University of Piraeus
Department of Digital Systems
Postgraduate Programme " Techno-economic Management & Security of Digital Systems "

SECURITY PROTOCOLS FOR IOT ENVIRONMENTS

Master Thesis Fall 2017

Combining existing communication protocols and security protocols for safer and cheaper communication

Advisor: Konstantinos Lamprinoudakis
Dorin Bugneac MTE-1501
dorinbug@hotmail.com

ABSTRACT

The aim of this thesis is to test the possibility of using the DTLS security protocol with the MQTT communication protocol on Wireless Sensor Network (WSN). Both protocols have been developed long before the inspiration of the Internet of Things (IoT). Because their original purpose is compatible with implementing the IoT many are trying modifying them so they can be used in developing IoT applications based on the Contiki OS.

First thing before testing the two protocols is very important, it requires to create the environment where all simulations will be executed with the intention of exploring the protocols. Also, if or how they can be combined. Next step involved running and testing of the MQTT protocol with intention of identifying how applications which includes this protocol work. Last step was to modify the tinyDTLS protocol created for tinyOS to work with the Contiki OS and to develop a client/server application to test the protocol.

After experimenting with the two protocols has shown that both protocols are important for the IoT development and can later be used to accomplish many goals related to stable and secure communication. However, there are some compatibility issues related to the structure of the two protocol that have to be resolved before combining them together successfully.

TABLE OF CONTENTS

1 INTRODUCTION

The evolution of technology in the recent years has opened a lot of new possibilities of using old and new technologies in an innovating ways. As IPv4 has reached a limit of covering the need of connected devices, a new technology, IPv6, has been developed to replace the old one. IPv6's capability of covering a huge range of connected devices has given birth to the Internet of Things (IoT).

"Smart systems that encompass integrated computational and physical components to sense the changing state of the real world. Devices, objects and systems connected in simple and transparent ways to enable information sharing to enable autonomous capabilities." [NIST]

The scope of the IoT is to create an intelligent world were the physical and virtual world connect with each other. The new smart environment is supposed to provide a smarter way using the energy, health, transport, cities, industry, buildings and many other areas of our daily life. This can be achieved by creating multiple islands of smart networks which enables the collection and exchange of information anytime, anywhere, anything and anyone through every available network.

The integration of this technology ensures a wide range of applications. Some of the most common areas to use the advantages of the IoT are:

- HEALTHCARE: Remote diagnostics and examinations, remote patient monitoring, tracking and telemetry, remote treatment and surgery, patient prescription management, medical asset tracking (tools, ambulances, and pharmaceuticals), centralized and up to date medical records, accessibility, including multi-lingual support, aural directions, and others.
- ICT: Network device tracking, inventory, automation, and remote/predictive management, physical security and monitoring (wires, fibers, poles, telecom vaults).
- MANUFACTURING AND HEAVY INDUSTRY: Process monitoring and management, equipment monitoring and management, health and safety monitoring and management, inventory management, shipping tracking, defect and recall management, proactive servicing and warranty ("tires will malfunction soon"), product-service bundles and mating ("keys in couch"), access control and monitoring for employees, suppliers, and contractors entering premises.
- FINANCE AND BANKING: Retail Point of Sale (POS) terminals, remotely located Automated Teller Machines (ATMs), on-line desktop banking, on-line mobile device banking, Food and Farming
- CHEMICAL AND ENVIRONMENTAL CONDITIONS (FOR PRODUCERS): monitoring of produce, livestock and processed foods for quality and defect management, spoilage, and expiry, tracking of heirloom varieties at the risk of being lost, access control and monitoring of food processing facilities, automation of ordering services and billing, automation of delivery processes and accounting.
- TRANSPORTATION: Smart trains, planes, automobiles, boats, and spacecraft. Traffic signals that respond to traffic conditions. Roads, rails, runways and piers that report wear and tear and proactively schedule maintenance. Road Side Units (RSUs) for safety, including crash prevention.
- DOMESTIC: Home automation, home security and monitoring, smart appliances.
- WATER: Reservoir levels, water quality (source, clear well, distribution network, demarcation point - home/building), system pressure, leak detection and localization.
- EDUCATION: Enhanced reality for learning, real-time empirical data from topic of study (video for marine probes used to study ocean biology), Security in schools such as perimeter monitoring,

automated student identity services, inventory and tracking of valuable educational assets and goods, remote education and engagement.

- ENERGY: Smart metering and time-of-use billing for consumers and businesses, coordination of batteries and storage systems for load balancing, pipeline and transmission lines which report on status and proactively adjust to loads and failures, coordination of generation and storage systems, smart incentives for load balancing.
- ENTERTAINMENT AND SPORTS: Enhanced reality for gaming -- Role-playing and first-person in real physical environments ("holo-deck" type), enhanced reality for tourism -- Multi-lingual tours for sites and monuments, simulated physical locations for promotion, embedded promotion and advertising, enhanced reality for sports -- player stats and health, equipment and material descriptions and information, ticketing and admission management.
- PUBLIC SAFETY AND MILITARY: Border and perimeter protection and surveillance, surveillance video camera network, asset tracking and localization, remote asset control, e.g. robots and drones. Weapons tracking and identification, sniper detection and localization, disaster management and response.
- RETAIL AND HOSPITALITY: Inventory management and logistics, highly targeted promotions based on physical location and environmental conditions, anti-theft and fraud, facilities monitoring and management, accessibility -- multi-lingual support / aural directions / others.
- GOVERNMENT: Remote service delivery and compliance monitoring, real-time environmental monitoring (air quality, water levels and pollution, earthquakes), municipal water and sewer monitoring and control, asset tracking and inventory, building and property management and maintenance (office campus, parks, and monuments), smart cities and smart communities.

All areas mentioned above are related to sensitive information. Some of this information is either related to the proper functions of a country, such as government and military, while the rest of the information is related to organizations that really depend on it for their survival. The reason being that the majority of the information collected concerns the privacy of individuals who either directly or indirectly use IoT applications in their daily life. Based on these reasons the need of sufficient security mechanisms is of high importance. Unfortunately, because the wireless networks created are composed of devices (nodes) with limited power, it is not feasible to utilize conventional security techniques. For this reason new techniques must be implemented and tested to ensure that they can provide integrity, confidentiality and authentication of the data collected. [3, 23]

1.1 Scope of the Thesis

The scope of the project is to test the compatibility of the MQTT and the DTLS protocols, based on the Contiki OS, as well as their security features. Basically, the reason of testing these two protocols is to see if they can be combined and used together in order to accomplish a secure way for the Wireless Sensor Network (WSN) to communicate with the global network. Both protocols have been developed, prior to the IoT, for power limited devices and thus they attract the interest for adoption in IoT infrastructures.

1.2 Structure of the Thesis

Chapter 2, Theoretical framework, includes basic information related to different parts of the Internet of Things. This includes known operating systems that are used for WSN, important protocols used to create

a fully functional IoT application and the security protocol tested. Generally, it provides the theoretical background of this work.

Chapter 3, Building the testing environment, presents how to create a virtual machine which includes the Contiki operating system and the msp430-gcc experimental compiler. Specifically it includes detailed steps on how the testing environment was created and what else is needed to test all aspects of the project.

Chapter 4, Experiments and results, describes how to create different IoT applications, based on the MQTT communication protocol and the DTLS security protocol. It describes all the parts needed to create an IoT environment, based on the Contiki OS, and how to test it using the Cooja simulator.

2 THEORETICAL FRAMEWORK

The increase of connected devices has shown the shortcomings that the limited capabilities of IPv4. To resolve this problem IPv6 was implemented, which gives a much greater coverage of connected devices to the Internet. IPv6 promotes the concept of the Internet of Things and in order to support it many new or existing technologies have been engaged.

2.1 OS FOR INTERNET OF THINGS

An operating system (OS) is a system program used for controlling the basic functions of the hardware, such as memory, storage, network, input/output, or intangible resources. For sensors to be able to function properly the need of compatible OS is required so that the restricted resources of sensors can be properly utilized giving the necessary results. Some of the most known OS for IoT low-power devices are tinyOS, RIOT and Contiki. [25]

2.1.1 TinyOS

TinyOS is an open source, BSD-licensed operating system designed for low-power wireless devices, such as those used in sensor networks, ubiquitous computing, personal area networks, smart buildings, and smart meters. A worldwide community from academia and industry use, develop, and support the operating system as well as its associated tool. (TinyOS, 2013)

It began as a collaboration between the University of California, Berkeley, Intel Research and Crossbow technology. TinyOS applications are written in a C programming language dialect called nesC, which is optimized for the memory limits of sensor networks. Also, both the OS and applications are composed of components which offer three types of elements, commands, events and task. Commands are requests to a component to do something, for example to query a sensor or to start a computation. Events are mostly used to signal the completion of such a request. Tasks are not executed immediately, when a task is posted the currently running program will continue its execution and the posted task will later be executed based on the schedule, first in first out. [9, 10]

2.1.2 RIOT

RIOT is a free, open source operating system developed by a grassroots community gathering companies, academia, and hobbyists, distributed all around the world. RIOT implements all relevant open standards supporting an Internet of Things that is connected, secure, durable, and privacy-friendly. (RIOT, 2017)

Basically RIOT is a simple and easy solution for a basic IoT OS to develop and run applications on constraint devices. From a developer point of view the benefits RIOT offers are that it uses standard C and C++ programming for the applications to be deployed and uses standard tools such as gcc, gdb, valgrid. Also, as an OS it minimizes hardware dependent code which can run on 8-bit platforms (e.g. Arduino Mega 2560), 16-bit platforms (e.g. MSP430), and on 32-bit platforms (e.g. ARM).

Furthermore, RIOT is also a resource friendly OS mostly because of the microkernel architecture and a tickless scheduler for very lightweight devices. The benefits are related to the robustness & code-footprint flexibility, the fact that it enables maximum energy-efficiency, real-time capability due to ultra-low interrupt latency (~50 clock cycles) and priority-based scheduling and multi-threading with ultra-low threading overhead (<25 bytes per thread).

From the point of view of the IoT the advantages are that it makes the applications ready for the smaller devices in the Internet with a common system support. It supports Internet of Things protocols like 6LoWPAN, IPv6, RPL, and UDP, CoAP and CBOR for easy integration on the applications to be used on nodes. Also, it supports both static and dynamic memory allocation, high resolution and long-term timers, different tools and utilities such as system shell, SHA-256, Bloom filters. [12, 20]

2.1.3 Contiki

Contiki is an open source operating system for the Internet of Things. It connects tiny low-cost, low-power microcontrollers to the Internet. Contiki is a powerful toolbox for building complex wireless systems and provides powerful low-power Internet communication. Also, it supports fully standard IPv6 and IPv4, along with the recent low-power wireless standards: 6loWPAN, RPL, CoAP. With Contiki's ContikiMAC and sleepy routers, even wireless routers can be battery-operated. (Contiki, 2016)

Some of the basic features that Contiki provides are related to memory allocation as it is designed for tiny systems, having only a few kilobytes of memory available, which makes it highly memory efficient and provides a set of specific mechanisms for memory allocation. Other mechanisms provided by Contiki are related to system power consumption and for understanding where the power was spent, generally Contiki is designed to operate in extremely low-power systems that may operate for years on AA batteries. Also, it provides a full IP network stack with standard IP protocols such as UDP, TCP and HTTP and more new low-power standard like 6LowPAN, RPL, CoAP and MQTT. Other protocols supported are for low-power IPv6 networking, which includes the 6LowPAN adaption layer, RPL IPv6 multi-hop routing protocol and the CoAP RESTful application-layer protocol.

