEE 802.15.4 packet sniffer or for other purposes.

The CC2531 USB Dongle **(Fig. 3.6)** is a fully operational USB device that can be plugged into a PC.

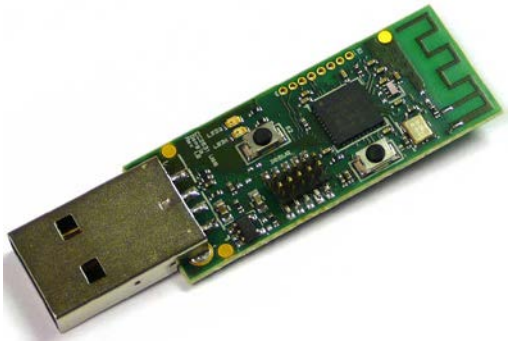

Fig. 3.6 CC2531 USB Dongle [8].

The dongle has 2 LEDs, two small push-buttons and connector holes that allow connection of external sensors or devices. The dongle also has a connector for programming and debugging of the CC2531 USB controller.

The dongle comes preprogrammed with firmware such that it can be used as a packet sniffer device.

CHAPTER 4. SETUP & TESTS

This chapter begins addressing the configuration of the test scenario both hardware and software levels. This will allow us to develop some tests in the following phases, such as the basic connectivity between devices. Some devices were configured in standalone mode in order to be able to perform different tests using 4 nodes, among which only 2 are directly accessible. In addition, we will check the visibility of the nodes connected to the same network and also the ones that are not joined. The next step will be the revision of the routing tables. Finally, through different captures, we will analyze the behavior after some changes of scenario.

4.1. Test Scenario installation & Configuration

The test scenario is composed by an openmote kit composed by 4 nodes, 2 operated by OpenBattery and the other 2 with OpenBase allowing to access console to interact with the scenario.

To flash code in OpenMote there are 2 methods:

- Using UART port and the bootloader backdoor mechanism present in TI CC2538 SoC. Although I followed the manufacturer's instructions I was not able to load code so I use the second option.
- Using a J-Link ARM debug probe (SEGGER j-link edu in my case) and an ARM-JTAG-20-10 adapter to connect the OpenBase JTAG port.

At software level we are going to use an Ubuntu Linux as OS to compile and flash devices and Github repository for OpenThread code.

First of all, we need to install the tools to cross-compile C/C++ code for ARM microcontrollers in Linux host.

```
apt-get install gcc-arm-none-eabi gdb-arm-none-eabi git make
```

To load code and debug we also need to download and install the latest version of the Segger J-Link drivers for GNU/Linux.

```
wget https://www.segger.com/downloads/jlink/jlink_6.0.7_x86_64.deb
dpkg --install jlink_6.0.7_x86_64.deb
```

Next step is to install git tools and download the OpenThread code from Github repository.

```
apt-get install git
git clone https://github.com/openthread/openthread.git ~/OpenThread
```

At this point we have all the tools and code to compile code and flash to OpenMote devices.

```
cd ~/OpenThread/openthread
./bootstrap
make -f examples/Makefile-cc2538
```

Once compiled binary file are ready to flash and are available at:

```
~/OpenThread/openthread/output/bin/arm-none-eabi-ot-cli-ftd
```

To flash OpenMote using the J-link method we are going to use the Segger tools provided and previously installed.

Now that the code is compiled, we need to upload it to the OpenMote. OpenBase is connected to the computer via the USB_FTDI USB port (on/off switch is in the USB_FTDI position) Then, OpenBase is connected using the j-link port to the Segger j-link and this is connected by USB to linux host.

Now is time to start the Segger GDB Server

```
/opt/SEGGER/JLink/JLinkGDBServer -device CC2538SF53
```

To load code we provide a script to connect the server and flash the code. Create a file containing all the parameters and save it as *flash.gdb*

```
arm-none-eabi-gdb
target remote localhost:2331
monitor interface jtag
monitor speed 5000
monitor endian little
monitor flash download = 1
monitor flash breakpoints = 1
monitor reset
load ~/OpenThread/openthread/output/bin/arm-none-eabi-ot-cli-ftd
```

Execute the GDB command using the file previously created.

```
arm-none-eabi-gdb –x flash.gdb
```

Once the code has been successfully loaded into the OpenMote-CC2538 board, it can be connected using serial port. Pressing the reset button OpenMote device are ready to be used as standalone connected to an OpenBattery or in an OpenBase to allow the cli interaction.

4.2. Basic tests

The first test is to check if there are communication with two devices.

Using two nodes connected to OpenBase we can check the connectivity using cli commands.

