POLITECNICO DI MILANO
DIPARTIMENTODI ELETTRONICA, INFORMAZIONE E BIOINGEGNERIA MASTER DI RICERCA IN INGEGNERIA DELL'INFORMAZIONE

Integrating low power devices with ELIoT using standard protocols

Advisor: Prof. Gianpaolo Cugola

Tutor: Prof. Alessandro Sivieri

Master Thesis of: HONG RUN
Matr.n: 796200
Master Thesis of: GAO XIAORUI
Matr.n: 797220

June 2014

ACRONYMS

ELIoT ……………………..*Erlang language for the Internet of Things*
WSN………………………………………..*wireless sensor network*
IoT …………………………………………………..*Internet of Things*
CoAP ………………………………*Constrained Application Protocol*
URI……………………………………….*Uniform Resource Identifier*
RFID……………………………………*Radio-frequency identification*
M2M……………………………………………. *machine-to-machine*
MEMS ………………………………*micro-electro-mechanical system*
UWB…………………………………………………*Ultra Wide Band*
SMAC……………………………………*Sensor media access control*
SPIN………………...*Sensor Protocols for Information via Negotiation*
PSFQ ……………………………………*Pump-Slowly, Fetch-Quickly*
CODA…………………………*Congestion Detection and Avoidance*
QoS………………………………………………..*Quality of Service*
LLN………………………………..*Low power and Lossy Networks*
RPL………………………………………*Routing Protocol for LLN*
DODAG………………...*Destination Oriented Directed Acyclic Graph*
DIO……………………………………….*DODAG information object*
6LowPan…………………...*IPv6 over Low-power and Lossy Network*
DAG…………………………………………*Directed Acyclic Graph*
RoLL…………………..*Routing over Lossy and Low-power Networks*
LBR…………………………………………*LLN border router LBR*

ABSTRACT

Nowadays, the rise of Internet of Things (IoT) has raised new demands for the traditional Internet. It is is driven by an expansion of the Internet through the inclusion of physical objects combined with an ability to provide smarter services to the business processes as more information becomes available. Within the network all the objects are required to be uniquely addressed. But due to the limitation of IPv4, even if the supply of IPv4 addresses were not to be exhausted soon, the size of IPv4 itself is not large enough to support the Internet of Things, so transition to IPv6 is inevitable and around the corner.

However, conventional IP protocol stack is not suitable for wireless sensor network, where the resources and energy are constrained. 6LoWPAN offers a feasible solution for communication between IPv6 network and WSN. Constrained Application Protocol (CoAP) is a lightweight application layer protocol that is specifically designed for resource-constrained internet devices and constrained networks. It is intended to reduce the power consumption of the sensors and extend battery life.

As a sequel to the study described above, the overall purpose of this thesis is dedicated to the implementation of the CoAP protocol not only on x86 platform but also on embedded devices, and make it possible to interconnect to each other.

The implementation consists of a client, a server and a gateway. The client part adopts a framework called ELIoT (Erlang language for the Internet of Things), sends CoAP request to access to the information or functionality that is available. The server part is developed on an open source operating system called CONTIKI, which offers standard protocol like IPv4, IPv6, 6LoWPAN, RPL. It is responsible for gathering the environment data according to the CoAP request. The gateway is placed between the server and client, it will automatically convert the target protocol type and forward the data through wireless or wired interface. It plays the role of interconnecting the different platforms and is easy to deploy and manage as the entire process.

CONTENTS

LIST OF FIGURES

1 Introduction

1.1 Internet and the Internet Of Things

What is Internet?

Internet is short for internetwork, it began in USA in 1969. It is a public worldwide computer communication network system. Nowadays it has become the most popular thing around the world. According to the statistics by 2007 more than 97% of all telecommunicated information was carried over the Internet.

What is the Internet of things? And what is the difference and relations between Internet and Internet of things?

At first, Radio-frequency identification (RFID), Infrared sensor, GPS and all other information sensing devices were seen as a prerequisite for the Internet of Things. Today however, the term Internet of Things (commonly abbreviated as IoT) is used to denote advanced connectivity of devices, systems and services that goes beyond machine-to-machine communications (M2M) and covers a variety of protocols, domains and applications.

As above says, in the future, not only computer systems and our cell-phones can connect to the Internet and get information, but also devices like low-power sensors and controllers, machines, will be connected by the Internet, and the Internet will be expanded, so that this network will generate and exchange information not only between human beings but also between things.

1.2 Aim of the thesis

In this thesis we will use a new framework called ELIoT (Erlang language for the Internet of Things), which allows us to use high-level programming language to develop applications not only on x86 system but also on embedded devices, and integrate CoAP, intended to be used in simple electronic devices, to allow intercommunication between the two devices.

An open source operating system called CONTIKI, which provides fully standard IPv4 and IPv6 along with some standard protocol like 6lowpan, RPL, is the OS used by the simple electronic devices mentioned above to run CoAP services. And use this operating system to develop an application and make those low-power device to communicate with other device using CoAP protocol. This is another part of the thesis.

1.3 Structure of the Thesis

This thesis is structured as follows:

Chapter 2 will introduce some background of this thesis, in particular, what is Internet of Things, the background of the Contiki system, Eralng and CoAP protocol and so on.

Chapter 3 will describe the scenario to apply CoAP and the way that we design the applications, include the client, server and proxy, and how to implement them with Erlang and Contiki.

Chapter 4 is going to have a test on our scenario, we are going to describe what the test environment is going to be, and then implement the test with the round trip time, system performance and the tmote sky RF system.

In chapter 5, we are going to tell you some existing related works with Internet of Things and CoAP framework. And also different choices that have been with different language and platform.

The last chapter, chapter 6, we conclude everything what we have done, and then gives some suggestion to the future work.

2 Background

2.1 Internet of things

The Internet of Things (IoT) is a scenario in which objects, animals or people are provided with unique identifiers and the ability to automatically transfer data over a network without requiring human-to-human or human-to-computer interaction. The term Internet of Things was proposed by Kevin Ashton in 1999 though the concept has been discussed since at least 1991. The concept of the Internet of Things first became popular through the Auto-ID Center at MIT and related market analysis publications.

The Internet of Things (IoT) refers to uniquely identifiable objects and their virtual representations in an Internet-like structure. A thing, in the Internet of Things, can be a person with a heart monitor implant, a farm animal with a biochip transponder, an automobile that has built-in sensors to alert the driver when tire pressure is low -- or any other natural or man-made object that can be assigned an IP address and provided with the ability to transfer data over a network. So far, the Internet of Things has been most closely associated with machine-to-machine (M2M) communication in manufacturing and power, oil and gas utilities. Products built with M2M communication capabilities are often referred to as being smart.

The first Internet appliance, for example, was a Coke machine at Carnegie Melon University in the early 1980s. The programmers could connect to the machine over the Internet, check the status of the machine and determine whether or not there would be a cold drink awaiting them, should they decide to make the trip down to the machine. Radio-frequency identification (RFID) was seen as a prerequisite for the Internet of Things in the early days. If all objects and people in daily life were equipped with identifiers, they could be managed and inventoried by computers. Besides using RFID, the tagging of things may be achieved through such technologies as near field communication, barcodes, QR codes and digital watermarking. Today however, the term Internet of Things (commonly abbreviated as IoT) is used to denote advanced connectivity of devices, systems and services that goes beyond the traditional machine-to-machine (M2M) and covers a variety of protocols, domains and applications[1].

According to Gartner, there will be nearly 26 billion devices on the Internet of Things by 2020. According to ABI Research, more than 30 billion devices will be wirelessly connected to the Internet of Things (Internet of Everything) by 2020. Cisco created a dynamic "connections counter" to track the estimated number of connected things from July 2013 until July 2020 (methodology included). This concept, where devices connect to the internet/web via low-power radio, is the most active research area in IoT. The low-power radios do not need to use Wi-Fi or Bluetooth. Lower-power and lower-cost alternatives are being explored under the category of Chirp Networks.

IPv6's huge increase in address space is an important factor in the development of the Internet of Things. According to Steve Leibson, who identifies himself as "occasional docent at the Computer History Museum," the address space expansion means that we could "assign an IPV6 address to every atom on the surface of the earth, and still have enough addresses left to do another 100+ earths." In other words, humans could easily assign an IP address to every "thing" on the planet. An increase in the number of smart nodes, as well as the amount of upstream data the nodes generate, is expected to raise new concerns about data privacy, data sovereignty and security.

"Today computers -- and, therefore, the Internet -- are almost wholly dependent on human beings for information. Nearly all of the roughly 50 petabytes (a petabyte is 1,024terabytes) of data available on the Internet were first captured and created by human beings by typing, pressing a record button, taking a digital picture or scanning a bar code.

The problem is, people have limited time, attention and accuracy -- all of which means they are not very good at capturing data about things in the real world. If we had computers that knew everything there was to know about things -- using data they gathered without any help from us -- we would be able to track and count everything and greatly reduce waste, loss and cost. We would know when things needed replacing, repairing or recalling and whether they were fresh or past their best.”

2.2: WSN(wireless sensor network)

The advances in the integration of micro-electro-mechanical system (MEMS), microprocessor, and wireless communication technology have enabled the deployment of large-scale sensor networks. A wireless sensor network consists of two kinds of entities: the sink that queries and collects information; and the sensor that senses environmental phenomena and reports data to the sink. In general, a few sinks and a huge set of small uncontrolled sensors are randomly deployed in a multi-hop and ad-hoc fashion to cooperatively pass their data through the network to a main location. The more modern networks are bi-directional, also enabling control of sensor activity.

The WSN formed by hundreds or thousands of motes that communicate with each other and pass data along from one to another. Each such sensor network node has several basic components: a CPU, a radio transceiver with an internal antenna or connection to an external antenna, and a sensor array and usually a battery or an embedded form of energy harvesting. Research done in this area focus mostly on energy aware computing and distributed computing. These research works have spanned over all layers of the network protocol stack. At the physical and the data link layer, energy-efficient and robust schemes, such as SMAC and UWB (Ultra Wide Band) have been proposed. Above these layers, new routing and transport protocols, such as Directed diffusion, SPIN , PSFQ , and CODA , have been proposed to fulfill the requirements of sensor networks, which include data-centric processing, in-network processing, and attribute-based naming[1].

In the past decade, many wireless sensor networks have been deployed. Based on basic functionalities, we categorize the applications into three basic applications: data collecting, event monitoring, and object tracking application. Most of wireless sensor networks will fall into one of these basic classes or hybrid class which combines more than two basic applications.

Environmental Applications:

The autonomous coordination capabilities of WSNs are utilized in the realization of a wide variety of environmental applications.

Some of the environmental applications of sensor networks are Area monitoring, Air pollution monitoring, Forest fire detection, Water quality monitoring and so on.

Home Applications:

As technology advances, smart sensor nodes and actuators can be buried in appliances such as vacuum cleaners, microwave ovens, refrigerators, and DVD players. These sensor nodes inside domestic devices can interact with each other and with the external network via the Internet or satellite. They allow end-users to more easily manage home devices both locally and remotely. Accordingly, WSNs enable the interconnection of various devices at residential places with convenient control of various applications at home.

Industrial Applications:

Wireless sensor networks have been developed for machinery condition-based maintenance (CBM) as they offer significant cost savings and enable new functionality. In wired systems, the installation of enough sensors is often limited by the cost of wiring. Previously inaccessible locations, rotating machinery, hazardous or restricted areas, and mobile assets can now be reached with wireless sensors.

There are some unique characteristics that set WSNs apart from other communication networks:

Compared to traditional communication networks, individual node Identifiers (IDs) are not important. Instead, WSNs are data-centric meaning that the communication should be targeted to nodes in a given location or with defined data content.

In a typical WSN, node platforms are ability to withstand harsh environmental conditions. Communication links between nodes are not stable due to node errors, unreliable and simple modulations, mobility of nodes, and environmental interferences.

Compared to other wireless networks, the number of nodes comprising WSNs may be huge. It should be easy to scalability to large scale of networks and also sometimes It should be deployed remotely.

A typical WSN node is small in physical size and with limited energy sources. This implies that computation, communication, and memory resources in nodes are very limited. Typically, severely energy constrained[2].

In large-scale WSNs, the deployment of nodes is random and their maintenance and replacement is impractical. So it should have some sort of Self-organizing and self-healing Still, the requirements and applications of the deployed WSN may alter, which implicate that runtime reconfiguration and reprogramming are needed.

To reduce their design complexity, most networks are organized as a stack of layers or levels, each one built upon the one below it. The number of layers, the name of each layer, the contents of each layer, and the function of each layer differ from network to network. The purpose of each layer is to offer certain services to the higher layers, shielding those layers from the details of how the offered services are actually implemented. The traditional layered approach brings three main problems to us: cannot share different information among different layers，which leads to each layer not having complete information; does not have the ability to adapt to the environmental change; duo to interference between different users, access confliction, and the change of environment in the wireless sensor networks, traditional layered approach for wired networks is not applicable to wireless networks. So we use cross-layer to make the optimal modulation to improve the transmission performance, such as data rate, energy efficiency, QoS (Quality of Service).

Hardware:

WSNs are composed of individual sensor nodes which are capable of interacting with their environment through various sensors, processing information locally and communicating information wirelessly with their neighbors

One major challenge is to produce low cost and tiny sensor nodes. Commercial platforms typically consist of three components and can be either an individual board or embedded into a single system:

Wireless modules or motes are the key components of the sensor network as they possess the communication capabilities and the programmable memory where the application code resides. A mode usually consists of a microcontroller, transceiver, power source, memory unit and may contains few sensors.

A sensor board is mounted on the mote and is embedded with multiple types of sensors. Available sensor boards include the MTS300/400 and MDA100/300 that are used in the Mica family of motes. Alternatively, the sensors can be integrated into the wireless module such as in the Telos or the SunSPOT platform.

A programming board, also known as the gateway board, provides multiple interfaces including Ethernet, WiFi, USB, or serial ports for connecting different motes to an enterprise or industrial network or locally to a PC/laptop. These boards are used either to program the motes or gather data from them

Software:

Energy is the scarcest resource of WSN nodes, and it determines the lifetime of WSNs. WSNs are meant to be deployed in large numbers in various environments, including remote and hostile regions, where ad hoc communications are a key component. For this reason, algorithms and protocols need to address the three main issues: lifetime maximization, robustness and fault tolerance and self-configuration.