More features that Contiki supports is dynamic module loading, dynamic loading and linking of modules at run-time, which provides the capability of changes after deployment. The dynamic module loader can load, relocate and link *ELF* files that can optionally be stripped off debugging symbols so to keep their size down. Contiki provides plenty of examples to get familiar with the Contiki system, the platform hardware and how to program network code which can be tested through the Cooja network simulator. Cooja provides a simulation environment that allows developers to test their wireless networks, how their applications run in large-scale network before being deployed on actual devices.

Furthermore, Contiki uses a mechanism called protothreads, which is a mixture of the event-driven and the multi-threaded programming mechanisms that can save memory and providing nice control flow in the code. With protothreads, event-handlers can be made to block, waiting for events to occur. Contiki provides a lightweight flash file system called Coffee. With which, application programs can open, close, read from, write to, and append to files on the external flash, without having to worry about flash sectors needing to be erased before writing or flash wear-leveling.

Another feature that Contiki provides is tailored wireless networking stack called Rime. The Rime stack supports simple operations such as sending a message to all neighbors or to a specified neighbor, as well as more complex mechanisms such as network flooding and address-free multi-hop semi-reliable scalable data collection. Everything runs with sleepy routers to save power. [11]

2.1.4 Comparison of Current IoT OS's

Although, there are many different operating systems which can be used for Internet of Things constricted devices, the three operating systems (TinyOS, RIOT, and Contiki) which have been briefly described are

the ones that are around for a long time and have been mostly used. Of course, there are differences between them, the most important ones being the following (Table 1).

OS	Min RAM	Min ROM	C Support	C++ Support	Multi-Threading	MCU w/o MMU	Modularity	Real-Time
Contiki	< 2kB	< 30kB	●	✗	●	✓	●	●
Tiny OS	< 1kB	< 4KB	✗	✗	●	✓	✗	✗
RIOT	~ 1.5kB	~ 5kB	✓	✓	✓	✓	✓	✓

Table 1. IoT OS's Differences [12]

Full support ✓
Partial support ●
No support ✗

As highlighted on Table 1 RIOT is the lightest operating system while it fully supports all basic needs of an OS. The biggest difference is with TinyOS which is written in nesC, a dialect of C contrary to the other two which support C (Contiki), C and C++ (RIOT) which are some of the most popular programming languages.

There are some other differences between RIOT and Contiki, related to the hardware they support and the, different types of nodes, generally there are some common nodes supported by both OSs' although Contiki has more options available. Another difference is the platform which they support, for example Contiki supports the 20-bit MSP430 platform which is necessary for more memory consumption protocols like MQTT and DTLS. The biggest difference is the Cooja simulator provided by Contiki for testing wireless networks based on the available hardware. [12]

2.2 PROTOCOLS

Standard network protocols like UDP, TCP, TLS, IPv4 and IPv6 are important for communication through the internet. Especially for IPv6 and UDP, which are important for the Internet of Things as they are highly used on constricted devices. Low-power standard protocols have been developed based on UDP, TCP, TLS IPv4/Ipv6 to be used on IoT devices, this protocols are MQTT, which is based on TCP communication, CoAP based on UDP, tinyDTLS which is based on DTLS (Datagram TLS for UDP communication) and 6LowPAN which is based on IPv6.

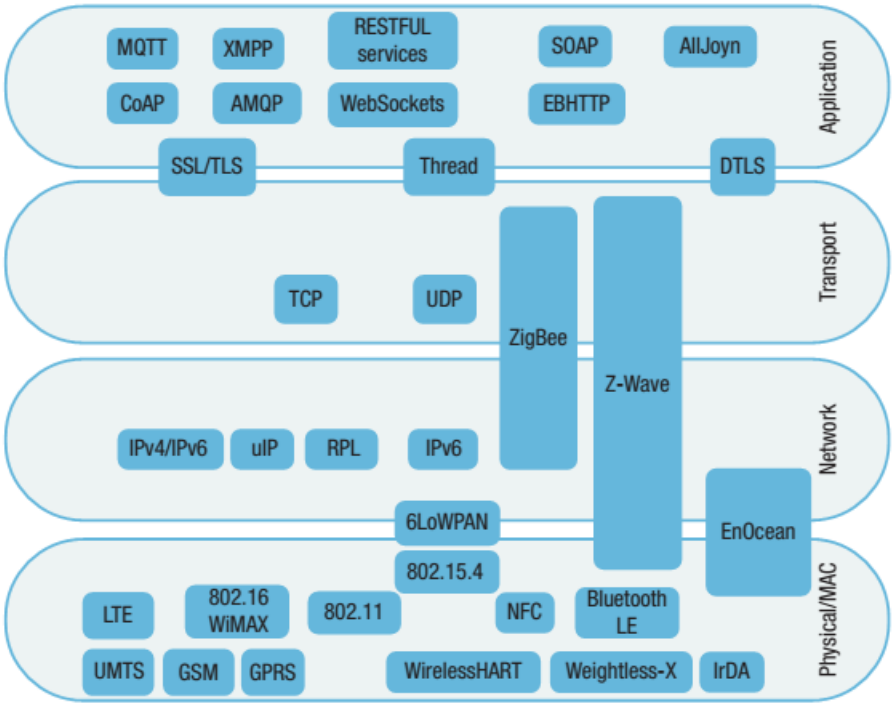

Figure 1. Common protocols used in the IoT [3]

2.2.1 MQTT

MQTT was developed by Andy Stanford-Clark (IBM) and Arlen Nipper (Eurotech; now Cirrus Link) in 1999 for the monitoring of an oil pipeline through the desert. The goals were to have a protocol, which is bandwidth-efficient and uses little battery power, because the devices were connected via satellite link and this was extremely expensive at that time.

The protocol uses a publish/subscribe architecture in contrast to HTTP with its request/response paradigm. Publish/Subscribe is event-driven and enables messages to be pushed to clients. The central communication point is the MQTT broker, it is in charge of dispatching all messages between the senders and the rightful receivers. Each client that publishes a message to the broker, includes a topic into the message. The topic is the routing information for the broker. Each client that wants to receive messages subscribes to a certain topic and the broker delivers all messages with the matching topic to the client. Therefore the clients don't have to know each other, they only communicate over the topic. This architecture enables highly scalable solutions without dependencies between the data producers and the data consumers.

The difference to HTTP is that a client doesn't have to pull the information it needs, but the broker pushes the information to the client, in the case there is something new. Therefore each MQTT client has a permanently open TCP connection to the broker. If this connection is interrupted by any circumstances, the MQTT broker can buffer all messages and send them to the client when it is back online. [13, 14, 15, 17]

2.2.2 CoAP

CoAP, on the other hand, is a new standard developed by the IETF Constrained Resource Environments (CoRE) group that is often described as a lightweight analog HTTP. Highly scalable and REST-savvy, CoAP

trades off the transmission guarantees of TCP (used by MQTT) for the smaller packets and lower overhead of UDP. CoAP employs a client-server model and request/response message pattern, where client devices send information requests directly to server devices, which then respond. Support for an Observer message pattern enables clients to receive an update whenever a requested state changes, for example a valve opening or closing, while confirmed message delivery provides some level of assurance under the connectionless UDP transport.

At the end of the day, the decision of what protocol to adopt depends entirely on the specifics of your particular device deployment. In some cases, the hub-and-spoke, brokered architecture of MQTT may offer advantages, while in others the decentralized approach employed by CoAP may be best. In the same vein, CoAP's UDP foundation is generally friendlier to battery-powered devices, while MQTT based on TCP can offer greater assurance of message delivery. [4, 17]

2.2.3 6LoWPAN

6LoWPAN is an acronym of IPv6 over Low power Wireless Person Network, with the concept originated that low-power devices with limited processing capabilities should be able to participate in the IoT and the need of IP address were important for the appropriate function of those devices. Generally, 6LoWPAN is IPv6 modified for low powered WPAN communication over the IEEE 802.15.4 standard, with the role of routing resource-constrained nodes over low-powered and lossy networks.

6LoWPAN has been created for devices with typical characteristics such as having limited processing capabilities, with small memory capacity, working on low power consumption for short range communication and all that for a low cost. Important characteristics of the protocol is the small packet size mostly because of the access control layer and link layer security overhead imposed to the data packets. Also, using the IEEE 802.15.4 standard which allows both 64-bit extended addresses and 16-bit addresses unique within the PAN (Personal Area Network). Another characteristic is the low bandwidth and it includes both star and mesh topologies, furthermore, it provides the sleeping mode as many devices may sleep for prolonged periods of time in order to save energy, which makes them unable to communicate during these sleep periods.

IPv6 networking provides some benefits to the 6LoWPAN such as:

- The pervasive nature of IP networks allows leveraging existing infrastructure.
- IP-based technologies already exist, are well-known, proven to be working and widely available. This allows for an easier and cheaper adoption, good interoperability and easier application layer development.
- IP networking technology is specified in open and freely available specifications, which is able to be better understood by a wider audience than proprietary solutions.
- Tools for IP networks already exist.
- IP-based devices can be connected readily to other IP-based networks, without the need for intermediate entities like protocol translation gateways or proxies.
- The use of IPv6, specifically, allows for a huge amount of addresses and provides for easy network parameters auto-configuration (SLAAC). This is paramount for 6LoWPANs where large number of devices should be supported.

However using IP communication in 6LoWPAN raises some issues which are:

- **IP Connectivity**: One of the characteristics of 6LoWPANs is the limited packet size, which implies that headers for IPv6 and layers above must be compressed whenever possible.
- **Topologies**: 6LoWPANs must support various topologies including mesh and star: Mesh topologies imply multi-hop routing to a desired destination. In this case, intermediate devices act as packet forwarders at the link layer. Star topologies include provisioning a subset of devices with packet forwarding functionality. If, in addition to IEEE 802.15.4, these devices use other kinds of network interfaces such as Ethernet or IEEE 802.11, the goal is to seamlessly integrate the networks built over those different technologies. This, of course, is a primary motivation to use IP to begin with.
- **Limited Packet Size**: Applications within 6LoWPANs are expected to originate small packets. Adding all layers for IP connectivity should still allow transmission in one frame, without incurring excessive fragmentation and reassembly. Furthermore, protocols must be designed or chosen so that the individual "control/protocol packets" fit within a single 802.15.4 Frame.
- **Limited Configuration and Management**: Devices within 6LoWPANs are expected to be deployed in exceedingly large numbers. Additionally, they are expected to have limited display and input capabilities. Furthermore, the location of some of these devices may be hard to reach. Accordingly, protocols used in 6LoWPANs should have minimal configuration, preferably work "out of the box", be easy to bootstrap, and enable the network to self-heal given the inherent unreliable characteristic of these devices.
- **Service Discovery**: 6LoWPANs require simple service discovery network protocols to discover, control and maintain services provided by devices.
- **Security**: IEEE 802.15.4 mandates link-layer security based on AES, but it omits any details about topics like bootstrapping, key management, and security at higher layers. Of course, a complete security solution for 6LoWPAN devices must consider application needs very carefully.