NODE 1

Configure panid

```
> panid 0x1234
```

Configure channel

```
> channel 11
Done
```

Bring up interface

```
> ifconfig up
Done
```

Start Thread Operation

```
> thread start
Done
```

Check Thread state

```
> state
leader
Done
```

View ip

```
> ipaddr
fdde:ad00:beef:0:0:ff:fe00:fc00
fe80:0:0:0:dcba:44f7:10d6:6aad
fdde:ad00:beef:0:ed55:920e:5392:3efc
Done
```

NODE 2

Configure panid

```
> panid 0x1234
```

Configure channel

```
> channel 11
Done
```

Bring up interface

```
> ifconfig up
Done
```

Start Thread Operation

```
> thread start
Done
```

Check Thread state

```
> state
router
Done
```

View ip

```
> ipaddr
fe80:0:0:0:58b1:774b:b4e7:36df
fdde:ad00:beef:0:0:ff:fe00:0
fdde:ad00:beef:0:12b5:5701:a0f2:932d
Done
```

As we can see both devices are up and running and the node1 become leader and the node 2 router. We also can see using scan commands the nodes available in the network.

```
>scan
| J | Network Name   | Extended PAN   | PAN | MAC Address   | Ch | dBm | LQI |
+---+----------------+----------------+-----+----------------+----+-----+-----+
> | 0 | OpenThread    | dead00beef00cafe | 1234 | deba44f710d66aad | 11 | -50 | 108 |
```

Because both nodes are in the same network we can ping from node 1 to node 2.

```
> ping fdde:ad00:beef:0:ed55:920e:5392:3efc
> 16 bytes from fdde:ad00:beef:0:ed55:920e:5392:3efc: icmp_seq=1
hlim=64 time=2ms
```

4.3. CLI for standalone devices

One of the requisites to allow the standalone operation is the OpenBattery nodes are able to start services and join network autonomously. This can be configured adding some code snippet in *openthread/examples/apps/cli/main.c* file before the *while(1).*

```
otSetPanId(sInstance,0x1234);
otSetChannel(sInstance, 11);
otInterfaceUp(sInstance);
otThreadStart(sInstance);
```

This configure channel, panid, raise up interface and start Thread protocol.

4.4. Scan & Discover

This test allows to discover all devices in the range of this device (in any channel).

```
> scan
| J | Network Name     | Extended PAN     | PAN  | MAC Address      | Ch | dBm | LQI |
+---+------------------+------------------+------+------------------+----+-----+-----+
| 0 | OpenThread       | dead00beef00cafe | 1234 | 262a9a8dc82cda79 | 11 | -34 | 108 |
| 0 | OpenThread       | dead00beef00cafe | 1122 | 3a4f64ce2c1ad519 | 12 | -37 | 108 |
| 0 | OpenThread       | dead00beef00cafe | 1144 | deba44f710d66aad | 14 | -39 | 108 |
Done
```

But if we want to check all the devices in the same network we can use next commands.

```
> discover
| J | Network Name     | Extended PAN     | PAN  | MAC Address      | Ch | dBm | LQI |
+---+------------------+------------------+------+------------------+----+-----+-----+
| 0 | OpenThread       | dead00beef00cafe | 1234 | 262a9a8dc82cda79 | 11 | -33 | 108 |
| 0 | OpenThread       | dead00beef00cafe | 1122 | 3a4f64ce2c1ad519 | 12 | -34 | 108 |
```

4.5. Router Table

In this test we have 4 nodes all configured in the same panid and channel and we can see that from one node (id=0) we can reach directly all the other 3 nodes.

```
> router table
| ID | RLOC16 | Next Hop | Path Cost | LQI In | LQI Out | Age | Extended MAC     |
+----+--------+----------+-----------+--------+---------+-----+------------------+
| 0  | 0x0000 |      0   |     0     |     3  |    3    |  9  | 5ab1774bb4e736df |
| 5  | 0x1400 |     63   |     0     |     0  |    0    | 63  | 262a9a8dc82cda79 |
|32  | 0x8000 |     32   |     0     |     3  |    3    | 12  | 3a4f64ce2c1ad519 |
|33  | 0x8400 |     33   |     0     |     0  |    9    | 159 | deba44f710d66aad |

Done
```

4.6. Diag mode

The OpenThread diagnostics module is a tool for debugging platform hardware manually.

Using the diagnostics module we can modify some parameters like RX power.

```
> diag power -10
set tx power to -10 dBm
status 0x00
```

Check network statistics.

```
> diag stats
received packets: 19
sent packets: 19
first received packet: rssi=-95, lqi=108
```

Or print statistics during diagnostics mode.

```
> diag stats
received packets: 10
sent packets: 10
first received packet: rssi=-65, lqi=101
```

4.7. Captured frames

- **NODE START:**

Captured frames from node *0x3a4f64ce2c1ad519* start **(Fig 4.1)** with no other devices in range.