Lifetime maximization: Energy/Power Consumption of the sensing device should be minimized and sensor nodes should be energy efficient since their limited energy resource determines their lifetime. To conserve power the node should shut off the radio power supply when not in use.

Some of the important topic in WSN (Wireless Sensor Networks) software research are operating system.

Operating systems for wireless sensor network nodes are typically less complex than general-purpose operating systems. Several software platforms have also been developed specifically for WSNs. Among these, the most accepted platform is TinyOS and Contiki.

TinyOS is a free and open source software component-based operating system and platform targeting wireless sensor networks (WSNs). Incorporates a component-based architecture (wide available library).It is written in nesC and based on an event-driven execution model that enables fine-grained power instead of multithreading.

Contiki is an OS which uses a simpler programming style in C while providing advances such as 6LoWPAN and Protothreads.

2.3:Tinyos and Contiki

Operating systems that are designed for wireless sensor networks are very different from operating systems for desktop/laptop computers like Windows or Linux or operating systems for powerful embedded systems like smart phones.

The biggest difference is the hardware on which the operating systems are running. The wireless sensor nodes usually have a microcontroller as a CPU that is not very powerful because the main focus of those motes lies in minimal power consumption since they are often designed to run on battery power for very long periods of time. And even though the microcontroller and all other components of motes are designed as low power devices, running them all at full power at all times would still consume way too much energy. So for that matter the main focus of those operating systems is energy conservation optimal usage of limited resources.

Operating systems for motes are very simple compared to other operating systems. But they still are often required to handle many different operations at the same time. A mote could for example be required to collect data from a sensor, process the data in some way and send the data to a gateway at the same time. Since the microcontrollers are only able to execute one program at the time, the operating systems have to have a scheduling system that shares the CPU resources between the different tasks, so that all of them can finish in the desired time frame. Since the requirements for the operating system vary between applications it is in most cases not possible to exchange the client program on a mote without changing the operating system. In fact in most cases the operating system behaves more like a library: It gets integrated into the application and both the application and the operating system are compiled into one single binary that is then deployed onto the sensor node.

TinyOS provides a programming framework to build application-specific OS instances. Programming framework made of scheduler, components and interfaces.

But there are some issues with TinyOS[3]:

Components linked to whole image of system, once linked can’t be updated without re-linking whole thing

No threads, event-driven paradigm, buffers can overflow while waiting for long-running task to complete more complex tasks make this more common (aggregations, encryption, signal processing), need threads (or breaking down into small execution modules) if want to do this

All memory pre-allocated

2.3.1 Contiki

Contki is an open source, highly portable, multi-tasking operating system for memory-efficient networked embedded systems and wireless sensor networks. It has been used in a variety of projects, Examples of where Contiki is used include street lighting systems, sound monitoring sm art cities, radiation monitoring systems, and alarm systems.

It was created by Adam Dunkels in 2002 and has been further developed by a world-wide team of developers from Atmel, Cisco, Enea, ETH Zurich, Redwire, RWTH Aachen University, Oxford Univ ersity, SAP, Sensinode, SICS, ST Microelectronics, Zolertia, and ma ny others1.The name Contiki comes from Thor Heyerdahl's famous Kon-Tikiraft.Kon-Tiki was the name of the raft used by Norwegian exploer and writer Thor Heyerdahl in his 1947 expedition across the Pacific Ocean from South America to the Polynesian islands. It was named after the Inca sun god, Viracocha, for whom "Kon-Tiki" was said to be an old name. Kon-Tiki is also the name of the popular book that Heyerdahl wrote about his adventures.

The name was given by the developers of the Contiki OS to the web browser they created. Later it became the name of the whole Operating System. While designing the cover page, we decided to connect the history of the name with a symbol for Safety.

Despite providing multitasking and a built-in TCP/IP stack, Contiki only needs about 10 kilobytes of RAM and 30 kilobytes of ROM. A full system, complete with a graphical user interface, needs about 30 kilobytes of RAM.

Contiki is designed to run on classes of hardware devices that are severely constrained in terms of memory, power, processing power, and communication bandwidth. A typical Contiki system has memory on the order of kilobytes, a power budget on the order of miniwatts, processing speed measured in megahertz, and communication bandwidth on the order of hundreds of kilobits/second. This class of systems includes both various types of embedded systems as well as a number of old 8-bit computers.

Many key mechanisms and ideas from Contiki have been widely adopt ed in the industry. The uIP (micro IP) embedded IP stack, originally releas ed in 2001, is today used by hundreds of companies in systems such as fr eighter ships, satellites and oil drilling equipment.

Three of them are very important: the uIP TCP/IP stack, which provides IPv4 networking,theuIPv6stack, whichprovides IPv6 networking, and the Rime stack, which is a set of custom lightweight networking protocols designed specifically for low-power wireless networks. The IPv6 stack was contributed by Cisco and was, at the time of release, the smallest IPv6 stack to receive the IPv6 Ready certification. The IPv6 stack also contains the RPL routing protocol for low-power lossy IPv6 networks and the 6LoWPAN header compression and adaptation layer for IEEE 802.15.4 links.

The Rime stack implements sensor network protocols ranging from reliable data collection and best-effort network flooding to multi-hop bulk data transfer and data dissemination. IP packets are tunneled over multi-h op routing via the Rime stack.

Many Contiki systems are severely power-constrained. To provid e a long sensor network lifetime, it is crucial to control and reduce the po wer consumption of each sensor node. Contiki provides a software-base d power profiling mechanism that keeps track of the energy expenditure o f each sensor node. Being software-based, the mechanism allows power p rofiling at the network scale without any additional hardware. Contiki's p ower profiling mechanism is used both as a research tool for experimenta l evaluation of sensor network protocols, and as a way to estimate the lifet ime of a network of sensors.

To run efficiently on memory-constrained systems, the Contiki pro-gramming model is based on protothreads. A protothread is a memo-ry-efficient programming abstraction that shares features of both multi-threading and event-driven programming to attain a low memory overhead of each protothread. The kernel invokes the prototh-read of a process in response to an internal or external event. Examples of internal events are timers that fire or messages being posted from other processes. Examples of external events are sensors that trigger or incom-ing packets from a radio neighbor.

In addition to protothreads, Contiki also supports per-process option al multithreading and interprocess communication using message passing. as well as an optional GUI subsystem with either direct graphic support for locally connected terminals or networked virtual display with VNC or over Telnet.

As presented operating systems for wireless sensor nodes have to fulfill a few requirements. After the look at both TinyOS and Contiki we now compare both operating systems by means of these requirements:

• Limited resources: The hardware platforms other very limited resources so the operating system should use them efficiently.

• Concurrency: The operating system should be able to handle different tasks at the same time.

• Flexibility: Since the requirements for different applications vary wildly, the operating system should be able to be flexible to handle those.

• Low Power: Energy conservation should be one of the main goals for the operating system..

Both operating systems can generally fulfill all of the discussed requirements. In details there are differences, so while TinyOS is better suited when resources are really scarce and every little bit of saved memory or computing power can help, Contiki might be the better choice when flexibility is most important, for example when the node software has to be updated often for a large amount of nodes[4].

2.4 Erlang

Erlang was designed by Ericsson to support distributed, fault-toleran t, soft-real-time, non-stop applications. It is a general-purpose concurr ent, garbage-collected programming language and runtime system. It s upports hot swapping, so that code can be changed without stopping system.

The main characteristics for Erlang are included:

High-Level Constructs:

Erlang is a declarative language. Declarative languages has the principle of trying to describe what should be computed, rather than saying how this value is calculated. A function definition—particularly one that uses pattern matching to select among different cases, and to extract components from complex data structures—will read like a set of equations.

In Erlang, you can pattern-match not only on high-level data but also on bit sequences, allowing a startlingly high-level description of protocol manipulation functions.

Concurrent Processes and Message Passing[5]:

Erlang's main strength is support for concurrency. It has a small but powerful set of primitives to create processes and communicate among them. Erlang's concurrency implementation is the Actor model. They are neither operating system processes nor operating system threads, but lightweight processes. While threads require external library support in most languages, Erlang provides language-level features for creating and managing processes with the aim of simplifying concurrent programming. Though all concurrency is explicit in Erlang, processes communicate using message passing instead of shared variables, which removes the need for explicit locks. Processes communicate with each other via message passing, where the message can be any Erlang data value. Message passing is asynchronous, so once a message is sent, the process can continue processing. Messages are retrieved from the process mailbox selectively, so it is not necessary to process messages in the order they are received. This makes the concurrency more robust, particularly when processes are distributed across different computers and the order in which messages are received will depend on ambient network conditions.

Scalable, Safe, and Efficient Concurrency[5]:

Erlang concurrency is fast and scalable. The estimated minimal overhead for each process is 300 words. Thus, many processes can be created without degrading performance. Its processes are lightweight in that the Erlang virtual machine does not create an OS thread for every created process. They are created, scheduled, and handled in the VM, independent of the underlying operating system. As a result, process creation time is of the order of microseconds and independent of the number of concurrently existing processes. Compare this with Java, where for every process an underlying OS thread is created: you will get some very competitive comparisons, with Erlang greatly outperforming both languages.

Soft Real-Time Properties:

Even though Erlang is a high-level language, you can use it for tasks with soft real-time constraints. Storage management in Erlang is automated, with garbage collection implemented on a per-process basis. This gives system response times on the order of milliseconds even in the presence of garbage-collected memory. Because of this, Erlang can handle high loads with no degradation in throughput, even during sustained peaks.

Language-level Dynamic Software Updating:

Erlang supports language-level Dynamic Software Updating. To implement this, code is loaded and managed as "module" units; the module is a compilation unit. The system can keep two versions of a module in memory at the same time, and processes can concurrently run code from each. The versions are referred to as the "new" and the "old" version. A process will not move into the new version until it makes an external call to its module.

Robustness:

Thanks to a set of simple but powerful error-handling mechanisms and exception monitoring constructs, very general library modules have been built, with robustness designed into their core. By programming for the correct case and letting these libraries handle the errors, not only are programs shorter and easier to understand, but they will usually contain fewer bugs.

Open Language:

Erlang is an open language allowing you to integrate legacy code or new code where programming languages other than Erlang. As a result, there are mechanisms for interworking with C, Java, Ruby, and other programming languages, including Python, Perl, and Lisp. High-level libraries allow Erlang nodes to communicate with nodes executing Java or C, making them appear and behave like distributed Erlang nodes. Other external languages can be tied in more tightly using drivers that are linked into the Erlang runtime system itself, as a device driver would be, and sockets can also be used for communication between Erlang nodes and systems written in other languages using popular protocols such as HTTP, SNMP, and IIOP.

Erlang and Multicore:

Separate processes with no shared memory communicating via message passing， naturally transfers to multicore processors in a way that is largely transparent to the programmer, so that you can run your Erlang programs on more powerful hardware without having to redesign them. With its approach that avoids shared data, Erlang is the perfect fit for multi-core processors, in effect solving many of the synchronization problems and bottlenecks that arise with many conventional programming languages.

Its declarative nature makes Erlang programs short and compact, and its built-in features make it ideal for fault-tolerant, soft real-time systems. Erlang also comes with very strong integration capabilities, so Erlang systems can be seamlessly incorporated into larger systems. This means that gradually bringing Erlang into a system and displacing less-capable conventional languages is not at all unusual. The language itself, the virtual machine, and its libraries have been keeping pace with the rapidly changing requirements of the software industry.

Our target system is a high-level, concurrent, robust, soft real-time system that will scale in line with demand, make full use of multi-core processors, and integrate with components written in other languages, so we choose Erlang.

2.5 Erlang for the Internet of Things

2.5.1 Introduction of ELIOT

ELIOT is a programming framework for the Internet of Things, by adapting the Erlang programming language to the requirements and needs of the embedded communication-oriented world, allowing localized and remote interactions, using high-level languages apt to manipulate data and protocols, offering powerful tools to debug, test and verify applications before their deployment, providing a Virtual Machine and language constructs apt to the development of distributed mobile. simulating tens or hundreds of devices and at least some of the characteristics of the environment, providing an abstraction over the hardware on which the applications will be executed, introducing new language constructs apt to the so-called “Internet of Things” world. At the same time, such a framework would cover a great part of the device spectrum, at worst with acceptable compromises for the less powerful devices.

The framework will also provide developers with tools for verifying their code: a simulator that allows executing the application on a simulated environment without changing a single line of code, static analysis tools already provided with the Erlang libraries and compatible with ELIoT, and a model checker that verifies properties of the application being developed. Other virtual machine capabilities are leveraged to allow hot swap of code and live interactions with the running system.

2.5.2 The framework of ELIOT

This section shows the differences in syntax and semantics with respect to Erlang; it introduces the ELIoT Virtual Machine, a modification of the original VM.

The ELIOT language[6]:

Here we describe ELIoT's dedicated language constructs, which concern three key aspects of inter-process communication when developing IoT applications:

- handling different communication guarantees
- supporting extended addressing schemes
- providing access to low-level information from the networking stack.

Erlang inter-process communication is based on the *!* operator, ELIoT complements Erlang's *!* operator with a new operator: ~, which models unreliable, best effort, sending of messages. Besides adding the ~ operator, ELIoT addresses possible faults of the underlying communication protocol by slightly changing the behavior of the *!* operator, the network cannot guarantee some properties anymore, and (as a consequence) neither the network protocol; this means that we needed to give the developer a way to know if the communication fails: in Erlang the send primitive returns immediately, regardless of the destiny of the message; in ELIoT, instead, in presence of communication faults that cannot be resolved, the framework places a special nack message into the sender's incoming message queue. Programmers can realize application-specific failure handling mechanisms based on such notifications.

In IoT applications a process needs to send a message to all other reachable processes. This form of broadcast communication is often be used, either as a primitive at the application level or as a low-level mechanism to implement higher-level communication protocols. ELIoT supports these scenarios by offering a richer addressing scheme than Erlang. In particular, ELIoT messages addressed to {n, all} reach processes with name n running on all reachable VMs.