The goals that 6LoWPAN wants to accomplish are:

- Fragmentation and Reassembly layer: IPv6 specification [RFC2460] establishes that the minimum MTU that a link layer should offer to the IPv6 layer is 1280 bytes. The protocol data units may be as small as 81 bytes in IEEE 802.15.4. To solve this difference a fragmentation and reassembly adaptation layer must be provided at the layer below IP.
- Header Compression: Given that in the worst case the maximum size available for transmitting IP packets over an IEEE 802.15.4 frame is 81 octets, and that the IPv6 header is 40 octets long, (without optional extension headers), this leaves only 41 octets for upper layer protocols, like UDP and TCP. UDP uses 8 octets in the header and TCP uses 20 octets. This leaves 33 octets for data over UDP and 21 octets for data over TCP. Additionally, as pointed above, there is also a need for a fragmentation and reassembly layer, which will use even more octets leaving very few octets for data. Thus, if one were to use the protocols as is, it would lead to excessive fragmentation and reassembly, even when data packets are just 10s of octets long. This points to the need for header compression.
- Address Auto-configuration: specifies methods for creating IPv6 stateless address auto configuration (in contrast to stateful) that is attractive for 6LoWPANs, because it reduces the configuration overhead on the hosts. There is a need for a method to generate the IPv6 IID (Interface Identifier) from the EUI-64 assigned to the IEEE 802.15.4 device.

- Mesh Routing Protocol: A routing protocol to support a multi-hop mesh network is necessary. Care should be taken when using existing routing protocols (or designing new ones) so that the routing packets fit within a single IEEE 802.15.4 frame. The mechanisms defined by 6lowpan IETF WG are based on some requirements for the IEEE 802.15.4 layer: IEEE 802.15.4 defines four types of frames: beacon frames, MAC command frames, acknowledgement frames and data frames. IPv6 packets must be carried on data frames.
- Data frames may optionally request that they be acknowledged. It is recommended that IPv6 packets be carried in frames for which acknowledgements are requested so as to aid link-layer recovery.
- The specification allows for frames in which either the source or destination addresses (or both) are elided. Both source and destination addresses are required to be included in the IEEE 802.15.4 frame header.
- The source or destination PAN ID fields may also be included. 6LoWPAN standard assumes that a PAN maps to a specific IPv6 link.
- Both 64-bit extended addresses and 16-bit short addresses are supported, although additional constraints are imposed on the format of the 16-bit short addresses.
- Multicast is not supported natively in IEEE 802.15.4. Hence, IPv6 level multicast packets must be carried as link-layer broadcast frames in IEEE 802.15.4 networks. This must be done such that the broadcast frames are only heeded by devices within the specific PAN of the link in question. [3]

2.2.4 DTLS & tinyDTLS

DTLS is an adaption of TLS for datagram based communication which provides equivalent security guarantees. It supports both client and server authenticated handshakes, also DTLS reuses almost all the protocol elements of TLS, with minor but important modifications for it to work. TLS depends on a subset of TCP features: reliable, in-order packet delivery and replay detection. Unfortunately, all of these features are absent from datagram transport.

The general idea of DTLS is that the client and the server want to send each other a lot of data as "datagrams", it wants to send a long sequence of bytes, with a defined order, but do not enjoy the luxury of TCP. TCP provides a reliable bidirectional tunnel for bytes, where all bytes eventually reach the receiver in the same order as what the sender used; TCP achieves that through a complex assembly of acknowledge messages, transmission timeouts, and reemissions. This allows TLS to simply assume that the data will go unscathed under normal conditions. On the other hand, DTLS works over datagrams which can be lost, duplicated, or received in the wrong order. To cope with that, DTLS uses some extra mechanisms and some extra leniency.

Main differences between TLS and DTLS are:

- Explicit records. With TLS, you have one long stream of bytes, which the TLS implementation decides to split into records as it sees fit; this split is transparent for applications. Not so with DTLS: each DTLS record maps to a datagram. Data is received and sent on a record basis, and a record is either received completely or not at all. Also, applications must handle path MTU discovery themselves.
- Explicit sequence numbers. TLS records include a MAC which guarantees the record integrity, and the MAC input includes a record sequence number which thus verifies that no record has been lost, duplicated or reordered. In TLS, this sequence number (a 64-bit integer) is implicit. In DTLS,

the sequence number is explicit in each record (8-bit overhead). The sequence number is furthermore split into a 16-bit "epoch" and a 48-bit subsequence number, to better handle cipher suite renegotiations.

- Alterations are tolerated. Datagrams may be lost, duplicated, reordered, or even modified. This is a "fact of life" which TLS would abhor, but DTLS accepts. Thus, both client and server are supposed to tolerate a bit of abuse; records which are "a bit early" are simply dropped. This means that DTLS implementation does not distinguish between normal "noise" (random errors which can occur) and an active attack.
- Stateless encryption. Since records may be lost, encryption must not use a state which is modified with each record. In practice, this means no RC4.
- No verified termination. DTLS has no notion of a verified end-of-connection like what TLS does with the close_notify alert message. This means that when a receiver ceases to receive datagrams from the peer, it cannot know whether the sender has voluntarily ceased to send, or whether the rest of the data was lost. It is up to whatever data format which is transmitted within DTLS to provision for explicit termination, if is needed.
- Fragmentation and reemission. Handshake messages may exceed the natural datagram length, and thus may be split over several records. The syntax of handshake messages is extended to manage these fragments. Fragment handling requires buffers, therefore DTLS implementations are likely to require a bit more RAM than TLS implementations.
- Protection against DoS/spoof. Since a datagram can be sent "as is", it is subject to IP spoofing: an evildoer can send a datagram with a fake source address. In particular a ClientHello message. If the DTLS server allocates resources when it receives a ClientHello, then there is ample room for DoS. In the case of TLS, a ClientHello occurs only after the three-way handshake of TCP is completed, which implies that the client uses a source IP address that it can actually receive. Being able to DoS a server without showing your real IP is a powerful weapon; hence DTLS includes an optional defense. The defensive mechanism in DTLS is a "cookie": the client sends its ClientHello, to which the server responds with a HelloVerifyRequest message which contains an opaque cookie, which the client must send back as a second ClientHello. The server should arrange for a type of cookie which can be verified without storing state; this cookie mechanism is really an emulation of the TCP three-way handshake (Figure 2). [24]

Using DTLS as the sole security suite for IoT, the following security protection can be achieved.

- **Network Access**: DTLS as an authentication protocol that can be used to authenticate new devices joining the network either using the PSK mode, raw public key, or public key certificate. The result of a successful DTLS handshake creates a secure channel between the new device and the authorizing entity. This secure channel enables the authorizing entity to distribute the session key securely to the joining device based on rules which have been configured by the network owner. If the new device and the authorizing entity are one-hop at the MAC layer, then the DTLS handshake messages are not dropped by the MAC layer. However, if new device and the authorizing entity are multi-hop at the MAC layer, the DTLS messages are dropped at the MAC layer since they are not yet protected with the session key.
- **Secure Communication channel**: a DTLS end-to-end session can be established between two communicating devices, one inside the 6LoWPAN and the other outside, to securely transport application data (CoAP messages). The application data is protected by the DTLS record layer,

authenticated and encrypted with a fresh and unique session key. DTLS consists of two layers: the lower layer contains the Record protocol and the upper layer contains either of the three protocols namely Handshake, Alert, and ChangeCipherSpec, or application data. The Record protocol is a carrier for the upper layer protocols. The Record header contains among others content type and fragment fields. Based on the value in the content type, the fragment field contains either the Handshake protocol, Alert protocol, ChangeCipherSpec protocol, or application data. The Record header is primarily responsible to cryptographically protect the upper layer protocols or application data once the handshake process is completed. The Record protocols protection includes confidentiality, integrity protection and authenticity. Handshake protocol is a complex chatty process and contains numerous message exchanges in an asynchronous fashion. The handshake messages, usually organized in flights, are used to negotiate security keys, cipher suites and compression methods (Figure 2).

- **Key Management**: As DTLS has the capability of renewing session keys, this mechanism can be utilized to support key management in a 6LoWPAN network. During the Network Access phase, the 6LBR distributes a primary key as part of the network access authentication procedure. It is thus possible to reuse the same channel to facilitate key management, thus enabling the primary key to be updated by the 6LBR when necessary.

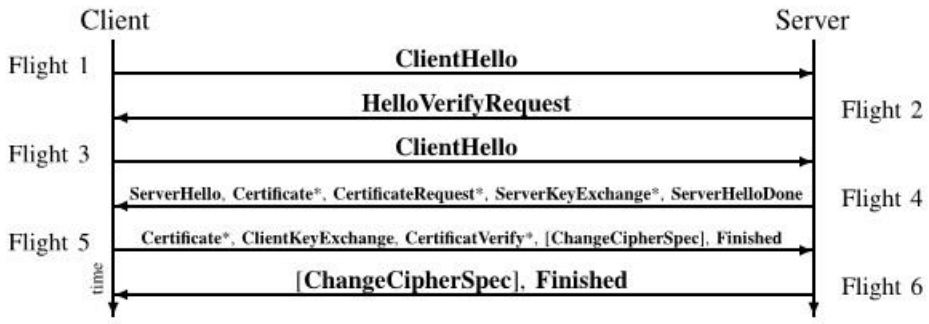

Figure 2. DTLS Handshake protocol

To sum up, DTLS extra features are conceptually imports from TCP (receive window, reassembly with sequence numbers, reemissions, connection cookie...) thrown over a normal TLS (the one important omission is the lack of acknowledge messages). The protocol is more lenient with regards to alterations, and does not include a verified "end-of-transmission". [24]

TinyDTLS is a software library that provides a very simple datagram server with DTLS support. It is designed to support session multiplexing in single-threaded applications and thus specifically targets embedded systems. Its salient features include:

- Basic support for DTLS with PSK only
- No support for public key cryptography
- Supports HMAC-SHA256
- Supports Rijndael (AES)
- Supports clock handling, NetQ, PN number generation

The TinyDTLS implementation library consists of the following core modules:

- **DTLS**: State machine used to establish a DTLS session, Handshake protocol definition, Structure of different messages used by the protocol
- **Cryptography**: Encryption operations, Decryption operations
- Keyed-Hash Message Authentication Code (HMAC): Algorithm used for message authentication
- **Counter with Cipher Block Chaining Message**
- **Authentication Code (Counter with CBC-MAC or CCM)**: Actual implementation of encryption function, Actual implementation of decryption function
- **Rijndael Cipher**: Implementation of AES
- **Secure Hash Algorithm (SHA-2)**: Implementation of a set of cryptographic hash functions

TinyDTLS contains all the logic required to handle secure communications including data sessions, handshake protocol definition and structures of different messages belonging to the security protocol. The Crypto module handles all the authentication and encrypt/decrypt operations. As a lightweight DTLS implementation, the crypto component supports only DTLS_PSK_WITH_AES_128_CBC_SHA-256, which is composed of the pre-shared key exchange algorithm and the 128 bit AES algorithm in CCM mode. Message integrity and a second check for message authentication is achieved by the HMAC component, which calculates a MAC through the SHA-256 function in combination with a secret cryptographic key generated from the master secret realized during the handshake phase. The Memory Allocation module allocates memory to peers as well as help freeing memory allocated to them. The Network Packet Queue (NetQ) utility functions implement an ordered queue of data packets to send over the network and can also be used to queue received packets from the network. [4, 5, 6, 7, 8]

2.3 PLATFORMS

There are many platforms that can be used for simulation of IoT environments depending on the needs and the goals that must be accomplished. Two renowned platforms of the zolertia family are the *Z1 mote* and *RE-Mote* which have great features giving them the possibility to accomplish more demanding tasks.