P.nbr.	Time (us)	Length	Frame control field	Sequence number	Dest. PAN	Dest. Address	Source Address	MAC payload
RX 1	+0 =0	63	Type Sec Pnd Ack.req PAN_compr DATA 0 0 0 1	0xC0	0x1122	0xFFFF	0x3A4F64CE2C1AD519	7F 3B 02 F0 4D 4C 4D 4C 0B E5 00 15 7A 8D 01 00 00 00 00 00 01 8A A8 9F 30 80 AF 7C B0 60 A8 A9 14 97 76 7C D6 C7 F0 9F D0 81 90 DE AA 92
RX 2	+1001030 =1001030	63	DATA 0 0 0 1	0xC1	0x1122	0xFFFF	0x3A4F64CE2C1AD519	7F 3B 02 F0 4D 4C 4D 4C D0 AF 00 15 7B 8D 01 00 00 00 00 00 01 06 20 C0 79 3A E7 F0 C0 64 5E 61 0B 8F 8C 40 9B 79 77 D9 0B F9 70 EE B2 12
RX 3	+3001083 =4002113	63	DATA 0 0 0 1	0xC2	0x1122	0xFFFF	0x3A4F64CE2C1AD519	7F 3B 02 F0 4D 4C 4D 4C D8 E6 00 15 7C 8D 01 00 00 00 00 00 01 A9 87 36 0D 6B B1 04 A6 B0 05 87 9B 13 F0 4C 49 D3 8A 5E 5B BA 0A 39 B9 90
RX 4	+1001031 =5003144	63	DATA 0 0 0 1	0xC3	0x1122	0xFFFF	0x3A4F64CE2C1AD519	7F 3B 02 F0 4D 4C 4D 4C 91 CE 00 15 7D 8D 01 00 00 00 00 00 01 4D C8 FD C8 AE 40 F5 36 2D 86 E8 53 BF 2D 26 BE 5B 1B 54 D2 39 8D BD 6D 24
RX 5	+2007247 =7010391	69	DATA 0 0 0 1	0xC4	0x1122	0xFFFF	0x3A4F64CE2C1AD519	7F 3B 01 F0 4D 4C 4D 4C E8 49 00 15 7E 8D 01 00 00 00 00 00 01 F5 CD 58 A9 EC 15 51 C5 C9 1D 91 67 A0 F4 E1 04 65 62 36 8A 6C 4F 05 EC 79 9C E8 24 9E A4 B8
RX 6	+97958 =7108349	78	DATA 0 0 1 1	0xC5	0x1122	0x5AB1774BB4E736DF	0x3A4F64CE2C1AD519	7F 33 F0 4D 4C 4D 4C E1 ED 00 15 7F 8D 01 00 00 00 00 00 01 9E 80 34 BD 03 A0 05 27 3E D1 AC 29 7B 13 D9 23 FF 2D A9 03 DD 72 9B 99 16 58 85 6D 2E 0C FE 02 49 5F F5
RX 7	+18428 =7126777	78	DATA 0 0 1 1	0xC5	0x1122	0x5AB1774BB4E736DF	0x3A4F64CE2C1AD519	7F 33 F0 4D 4C 4D 4C E1 ED 00 15 7F 8D 01 00 00 00 00 00 01 9E 80 34 BD 03 A0 05 27 3E D1 AC 29 7B 13 D9 23 FF 2D A9 03 DD 72 9B 99 16 58 85 6D 2E 0C FE 02 49 5F F5
RX 8	+18969 =7145746	78	DATA 0 0 1 1	0xC5	0x1122	0x5AB1774BB4E736DF	0x3A4F64CE2C1AD519	7F 33 F0 4D 4C 4D 4C E1 ED 00 15 7F 8D 01 00 00 00 00 00 01 9E 80 34 BD 03 A0 05 27 3E D1 AC 29 7B 13 D9 23 FF 2D A9 03 DD 72 9B 99 16 58 85 6D 2E 0C FE 02 49 5F F5
RX 9	+21991 =7167737	78	DATA 0 0 1 1	0xC5	0x1122	0x5AB1774BB4E736DF	0x3A4F64CE2C1AD519	7F 33 F0 4D 4C 4D 4C E1 ED 00 15 7F 8D 01 00 00 00 00 00 01 9E 80 34 BD 03 A0 05 27 3E D1 AC 29 7B 13 D9 23 FF 2D A9 03 DD 72 9B 99 16 58 85 6D 2E 0C FE 02 49 5F F5
RX 10	+22002 =7189739	78	DATA 0 0 1 1	0xC5	0x1122	0x5AB1774BB4E736DF	0x3A4F64CE2C1AD519	7F 33 F0 4D 4C 4D 4C E1 ED 00 15 7F 8D 01 00 00 00 00 00 01 9E 80 34 3E D1 AC 29 7B 13 D9 23 FF 2D A9 03 DD 72 9B 99 16 58 85 6D 2E 0C FE