The ELIOT virtual machine[6]:

VM consumes large quantities of memory to load some of these modules and launch several services during startup. To address this issue, ELIOT have developed a custom version of the VM, wiping off all the libraries that are not needed to run, These libraries can be re-added if necessary, but the network communication modifications may require them to be modified: only those provided by ELIoT have already been adapted to the new VM;

Several aspects of the VM mechanisms also have been changed, in particular regarding the network stack.

The structure of an ELIoT deployment is :the VM runs on the GNU/Linux operating system .with a network driver developed distinctly and adaptable to different network types .On top of the VM run the OTP libraries4, the hardware interfaces to interact with sensors and actuators, and the ELIoT ,which contains a small number of functions that substitute the equivalent ones from Erlang standard library and allow the simulator to run the applications unmodified On top of these libraries runs the ELIoT application itself .

2.6: COAP

2.6.1 Introduction of COAP

Constrained Application Protocol (CoAP) is a specialized web transfer protocol intended to be used in very simple electronics devices that allows them to communicate interactively over the Internet. It is particularly targeted for small low power sensors, switches and similar components that need to be controlled or supervised remotely, through standard Internet networks. CoAP is an application layer protocol that is intended for use in resource-constrained internet devices and constrained networks. The nodes often have 8-bit microcontrollers with small amounts of ROM and RAM, while constrained networks such as 6LoWPAN often have high packet error rates and a typical throughput of 10s of kbit/s. CoAP is designed for machine-to-machine (M2M) applications such as smart energy and building automation. Which tend to be deeply embedded and have much less memory and power supply than traditional internet devices have. Therefore, efficiency is very important. It is designed to easily interface with HTTP for integration with the Web while meeting specialized requirements such as multicast support. CoAP can run on most devices that support UDP or a UDP analogue

One of the main goals of CoAP is to design a generic web protocol for the special requirements of this constrained environment, especially considering energy, building automation and other machine-to-machine (M2M) applications. Although CoAP could be used for refashioning simple HTTP interfaces into a more compact protocol, it more importantly also offers features for M2M. CoAP provides a request/response interaction model between application endpoints, supports built-in discovery of services and resources, and includes key concepts of the Web such as URIs and Internet media types.

CoAP has the following main features:

Constrained web protocol fulfills M2M requirements. UDP binding with optional reliability supports unicast and multicast requests. Asynchronous message exchanges.

Low header overhead and parsing complexity

URI and Content-type support.

Simple caching based on max-age.

The mapping of CoAP with HTTP is stateless, allowing proxies to be built providing access to CoAP resources via HTTP in a uniform way. access to CoAP resources via HTTP in a uniform way or for HTTP simple interfaces to be realized alternatively over CoAP.

Simple subscription for resources, and resulting push notifications.

2.6.2 COAP Message

The interaction model of CoAP is similar to the client/server model of HTTP. However, machine-to-machine interactions typically result in a CoAP implementation acting in both client and server roles. A CoAP request is equivalent to that of HTTP, and is sent by a client to request an action (using a method code) on a resource (identified by a URI) on a server. The server then sends a response with a response code; this response may include a resource representation.

Unlike HTTP, CoAP deals with these interchanges asynchronously over a datagram-oriented transport such as UDP. This is done logically using a layer of messages that supports optional reliability (with exponential back-off). CoAP defines four types of messages: Confirmable, Non-confirmable, Acknowledgement, Reset; method codes and response codes included in some of these messages make them carry requests or responses. The basic exchanges of the four types of messages are somewhat orthogonal to the request/response interactions; requests can be carried in Confirmable and Nonconfirmable messages, and responses can be carried in these as well as piggy-backed in Acknowledgement messages.

One could think of CoAP logically as using a two-layer approach, a

CoAP messaging layer used to deal with UDP and the asynchronous nature of the interactions, and the request/response interactions using Method and Response codes (see the follow figure). CoAP is however a single protocol, with messaging and request/response just features of the CoAP header.

```
+----------------------+
|      Application     |
+----------------------+
+----------------------+
|  Requests/Responses  |
|----------------------|  CoAP
|       Messages       |
+----------------------+
+----------------------+
|          UDP         |
+----------------------+
```

Figure 2.1: Abstract layering of CoAP

Messaging Model[7]:

CoAP makes use of two message types, requests and responses, using a simple binary base header format. The base header may be followed by options in an optimized Type-Length-Value format. CoAP is by default bound to UDP and optionally to DTLS, providing a high level of communications security.

Any bytes after the headers in the packet are considered the message body if any. The length of the message body is implied by the datagram length. When bound to UDP the entire message MUST fit within a single datagram. When used with 6LoWPAN as defined in RFC 4944, messages SHOULD fit into a single IEEE 802.15.4 frame to minimize fragmentation.

Reliability is provided by marking a message as Confirmable (CON). A Confirmable message is retransmitted using a default timeout and exponential back-off between retransmissions, until the recipient sends an Acknowledgement message (ACK) with the same Message ID (in this example, 0x7d34) from the corresponding endpoint; see the fellow figure . When a recipient is not at all able to process a Confirmable message. It replies with a Reset message (RST) instead of an Acknowledgement (ACK)

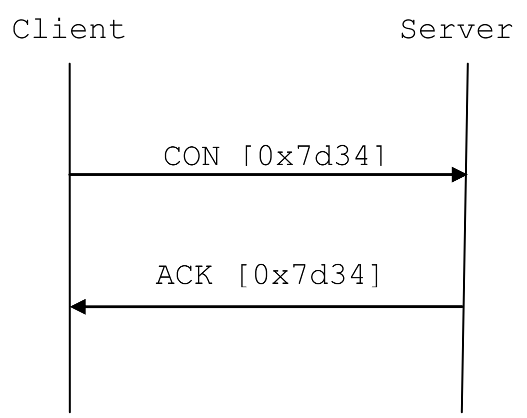


Figure 2.2 :Reliable message transmission

A message that does not require reliable transmission, for example each single measurement out of a stream of sensor data, can be sent as a Non-confirmable message (NON). These are not acknowledged, but still have a Message ID for duplicate detection (in this example,0x01a0); see Figure 3. When a recipient is not able to process a Non-confirmable message, it may reply with a Reset message (RST).

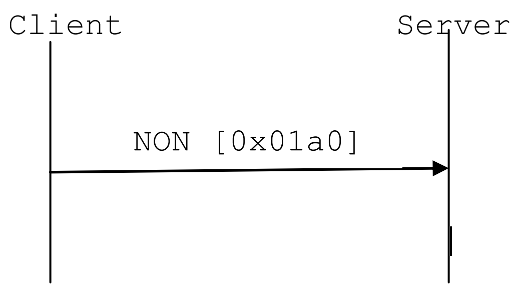


Figure 2.3:Unreliable message transmission

Request/Response Model:

CoAP request and response semantics are carried in CoAP messages, which include either a Method code or Response code, respectively. Optional (or default) request and response information, such as the URI and payload media type are carried as CoAP options. A Token is used to match responses to requests independently from the underlying messages (Note that the Token is a concept separate from the Message ID.)

A request is carried in a Confirmable (CON) or Non-confirmable (NON) message, and if immediately available, the response to a request carried in a Confirmable message is carried in the resulting Acknowledgement (ACK) message. This is called a piggy-backed response. (There is no need for separately acknowledging a piggy-backed response, as the client will retransmit the request if the Acknowledgement message carrying the piggy-backed response is lost.) Two examples for a basic GET request with piggy-backed response are shown in the follow figures , one successful, one resulting in a 4.04 (Not Found) response.

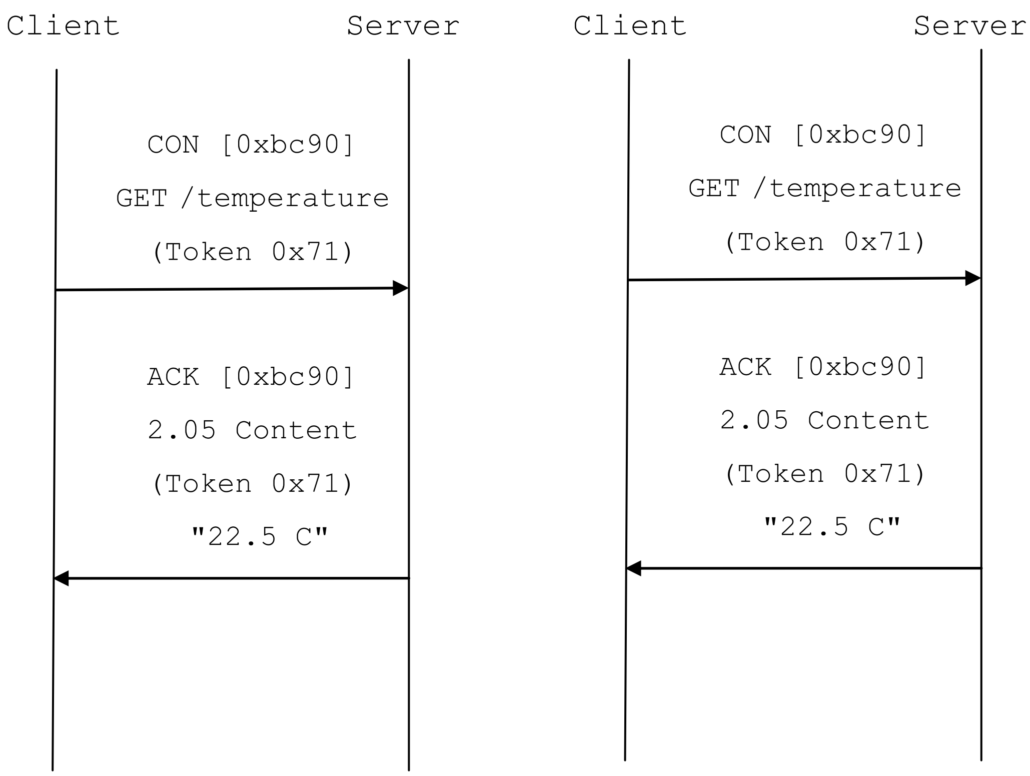


Figure 2.4: Two GET requests with piggy-backed responses

If the server is not able to respond immediately to a request carried in a Confirmable message, it simply responds with an Empty Acknowledgement message so that the client can stop retransmitting the request. When the response is ready, the server sends it in a new Confirmable message (which then in turn needs to be acknowledged by the client). This is called a separate response, as illustrated in the following figure.

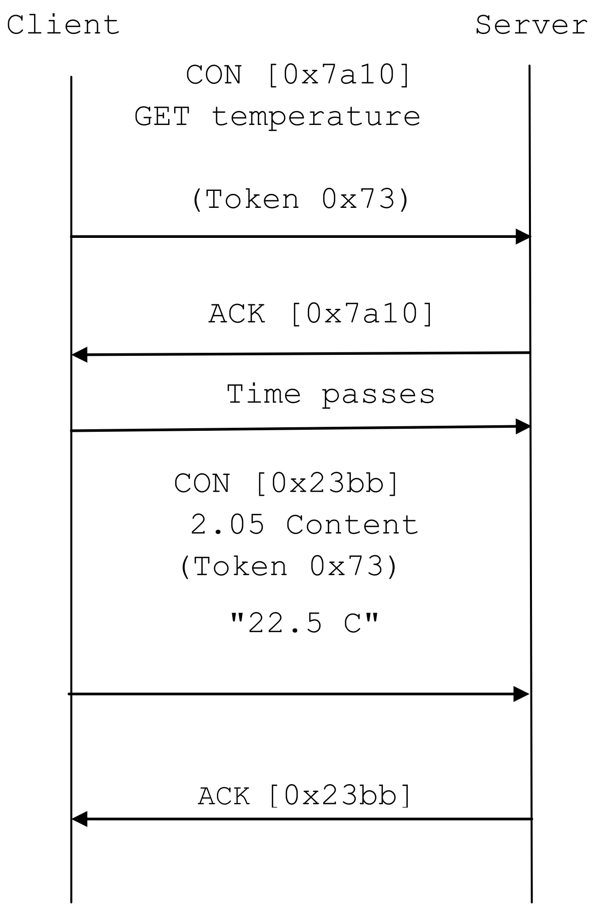


Figure 2.5 :A GET request with a separate response

If a request is sent in a Non-confirmable message, then the response is sent using a new Non-confirmable message, although the server may instead send a Confirmable message. This type of exchange is illustrated in the following figure.

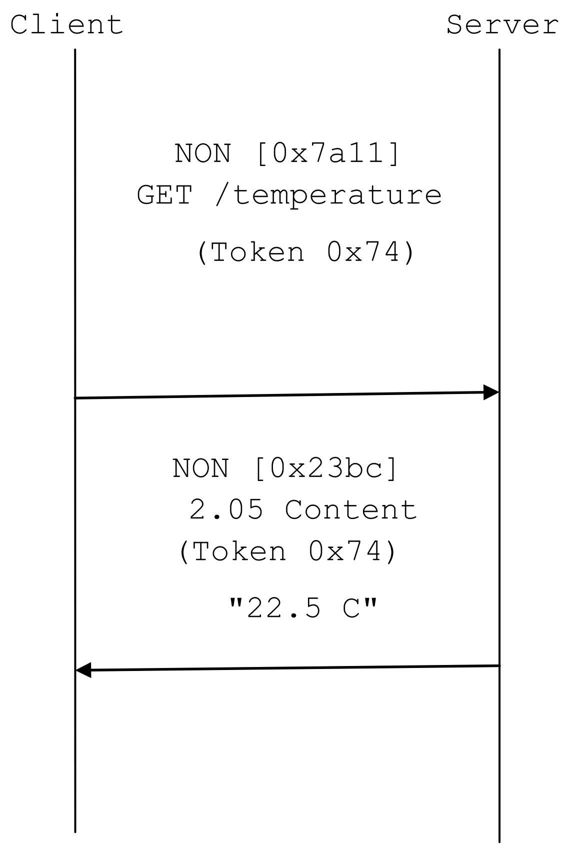


Figure 2.6:A NON request and response

2.6.3 Method Definitions

In this section each method is defined along with its behavior. A request with an unrecognized or unsupported Method Code MUST generates a 4.05 (Method Not Allowed) piggy-backed response.