2.3.1 Zolertia Z1 mote

The Z1 platform is a general purpose development platform for wireless sensor networks (WSN) designed for researchers, developers, enthusiasts and hobbyists. Z1 is equipped with a second generation MSP430F2617 low power microcontroller, which features a powerful 16-bit RISC CPU @16MHz clock speed, built-in clock factory calibration, 8KB RAM and a 92KB Flash memory. Also includes the well-known CC2420 transceiver, IEEE 802.15.4 compliant, which operates at 2.4GHz with an effective data rate of 250Kbps. Z1 hardware selection guarantees the maximum efficiency and robustness with low energy cost.

Z1 platform features:

- Ultra-Low Power MCU and 2.4GHz Transceiver
- 2 x Digital Built-in sensors (temperature and 3-axis accelerometer)
- USB Programming Ready
- Supported in Open Source Operative Systems as Contiki, RIOT, tinyOS, MansOS and OpenWSN.
- Flexible Powering: Battery Pack (2xAA or 2xAAA), Coin Cell (up to 3.6V), USB Powered, Directly Connected through two wires coming from a power source. USB VCC and GND pins are available on the digital buses expansion port. You can connect to this pins any power of source from 4V to 5.25V and it will be regulated to 5V and 3V. [21]

2.3.2 Zolertia RE-Mote

The RE-Mote is a hardware development platform designed jointly with universities and industrial partners, in the frame of the European research project RERUM (Reliable, Resilient and secure IoT for smart city applications). The RE-Mote aims to fill the gap of existing IoT platforms lacking an industrial-grade design and ultra-low power consumption, yet allowing makers and researchers alike to develop IoT applications and connected products. The platform is based in the Texas Instruments CC2538 ARM Cortex-M3 system on chip (SoC), with an on-board 2.4 GHz IEEE 802.15.4 RF interface, running at up to 32 MHz with 512 KB of programmable flash and 32 KB of RAM, bundled with a Texas Instruments CC1200 868/915 MHz RF transceiver to allow dual band operation.

RE-Mote features:

- ISM 2.4-GHz IEEE 802.15.4 & Zigbee compliant radio.
- ISM 863-950-MHz ISM/SRD band IEEE 802.15.4 compliant radio.
- ARM Cortex-M3 32 MHz clock speed, 512 KB flash and 32 KB RAM (16 KB retention)
- AES-128/256, SHA2 Hardware Encryption Engine
- ECC-128/256, RSA Hardware Acceleration Engine for Secure Key Exchange
- User and reset button
- Consumption down to 150 nA using the shutdown mode.
- Programming over BSL without requiring to press any button to enter bootloader mode.
- Built-in battery charger (500 mA), facilitating Energy Harvesting and direct connection to Solar Panels and to standards LiPo batteries.
- Wide range DC Power input: 3.3-16 V.
- Small form-factor (73 x 40 mm).
- MicroSD (over SPI).
- On board RTCC (programmable real time clock calendar) and external watchdog timer (WDT).
- Programmable RF switch to connect an external antenna either to the 2.4 GHz or to the Sub 1 GHz RF interface through the RP-SMA connector.
- Supported in Open Source Operative Systems as Contiki, RIOT and OpenWSN (in progress). [22]

3 BUILDING THE TESTING ENVIRONMENT

For evaluating the different protocols and the hardware used for developing IoT applications and wireless environments it is necessary to build an appropriate testing environment. The base of the environment created is an Ubuntu 16.04 virtual machine, alongside it was used Contiki version 3.0 operating system with an msp430-gcc experimental compiler version 4.7. This version of the compiler is necessary for the reason that it reduces the memory ROM used of the nodes.

3.1 CONTIKI INSTALLATION

There are alternative ways to install Contiki from scratch by installing from sources or using virtual environments. To work with Contiki three things are necessary.

- The Contiki source code.
- A target platform.
- A toolchain to compile the source code for such target platform.

First thing to do is to install the toolchain needed to compile the source code of Contiki and applications created with Contiki OS. For the Linux Ubuntu (version 12.04 and above) virtual machine to install the toolchain and required dependencies it is needed to run in a terminal the following commands:

```
$ sudo dpkg -r texinfo
$ sudo apt-get update
$ sudo apt-get install gcc-arm-none-eabi gdb-arm-none-eabi
$ sudo apt-get -y install build-essential automake gettext
$ sudo apt-get -y install gcc-arm-none-eabi curl graphviz unzip wget
$ sudo apt-get -y install gcc gcc-msp430
$ sudo apt-get -y install openjdk-7-jdk openjdk-7-jre ant (for latest version
of Ubuntu Java jdk and jre version 8 is needed)
```

Next step is to install the Contiki source code which can be installed by using a terminal and typing the following commands.

```
$ sudo apt-get -y install git
$ git clone --recursive https://github.com/contiki-os/contiki.git
```

The first command is to install the git control system and the second one is to clone the Contiki source code from GitHub. Additionally, to the master branch included in the Contiki repository it is recommended to install the iot-workshop branch to keep track of the development done in the master branch of any changes and updates. This branch has already examples and applications to guide through Contiki and building real IoT applications. Again as with the previous steps in a terminal running the following commands are needed.

```
$ cd (path to contiki folder)
$ git remote add iot-workshop https://github.com/alignan/contiki
```

```
$ git fetch iot-workshop
$ git checkout iot-workshop
```

3.2 MSP430-GCC INSTALLATION

Installing the msp430-gcc version 4.7 experimental compiler has its own problems. Basically, the problem is that the installation must be done manually following specific steps to have a successful installation. There are a few ways of succeeding in the installation of the compiler that can be found because the compiler depends on some specific components. To make the installation successful is better to build it on an older version of Ubuntu, preferable 12.04 or 14.04 version depending on which solution[1] is followed. Below is a step by step guide of how to install and configure the msp430-gcc version 4.7 experimental compiler.

To begin the installation is preferable to create a folder where all downloaded files and patches will be placed. So the commands to start are the following:

```
$ cd~
$ mkdir #new folder#
$ cd #new folder#
```

The above commands create a folder on the home directory and after that changing the directory to the newly created folder. Next step is to install all dependencies needed for the compiler. The required commands to download and install are shown below.

```
$ sudo apt-get install patch
$ sudo apt-get install ncurses-dev
$ sudo apt-get install build-essential
$ sudo apt-get install bison
$ sudo apt-get install flex
$ sudo apt-get install zlib1g-dev
$ sudo apt-get install sed
$ sudo apt-get install automake
$ sudo apt-get install gawk
$ sudo apt-get install mawk
$ sudo apt-get install libusb-1.0.0
$ sudo apt-get install libusb-1.0.0-dev
$ sudo apt-get install dos2unix
$ sudo apt-get install srecord
```

[1] Links to different solutions. https://lab11.eecs.umich.edu/wiki/doku.php?id=msp430:mspgcc:start, https://richardnaud.wordpress.com/2013/10/02/installation-of-eclipse-cdt-msp430-gcc-4-7-0-mspdebug-0-21-and-more-on-ubuntu/, https://github.com/contiki-os/contiki/wiki/MSP430X#MSP430X, https://github.com/tecip-nes/contiki-tres/wiki/Building-the-latest-version-of-mspgcc, https://m8051.blogspot.gr/2012/07/compiling-msp430-gcc-in-linux-upto-date.html, http://colotronics.blogspot.gr/2014/08/msp430-toolchain-in-ubuntu-1404-with.html.

After installing the dependencies which are required for the proper function of the compiler next is to download all basic components needed. The reason why it is better to use an older version of Ubuntu is because of these components as it creates multiple errors installing them when on a newer version of Ubuntu.

```
$ wget http://sourceforge.net/projects/mspgcc/files/mspgcc/DEVEL-4.7.x/mspgcc-20120911.tar.bz2
$ wget http://sourceforge.net/projects/mspgcc/files/msp430mcu/msp430mcu-20130321.tar.bz2
$ wget http://sourceforge.net/projects/mspgcc/files/msp430-libc/msp430-libc-20120716.tar.bz2
$ wget http://ftpmirror.gnu.org/binutils/binutils-2.22.tar.bz2
$ wget http://ftp.gnu.org/pub/gnu/gcc/gcc-4.7.0/gcc-4.7.0.tar.bz2
$ wget http://ftp.gnu.org/pub/gnu/gdb/gdb-7.2a.tar.bz2
$ wget http://sourceforge.net/p/mspgcc/bugs/_discuss/thread/fd929b9e/db43/attachment/0001-SF-357-Shift-operations-may-produce-incorrect-result.patch
$ wget http://sourceforge.net/p/mspgcc/bugs/352/attachment/0001-SF-352-Bad-code-generated-pushing-a20-from-stack.patch
$ wget -O gdb.patch https://sourceware.org/git/?p=gdb.git;a=patch;h=7f62f13c2b535c6a23035407f1c8a36ad7993dec
```

Furthermore, *texinfo* may create additional errors while installing some of the patches, the simplest solution to this problem is to downgrade texinfo on the virtual machine before installing all components and patches needed for the compiler. The following commands downloads an older version of texinfo, removes the one already installed and installs an older version that does not create any problems. After the successful installation of msp430-gcc compiler it can be updated to it latest version without causing any future problem.

```
$ wget http://ftp.br.debian.org/debian/pool/main/t/texinfo/texinfo_4.13a.dfsg.1-10_amd64.deb
$ sudo dpkg -r texinfo
$ sudo dpkg -i texinfo_4.13a.dfsg.1-10_amd64.deb
```

After downloading all the necessary patches and components, which are compressed, next is to extract all files by using the *"tar xvfj"* command.

```
$ tar xvfj mspgcc-20120911.tar.bz2
$ tar xvfj binutils-2.22.tar.bz2
$ tar xvfj gcc-4.7.0.tar.bz2
$ tar xvfj gdb-7.2a.tar.bz2
$ tar xvfj msp430mcu-20130321.tar.bz2
$ tar xvfj msp430-libc-20120716.tar.bz2
```

For a better and cleaner installation it is suggested to create individual folders for each component of the compiler in case of errors during installation.

```
$ mkdir build
$ cd build
```

```
$ mkdir binutils
$ mkdir gcc
$ mkdir gdb
$ cd ..
```

First thing to be installed is the *binutils version 2.22,* by changing directory in the folder extracted, after that follows the installation the corresponding patch and configuring the target and prefix we need, and later using the *make* command to compile it.

```
$ cd binutils-2.22
$ patch -p1<../mspgcc-20120911/msp430-binutils-2.22-20120911.patch
$ cd ../build/binutils
$ ../../binutils-2.22/configure --target=msp430 --prefix=/usr/local/msp430 2>&
1 | tee co
$ make 2>&1 | tee mo
$ sudo make install 2>&1 | tee moi
```

Next is the *gcc version 4.7.0* which has three different patches, the first patch is related to the compiler and the next two are to fix some problems that may occur due to the installation of the first patch.

```
$ cd ../../gcc-4.7.0
$ patch -p1 < ../mspgcc-20120911/msp430-gcc-4.7.0-20120911.patch
$ patch -p1< ../0001-SF-352-Bad-code-generated-pushing-a20-from-stack.patch
$ patch -p1< ../0001-SF-357-Shift-operations-may-produce-incorrect-result.patc
h
$ ./contrib/download_prerequisites
```

After that follows the replacement of the *ira-int.h* file and the configuration of the gcc in the same way as binutils.

```
$ cd gcc
$ rm ira-int.h
$ wget -O ira-int.h https://gcc.gnu.org/viewcvs/gcc/branches/gcc-4_7-branch/gc
c/ira-int.h?revision=191605&view=co&pathrev=191605
$ cd ../../build/gcc
$ ../../gcc-4.7.0/configure --target=msp430 --enable-languages=c,c++ --prefix=
/usr/local/msp430 2>&1 | tee co
$ make 2>&1 | tee mo
$ sudo make install 2>&1 | tee moi
```