RX 11	+662537 =7852276	69	DATA 0 0 0 1	0xC6	0x1122	0xFFFF	0x3A4F64CE2C1AD519	7F 3B 01 F0 4D 4C 4D 4C E7 FB 00 15 80 8D 01 00 00 00 00 00 01 61 76 17 44 9F 94 B2 57 47 4E D7 0E 22 78 AE F4 EB D8 25 11 98 19 32 78 60 65 FE 9F 28 6C 60
RX 12	+2112064 =9964340	69	DATA 0 0 0 1	0xC7	0x1122	0xFFFF	0x3A4F64CE2C1AD519	7F 3B 01 F0 4D 4C 4D 4C FB DC 00 15 81 8D 01 00 00 00 00 00 01 27 BF AB 3A 28 4B 9F 89 0E 0E 77 D8 7D 2E 98 49 E0 72 29 5C 93 FC 66 0C E3 77 72 C4 3E 07 DE
RX 13	+3811118 =13775458	69	DATA 0 0 0 1	0xC8	0x1122	0xFFFF	0x3A4F64CE2C1AD519	7F 3B 01 F0 4D 4C 4D 4C 39 C3 00 15 82 8D 01 00 00 00 00 00 01 1C 4C A6 D8 5E 3B 6D BD E5 E3 64 73 90 86 63 3A 49 D3 C8 A2 6C 73 10 67 3A 44 6E A2 54 9D 5C
RX 14	+4833149 =18608607	69	DATA 0 0 0 1	0xC9	0x1122	0xFFFF	0x3A4F64CE2C1AD519	7F 3B 01 F0 4D 4C 4D 4C 83 B4 00 15 83 8D 01 00 00 00 00 00 01 AF 1C D1 D4 E3 04 9F 70 E9 21 57 84 CB FA EB B7 45 2B FB 0B 73 B5 0E 05 8F BE D0 D7 C9 76 DD
RX 15	+14839437 =33448044	69	DATA 0 0 0 1	0xCA	0x1122	0xFFFF	0x3A4F64CE2C1AD519	7F 3B 01 F0 4D 4C 4D 4C F9 83 00 15 84 8D 01 00 00 00 00 00 01 AA E5 EA 38 24 06 82 7A 51 E9 49 C6 EA 9E C0 F6 C8 AE D7 FC 59 F8 60 A8 16 BC 8B C4 3C 97 38
RX 16	+27962840 =61410884	69	DATA 0 0 0 1	0xCB	0x1122	0xFFFF	0x3A4F64CE2C1AD519	7F 3B 01 F0 4D 4C 4D 4C 7C 5B 00 15 85 8D 01 00 00 00 00 00 01 45 29 DD D9 32 26 FD 86 13 E3 DD 60 4A 0C C0 0E A0 54 BB 6D D0 87 98 A1 97 90 4D 27 DF 11 47
RX 17	+37308123 =98719007	69	DATA 0 0 0 1	0xCC	0x1122	0xFFFF	0x3A4F64CE2C1AD519	7F 3B 01 F0 4D 4C 4D 4C 4D D5 00 15 86 8D 01 00 00 00 00 00 01 11 69 3C 00 94 3F 2C 26 F3 B3 6E A5 A0 80 FE 32 9F 1F 78 BF 76 A1 6F 75 5B 04 26 1A 54 01 C2
RX 18	+20903625 =119622632	69	DATA 0 0 0 1	0xCD	0x1122	0xFFFF	0x3A4F64CE2C1AD519	7F 3B 01 F0 4D 4C 4D 4C AD 5E 00 15 87 8D 01 00 00 00 00 00 01 B4 D3 4F A7 06 91 54 FF 39 06 87 7E 59 D3 70 CF B8 F2 16 24 02 D3 26 4F B6 64 C3 5B FC 64 C1
RX 19	+37056119 =156678751	69	DATA 0 0 0 1	0xCE	0x1122	0xFFFF	0x3A4F64CE2C1AD519	7F 3B 01 F0 4D 4C 4D 4C 84 7D 00 15 88 8D 01 00 00 00 00 00 01 4B 77 9E CA 4D 7E C6 E9 3B 0E 11 7B 78 D4 3F F9 E2 05 22 10 A7 8E 7F 15 35 6C CA B0 F4 E0 D8
RX 20	+30848934 =187527685	69	DATA 0 0 0 1	0xCF	0x1122	0xFFFF	0x3A4F64CE2C1AD519	7F 3B 01 F0 4D 4C 4D 4C 0E 75 00 15 89 8D 01 00 00 00 00 00 01 D3 3F E3 30 AB 18 CE 01 DD 34 C3 30 C1 27 02 CB 8D 70 1A A4 17 FB 88 D2 8E 42 85 85 3D A0 D6
RX 21	+31495432 =219023117	69	DATA 0 0 0 1	0xD0	0x1122	0xFFFF	0x3A4F64CE2C1AD519	7F 3B 01 F0 4D 4C 4D 4C 75 85 00 15 8A 8D 01 00 00 00 00 00 01 A6 7C 1E 12 0F 98 57 73 9B 36 6D 55 AA 86 44 FF A2 F7 E7 9A 97 33 5A 2A DD 01 C8 DF C7 47 E7