GET:

The GET method retrieves a representation for the information that currently corresponds to the resource identified by the request URI. If the request includes an Accept Option, that indicates the preferred content-format of a response. If the request includes an ETag Option, the GET method requests that ETag be validated and that the representation be transferred only if validation failed. Upon success a 2.05 (Content) or 2.03 (Valid) response code SHOULD be present in the response.

The GET method is safe and idempotent.

POST:

The POST method requests that the representation enclosed in the request be processed. The actual function performed by the POST method is determined by the origin server and dependent on the target resource. It usually results in a new resource being created or the target resource being updated.

If a resource has been created on the server, the response returned by the server SHOULD have a 2.01 (Created) response code and SHOULD include the URI of the new resource in a sequence of one or more Location-Path and/or Location-Query Options (Section 5.10.7). If the POST succeeds but does not result in a new resource being created on the server, the response SHOULD have a 2.04 (Changed) response code. If the POST succeeds and results in the target resource being deleted, the response SHOULD have a 2.02 (Deleted) response code.

POST is neither safe nor idempotent.

PUT:

The PUT method requests that the resource identified by the request URI be updated or created with the enclosed representation. The representation format is specified by the media type and content coding given in the Content-Format Option, if provided. (Changed) response code SHOULD be returned.

If no resource exists then the server MAY create a new resource with that URI, resulting in a 2.01 (Created) response code. If the resource could not be created or modified, then an appropriate error response code SHOULD be sent.

Further restrictions to a PUT can be made by including the If-Matchor If-None-Match options in the request.

PUT is not safe, but is idempotent.

DELETE:

The DELETE method requests that the resource identified by the request URI be deleted. A 2.02 (Deleted) response code SHOULD be used on success or in case the resource did not exist before the request.

DELETE is not safe, but is idempotent.

2.6.4 CoAP URIs

CoAP uses the "coap" URI schemes for identifying CoAP resources and providing a means of locating the resource. Resources are organized hierarchically and governed by a potential CoAP origin server listening for CoAP requests ("coap") on a given UDP port. The CoAP server is identified via the generic syntax's authority component, which includes a host component and optional UDP port number. The remainder of the URI is considered to be identifying a resource which can be operated on by the methods defined by the CoAP protocol. The "coap" URI schemes can thus be compared to the "http" URI schemes.

<u>CoAP URI Scheme:</u>

coap-URI = "coap:" "//" host [":" port] path-abempty ["?" query]

If the host component is provided as an IP-literal or IPv4address, then the CoAP server can be reached at that IP address. If host is a registered name, then that name is considered an indirect identifier and the endpoint might use a name resolution service, such as DNS, to find the address of that host. The host MUST NOT be empty; if a URI is received with a missing authority or an empty host, then it MUST be considered invalid. The port subcomponent indicates the UDP port at which the CoAP server is located. If it is empty or not given, then the default port 5683 is assumed.

The path identifies a resource within the scope of the host and port. It consists of a sequence of path segments separated by a slash character (U+002F SOLIDUS "/"). The query serves to further parameterize the resource. It consists of a sequence of arguments separated by an ampersand character (U+0026 AMPERSAND "&"). An argument is often in the form of a"key=value" pair.

The "coap" URI scheme supports the path prefix "/.well-known/" defined by [RFC5785] for "well-known locations" in the name-space of a host. This enables discovery of policy or other information about a host ("site-wide metadata"), such as hosted resources .

Decomposing URIs into Options:

The steps to parse a request's options from a string |url| are as follows. These steps either result in zero or more of the Uri-Host, Uri-Port, Uri-Path and Uri-Query Options being included in the request, or they fail.

1. If the |url| string is not an absolute URI ([RFC3986]), then fail this

algorithm.

2. Resolve the |url| string using the process of reference resolution defined by [RFC3986]. At this stage the URL is in ASCII encoding [RFC0020], even though the decoded components will be interpreted in UTF-8 [RFC3629] after step 5, 8 and 9.

It doesn’t matter what it is resolved relative to, since we already know it is an absolute URL at this point.

3. If |url| does not have a <scheme> component whose value, when converted to ASCII lowercase, is "coap" or "coaps", then fail this algorithm.

4. If |url| has a <fragment> component, then fail this algorithm.

5. If the <host> component of |url| does not represent the request’s destination IP address as an IP-literal or IPv4address, include a Uri-Host Option and let that option’s value be the value of the <host> component of |url|, converted to ASCII lowercase, and then converting all percent-encodings ("%" followed by two hexadecimal digits) to the corresponding characters.

In the usual case where the request’s destination IP address is derived from the host part, this ensures that a Uri- Host Option is only used for a <host> component of the form regname

6. If |url| has a <port> component, then let |port| be that component’s value interpreted as a decimal integer; otherwise, let |port| be the default

port for the scheme.

7. If |port| does not equal the request’s destination UDP port, include a Uri-Port Option and let that option’s value be |port|.

8. If the value of the <path> component of |url| is empty or consists of a single slash character (U+002F SOLIDUS "/"), then move to the next step.

Otherwise, for each segment in the <path> component, include a

Uri-Path Option and let that option’s value be the segment (not including the delimiting slash characters) after converting each percent-encoding ("%" followed by two hexadecimal digits) to the corresponding byte.

9. If |url| has a <query> component, then, for each argument in the <query> component, include a Uri-Query Option and let that option’s value be the argument (not including the question mark and the delimiting ampersand characters) after converting each percent-encoding to the corresponding byte.

Note that these rules completely resolve any percent-encoding.

Composing URIs from Options:

The steps to construct a URI from a request’s options are as follows. These steps either result in a URI, or they fail. In these steps, percent-encoding a character means replacing each of its (UTF-8 encoded) bytes by a "%" character followed by two hexadecimal digits representing the byte, where the digits A-F are in upper case (as defined in [RFC3986] to reduce variability, the hexadecimal notation for percent-encoding in CoAP URIs MUST use uppercase letters). The definitions of "unreserved" and "sub-delims" are adopted from [RFC3986].

1. If the request is secured using DTLS, let |url| be the string "coaps://". Otherwise, let |url| be the string "coap://".

2. If the request includes a Uri-Host Option, let |host| be that option's value, where any non-ASCII characters are replaced by their corresponding percent-encoding. If |host| is not a valid reg-name or IP-literal or IPv4address, fail the algorithm. If the request does not include a Uri-Host Option, let |host| be the IP-literal (making use of the conventions of [RFC5952]) or IPv4address representing the request's destination IP address.

3. Append |host| to |url|.

4. If the request includes a Uri-Port Option, let |port| be that option's value. Otherwise, let |port| be the request's destination UDP port.

5. If |port| is not the default port for the scheme, then append a single U+003A COLON character (:) followed by the decimal representation of |port| to |url|.

6. Let |resource name| be the empty string. For each Uri-Path Option in the request, append a single character U+002F SOLIDUS(/) followed by

the option’s value to |resource name|, after converting any character that is not either in the "unreserved"set, "sub-delims" set, a U+003A COLON (:) or U+0040 COMMERCIALAT (@) character, to its percent-encoded form.

7. If |resource name| is the empty string, set it to a single character U+002F SOLIDUS (/).

8. For each Uri-Query Option in the request, append a single character U+003F QUESTION MARK (?) (first option) or U+0026 AMPERSAND (&) (subsequent options) followed by the option’s value to |resource name|, after converting any character that is not either in the "unreserved" set, "sub-delims" set (except U+0026 AMPERSAND (&)), a U+003A COLON (:), U+0040 COMMERCIAL AT(@), U+002F SOLIDUS (/) or U+003F QUESTION MARK (?) character, to its percent-encoded form.

2.7 Tmote-sky

The Tmote Sky module is a low power “mote” with integrated sensors, radio, antenna, microcontroller, and programming capabilities.

Power:

Tmote Sky is powered by two AA batteries. The module was designed to fit the two AA battery form factor. AA cells may be used in the operating range of 2.1 to 3.6V DC, however the voltage must be at least 2.7V when programming the microcontroller flash or external flash.

If the Tmote Sky module is plugged into the USB port for prog-ramming or communication, it will receive power from the host co-mputer. The mote operating voltage when attached to USB is 3V.If Tmote will always be attached to a USB port, no battery pack is necessary. In our case, we use this power supply mode.

Microprocessor[8]:

The low power operation of the Tmote Sky module is due to the ultra low power Texas Instruments MSP430 F1611 microcontroller featuring 10kB of RAM, 48kB of flash, and 128B of information storage. This 16-bit RISC processor features extremely low active and sleep current consumption that permits Tmote to run for years on a single pair of AA batteries. The MSP430 has an internal digitally controlled oscillator (DCO) that may operate up to 8MHz. The DCO may be turned on from sleep mode in 6 μ s, however 292ns is typical at room temperature. When the DCO is off, the MSP430 operates off an eternal 32768Hz watch crys-tal. Although the DCO frequency changes with voltage and temperature, it may be calibrated by using the 32kHz oscillator.

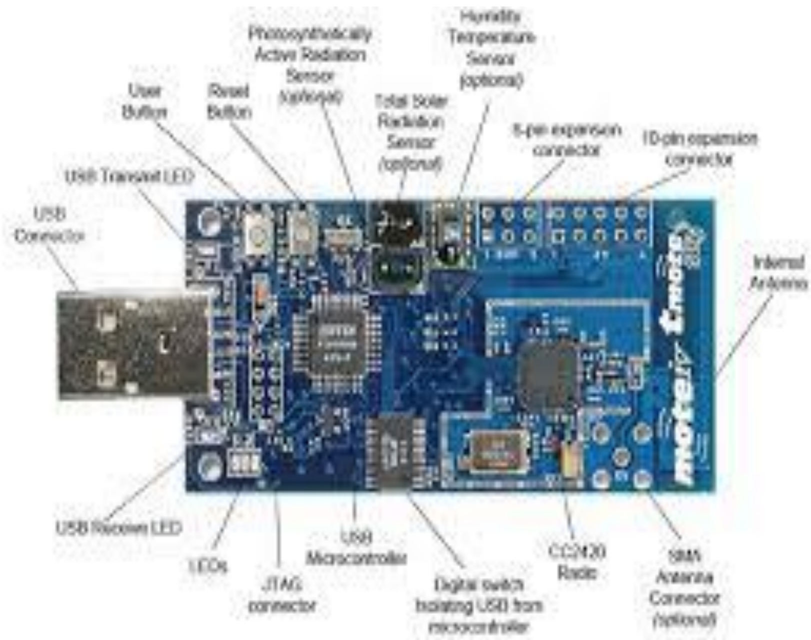
In addition to the DCO, the MSP430 has 8 external ADC ports and 8 internal ADC ports. The ADC internal ports may be used toread the in-ternal thermistor or monitor the battery voltage.

A variety of peripherals are available including SPI, UART, digital I/O ports, Watchdog timer, and Timers with capture and compare functionality. The F1611 also includes a 2-port 12-bit DAC module, Supply Voltage Supervisor, and 3-port DMA controller.

Programming:

The Tmote Sky module is programmed through the onboard USB connector. A modified version of the MSP430 Bootstrap Loader, msp430-bsl, programs the microcontroller's flash.

Tmote Sky has a unique hardware circuit that prevents spurious resets. This hardware circuit makes it necessary for a special sequence to be sent to the module in order to program it.



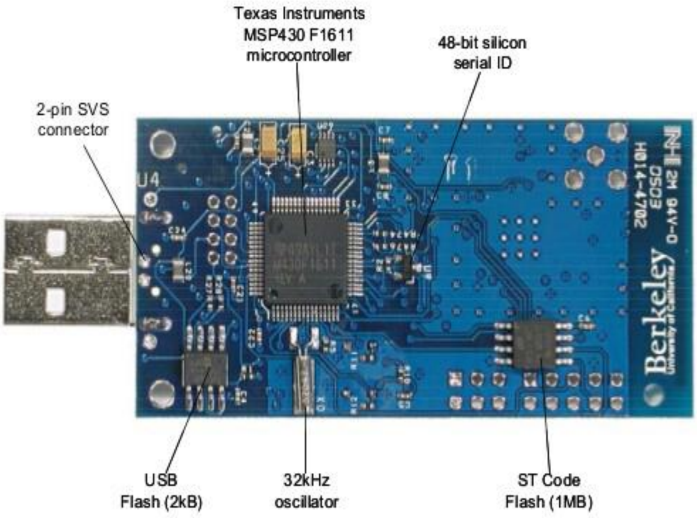


Figure 2.7:Front and back of the Tmote Sky module

3 Design

3.1 The scenario

This chapter we will introduce the scenario used in this project, through this document we will illustrate and validate the implementation of CoAP protocol.

With the usual TCP/IP stack, the CoAP protocol should be put on top of the transport Layer that is, the Application layer, like the following figure shows,

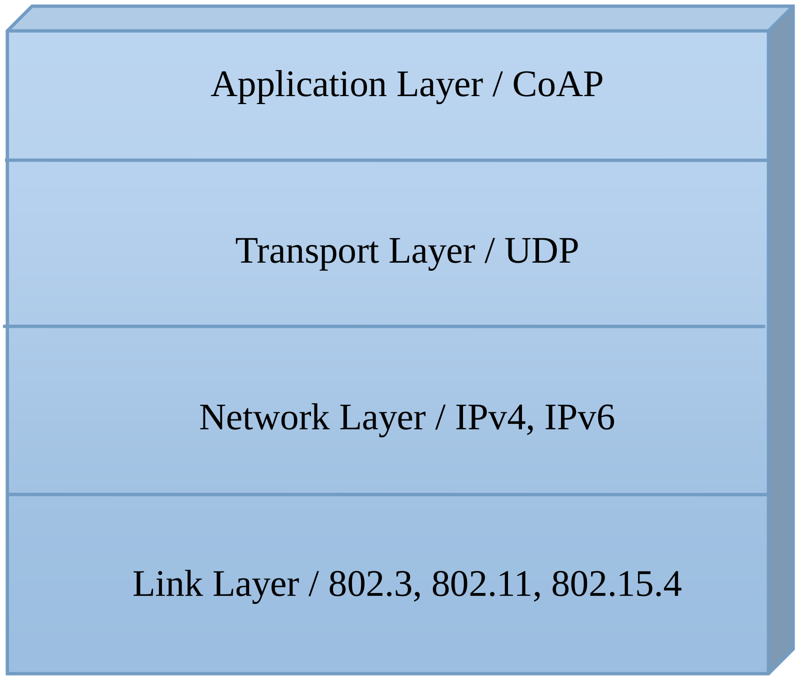


Figure 3.1: protocol stack

3.1.1 System aim

In particular, the design is aimed to provide an Erlang API for CoAP protocol, a gateway for network layer conversion, and a Contiki CoAP application to process the request. According to the aim of the thesis, we designed and program the application, make all the modules work efficiently and reliably and then test them, producing the results described in the next chapters.