If until now the process is error free then next action is to update the PATH environment for the newest msp430 compiler so that when compiling any application the compiler used will be the experimental one (msp430-gcc version 4.7).

```
$ export PATH=/usr/local/msp430/bin/:$PATH
$ sudo sed -e '/^PATH/s/"$/:\/usr\/local\/msp430\/bin"/g' -i /etc/environment
```

Next step is to check that everything went well with the installation of msp430-gcc experimental compiler and that the environment PATH has been updated. To do that is to check the version of msp430 compiler which must return version 4.7.0 20120322. The command for checking the compiler's version is:

```
$ msp430-gcc --version
```

If everything until now is successful, than next follows the installation of the remaining parts needed for the compiler to be fully function. Next thing on the list to be installed is gdb following the same steps with the previous components.

```
$ cd ../../gdb-7.2
$ patch -p1 < ../mspgcc-20120911/msp430-gdb-7.2a-20111205.patch
$ patch -p1< ../gdb.patch
$ cd ../build/gdb
$ ../../gdb-7.2/configure --target=msp430 --prefix=/usr/local/msp430 2>&1 | te
e co
$ make 2>&1 | tee mo
$ sudo make install 2>&1 | tee moi
```

To check that gdb and the patch were installed successfully is the same as with the msp430-gcc only now it is for msp430-gdb and the version it should have is 7.2.

```
$ msp430-gdb --version
```

The next and last two components to be installed are the msp430mcu and msp430-libc, as with the previous ones they follow the same steps.

```
$ cd ../../msp430mcu-20130321/
$ sudo MSP430MCU_ROOT=`pwd` ./scripts/install.sh /usr/local/msp430 | tee so
$ cd ../msp430-libc-20120716/src/
$ make 2>&1 | tee mo
$ sudo PATH=$PATH make PREFIX=/usr/local/msp430 install 2>&1 | tee moi
$ cd ../..
```

Finally, is to remove the old texinfo and to install the latest version available as now it should not create any problems with the function of the experimental compiler (msp430-gcc version 4.7).

```
$ sudo dpkg -r texinfo
$ sudo apt-get install texinfo
```

After the successful installation of the experimental compiler msp430-gcc version 4.7 the folder created at the beginning where all components and patches have been stored can be deleted. Also, the virtual machine can be updated from the oldest version (12.04 or 14.04) to the newest available version. If the entire procedure of creating the virtual machine, installing the Contiki OS and msp430-gcc experimental compiler seem problematic then there is already a virtual machine for VMware workstation to be

downloaded[2]. It is a Linux Ubuntu 16.04 virtual machine with the latest Contiki, including the iot-workshop branch, and msp430 compiler.

[2] VM website: https://sourceforge.net/projects/zolertia/files/VM/

4 EXPERIMENTS AND RESULTS

With the testing environment successfully established, what follows is to develop the required testing applications. Implementing any kind of application, being either CoAP, MQTT or a simple UDP application have some common structure like having a project configuration file. It is also necessary to have the Makefile.h, Makefile.include and the Makefile.traget which specifies the platform targeted.

A Makefile.h is a file containing a set of directions used with the make build automation tool. The main purpose of a Makefile.h is to reduce build times if only a few source files have changed. Also, the Makefile.h links the source code to the project configuration file (project-conf.h) and Makefile.include, both files include general specifics for applications and can be used by multiple applications.

First part of the experiment is related to testing, understanding and developing a client application based on the MQTT protocol. The second part is to modify the tinyDTLS protocol for the Contiki OS and develop a client and server application based on Contiki OS for secure message exchanges. To make the experiments what were used are the Contiki OS and the Cooja simulation to test the applications. Specifically for the MQTT protocol experiment is necessary to use a broker as server (*mosquitto*[3] *broker*) and the Cooja simulation for the client examples. For the tinyDTLS protocol both the client and the server are tested with Cooja simulation. The reason is to see if it is possible to combine the MQTT and DTLS protocols and develop a more secure communication method for a WSN.

4.1 MQTT TESTING

For the proper function of the MQTT protocol it is important to have a broker to which the local wireless network has to connect. A broker is a server which is configured to support the MQTT protocol and all functions related to the protocol. The one used during the testing is the mosquitto broker a simple broker to install and use. By default, mosquitto does not need a specified configuration file and will use default values when started. If changes are needed to manipulate the broker's functionality that can be accomplished by creating a personal configuration file (mosquito.conf[4]). To start the mosquitto broker all that is needed is the start command shown in "1" and in case of using a personal configuration file a simple modification to the command specifying the path and name of the configuration file "2". The –v signifies the verbose mode which shows the packets exchanged between server and client.

```
1. $ sudo mosquitto –v
2. $ sudo mosquitto –v –c "/configuration_file_path/configuration_ file_name"
```

The mosquitto broker is the server running in a terminal of the virtual machine. MQTT client, based on Contiki OS, is done in C by following a specific structure so that a node could function properly (Figure 3). Though it follows the basic principles of C program language it has different libraries and standard functions. Instead of the *main()* function, which characterizes the C programming language, for Contiki OS applications the main function is PROCESS_THREAD().

[3] Mosquitto broker web site: https://mosquitto.org/man/mosquitto-8.html
[4] Mosquitto configuration file: https://mosquitto.org/man/mosquitto-8.html

As with traditional C programming language, programming a Contiki OS application, the command *include* is used to identify which libraries are required, the *define* command is used for stable variables such as IP address, port or ID etc. Generally, a Contiki OS application is a process thread invoking the different functions of nodes (Figure 3).

```
/*---------------------------------------------------------------------------*/
/* This is the main contiki header, it should be included always */
#include "contiki.h"

/* This is the standard input/output C library, used to print information to the
 * console, amongst other features
 */
#include <stdio.h>
/*---------------------------------------------------------------------------*/
/* We first declare the name or our process, and create a friendly printable
 * name
 */
PROCESS(hello_world_process, "Hello world process");

/* And we tell the system to start this process as soon as the device is done
 * initializing
 */
AUTOSTART_PROCESSES(&hello_world_process);
/*---------------------------------------------------------------------------*/
/* We declared the process in the above step, now we are implementing the
 * process.  A process communicates with other processes and, thats why we
 * include as arguments "ev" (event type) and "data" (information associated
 * with the event, so when we receive a message from another process, we can
 * check which type of event is, and depending of the type what data should I
 * receive
 */
PROCESS_THREAD(hello_world_process, ev, data)
{
  /* Every process start with this macro, we tell the system this is the start
   * of the thread
   */
  PROCESS_BEGIN();

  static uint16_t num = 0xABCD;
  static const char *hello = "Hello world, again!";

  /* Now we are printing this message to the console over the USB port, use
   * "make login" and hit the reset button to catch this message, as well as the
   * booting information of the node
   */
  printf("Hello, world\n");

  /* We might as well also use */
  printf("%s\n", hello);

  /* And mix numeric values with strings */
  printf("This is a value in hex 0x%02X, the same as %u\n", num, num);

  /* This is the end of the process, we tell the system we are done */
  PROCESS_END();
}
/*---------------------------------------------------------------------------*/
```

Figure 3. Node Application Example

4.1.1 The Makefiles

A Contiki project has two makefiles, the most important one is the Makefile.h and the second file is the Makefile.target. The Makefile.target is simply for specifying the targeted platform to which the application should compile.

```
1  TARGET = z1
```

Figure 4. Makefile.target

The second makefile, Makefile.h, is much more important as it is what defines the base of the project. It directs the *make* command on how to compile and link the application. It contains the dependencies of the project and helps at minimizing the time and complexity of the project if changes have been made,

especially after the first compiling of the project. In Figure 5 is a simple example of how a Makefile.h looks like, in this example the project configuration file is defined at the beginning, both showing the name of the file and location (same directory, "../" to change directory). Another important part is the definition of the base of the project in this example is MQTT (APPS += mqtt). The last line includes the Makefile.include, which belongs to Contiki OS and defines the configuration of the Contiki OS.

```
1  DEFINES+=PROJECT_CONF_H=\"project-conf.h\"
2
3  all: mqtt-example
4
5  APPS += mqtt
6
7  # Linker size optimization
8  SMALL = 1
9
10 # Use IPv6 only
11 CONTIKI_WITH_IPV6 = 1
12
13 CONTIKI = ../../../..
14 include $(CONTIKI)/Makefile.include
15
```

Figure 5. Makefile.h

Many changes can be done to the Makefile.h and more can be added based on the needs of the project. First thing that must be done is to identify what are the needs of the project and then to make the appropriate changes.

4.1.2 Project Configuration

The next important file of a project is the configuration file where can be defined all the projects configurations. The project configuration file (Figure 6) shows some basic configurations to be made for a node which uses the MQTT protocol for communication. In this file all the stable variables of the project can be defined such as the server's IP and PORT, also the node's ID and the rest of the nodes behavior functions. Except of the configuration related to the communication protocol or other kinds of protocols there are a few specific configurations related to the proper way of transmitting the data:

- IEEE 802.15.4 PAN ID, personal area network id to be sent from the wireless local network.
- NETSTACK_CONF_RDC, specifies the Radio Duty Cycling (RDC) layer.
- MAX_TCP_SEGMENT_SIZE, parameter of the options field of the TCP header that specifies the largest amount of data received in a single TCP segment.

For the MQTT protocol there are two additional configurations that are needed so the nodes can function properly. Both of them are related to the size of the local wireless network.

- NBR_TABLE_CONF_MAX_NEIGHBORS, indicates the number of neighbors a node can have.
- UIP_CONF_MAX_ROUTES, indicates the number of routes a node can have for transmitting the data.

These configurations help to build bigger wireless networks by having more nodes and routes available for a larger area coverage.

```
/*---------------------------------------------------------------------------*/
#ifndef PROJECT_CONF_H_
#define PROJECT_CONF_H_
/*---------------------------------------------------------------------------*/
/*
 * If you have an IPv6 network or a NAT64-capable border-router:
 * test.mosquitto.org
 * IPv6 "2001:41d0:a:3a10::1"
 * NAT64 address "::ffff:5577:53c2" (85.119.83.194)
 *
 * To test locally you can use a mosquitto local broker in your host and use
 * i.e the fd00::1/64 address the Border router defaults to
 */
#define MQTT_DEMO_BROKER_IP_ADDR  "fd00::1"
/*---------------------------------------------------------------------------*/
/* Default configuration values */
#define DEFAULT_BROKER_PORT          1883
#define DEFAULT_PUBLISH_INTERVAL     (45 * CLOCK_SECOND)
#define DEFAULT_KEEP_ALIVE_TIMER     60

#undef IEEE802154_CONF_PANID
#define IEEE802154_CONF_PANID        0xABCD

/* The following are Zoul (RE-Mote, etc) specific */
#undef CC2538_RF_CONF_CHANNEL
#define CC2538_RF_CONF_CHANNEL       26

/* Specific platform values */
#if CONTIKI_TARGET_ZOUL
#define BUFFER_SIZE                  64
#define APP_BUFFER_SIZE              512

#else /* Default is Z1 */
#define BUFFER_SIZE                  48
#define APP_BUFFER_SIZE              256
#define BOARD_STRING                 "Zolertia Z1 Node"
#undef NBR_TABLE_CONF_MAX_NEIGHBORS
#define NBR_TABLE_CONF_MAX_NEIGHBORS 3
#undef UIP_CONF_MAX_ROUTES
#define UIP_CONF_MAX_ROUTES          3
#endif
/*---------------------------------------------------------------------------*/
#undef NETSTACK_CONF_RDC
#define NETSTACK_CONF_RDC          nullrdc_driver

/* Maximum TCP segment size for outgoing segments of our socket */
#define MAX_TCP_SEGMENT_SIZE       32

#endif /* PROJECT_CONF_H_ */
/*---------------------------------------------------------------------------*/
/** @} */
```