Fig. 4.1 Node start capture.

We observe the node *0x3a4f64ce2c1ad519* broadcast 5 times the Data packet, latter they ask to node *0x5ab1774bb4e736df* same packet 4 times (this node it's not present at scenario and although I reset the node still appearing references to that). They broadcast again 5 packets like the beginning. Packets from 16 to 21 are periodic packets sent every 40 sec.

- **PING:**

Captured packages ping **(Fig 4.2)** from *0xdeba44f710d66aad* to *0x262a9a8dc82cda79*.

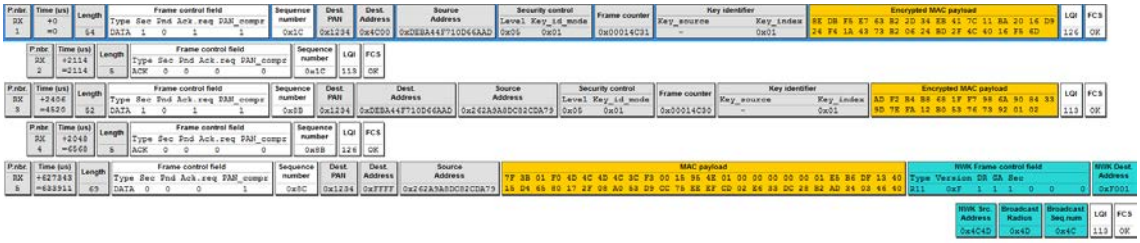

Fig. 4.2 Ping capture.

Node *0xdeba44f710d66aad* sends a packet to address *0x4C00* and frame 2 is the ack. In the next packet pinged node *0x262a9a8dc82cda79* replies to *0xdeba44f710d66aad* and got the ack confirming the package. Last frame is sent from node *0xdeba44f710d66aad* to broadcast.

- **Join Node:**

Captured packages when node *0x5ab1774bb4e736df* join the network **(Fig 4.3)** composed by devices *0xdeba44f710d66aad* and *0x262a9a8dc82cda79.*

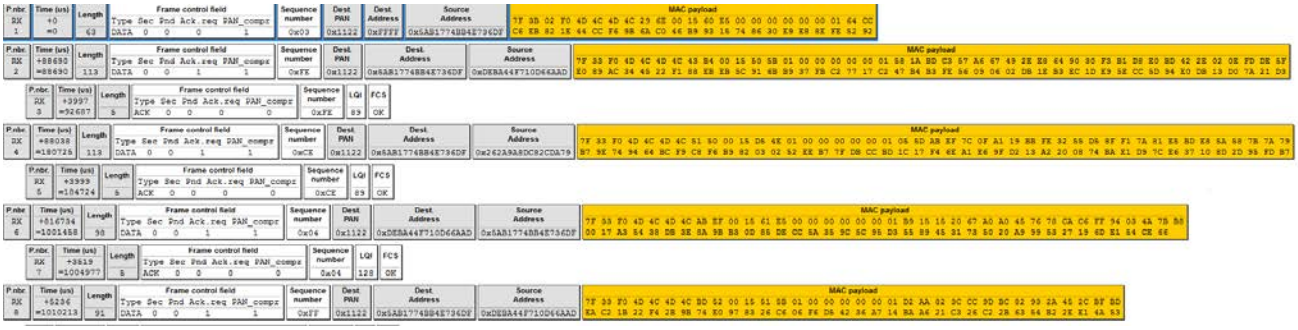

Fig. 4.3 Join node capture.

When start node *0x5ab1774bb4e736df* sends a packet to broadcast and node *0xdeba44f710d66aad* reply to already joined node *0x5ab1774bb4e736df* with a DATA packet receiving the ack. Node *0x262a9a8dc82cda79* do the same and node *0x5ab1774bb4e736df sends* a data packet to *0xdeba44f710d66aad* receive ack and last *0xdeba44f710d66aad sends* a packet to *0x5ab1774bb4e736df.*