3.2 Analysis

The scenario which we considered is very simple, in particular, the basic design involves three parts: client part, gateway part, and server part, like what shows in the figure follows,

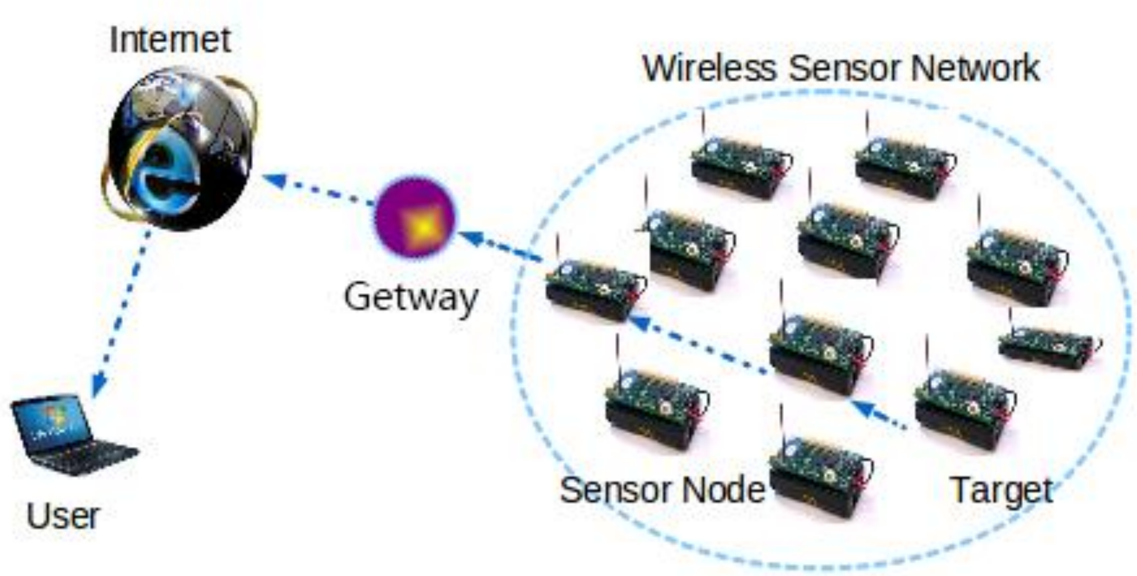


Figure 3.2 : The scenario of the thesis

3.2.1 Client

Client's behavior likes a HTTP browser, it needs some information from remote server and then the URI(Uniform Resource Identifier) of the resource, just like when you need to visit Politecnico di Milano English version web page, you need to know not only the address of the server(www.polimi.it), but also the URI of the English version URI(/en/English-version/).

After you get all of this information, now you can send a packet as follows:

IP HEADER || UDP HEADER || COAP HERDER || COAP_OPTION_URI = LIGHT

Then the client waits for a response from the server.

3.2.2 Server

In this scenario server is one of the sensor nodes. It opens a UDP port and listens for a request. Once get the request packet from a client, it reads the packet, according to the options in the CoAP packet. For example, once sensor node get a CoAP packet with URI(Uniform Resource Identifier) option “light”, the sensor node begins to turn on the light sensor，get the instant value, and encapsulate it into the response packet, and then send it out. The packet is as follows；

IP HEADER || UDP HEADER || COAP HERDER || COAP_PAYLOAD = 100

Now let us see a communication example between client and server:

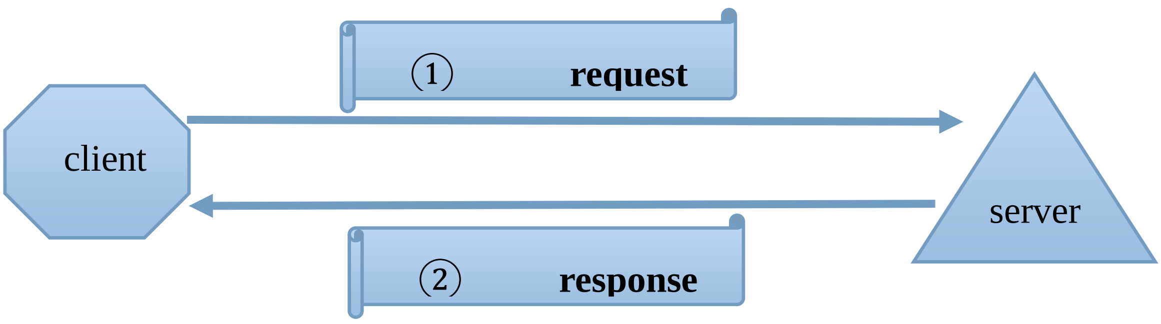


Figure 3.3 : C/S communication model

CoAP request and response semantics are carried in CoAP messages, which include either a Method code or Response code. Optional request and response information, such as the URI , URI_QUERY, or payload. A Token is used to match responses to requests independently from the underlying messages.(Note that the Token is a concept different from the Message ID.)

CON [0xbc90]
GET /temperature
(Token 0x71)

Figure 3.4 : Request Packet

ACK [0xbc90]
2.05 Content
(Token 0x71)
"22.5 C"

Figure 3.5 : Response Packet

3.2.3 Gateway

Things are not as easy as above said. Because now we want clients to work in the normal Internet environment, it means to use standard TCP/IP stack(IP version 4 or IP version 6) and standard IEEE 802.3 or IEEE 802.11 at the link layer, but our sensor nodes use IEEE 802.15.4 at link layer and IPv6 at network layer. So what the gateway should do , is like a converter, that means convert both the link layer and network layer into different protocol environment. For example from IEEE 802.11 and IPv4 stack to the IEEE802.15.4 and IPvt6 stack.

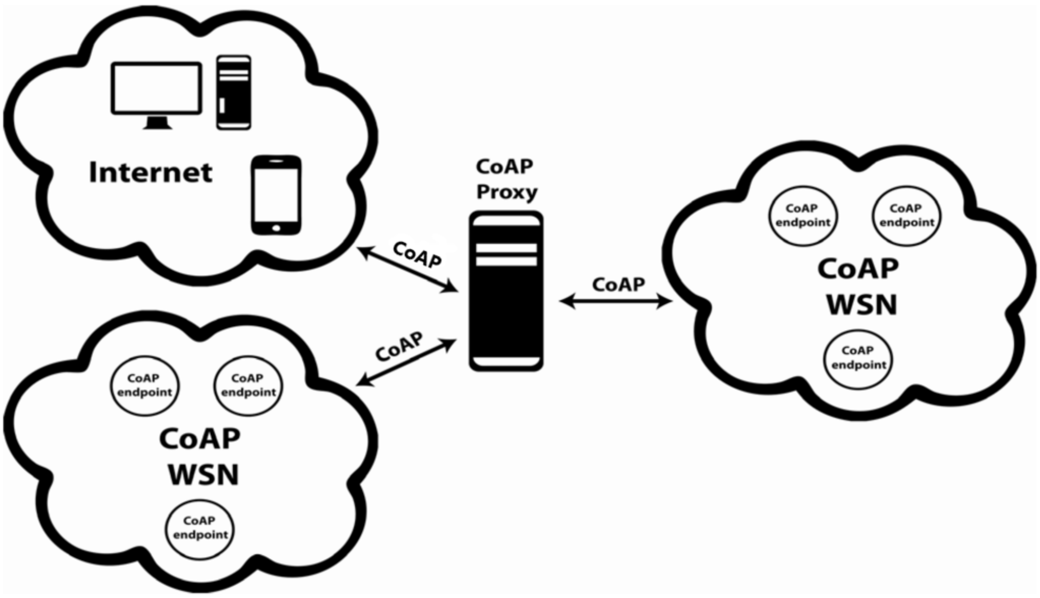


Figure 3.6:Application Model

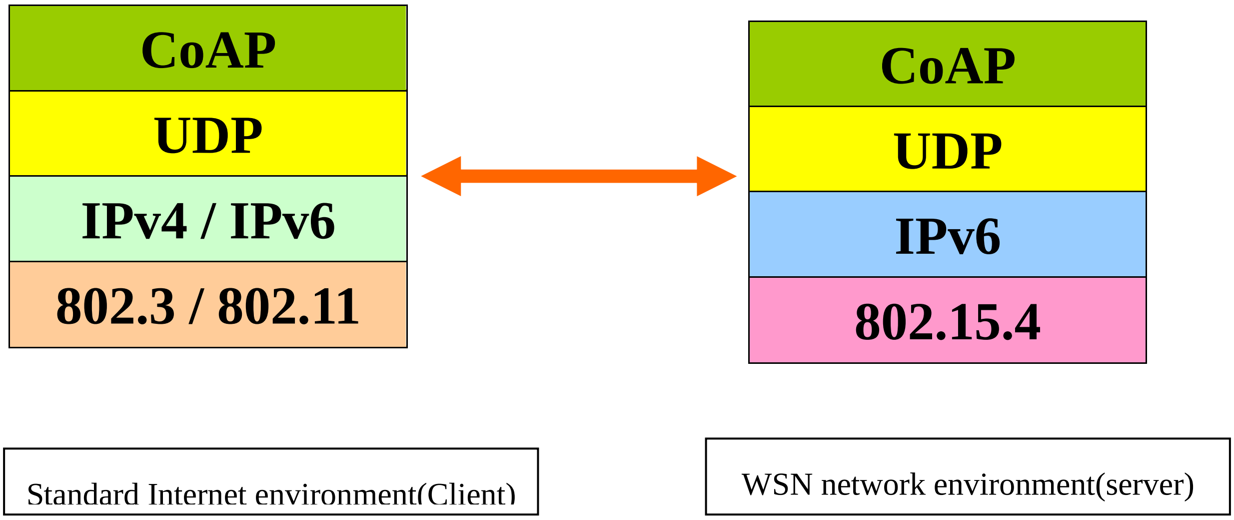


Figure 3.7: Protocol stack for the application model

3.3 Implementation

In the last subchapter we described the aim of the system and what we need in the system, now let us begin to implement them with Erlang and Contiki.

3.3.1 CoAP over Erlang

Erlang is a general-purpose concurrent, garbage-collected programming language and runtime system. The sequential subset of Erlang is a functional language, with eager evaluation, single assignment, and dynamic typing. It was designed by Ericsson to support distributed, fault-tolerant, soft-real-time, non-stop applications. It support shot swapping, so that code can be changed without stopping a system.

While threads require external library support in most languages, Erlang provides language-level features for creating and managing processes with the aim of simplifying concurrent programming. Though all concurrency is explicit in Erlang, processes communicate using message passing instead of shared variables, which removes the need for explicit locks (a locking scheme is still used internally by the VM).

This subchapter introduce how to implement an Erlang API to analysis and encapsulate CoAP packet.

Example of NIF interface in Erlang language

Erlang is a high-level language, although it support binary structure, but sometimes we already have libraries that are written in other languages like C or C++. So we do not need to rewrite a Erlang library to encapsulate and analysis packets. In order to use C library we need a technique called NIF(Native Implemented Function).

A NIF (Native Implemented Function) is a function that is implemented in C instead of Erlang. NIF appears as any other functions to the callers. They belong to a module and are called like any other Erlang functions. The NIFs of a module are compiled and linked into a dynamic loadable shared library (SO in Unix, DLL in Windows). The NIF library must be loaded at runtime by the Erlang code of the module.

Since a NIF library is dynamically linked into the emulator process, this is the fastest way of calling C-code from Erlang (alongside port drivers). Calling NIFs requires no context switches.

Even all the functions of a module written by C, and implemented by NIFs, we still need a Erlang module to load the code. Let us see a easy NIF example:

```
1.  niftest.c
2.  #include "erl_nif.h"
3.  static ERL_NIF_TERM hello(ErlNifEnv* env, int argc, const ERL_NIF_TERM argv[])
4.  {
5.      return enif_make_string(env, "Hello world!", ERL_NIF_LATIN1);
6.  }
7.  static ErlNifFunc nif_funcs[] =
8.  {
9.      {"hello", 0, hello}
10. };
11. ERL_NIF_INIT(niftest,nif_funcs,NULL,NULL,NULL,NULL)
```

The first argument of the ERL_NIF_INIT function should be the module name of the Erlang module, the second argument of this function is the structure array name from line 7 to line 10. The argument left are pointers to the callback functions which can be used to initialize the NIF library, we do not use them.

On Linux platform we can compile this file by this way:

```
12. gcc -fPIC -shared -o niftest.so niftest.c
```

Niftest.c will be compiled to a dynamic library niftest.so, so that when Erlang is needed, the Erlang VM can load it.

Let us see the Erlang module:

```
niftest.erl
-module(niftest).
-export([init/0, hello/0]).
init() ->
      erlang:load_nif("./niftest", 0).
hello() ->
      "NIF library not loaded".
```

Now let us begin to use this module,

```
1> c(niftest).
{ok,niftest}
2> niftest:hello().
"NIF library not loaded"
3> niftest:init().
ok
4> niftest:hello().
"Hello world!"
```

To encapsulate and analysis packet we use a library called Cantcoap. Cantcoap library offers a minimal set of functions to serialize and de-serialize CoAP packets, and it focuses on simplicity and a straightforward interface.

We use this library to help us building a Erlang NIF interface, so we can directly process the CoAP packet without rewriting Erlang low layer functions.