Figure 6. Project Configuration File

4.1.3 Application Files

The library for a MQTT example is the *mqtt.h,* which enables the use of its methods. Except of the library there are a few stable variables needed to be defined for the proper function of the library. This variables are:

- ID of the node, important to be unique
- IP of the server, specifically the PREFIX of the border router
- PORT of the broker, mosquitto broker port is 1883 (8883 if TLS is enabled)
- The topics to which a MQTT node will connect
- Authentication tokens, username and password of the node if required, if not required can be NULL

To initialize the connection of the MQTT client with broker it is necessary to do it in the PROCESS_THREAD of the application. To achieve this the mqtt_register and mqtt_set_username_password methods are needed to be called (Figure 7).

```
160   /* Create the connection */
161   mqtt_register(&conn, &mqtt_demo_process, DEFAULT_ORG_ID, mqtt_event,
162                 MAX_TCP_SEGMENT_SIZE);
163
164   /* User and token if required */
165   mqtt_set_username_password(&conn, DEFAULT_CONF_USER, DEFAULT_CONF_TOKEN);
```

Figure 7. Initializing the MQTT Connection

The mqtt_register method is responsible for creating the connection between the client and broker. The method has five parameters the conn, mqtt_demo_process, DEFAULT_ORG_ID, mqtt_event, MAX_TCP_SEGMENT_SIZE. The first parameter, conn, is a static struct mqtt_connection parameter to be used for the connection established by the method, the mqtt_demo_process is related to the PROCESS_THREAD, and more specifically is the name of the PROCESS. The third and fifth parameters are stable variables, the DEFAULT_ORG_ID variable is the ID of the node and MAX_TCP_SEGMENT_SIZE is defined in the configuration file and represents the size of the TCP segment, which is 32 bytes (Figure 6. Project Configuration FileFigure 6). The fourth parameter mqtt_event (Figure 8) is responsible for checking the state of the client.

```
77   /*---------------------------------------------------------------------------*/
78   PROCESS(mqtt_demo_process, "MQTT Simple Demo");
79   /*---------------------------------------------------------------------------*/
80   static void
81   mqtt_event(struct mqtt_connection *m, mqtt_event_t event, void *data)
82   {
83     switch(event) {
84     case MQTT_EVENT_CONNECTED:
85       printf("MQTT connection OK\n");
86       state = CONNECTED;
87       process_poll(&mqtt_demo_process);
88       break;
89
90     case MQTT_EVENT_DISCONNECTED:
91       state = DISCONNECTED;
92       process_poll(&mqtt_demo_process);
93       /* Print the disconnection reason */
94       printf("MQTT Disconnected=%u\n", *((mqtt_event_t *)data));
95       break;
96
97     case MQTT_EVENT_PUBLISH:
98       msg_ptr = data;
99       if(msg_ptr->first_chunk) {
100        msg_ptr->first_chunk = 0;
101        printf("Received PUB: topic '%s' (%i bytes)\n", msg_ptr->topic,
102                                                        strlen(msg_ptr->topic));
103      }
104      printf("Payload %s (%u bytes)\n", msg_ptr->payload_chunk,
105                                        msg_ptr->payload_length);
106      leds_toggle(LEDS_RED);
107      break;
108
109    case MQTT_EVENT_SUBACK:
110      state = READY;
111      process_poll(&mqtt_demo_process);
112      break;
113
114    case MQTT_EVENT_UNSUBACK:
115    case MQTT_EVENT_PUBACK:
116      break;
117
118    default:
119      break;
120    }
121  }
122  /*---------------------------------------------------------------------------*/
```

Figure 8. MQTT Event

Furthermore, the mqtt_connect_process method is responsible for connecting to broker and to subscribe to the subscription topic. The method is called in the PROCESS_THREAD inside a while loop statement.

```
123  static void
124  mqtt_connect_process() {
125
126      switch(state) {
127        case INIT:
128        /* Attempt to connect to the broker */
129        printf("MQTT connecting...\n");
130        mqtt_connect(&conn, DEFAULT_BROKER_IP, DEFAULT_BROKER_PORT,
131                     PUB_INTERVAL * 3);
132        break;
133
134        case CONNECTED:
135        printf("MQTT connected, subscribing...\n");
136        mqtt_subscribe(&conn, NULL, DEFAULT_SUB_TOPIC, MQTT_QOS_LEVEL_0);
137        break;
138
139        case READY:
140        printf("MQTT subscribed to %s\n", DEFAULT_SUB_TOPIC);
141        break;
142
143        case DISCONNECTED:
144        state = INIT;
145        printf("MQTT disconnected\n");
146        mqtt_disconnect(&conn);
147        break;
148      }
149  }
```

Figure 9. MQTT Connect Process

4.1.4 MQTT Example

Every local wireless network needs a node to play the role of a border router to forward the data to the global network. The border router operates as the server of the local WSN (wireless sensor network) and connects the local and global network. Another important part is tunslip6 which creates a bridge between local network and machine.

To compile tunslip6 the commands are:

- `$ cd contiki/tools`
- `$ sudo make tunslip65`
- `$ sudo ./tunslip6 fd00::1/64 –a 127.0.0.1`

To compile the border router the command is:

- `$ sudo make border-router TARGET=z1`[6] `PREFIX=fd00::1/64`

Starting the Cooja[7] simulation and adding the border router based on the z1 node. After that changing the node to Serial Socket server mode, which makes the node listening on PORT 60001. Next step is to add the MQTT nodes and creating the local wireless network (Figure 10). The network window shows the local wireless network, where node 1 is the border router and nodes from 2 to 7 are MQTT nodes. Simulation control helps controlling the execution of the simulation, Mote output shows the messages printed by the nodes and Radio messages captures the packets transmitted by the nodes.

[5] Error compiling tunslip6 related to gettimeofday() method, solution is to add the time.h library.
[6] Optional if Makefile.target is included and specifies the desired target.
[7] Location is the contiki/tools/cooja and the command is *sudo ant run*.

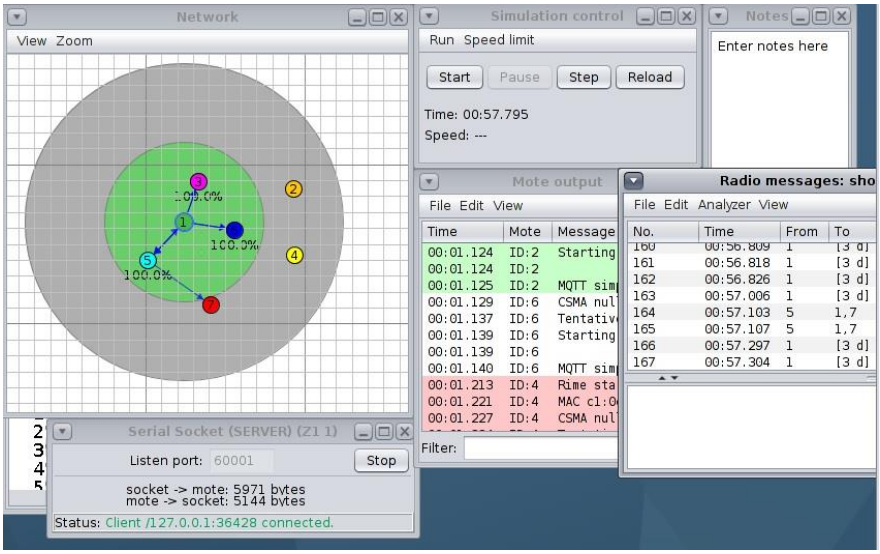

Figure 10. Network Simulation

Testing the network simulation to see how the simulation responds after changing the maximum number of neighbors and routes plays an important role of the behavior of the network also important is the location of the nodes.

4.2 TINYDTLS TESTING

tinyDTLS is a library based on DTLS which was created to be used with tinyOS operating system. There are a few versions of the library, most of them are not compatible with the Contiki OS. To make the library compatible with Contiki it is necessary to combine two tinyDTLS libraries[8] that shown the less problems while compiling them.

The first library is *armour-tinydtls* which is not fully compatible with Contiki as it does not provide with the appropriate command to compile the library for Contiki (./configurate --with-contiki). The second one, tinydtls library from cetic, is compatible with Contiki but the functions are not fully functional. To make a library that can be used with Contiki OS is to use the configuration files from the armour-tinydtls and the files containing the operational part of the library from cetic tinydtls library.

The parts taken from the cetic *tinydtls* library are both comprised of folders and files related to the configuration of the library. The folders taken are the aes, doc, ecc, platform-specific, sha2 and tests, next are the files .gitignore, ABOUT.md, CONTRIBUTING.md, LICENSE, Makefile.in, Makefile.tinydtls. README, configure.in. From the armour-tinydtls library provided by the iot-lab the files are:

- alert.h,
- ccm.c, ccm.h,
- crypto.c, crypto.h,
- dtls.c, dtls.h, dtls_debug.c, dtls_debug.h, dtls_time.c, dtls_time.h,
- global.h,
- hmac.c, hmac.h,

[8] First library is from cetic https://github.com/cetic/tinydtls, second one is from iot-lab https://github.com/iot-lab/armour-tinydtls.

- netq.c, netq.h,
- numeric.h,
- peer.c, peer.h,
- prng.h,
- session.c, session.h,
- state.h,
- tinydtls.h,
- uthash.h,
- utlist.h.

After combining the two libraries to install the new one follows the same principals with the rest. First thing to do is to copy the new library in the contiki/apps directory and next is to run the configuration file with the contiki option (--with-contiki). When including the library on a project it is important to specify it in the Makefile.h of the project (APPS += tinydtls/aes tinydtls/sha2 tinydtls/ecc tinydtls).