When finish router table are updated.

```
> router table
| ID | RLOC16 | Next Hop | Path Cost | LQI In | LQI Out | Age | Extended MAC     |
+----+--------+----------+-----------+--------+---------+-----+------------------+
|  0 | 0x0000 |        0 |         0 |      3 |       3 |   9 | 5ab1774bb4e736df |
| 19 | 0x4c00 |       19 |         0 |      0 |       0 | 106 | 262a9a8dc82cda79 |
| 52 | 0xd000 |       52 |         0 |      3 |       3 |  19 | deba44f710d66aad |
```

- **Node Disappeared:**

Captured packages when node *0x5ab1774bb4e736df* disappear from the network **(Fig 4.4)** composed by devices *0xdeba44f710d66aad* and *0x262a9a8dc82cda79.*

P.nbr.	Time (us)	Length	Frame control field	Sequence number	Dest. PAN	Dest. Address	Source Address	MAC payload
RX 1	+0 =0	70	Type Sec Pnd Ack.req PAN_compr DATA 0 0 0 1	0xF9	0x1122	0xFFFF	0x262A9A8DC82CDA79	7F 3B 01 F0 4D 4C 4D 4C D9 80 00 15 FF 4E 01 00 00 00 00 00 01 5B 00 00 3B 68 E3 EE 4C 4E CF 3F 3B B2 79 68 0A B6 1B 31 08 44 33 2F A2 A4 27 71 2C 03 19 59 CA

P.nbr.	Time (us)	Length	Frame control field	Sequence number	Dest. PAN	Dest. Address	Security control	Frame counter	Key identifier	Encrypted MAC payload
RX 2	+16593307 =16593307	37	Type Sec Pnd Ack.req PAN_compr DATA 1 1 1 1	0x4F	0x9FB3	0xEE0EC7862DFB6A1A	Level Key_id_mode 0x04 0x01	0x9C7F7386	Key_source – Key_index 0x36	64 32 D1 D7 5F 5D 8A 85 39 64 E5 D9 DE 2D B4 0A

P.nbr.	Time (us)	Length	Frame control field	Sequence number	Dest. PAN	Dest. Address	Source Address	MAC payload
RX 3	+12331464 =28924771	70	Type Sec Pnd Ack.req PAN_compr DATA 0 0 0 1	0xFA	0x1122	0xFFFF	0x262A9A8DC82CDA79	7F 3B 01 F0 4D 4C 4D 4C E1 84 00 15 00 4F 01 00 00 00 00 00 01 F6 D8 6F 8E 49 1E 82 9E 0C BF D5 0E 16 13 C2 7E CA 99 FC DA 99 F1 29 A2 95 AE 3D 84 C9 EC 0F 76
RX 4	+33448879 =62373650	70	Type Sec Pnd Ack.req PAN_compr DATA 0 0 0 1	0xFB	0x1122	0xFFFF	0x262A9A8DC82CDA79	7F 3B 01 F0 4D 4C 4D 4C 95 2B 00 15 01 4F 01 00 00 00 00 00 01 83 46 8A 3A F8 E9 C2 26 38 72 0B 2C 19 C7 1A 4B 0C 53 4E 50 87 C2 E1 4A D1 28 7C B9 5B 5D D0 3A
RX 5	+33780883 =96154533	70	Type Sec Pnd Ack.req PAN_compr DATA 0 0 0 1	0xFC	0x1122	0xFFFF	0x262A9A8DC82CDA79	7F 3B 01 F0 4D 4C 4D 4C B5 08 00 15 02 4F 01 00 00 00 00 00 01 96 87 31 4A 8C 22 F7 05 AA C5 53 50 1A CD 6A D7 EB 29 90 17 5F 9D 6B A2 EE C7 6C D3 C5 CF 69 AB
RX 6	+31979834 =128134367	70	Type Sec Pnd Ack.req PAN_compr DATA 0 0 0 1	0xFD	0x1122	0xFFFF	0x262A9A8DC82CDA79	7F 3B 01 F0 4D 4C 4D 4C 09 4F 00 15 03 4F 01 00 00 00 00 00 01 31 44 C1 A1 C8 3F 52 95 0F B3 F7 0A 2B C0 9C 11 5A EB 5F 82 B4 A7 1C D4 A1 79 B2 D0 62 EB 3A 8D
RX 7	+37167974 =165302341	70	Type Sec Pnd Ack.req PAN_compr DATA 0 0 0 1	0xFE	0x1122	0xFFFF	0x262A9A8DC82CDA79	7F 3B 01 F0 4D 4C 4D 4C D6 44 00 15 04 4F 01 00 00 00 00 00 01 D5 C3 1A 7A F9 2E 7F 71 1B 92 3D 83 8C 6B EE 03 DF B3 E3 85 A6 FC 74 44 03 77 87 B5 68 7F 55 9F
RX 8	+8596210 =173898551	69	Type Sec Pnd Ack.req PAN_compr DATA 0 0 0 1	0xFF	0x1122	0xFFFF	0x262A9A8DC82CDA79	7F 3B 01 F0 4D 4C 4D 4C BE 14 00 15 05 4F 01 00 00 00 00 00 01 03 93 7F F5 62 EC 8B CF 9F 06 4A CA 79 B8 4C 15 B6 17 0F 07 A8 33 98 FC 7F 34 4D 60 F6 78 AB
RX 9	+2006054 =175904605	69	Type Sec Pnd Ack.req PAN_compr DATA 0 0 0 1	0x00	0x1122	0xFFFF	0x262A9A8DC82CDA79	7F 3B 01 F0 4D 4C 4D 4C 86 46 00 15 06 4F 01 00 00 00 00 00 01 5A CE 9A 61 A3 74 3A C1 48 EF 6A 9C 27 82 36 EA 30 15 FE C3 D6 48 2C 52 16 95 3D C4 C4 4B 35
RX 10	+3292086 =179196691	69	Type Sec Pnd Ack.req PAN_compr DATA 0 0 0 1	0x01	0x1122	0xFFFF	0x262A9A8DC82CDA79	7F 3B 01 F0 4D 4C 4D 4C E6 24 00 15 07 4F 01 00 00 00 00 00 01 B9 9A FE 04 FB 9C 15 4A 47 26 26 9E 54 4A 1A E6 B9 C2 16 13 06 D2 AE DB EE AF 44 8C C9 DA 5E
RX 11	+5853157 =185049848	69	Type Sec Pnd Ack.req PAN_compr DATA 0 0 0 1	0x02	0x1122	0xFFFF	0x262A9A8DC82CDA79	7F 3B 01 F0 4D 4C 4D 4C A3 8A 00 15 08 4F 01 00 00 00 00 00 01 67 F9 34 9E 7E C3 69 DE D7 AC 21 7B F6 BD 0A D0 E9 96 F4 13 35 48 00 01 E4 5D 37 86 7F 8F F2