Here we will only show you the Erlang module part, you can find the C implementation in the attachment files.

```
-module(pdu).
-export([init/0,make_pdu/5,get_content/1,add_option/4,add_payload/3,get_header/1,get_token/1,get_URI/1,get_option/2]).
-on_load(init/0).
-define(APPNAME,wsngateway).
init()->
    case code:priv_dir(?APPNAME) of
        {error, _} ->
            error_logger:format("~wpriv dir not found~n", [?APPNAME]),
            exit(error);
        PrivDir ->
            erlang:load_nif(filename:join([PrivDir, "pdu_drv"]), 0)
    end.
%% @spec make_pdu(Type::integer(), Method::integer(), Token::list(), MID::integer(), URI::list()) ->
%% {ok, PDU::binary()} | {error, Reason::string()}
make_pdu(_Type,_Method,_Token,_ID,_URI)->
    erlang:nif_error(nif_not_loaded).
%% @spec get_content(Buffer::binary()) ->
%% {ok, newpdu::bianry()} | {error, Reason::string()}
get_content(_Buffer)->
    erlang:nif_error(nif_not_loaded).
%% @spec add_option(PDU::binary(), Optnum::integer(), Optlen::integer(), Optval::list()) ->
%% {ok ,newpdu::binary()} | {error, Reason::string()}
add_option(_PDU,_Optnum,_Optlen,_Opnval)->
    erlang:nif_error(nif_not_loaded).
%% @spec add_pauload(PDU::binary(), Payloadvalue::list(), Payloadlen::integer()) ->
%% {ok, newpdu::binary()} | {error, Reason::string()}
add_payload(_PDU,_Payloadvalue,_Payloadlen)->
    erlang:nif_error(nif_not_loaded).
%% @spec get_header(PDU::binary()) ->
%% {ok, Version::integer(), Type::integer(), Tokenlength::integer(), Code::integer(), message::integer()} |
%% {error, Reason::string()}
get_header(_PDU)->
    erlang:nif_error(nif_not_loaded).
%% @spec get_token(PDU::binary()) ->
%% {ok, Token::binary()} | {error, Reason::string()}
get_token(_PDU)->
    erlang:nif_error(nif_not_loaded).
%% @spec get_URI(PDU::binary()) ->
```

```
%% {ok, URI::atom()} | {error, Reason::string()}
get_URI(_PUD)->
    erlang:nif_error(nif_not_loaded).
%% @spec get_option(PDU::binary(), Optnum::integer()) ->
%% {ok, optionvalue::string()} | {error, Reason::string()}
get_option(_PDU,_Optnum)->
    erlang:nif_error(nif_not_loaded)
```

In this module, we give some APIs to allow you to make the CoAP PDU very convenient.

Function *init* will be automatically loaded when you first use the functions in this module.

Function *make_pdu*, it needs five parameters, and returns a tuple {ok, newpdu} or {error, reason}. So if everything goes well, you can get your PDU with only some mandatory things for a CoAP PDU.

For function *get_content*, you should give the binary PDU you have, and it will return back the payload of the PDU, {ok, pdu} or {error, reason}.

Function *add_option* and a*dd_payload* are offering a way to add options and payload to a exist CoAP PDU. It returns {ok, pdu} or {error, reason}.

Function *get_header* can help you to analysis the CoAP packet, gives it an exist CoAP packet, it returns a six member tuple {ok, version, type, token length, code, message ID} or {error, reason}.

Functions *get_token*, *get_URI* and *get_option* like what's their name says they return the token, URI and one of the option you need.

Use pdu module to have a client

What the client should do, just like a HTTP browse, we give the request, and wait for the response from the remote server.

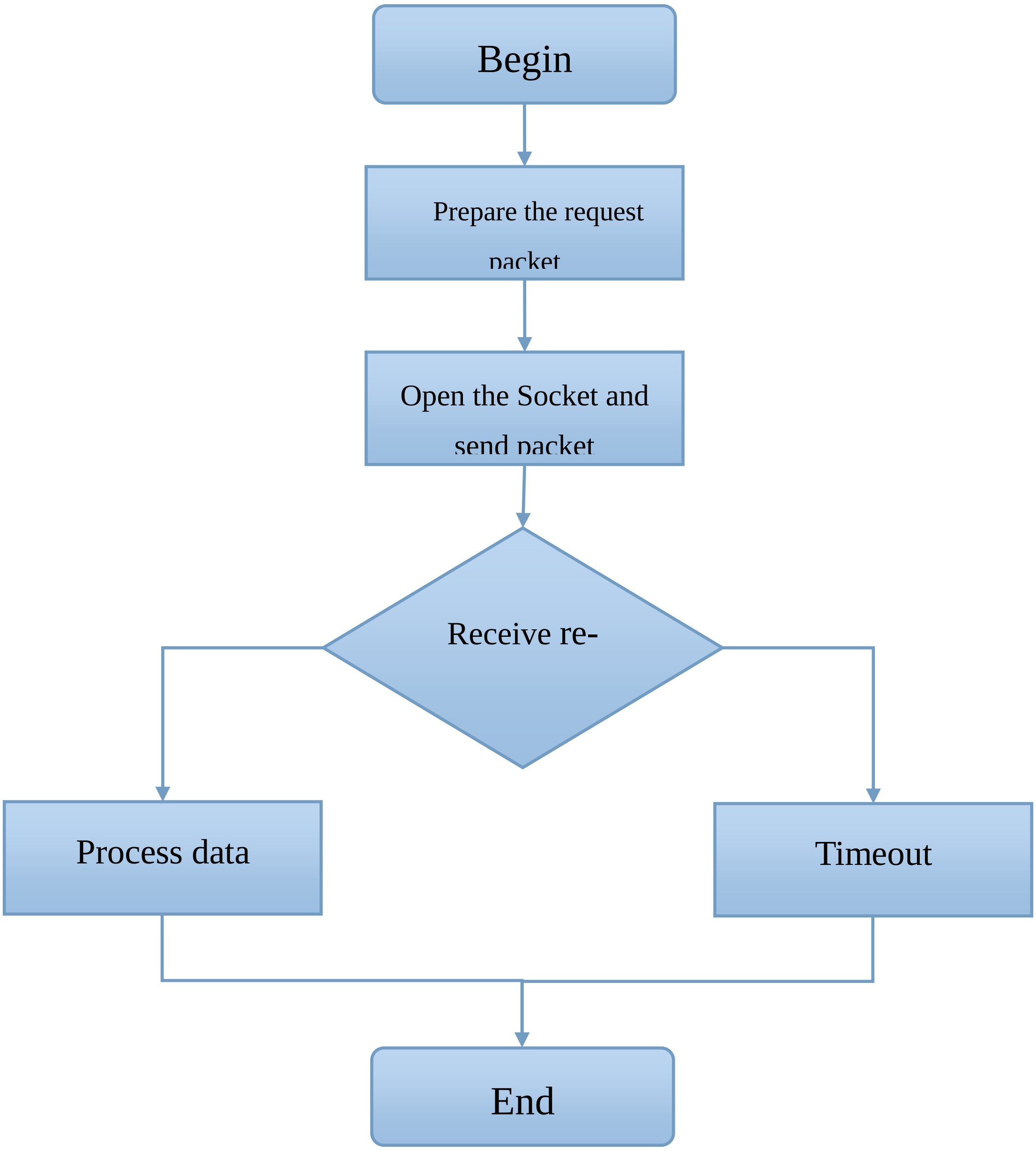


Figure 3.8: flow diagram for the client part

Let us have a look at one of the function, get method.

```
get(Host,URI,Para)->
    Token=make_token(),
    ID=make_message_id(),
```

```
{ok,PDU}=pdu:make_pdu(0,?COAP_GET,Token,ID,URI),
{ok,Newpdu}=getpara(PDU,Para),
{ok,Destaddr}=inet_parse:address(Host),
Case gen_udp:open(?TMP_PORT,[binary,inet,{active,false}])of
    {ok,Socket}->
        gen_udp:send(Socket,Destaddr,?PORT,Newpdu),
        Res=casegen_udp:recv(Socket,0,6000)of
            {ok,{Destaddr,?PORT,Packet}}->
    {ok,Ver,Type,Tkl,Code,MID}=pdu:get_header(Packet),
        io:format("~p~n",[{ok,Ver,Type,Tkl,Code,MID}]),
        {ok,Content}=pdu:get_content(Packet),
        io:format("value is: ~p~n",[Content]);
    {error,Reason}->
        {error,Reason}
    end,
    gen_udp:close(Socket),
    Res;
{error,Reason}->
    {error,Reason}
end.
```

Other method are very similar, just change the method code.

Proxy

As what we have discussed before, a proxy should be compatible to either normal Internet IP layer and link layer, or the WSN network layer and link layer.

For the normal Internet part, because we use linux based device to act as a proxy, we do not need to consider how to implement this part, just use what linux offers us. For the WSN network part, we need to implement the network layer part, because we use IPv6 instead of IPv4, we can use the APIs which Contiki offers us, but what we need to do is that how to have correspondence between IPv6 world and IPv4 world. We now use what Erlang is good at.

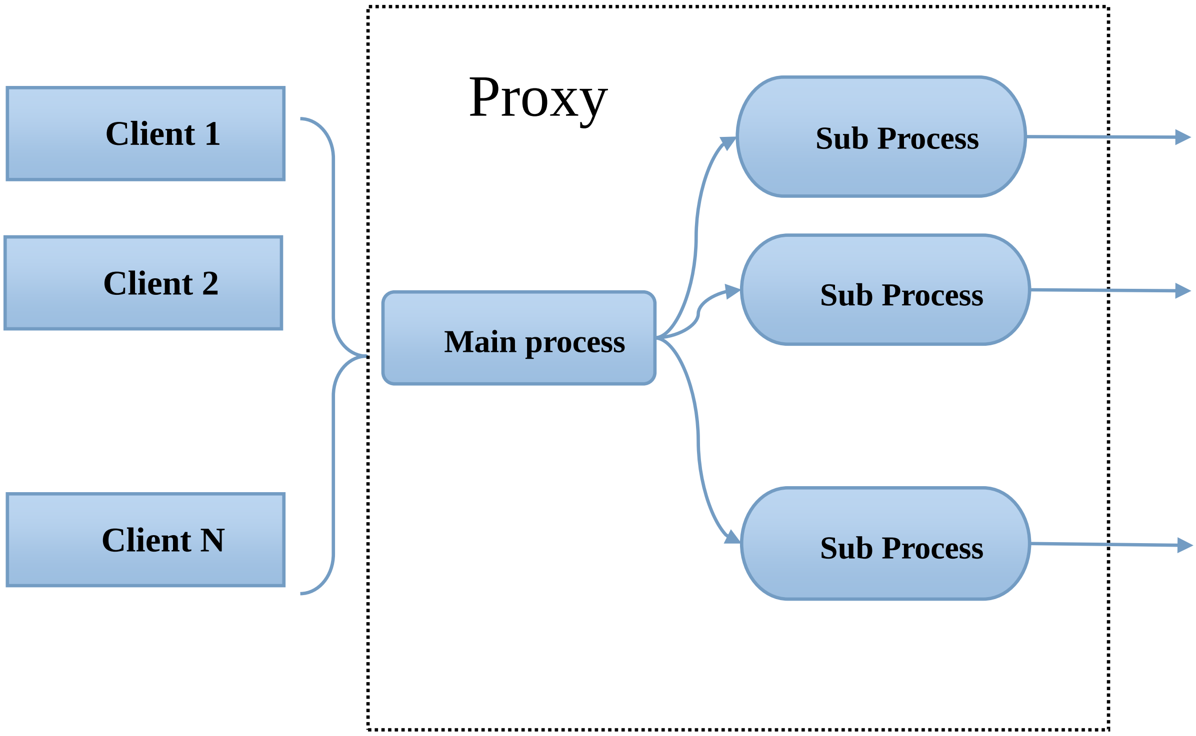


Figure 3.9: Spawn sub-process in proxy

```
proxy_recv(S,FromIPvN)->
    receive
        {udp,S,FromIP,FromPort,Bin}->
            %% spawn a process to send the message and wait the response
            spawn(fun()->wait_response(FromIP,FromPort,S,Bin,FromIPvN)end),
            proxy_recv(S,FromIPvN);
        {error,Reason}->
            io:format("{get error message, ~p}~n",[Reason]),
```

```
            proxy_recv(S,FromIPvN);
        stop->
            gen_udp:close(S),
            io:format("UDP port closed~n")
    End.
wait_response(FromIP,FromPort,Socket,Bin,FromIPvN)->
    casepdu:get_option(Bin,?COAP_OPTION_URI_HOST)of
        {ok, URI_HOST} ->
            io:format("Get URI_HOST: ~p~n",[URI_HOST]);
        {error, URI_HOST} ->
            io:format("get_option error ~p~n", [{error, URI_HOST}])
    end,
    {ok,RemoteIP}=inet_parse:address(URI_HOST),
    case pdu:get_option(Bin, ?COAP_OPTION_URI_PORT) of
        {ok, URI_PORT} ->
            io:format("Get URI_PORT: ~p~n",[URI_PORT]);
        {error, URI_PORT} ->
            io:format("get_option error ~p~n", [{error, URI_PORT}])
    end,
     RemotePort=list_to_integer(URI_PORT),
    Receive
        {udp,Socket_to,RemoteIP,RemotePort,B}->
            gen_udp:send(Socket,FromIP,FromPort,B)
        after5000->
            Case  pdu:make_pdu  (?COAP_ACKNOWLEDGEMENT,?COAP_GATEWAY_TIMEOUT,  binary_to_list(Token), _MessageID,atom_to_list(URI))of
        {ok,TimeoutResponse}->
            gen_udp:send(Socket,FromIP,FromPort,TimeoutResponse),
            io:format("Not received response in 5 seconds.~n");
        {error,Make_pdu}->
            io:format("Make pdu failed with reason: ~p~n",[Make_pdu])
    end
end.
```

Erlang can spawn a process very easily and without shared memory. Each process has its own variable, own socket, own memory. So our idea is that for each client's request, we open a process, and transfer the request to the real destination, and receive the response from the server and send it back. For each request, we have URI_HOST and URI_PORT option which tell the proxy, where is the real server and its port. After get this information, the server just change the network layer.