DTLS project is comprised of a server and a client. The difference between the server and the client is only the method *dtls_connect()* which is what characterizes the client. This method makes the client to initiate the handshake protocol and connect with the server. To create an application using the DTLS library for sending encrypted data between client and server there are a few crucial things needed to be implemented. First thing is to include the appropriate files from the library which are the tinydtls.h and dtls.h. Both of them enables the use of the methods required to create the connection between client and server and exchange data between them. Next is the definition of some stable variables needed to accomplish the application's goals (Figure 11).

```
59  /*---------------------------------------------------------------------------*/
60  /* Enables printing debug output from the IP/IPv6 libraries */
61  #define DEBUG DEBUG_PRINT
62  #include "net/ip/uip-debug.h"
63
64  #define UIP_IP_BUF   ((struct uip_ip_hdr *)&uip_buf[UIP_LLH_LEN])
65  #define UIP_UDP_BUF  ((struct uip_udp_hdr *)&uip_buf[UIP_LLIPH_LEN])
66
67  #define MAX_PAYLOAD_LEN      256
68
69  #define PSK_ID_MAXLEN        32
70  #define PSK_MAXLEN           32
71  #define PSK_DEFAULT_IDENTITY "Client_identity"
72  #define PSK_DEFAULT_KEY      "secretPSK"
73  #define PSK_OPTIONS          "i:k:"
74
75  #ifdef __GNUC__
76  #define UNUSED_PARAM __attribute__((unused))
77  #else
78  #define UNUSED_PARAM
79  #endif /* __GNUC__ */
80  /*---------------------------------------------------------------------------*/
```

Figure 11. tinyDTLS defined variables (client)

The UIP_IP_BUF and UIP_UDP_BUF are used to handle the messages received. The majority of them are related to the creation of the session key and encrypting the messages transmitted. The method to create the session key is the *get_psk_info()* (Figure 12), there are a few small differences between the client and server version of the method. The reason being that the client only is concerned by its own ID and secret key, but the server needs to be able to communicate with multiple clients. For the client it is required to define to *char* arrays, one for the ID and one for secret key. Also, it is needed to find the size of this two arrays.

```c
/*----------------------------------------------------------------------------*/
/* This function is the "key store" for tinyDTLS. It is called to
 * retrieve a key for the given identity within this particular
 * session. */
static int
get_psk_info(struct dtls_context_t *ctx, const session_t *session,
             dtls_credentials_type_t type,
             const unsigned char *id, size_t id_len,
             unsigned char *result, size_t result_length) {

  struct keymap_t {
    unsigned char *id;
    size_t id_length;
    unsigned char *key;
    size_t key_length;
  } psk[3] = {
    { (unsigned char *)"Client_identity", 15,
      (unsigned char *)"secretPSK", 9 },
    { (unsigned char *)"default identity", 16,
      (unsigned char *)"\x11\x22\x33", 3 },
    { (unsigned char *)"\0", 2,
      (unsigned char *)"", 1 }
  };

  if (type != DTLS_PSK_KEY) {
    return 0;
  }

  if (id) {
    int i;
    for (i = 0; i < sizeof(psk)/sizeof(struct keymap_t); i++) {
      if (id_len == psk[i].id_length && memcmp(id, psk[i].id, id_len) == 0) {
        if (result_length < psk[i].key_length) {
          printf("buffer too small for PSK");
          return dtls_alert_fatal_create(DTLS_ALERT_INTERNAL_ERROR);
        }

        memcpy(result, psk[i].key, psk[i].key_length);
        return psk[i].key_length;
      }
    }
  }

  return dtls_alert_fatal_create(DTLS_ALERT_DECRYPT_ERROR);
}
```

```c
/*----------------------------------------------------------------------------*/
/* PSK key values for the get_key_info method */
static unsigned char psk_id[PSK_ID_MAXLEN] = PSK_DEFAULT_IDENTITY;
static size_t psk_id_length = sizeof(PSK_DEFAULT_IDENTITY) - 1;
static unsigned char psk_key[PSK_MAXLEN] = PSK_DEFAULT_KEY;
static size_t psk_key_length = sizeof(PSK_DEFAULT_KEY) - 1;
```

```c
/*----------------------------------------------------------------------------*/
static int
get_psk_info(struct dtls_context_t *ctx UNUSED_PARAM,
             const session_t *session UNUSED_PARAM,
             dtls_credentials_type_t type,
             const unsigned char *id, size_t id_len,
             unsigned char *result, size_t result_length) {

  switch (type) {
  case DTLS_PSK_IDENTITY:
    if (result_length < psk_id_length) {
      printf("cannot set psk_identity -- buffer too small\n");
      return dtls_alert_fatal_create(DTLS_ALERT_INTERNAL_ERROR);
    }
    memcpy(result, psk_id, psk_id_length);
    return psk_id_length;
  case DTLS_PSK_KEY:
    if (id_len != psk_id_length || memcmp(psk_id, id, id_len) != 0) {
      printf("PSK for unknown id requested, exiting\n");
      return dtls_alert_fatal_create(DTLS_ALERT_ILLEGAL_PARAMETER);
    } else if (result_length < psk_key_length) {
      printf("cannot set psk -- buffer too small\n");
      return dtls_alert_fatal_create(DTLS_ALERT_INTERNAL_ERROR);
    }
    memcpy(result, psk_key, psk_key_length);
    return psk_key_length;
  default:
    printf("unsupported request type: %d\n", type);
  }

  return dtls_alert_fatal_create(DTLS_ALERT_INTERNAL_ERROR);
}
/*----------------------------------------------------------------------------*/
```

Figure 12. get_psk_info method (left server/right client)

There are special methods to send the messages also to read them. The method to send data *send_to_peer()* (Figure 14) its function is to start the handshake protocol between the client and server and the messages that are sent through this method are not encrypted. The *read_from_peer()* (Figure 13) is used to read the messages received, there are a few differences between the server and client regarding this method. The difference between client and server of the *read_from_peer()* method is that the client does not need the if statement.

```c
#define DTLS_SERVER_CMD_CLOSE       "server:close"
#define DTLS_SERVER_CMD_RENEGOTIATE "server:renegotiate"

static int
read_from_peer(struct dtls_context_t *ctx,
               session_t *session, uint8 *data, size_t len) {
  char *str;

  str = data;
  str[len] = '\0';
  printf("Received from the client: '%s'\n", str);

  if (len >= strlen(DTLS_SERVER_CMD_CLOSE) &&
      !memcmp(data, DTLS_SERVER_CMD_CLOSE, strlen(DTLS_SERVER_CMD_CLOSE))) {
    printf("server: closing connection\n");
    dtls_close(ctx, session);
    return len;
  } else if (len >= strlen(DTLS_SERVER_CMD_RENEGOTIATE) &&
      !memcmp(data, DTLS_SERVER_CMD_RENEGOTIATE, strlen(DTLS_SERVER_CMD_RENEGOTIATE))) {
    printf("server: renegotiate connection\n");
    dtls_renegotiate(ctx, session);
    return len;
  }

  return 0;
}
```

Figure 13. read_from_peer() method

```
116  /*---------------------------------------------------------------------------*/
117  static int
118  send_to_peer(struct dtls_context_t *ctx, session_t *session, uint8 *data, size_t len) {
119
120    struct uip_udp_conn *conn = (struct uip_udp_conn *)dtls_get_app_data(ctx);
121
122    uip_ipaddr_copy(&conn->ripaddr, &session->addr);
123    conn->rport = session->port;
124
125    PRINTF("send to ");
126    PRINT6ADDR(&conn->ripaddr);
127    PRINTF(":%u\n", uip_ntohs(conn->rport));
128
129    uip_udp_packet_send(conn, data, len);
130
131    /* Restore server connection to allow data from any node */
132    /* FIXME: do we want this at all? */
133    memset(&conn->ripaddr, 0, sizeof(conn->ripaddr));
134    memset(&conn->rport, 0, sizeof(conn->rport));
135
136    return len;
137  }
138  /*---------------------------------------------------------------------------*/
```

Figure 14. send_to_peer() method

Basically all three methods presented above are needed to achieve the handshake protocol of the tinyDTLS library. To call these methods it is required to define a handler (Figure 15) which actually knows the method it needs based on the necessary action, session key construction, send message or read message.

```
248  static dtls_handler_t cb = {
249      .write = send_to_peer,
250      .read  = read_from_peer,
251      .event = NULL,
252      .get_psk_info = get_psk_info
253  };
```

Figure 15. DTLS Handler

The method employed to receive the message and to prepare them for the *read_from_peer()* method is the *dtls_handler_read()* (Figure 16). The method prepares the information received and through the *dtls_handle_message()* and later are passed to the *read_from_peer()*. The *dtls_handle_message()* is internal method of the library and it is needed only to pass the method to it as its functions are predefined.

```
186  /*---------------------------------------------------------------------------*/
187  static int
188  dtls_handle_read(struct dtls_context_t *ctx) {
189    static session_t session;
190
191    if(uip_newdata()) {
192        uip_ipaddr_copy(&session.addr, &UIP_IP_BUF->srcipaddr);
193        session.port = UIP_UDP_BUF->srcport;
194        session.size = sizeof(session.addr) + sizeof(session.port);
195
196        ((char *)uip_appdata)[uip_datalen()] = 0;
197
198        PRINTF("Message received from ");
199        PRINT6ADDR(&session.addr);
200        PRINTF(":%d\n", UIP_HTONS(session.port));
201    }
202
203    return uip_datalen() < 0 ? uip_datalen() : dtls_handle_message(ctx, &session,
204                                                uip_appdata, uip_datalen());
205  }
206  /*---------------------------------------------------------------------------*/
```

Figure 16. dtls_handle_read() method

Except these methods there are a few more things needed to make the library work properly. The first thing is to specify the local and global addresses of the client and server. Next is the creation of the UDO connection based on the IP address and PORT and binding it, after that, based on the connection created,

it is required to create the DTLS context (*dtls_new_context()*). After the creation of the context it is needed to set the handlers based on the DTLS context and the DTLS handler (Figure 15). Next is for the client to call the *dtls_connect()* method to start the handshake protocol. After the successful conclusion of the handshake protocol to send encrypted messages the *dtls_write()* method is needed, as through this method the library calls all the encryption functions available.

After creating both the client and server nodes what follows is running the simulation and observing the reaction of the nodes. First step is to compile both nodes, DTLS client and server, and also the border router[9]. Next step is to add the nodes as zolertia z1 nodes, run tunslip6 and then to start the simulation. The client sends the messages to the border router which forwards them to the server.

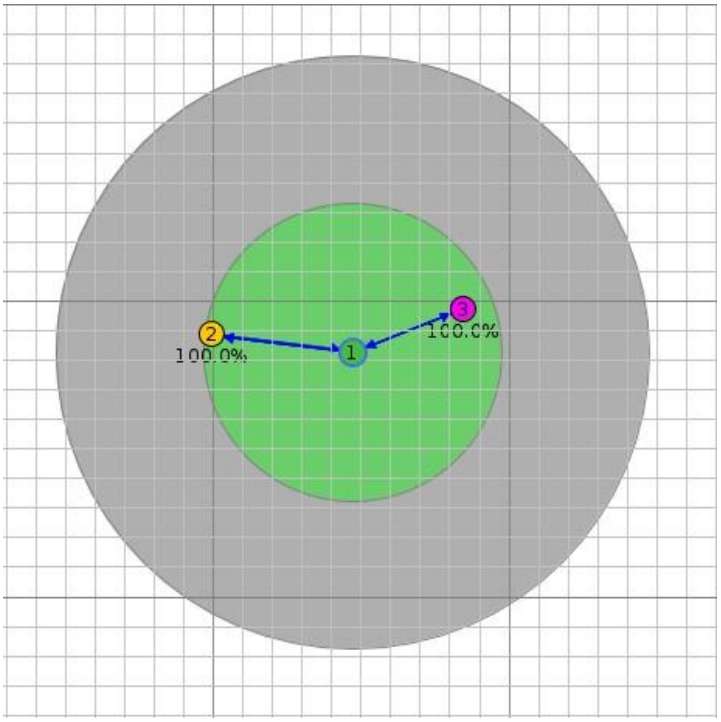

Figure 17. tinyDTLS Simulation

Node 1 border router
Node 2 tinyDTLS client
Node 3 tinyDTLS server

The first thing that follows is the handshake protocol, and after the handshake protocol has been completed what follows is the exchange of encrypted messages between client and server. The example shown in Figure 18 is a simple PING PONG client server example. It starts with the information from whom the message was sent and the size of the message. Next, the message is decrypted and the simulation shows the actual message in plaintext. Finally a response is sent back which is encrypted with the TLS_PSK_WITH_AES_128_CCM_8, a pre shared secret key AES 128 bit key with CCM mode for both authentication and confidentiality.