Fig. 4.4 Node disappeared capture.

First of all note frame 2 do not belongs to this scenario… seems a “real” ZigBee packet with security enabled. About the scenario, we can identify the periodic packets sent from leader node *0x262a9a8dc82cda7* every 40 seconds and when have elapsed 240 seconds (6 packets x 40 seconds) node start to send 4 consecutive packets (provably a routing table upgrade) to broadcast informing about the node lost. Note this 240 seconds coincide with the timer explained in figure 2.5.

```
kMleEndDeviceTimeout      = 240,          ///< MLE_END_DEVICE_TIMEOUT (seconds)
```

- **Discover (with 2 nodes more in range):**

Captured packages when node *0x262a9a8dc82cda79* send the discover **(Fig 4.5)**.

P.nbr.	Time (us)	Length	Frame control field	Sequence number	Dest. PAN	Dest. Address	Source PAN	Source Address	MAC payload	NWK Frame control field	NWK Dest Address
RX 1	+0 =0	37	Type Sec Pnd Ack.req PAN_compr DATA 0 0 0 0	0x80	0xFFFF	0xFFFF	0x1122	0x262A9A8DC82CDA79	7F 3B 02 F0 4D 4C 4D 4C 4D 3A FF 10 1A 04 80 02 20 00	Type Version DR GA Sec R11 0xF 1 1 1 0 0 0	0xFF00

P.nbr.	Time (us)	Length	Frame control field	Sequence number	Dest. PAN	Dest. Address	Source Address	MAC payload
RX 2	+2800 =2800	66	Type Sec Pnd Ack.req PAN_compr DATA 0 0 1 1	0x87	0x1122	0x262A9A8DC82CDA79	0xDEBA44F710D66AAD	7F 33 F0 4D 4C 4D 4C 15 C1 FF 11 1A 1E 81 02 28 00 02 08 DE AD 00 BE EF 00 CA FE 03 0A 4F 70 65 6E 54 68 72 65 61 64 12 02 03 E8

P.nbr.	Time (us)	Length	Frame control field	Sequence number	LQI	FCS
RX 3	+2497 =5297	5	Type Sec Pnd Ack.req PAN_compr ACK 0 0 0 0	0x87	110	OK

P.nbr.	Time (us)	Length	Frame control field	Sequence number	Dest. PAN	Dest. Address	Source Address	MAC payload
RX 4	+16245 =21542	66	Type Sec Pnd Ack.req PAN_compr DATA 0 0 1 1	0x7F	0x1122	0x262A9A8DC82CDA79	0x5AB1774BB4E736DF	7F 33 F0 4D 4C 4D 4C F7 32 FF 11 1A 1E 81 02 28 00 02 08 DE AD 00 BE EF 00 CA FE 03 0A 4F 70 65 6E 54 68 72 65 61 64 12 02 03 E8

P.nbr.	Time (us)	Length	Frame control field	Sequence number	LQI	FCS
RX 5	+2495 =24037	5	Type Sec Pnd Ack.req PAN_compr ACK 0 0 0 0	0x7F	110	OK

Fig. 4.5 node discover capture.