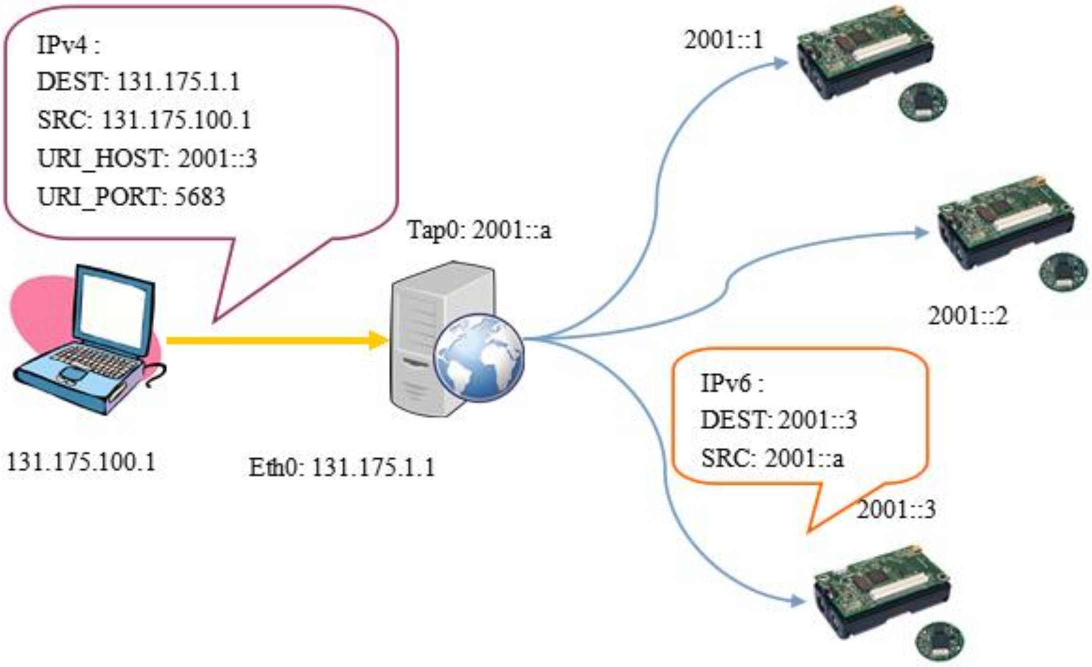


Figure 3.10: An example of the scenarios

and send it out, and wait for the server's response. If no response after a certain time, the Proxy send a gateway timeout message to the client, and end this process.

Between Erlang proxy and Contiki node:

Because of our proxy is using normal Linux based device, there is no hardware support IEEE 802.15.4, so what we need to do is let a Contiki node connect to the Linux based device(for example PC or Raspberry Pi) with USB port.

At first, we wanted to use the raw serial protocol to transmit message between them. But once we were going to use IPv6 to communicate between sensors, the raw serial protocol was never working. After some research, we decided that, since CoAP uses IPv6 on Contiki devices, we had to use SLIP protocol(Serial Line Internet Protocol) instead of raw serial protocol.

SLIP modifies a standard TCP/IP datagram by appending a special "END" byte to it, which distinguishes datagram boundaries in the byte stream, if the END byte occurs in the data to be sent, the two byte sequence ESC, ESC_END is sent instead, if the ESC byte occurs in the data, the two byte sequence ESC, ESC_ESC is sent. variants of the protocol may begin, as well as end, packets with END.

Hex value	Abbreviation	Description
0xC0	END	Frame End
0xDB	ESC	Frame Escape
0xDC	ESC_END	Transposed Frame End
0xDD	ESC_ESC	Transposed Frame Escape

RPL Border Router

In our case, there may be a mount of contiki servers. When the packets come from the serial port, we need a router to transmit them to the specified destination. Routing protocol plays an important role in the network, which is responsible for constructing the network topology, routing, data forwarding and some other functions. Traditional wired network routing protocols require to send large amounts of data packets to maintain the network topology, and also need a lot of storage space to save the routing entries, therefore not suitable for wireless sensor networks, in which transmission rate, capacity and processing is very limited. RPL is IPv6 routing protocol designed for low power and lossy networks, which is created by IETF ROLL working group as a proposed standard.

RPL forms a directed acyclic graph, that is, a tree-like topology. Each node except the root has its related parent acting like a gateway. The nodes in the network maintain a route table that stores all the routes down blow. If the destination does not exist in the routing table, the node will forward it to the related parent node until reaching the destination.

In RPL protocol, a Destination Oriented Directed Acyclic Graph (DODAG) is several nodes connected by directed edges, among which there is no cycles. Compared with the conventional tree topology, DODAG can provide additional paths. Nodes use DODAG information object (DIO) messages to create and maintain DODAG. The DODAG building process is explained in the following steps. Step 1: The network administrator configures (at application level) one or more nodes as a DODAG root. The DODAG roots starts sending the link local multicast DIO messages. A node may also solicit for DIO from the root in the mean time using DODAG Information Solicitations (DIS), in which case the DODAG root will send the DIO immediately. Step 2: The nodes nearby will receive the DIO from the root and will process it as it is from a lower rank node and will select root as their parent. Step 3. These nodes will now send link local multicast DIOs and the other nodes receiving the DIO may select them as parent. If a node receives DIOs from two or more parents, it will decide based on the objective function. This process will continue until all the nodes join the DODAG.

RPL uses "up" and "down" direction’s terminology regarding the movement of traffic. It can be divided into two main parts, either up the tree or down the tree. To send data up the tree, RPL needs the information in the DODAG. When sending a packet from a node to the root, it simply sends the packet to its preferred parent, and parents keep sending to their parents until the packet reaches the root. So each node only need to keep track of its parent. To send data down the tree, RPL uses DAO messages to maintain the routing table in support of downward traffic, all nodes need to keep track of all nodes below them. Parent routes are built with DIO messages. Child routes are built with DAO messages. Upward routing has great scaling properties since the number of upward routes is constant with network size. Downward routing does not scale as well because the number of routes each node needs to have room for increases linearly with network size.

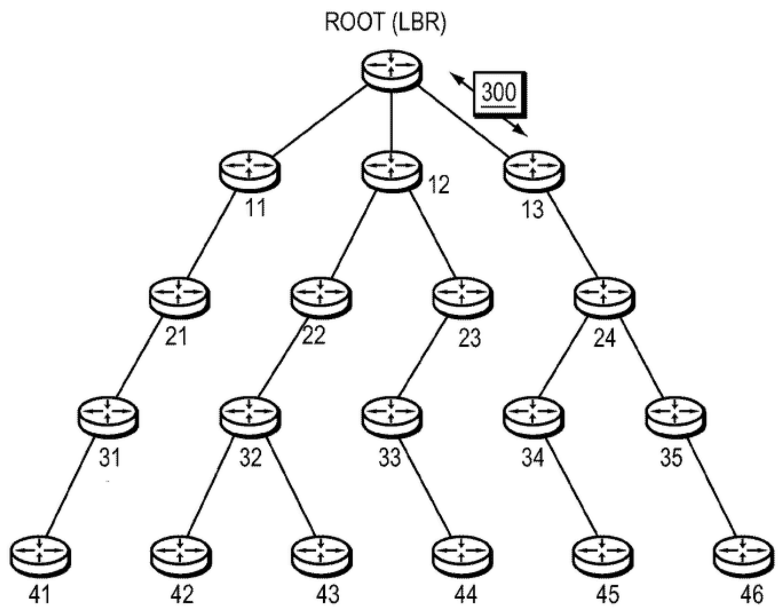


Figure 3.11:RPL routing protocol

Contiki RPL is an open-source implementation of IETF ROLL working group's RPL. A typical RPL network in contiki is shown above. An LLN border router (LBR) is located on the top of the topology. A border router is used in order to connect 6LoWPAN devices to the IP network and is responsible for handling traffic to and from the IPv6 and 802.15.4 interfaces. The border router creates a RPL DAG, the other nodes will be udp servers or clients. At first, it joins the network acting like a RPL router, and sends a UDP datagram to IPv6 address. Since we use the SLIP protocol, the border router must connect to computer over a SLIP tunnel. In this case, it can be formed by tunslip6, a tool that is provided in contiki. After assigning an IPv6 address, a connection will be made between the router and SLIP.

UDP server

In order to get the request from the Erlang client, the server must create a UDP connection and then bind it to a local port that it would listen on. Also the local port must be the same as the remote port specified in Erlang client. The main process thread for the UDP server application need not contain any large differences from regular Contiki applications. The code of this process is shown below:

```
udpconn = udp_new(remote_ip_address, port, appstate);
udp_bind(udpconn, port);
```

The udp_new function sets up a new connection. This function creates a new UDP connection with the specified remote endpoint, the format of the function is:

```
udp_new(remote_ip_address, port, appstate);
```

By specifying a remote_ip and port, the application only accepts connections from this certain IP address and port combination. However, our goal is to recieve the messages from any IP and any port, just one connection is not enough. So by setting the remote ip_address to NULL and the port to 0, this is easily achieved. Here we don't want to change the application state, so it is set to NULL. In order for the udpconn variable, which holds the UDP connection information, to be available it must first be declared. The declaration of this variable happens at the very beginning of the source code, immediately after any include and define statements, thereby making it a global variable:

```
static struct uip_udp_conn *udpconn;
```

These simple lines of code allow a Contiki application to set up a UDP server on a particular port. In our application, the UDP server is accepting connections from any IP address and port pair; the application is listening on port 5678.

Also each server must be different from another by assigning a unique IPv6 address, we must provide which server to communicate in Erlang client. The function is:

```
uip_ip6addr(&ipaddr, addr0, addr1, addr2, addr3, addr4, addr5, addr6, addr7);
uip_ds6_addr_add (uip_ipaddr_t *ipaddr, unsigned long vlifetime, uint8_t type);
```

The first function is to construct an IPv6 address from eight 16-bit words. The second function is to assign an address. The type is chosen among 0,1,2,3, which means any type, auto-config, DHCP and manually. In our case, set lifetime to 0 (permanently), adn set type to manually.

When a message packet arrives, just having a UDP server listen on a port is not useful unless we do something with it. In order to handle these message receive events we must create our own function and pass to it any events that occur. Within the main application process, we could create an infinite while loop, which passes the event and its associated data to a function.

```
while(1) {
    PROCESS_YIELD();
    if(ev == tcpip_event) {
        tcpip_handler();
        send_packet();
    }
```

The main process is yielded anytime an event occurs and it first checks if the event which caused the function to be called was of type tcpip_event. Then we have functions proceed to handle the data, and send it back.

Upon receiving an IPv6 packet from the client, the first thing to do is to get rid of the layer 2 information. To achieve this, we use the uip_appdata, a pointer to the application data in the packet buffer. This pointer points to the application data when the application is called. If the application wishes to send data, the application may use this space to write the data into before calling uip_send(). In tcpip_handler(), now we can unpack the CoAP packet.

In Contiki, there is a C CoAP implementation called Erbium. It is a low-power REST Engine for Contiki that was developed together with

SICS. The REST Engine includes a comprehensive embedded CoAP implementation, which became the official one for the Contiki OS. In our case, we use the newest version er-coap-13, it is located in apps folder. And it must be included in makefile in order to use it.

```
WITH_COAP = 1
CFLAGS += -DWITH_COAP
APPS += er-coap-13
APPS += erbium
```

The struct of the CoAP is defined as follow:

```
typedef struct {
  uint8_t *buffer;
  uint8_t version;
  coap_message_type_t type;
  uint8_t code;
  uint16_t mid;
  uint8_t options[COAP_OPTION_PROXY_URI / OPTION_MAP_SIZE + 1];
  coap_content_type_t content_type;
  uint32_t max_age;
  size_t proxy_uri_len;
  const char *proxy_uri;
  uint8_t etag_len;
  uint8_t etag[COAP_ETAG_LEN];
  size_t uri_host_len;
  const char *uri_host;
  size_t location_path_len;
  const char *location_path;
  uint16_t uri_port;
  size_t location_query_len;
  const char *location_query;
  size_t uri_path_len;
  const char *uri_path;
```

```
    uint16_t observe;
    uint8_t token_len;
    uint8_t token[COAP_TOKEN_LEN];
    uint8_t accept_num;
    uint16_t accept[COAP_MAX_ACCEPT_NUM];
    uint8_t if_match_len;
    uint8_t if_match[COAP_ETAG_LEN];
    uint32_t block2_num;
    uint8_t block2_more;
    uint16_t block2_size;
    uint32_t block2_offset;
    uint32_t block1_num;
    uint8_t block1_more;
    uint16_t block1_size;
    uint32_t block1_offset;
    uint32_t size;
    size_t uri_query_len;
    const char *uri_query;
    uint8_t if_none_match;
    uint16_t payload_len;
    uint8_t *payload;
} coap_packet_t;
```

The version, type, token, code and mid are the same as that in the CoAP packet, it's very convenient. There is a function allow us to transfer the informations into this struct:

```
coap_parse_message(void *packet, uint8_t *data, uint16_t data_len);
```

By passing the associated data of the event along with the data length, we achieve a fully organized CoAP data, and process according to the code, URI path and query respectively. Because of the limitation of the CoAP protocol, the maximum length of the URI path is about 13 bytes, we define a 4 bytes URI path also for convenient. The Tmote Sky is able to sense the temperature, light, humidity, battery and some other functions. The value obtained from sensor is not the regular one, it need to be converted by applying the given formula. Here is the code of getting the temperature:

```
int get_temperature(void)
{
    return ((sht11_sensor.value(SHT11_SENSOR_TEMP) / 10) - 396) / 10;
}
```