[9] The PREFIX for the example is aaaa::1/64

```
ID:3   Message received from aaaa::c30c:0:0:2:1025
ID:3   dtls_handle_message: FOUND PEER
ID:3   got packet 23 (33 bytes)
ID:3   decrypt_verify(): found 4 bytes cleartext
ID:3   new packet arrived with seq_nr: 4
ID:3   new bitfield is               : b
ID:3   ** application data:
ID:3   Received from the client: 'PING'
ID:3   OK
ID:3   dtls_prepare_record(): encrypt using TLS_PSK_WITH_AES_128_CCM_8
ID:3   send to aaaa::c30c:0:0:2:1025
ID:2   Client received message from serverdtls_handle_message: FOUND PEER
ID:2   got packet 23 (33 bytes)
ID:2   decrypt_verify(): found 4 bytes cleartext
ID:2   new packet arrived with seq_nr: 2
ID:2   new bitfield is               : 3
ID:2   ** application data:
ID:2   Received from the server: 'PONG'
ID:2   OK
ID:2   dtls_prepare_record(): encrypt using TLS_PSK_WITH_AES_128_CCM_8
ID:2   send to aaaa::c30c:0:0:3:1026
```

Figure 18. Message Exchange

In case that is required to keep track of the packets being exchanged between the nodes the option of the Radio messages can show everything. Also, if there is the need to analyze these packets than it can be saved as a PCAP file and using *Wireshark* to have a deeper look into the packets. By studying these files we can see that the payload sent between the nodes is encrypted. The only information that can be seen is the header of the messaged which are required to be in plaintext making forwarding feasible.

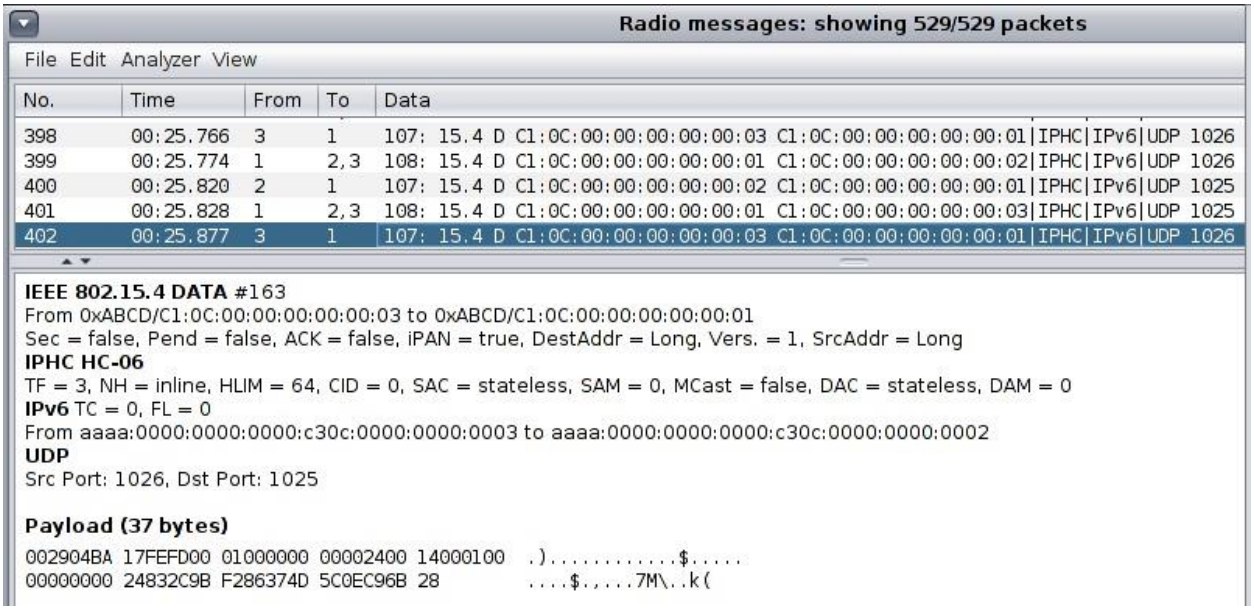

Figure 19. 6LowPAN PCAP Analyzer

5 CONCLUSION

There are quite a few old technologies which have been adapted to the Internet of Things. Some of these technologies are more compatible with the IoT than others but all have their own advantages and disadvantages. The MQTT protocol is one of them as it was created long before the idea and provides a solution on how to exchange data between the local wireless network and the broker.

A very important aspect of the IoT is the need of security measures, as its purpose is to collect data and send it to a server through global network. Of course, the data collect is sensitive and it should be protected for privacy and even economical reasons. Another security issue is that nodes belonging to local network can be hijacked and used for malicious reasons. Because the already known security measures are quite heavy to be used by nodes, which have limited processing capabilities, new ways to guaranty the security of both the data and the local wireless network must be implemented. IoT prefers to use the more lightweight UDP protocol for communication instead of TCP, and because TLS security measure is too demanding a more appropriate security measure is needed. One such protocol is Datagram TLS (DTLS) developed with the scope to implement TLS security techniques on UDP communication. DTLS is more lightweight so it can be used by nodes and provides security measures to achieve both authentication and confidentiality of data.

There have been successful attempts of combining the DTLS library and the CoAP communication protocol. However, combining the MQTT and DTLS protocols is a bit more difficult as these two protocols are based on different transport layer protocols, TCP and UDP respectively. At the moment the MQTT protocol and the brokers used to communicate with the MQTT nodes do not support DTLS security. It is not only the brokers that need to be updated to support DTLS as there is already a MQTT-SN (MQTT for sensor network) in development, but also to combine the MQTT and DTLS as a new library. Such kind of a library must be implemented to make the integration of DTLS security measurements available on MQTT based wireless network.

REFERENCES

[1] Sergey Andreevy, Olga Galinina, Alexander Pyattaev, Mikhail Gerasimenko, Tuomas Tirronen, Johan Torsner, Joachim Sachs, Mischa Dohler, and Yevgeni Koucheryavy. (2015) *Understanding the IoT Connectivity Landscape – A Contemporary M2M Radio Technology Roadmap*. IEEE Communications Magazine. Available at: http://ieeexplore.ieee.org/abstract/document/7263370/ (Accessed: 03/11/2017).

[2] Tara Salman. (2015) *Networking Protocols and Standards for Internet of Things*. Available at: http://www.cse.wustl.edu/~jain/cse570-15/ftp/iot_prot/index.html (Accessed: 28/10/2016).

[3] Antonio Liñán Colina, Alvaro Vives, Marco Zennaro, Antoine Bagula, Ermanno Pietrosemoli. (2016) *Internet of Things IN 5 DAYS*. Available at: http://wireless.ictp.it/school_2016/book/IoT_in_five_days-v1.0.pdf (Accessed: 10/10/2016).

[4] Vishwas Lakkundi, Keval Singh. (2014) *Lightweight DTLS implementation in CoAP-based Internet of Things*. Bangalore: International Conference on Advanced Computing and Communications ADCOM 201. Available at: https://www.researchgate.net/publication/265914358_Lightweight_DTLS_implementation_in_CoAP-based_Internet_of_Things (Accessed: 10/01/2017).

[5] Ajit A.Chavan, Mininath K. Nighot. (2014) *Secure CoAP Using Enhanced DTLS for Internet of Things*. City: International Journal of Innovative Research in Computer and Communication Engineering. Available at: https://www.rroij.com/open-access/secure-coap-using-enhanced-dtls-forinternet-of-things.pdf (Accessed: 12/01/2017).

[6] Thomas Kothmayr, Wen Hu, Corinna Schmit, Michael Br¨unig, Georg Carle. (2011) *Securing the Internet of Things with DTLS*. Available at: https://pdfs.semanticscholar.org/41f8/c9afb5e135a8d84a3b3614d12b14e6deaecb.pdf (Accessed: 10/01/2017).

[7] Nagendra Modadugu, Eric Rescorla. *The Design and Implementation of Datagram TLS*. Stanford University RTFM, Inc. Available at: https://crypto.stanford.edu/~nagendra/papers/dtls.pdf (Accessed: 15/01/2017).

[8] Thomas Kothmayr, Corinna Schmitt, Wen Hu, Michael Br¨unig and Georg Carle. (2012) *A DTLS Based End-To-End Security Architecture for the Internet of Things with Two-Way Authentication*. Clearwater, FL, USA: IEEE. Available at: http://Website URL (Accessed: 15/01/2017).

[9] TinyOS 2011, FAQ Wiki, accessed 19 October 2016, <http://tinyos.stanford.edu/tinyos-wiki/index.php/FAQ#What_is_TinyOS.3F>

[10] TInyOS Wiki 2016, Wikipedia, accessed 19 October 2016, <https://en.wikipedia.org/wiki/TinyOS>

[11] Contiki: The Open Source OS for the Internet of Things 2016, Official website, accessed 10 October 2016, <http://www.contiki-os.org/index.html#why>

[12] RIOT: The friendly Operating System for the Internet of Things 2016, Official website, accessed 11 October 2016, < https://riot-os.org/>

[13] MQTT Security Fundamentals 2017, HiveMQ, accessed 3 November 2016, <http://www.hivemq.com/blog/introducing-the-mqtt-security-fundamentals>

[14] MQTT 101 – How to Get Started with the lightweight IoT Protocol 2017, HiveMQ, accessed 3 November 2016, <http://www.hivemq.com/blog/how-to-get-started-with-mqtt>

[15] Get to Know MQTT: The Messaging Protocol for the Internet of Things 2016, the New Stack, accessed 3 November 2016, <https://thenewstack.io/mqtt-protocol-iot/>

[16] Which is more lightweight as IoT protocol - MQTT or CoAP? 2016, Quora, accessed 15 December 2016, <https://www.quora.com/Which-is-more-lightweight-as-IoT-protocol-MQTT-or-CoAP>

[17] Connecting the Iot: A Quick Look at Mqtt & Coap Protocols, I/O Hub the Logic Supply, accessed 17 December 2016, <https://www.logicsupply.com/explore/io-hub/connecting-the-iot-mqtt-coap-protocols/>

[18] DTLS Usage 2014, tinyDTLS 0.8.1, accessed 20 January 2017, <http://tinydtls.sourceforge.net/group__dtls__usage.html>

[19] Contiki 2014, tinyDTLS 0.8.1, accessed 20 January 2017, http://tinydtls.sourceforge.net/group__contiki.html

[20] Emmanuel Baccelli, Oliver Hahm, Mesut Gunes, Matthias Wahlisch Thomas C. Schmidt. (2013) RIOT OS: Towards an OS for the Internet of Things. INFOCOM'2013 Demo/Poster Session. Available at: http://www.inet.haw-hamburg.de/papers/bhgws-rotoi-13.pdf (Accessed: 10/06/2017).

[21] Zolertia Z1 mote 2016, GitHub Zolertia Resources, accessed 13 October 2016, <https://github.com/Zolertia/Resources/wiki/The-Z1-mote>

[22] Zolertia RE-Mote platform 2016, GitHub Zolertia Resources, accessed 13 October 2016, <https://github.com/Zolertia/Resources/wiki/RE-Mote>

[23] 10 Real World Applications of Internet of Things (IoT) 2016, Analytics Vidhya, accessed 11 October 2016, <https://www.analyticsvidhya.com/blog/2016/08/10-youtube-videos-explaining-the-real-world-applications-of-internet-of-things-iot/>

[24] Differences between TLS and DTLS 2016, Security Stackex change, accessed 25 January 2017, <https://security.stackexchange.com/questions/29172/what-changed-between-tls-and-dtls>

[25] Operating Systems 2017, Wikipedia the Free Encyclopedia, accessed 25 May 2017, <https://en.wikipedia.org/wiki/Operating_system>