Leader node *0x262a9a8dc82cda79* send a broadcast packet and both present nodes replies to leader and receive ack.

```
> discover
| J | Network Name     | Extended PAN     | PAN  | MAC Address      | Ch | dBm | LQI |
+---+------------------+------------------+------+------------------+----+-----+-----+
| 0 | OpenThread       | dead00beef00cafe | 1234 | deba44f710d66aad | 11 | -39 | 107 |
| 0 | OpenThread       | dead00beef00cafe | 1122 | 5ab1774bb4e736df | 12 | -44 | 108 |
```

CHAPTER 5. CONCLUSIONS

In this study we aimed to analyze Thread technology focusing on the OpenThread implementation. Moreover, we have performed some tests in a real platform that allowed us the possibilities and shortcomings of the system. Thread uses 6LoWPAN, which in turn uses the IEEE 802.15.4 wireless protocol with IPv6-based mesh communication allowing the connection to the internet and provides simple yet robust interface to the cloud. Thread is built upon a foundation of existing standards and can fulfill the requirements of low power, resilience, ip-based, security and friendly use. OpenThread is an open source implementation based on the draft Thread 1.0 specification of the Thread networking protocol and was released less than one year ago. Being so new, it has been a great challenge to explore this brand new technology but with the disadvantage of the lack of information and support, what makes the development of the work very complicated. For instance, some of the validated features were released in the last week.

Focusing on the tests, which were the most arduous and complex, we found strange behaviors. Although in some cases they could be caused by the lack of knowledge, in others we found that actually they had a malfunction. That prevents us of making more advanced tests, especially those related to the routing and comparison with other protocols. Instead, we were able to perform some basic tests such as check connectivity, see routing tables, etc.

In the near future the technology may be more mature and more people may be working on it, so there would be more applications and functionalities implemented. On the other hand, with the current functionalities we could study more thoroughly the part of the routing as well as study upper layers such as commissioning roles using wpantund as border router gateway between a common IPv6 network and a LR-WPAN Thread network.

Thread could be the future of mesh networking. It delivers on the promise of an IP-based mesh networking solution that is secure, reliable, scalable and optimized for low power operation. Nevertheless, it needs an industry-wide, standardized tunneling solution until ISPs provide native IPv6 to the home.

REFERENCES

[1] Z-Wave products commercial website http://www.z-wave.com

[2] EnOcean commercial website https://www.enocean.com

[3] Nest commercial website https://nest.com

[4] Thread website: https://threadgroup.org
- White papers:
  - Thread Overview.
  - Thread Stack Fundamentals.
  - 6LoWPAN.

[5] OpenThread Github website: https://github.com/openthread/openthread

[6] OpenMote website http://www.openmote.com/

[7] Segger j-lin edu https://www.segger.com/j-link-edu.html

[8] TI CC2531 USB Evaluation Module Kit http://www.ti.com/tool/CC2531emk

[9] Texas Instruments. CC2538 Powerful Wireless Microcontroller SoC website http://www.ti.com/product/CC2538

[10] IEEE 802.15.4-2006. “Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for Low-Rate Wireless Personal Area Networks (LR- WPANs)”.

[11] C. Gomez, J. Paradells, J.E. Caballero, "Sensors everywhere: wireless network technologies and solutions", Fundación Vodafone España.

[12] “UG103.11: Application Development Fundamentals: Thread”, SIlicon Labs.

[13] X. Vilajosana, P. Tuset, T. Watteyne, K. Pister: “OpenMote: Open-Source Prototyping Platform for the Industrial IoT”

[14] R. Romero “Demostrador d’un sistema de calefacció de la llar basat en OpenMote, OpenWSN (IEEE 802.15.4e) i thethings.iO”, Master Thesis UOC

[15] OpenThread wpantund Github https://github.com/openthread/wpantund

[16] “Ushering in a New Era of Internet Connectivity with Thread Networking Protocol”, SIlicon Labs.

[17] “RFC 2080, RFC 4862, RFC 4944, RFC 6282, RFC 6347, RFC 6775”, IETF

[18] D. Pollack “Understanding Thread Protocol Cheat Sheet”, Cheatography.

LIST OF ACRONYMS

6LoWPAN	IPv6 over Low power Wireless Personal Area Networks
CoAP	Constrained Application Protocol
DTLS	Datagram Transport Layer Security
HAN	Home Area Network
HAL	Hardware Abstraction Layer
IoT	Internet of Things
LR-WPAN	Low Rate Wireless Personal Area Network
MLE	Mesh Link Establishment
REED	Router Eligible End Device
SoC	System on Chip