It is necessary to create a CoAP packet for responding, the value of the sensor will be stored in the payload part. The declaration of this response happens at the beginning of the code, making it a global variable. Also the response code should be changed accordingly, now it is ready to send. The response code definition is shown below:

```
typedef enum {
  NO_ERROR = 0,
  CREATED_2_01 = 65,                  /* CREATED */
  DELETED_2_02 = 66,                  /* DELETED */
  VALID_2_03 = 67,                    /* NOT_MODIFIED */
  CHANGED_2_04 = 68,                  /* CHANGED */
  CONTENT_2_05 = 69,                  /* OK */
  BAD_REQUEST_4_00 = 128,             /* BAD_REQUEST */
  UNAUTHORIZED_4_01 = 129,            /* UNAUTHORIZED */
  BAD_OPTION_4_02 = 130,              /* BAD_OPTION */
  FORBIDDEN_4_03 = 131,               /* FORBIDDEN */
  NOT_FOUND_4_04 = 132,               /* NOT_FOUND */
  METHOD_NOT_ALLOWED_4_05 = 133,      /* METHOD_NOT_ALLOWED */
  NOT_ACCEPTABLE_4_06 = 134,          /* NOT_ACCEPTABLE */
```

```
    PRECONDITION_FAILED_4_12 = 140,       /* BAD_REQUEST */
    REQUEST_ENTITY_TOO_LARGE_4_13 = 141,  /* REQUEST_ENTITY_TOO_LARGE */
    UNSUPPORTED_MEDIA_TYPE_4_15 = 143,    /* UNSUPPORTED_MEDIA_TYPE */
    INTERNAL_SERVER_ERROR_5_00 = 160,     /* INTERNAL_SERVER_ERROR */
    NOT_IMPLEMENTED_5_01 = 161,          /* NOT_IMPLEMENTED */
    BAD_GATEWAY_5_02 = 162,              /* BAD_GATEWAY */
    SERVICE_UNAVAILABLE_5_03 = 163,       /* SERVICE_UNAVAILABLE */
    GATEWAY_TIMEOUT_5_04 = 164,          /* GATEWAY_TIMEOUT */
    PROXYING_NOT_SUPPORTED_5_05 = 165,    /* PROXYING_NOT_SUPPORTED */
    /* Erbium errors */
    MEMORY_ALLOCATION_ERROR = 192,
    PACKET_SERIALIZATION_ERROR,
    /* Erbium hooks */
    MANUAL_RESPONSE,
    PING_RESPONSE
} coap_status_t;
```

Sending messages over UDP is also simple while using 6lowpan with Contiki. The send_packet() function takes a CoAP packet as a parameter, which is the response. But in order to send the packet, we need to serialize it first and put it into a message buffer. This function is also implemented in the library:

```
coap_serialize_message(void *packet, uint8_t *buffer);
```

A uip_udp_packet_sendto() function is used in order to send the message buffer to a certain address and port combination:

```
uip_udp_packet_sendto(struct uip_udp_conn *c, const void *data, int len,
                  const uip_ipaddr_t *toaddr, uint16_t toport);
```

The c variable is what was used in order to create and hold information about the UDP connection. The toaddr and toport are the source address and port of the packet we receive from last time. They are in the UDP header of the buffer. These are the variables are all required in order to dispatch the message buffer back to the sender.

4 Evaluation

This chapter will describe the evaluation of the whole process, implemented mainly in a home scenario.

4.1 Test environment

- Client: PC
  - CPU: Intel(R) Core(TM) i3-2330M 2.2GHz
  - RAM: 4 GB
  - System: Ubuntu Linux 13.10 X86_64
- Proxy: Raspberry Pi
  - SOC: BroadcomBCM2835
  - CPU: ARM1176JZF-S 700MHz
  - RAM: 512 MB
  - System: Debian GNU/linux
- Server: Tmote sky
  - CPU: MSP430 8MHz
  - RAM: 10KB
  - System: Contiki OS

4.2 Round trip time

Round trip time is the length of time it takes for a signal to be sent plus the length of time it takes for an acknowledgment of that signal to be received. Here the RTT is consisted of,

- Time between client and proxy(time C-P),
- Proxy process time,
- Time between proxy and server(Time P-S),
- Server process time.

Now we have test 20 times, how much the round trip time will be when measuring temperature,

	time C-S	time P-S	time C-P
	0.532635	0.513119	0.019516
	0.593028	0.470485	0.122543
	0.721123	0.664892	0.056231
	0.597888	0.481923	0.115965
	0.504986	0.431622	0.073364
	0.589403	0.47157	0.117833
	0.708621	0.653421	0.0552
	0.417519	0.380856	0.036663
	0.376693	0.35531	0.021383
	0.468068	0.340414	0.127654
	0.629148	0.471335	0.157813
	0.563249	0.438224	0.125025
	0.553016	0.517456	0.03556
	0.4666917	0.365328	0.1013637
	0.488352	0.380098	0.108254

	0.398147	0.368432	0.029715
	0.590897	0.39926	0.191637
	0.356958	0.338605	0.018353
	0.482122	0.407069	0.075053
average	0.528344458	0.444706263	0.083638195
max	0.721123	0.664892	0.191637
min	0.356958	0.338605	0.018353

Table 4.1: : round trip time of measuring

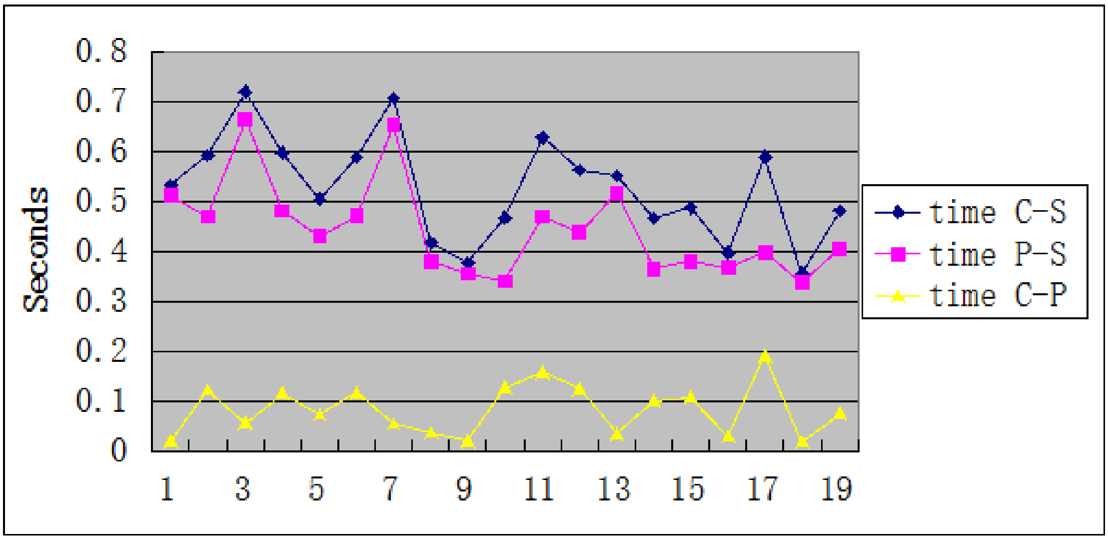


Figure 4.2: round trip time of measuring

As we can see from the table and figure, on average, the round trip time from client to server is 0.528 second, the time between proxy to server takes the most part, about 0.445 second, nearly 84.28%, the time between client and proxy only takes less than 20%.

Next we use light sensor to measure the round trip time,

	time C-S	time P-S	time C-P
	0.176034	0.143643	0.032391
	0.156586	0.126988	0.029598
	0.191075	0.172685	0.01839
	0.328711	0.298529	0.030182
	0.202629	0.171201	0.031428
	0.469357	0.438593	0.030764
	0.327093	0.295251	0.031842
	0.224221	0.192289	0.031932
	0.270377	0.237854	0.032523
	0.323381	0.292648	0.030733
	0.25253	0.216034	0.036496
	0.266794	0.232764	0.03403
	0.276943	0.246169	0.030774
	0.204786	0.174497	0.030289
	0.443354	0.391339	0.052015
	0.262673	0.229278	0.033395
	0.160034	0.130313	0.029721
	0.156803	0.125116	0.031687
	0.365741	0.280189	0.085552
	0.253461	0.23597	0.017491
average	0.26562915	0.2315675	0.03406165
max	0.469357	0.438593	0.085552
min	0.156586	0.125116	0.017491

Table4.3: round trip time of measuring light

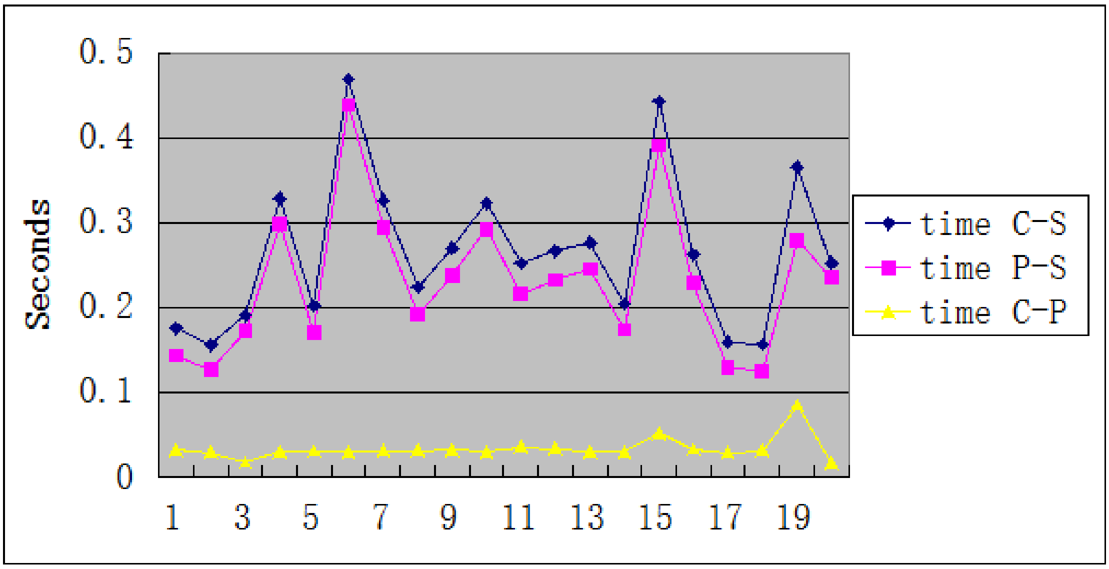


Figure 4.4: round trip time of measuring light

From the data above, we can see during measuring light, the average round trip time from client to server is about 0.266 seconds, and the time between proxy and server is almost 0.232 seconds. The round trip time between proxy and server takes more than 87.22%, time between client and proxy takes less than 15%.

We can see that use different sensors on the same server, the round trip time have bit gap, use light sensor, the average round trip time is only about 50% compare to measuring temperature. So let us see how much time will be used while measuring temperature. We design an test, measuring the time to get temperature:

```
3.   start = clock_time();
4.   value = get_temperature();
5.   end = clock_time();
```

```
// we have #define CLOCK_CONF_SECOND 128UL in platform-conf.h, so the time measured in seconds would be clock_ticks/128
duration = (end - start)/128;
printf("%u\n", duration);
```

We have tested for about 20 times, the result shows that to get the temperature we need about 0.231 seconds, so this shows that the result above is correct, the gap between measuring temperature and measuring light is because of the temperature sensor need time to wait the value.

4.3 System performance

Previously, we introduced the round trip time from the client to the server. This part we are going to test how much system resource the Erlang CoAP application will take from the device. Memory consumption, power consumption as well as spawn time of each Erlang process will be tested.

4.3.1 Memory consumption

We measure memory usage with pmap: a Linux utility that reports the entire memory allocated for a given application, including code, libraries, stack, and heap. This gives a precise indication of the amount of memory a device needs to run the application: devices with less memory would just be unable to run the same application implementation.

For better testing the memory consumption of the Erlang application, we separately access to the Erlang shell without load any applications and then load the application, it turns out that the Erlang shell is consume the most part of the memory, but the application itself take less(only takes a few hundreds KB on Raspberry Pi and few MB on PC).

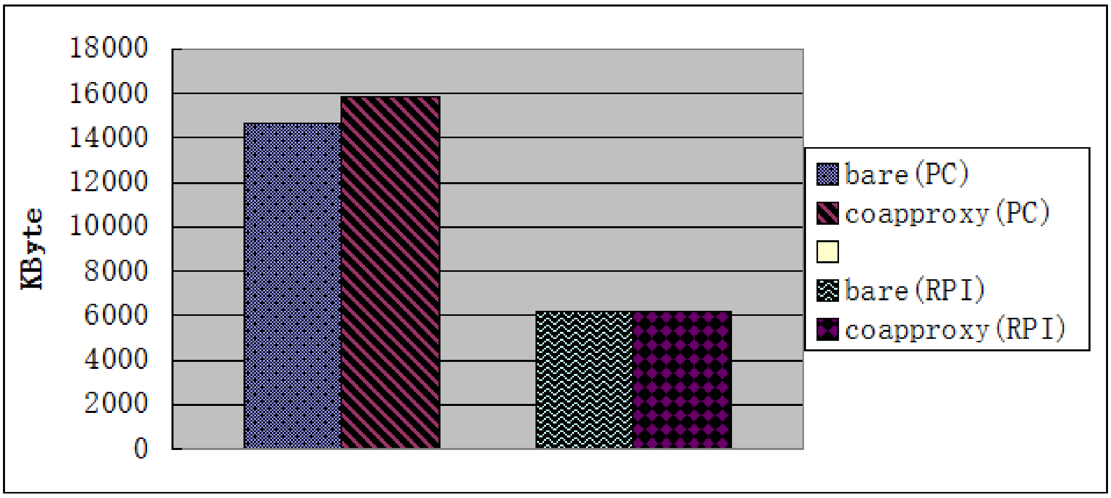


Figure 4.5: Memory consumption comparison

This point out that if we want to improve the memory consumption of the Erlang application, we need to reduce the memory consumption of the Erlang shell.

4.3.2 Power consumption

Such low power consumption sensor nodes should use low-power RF and at the same time, we need to support some Internet stack like IPv4 or

IPv6, so they use An Adaptation Layer to fit IPv6 over Low-Power wireless Area Networks, it is called 6LoWPAN. With 6LoWPAN technology, the packets can be efficiently compressed. For example, in the real network, a common MTU size for IPv6 packet is 1280 bytes, and after compression, in the 802.15.4 network, the MTU is 127 bytes.

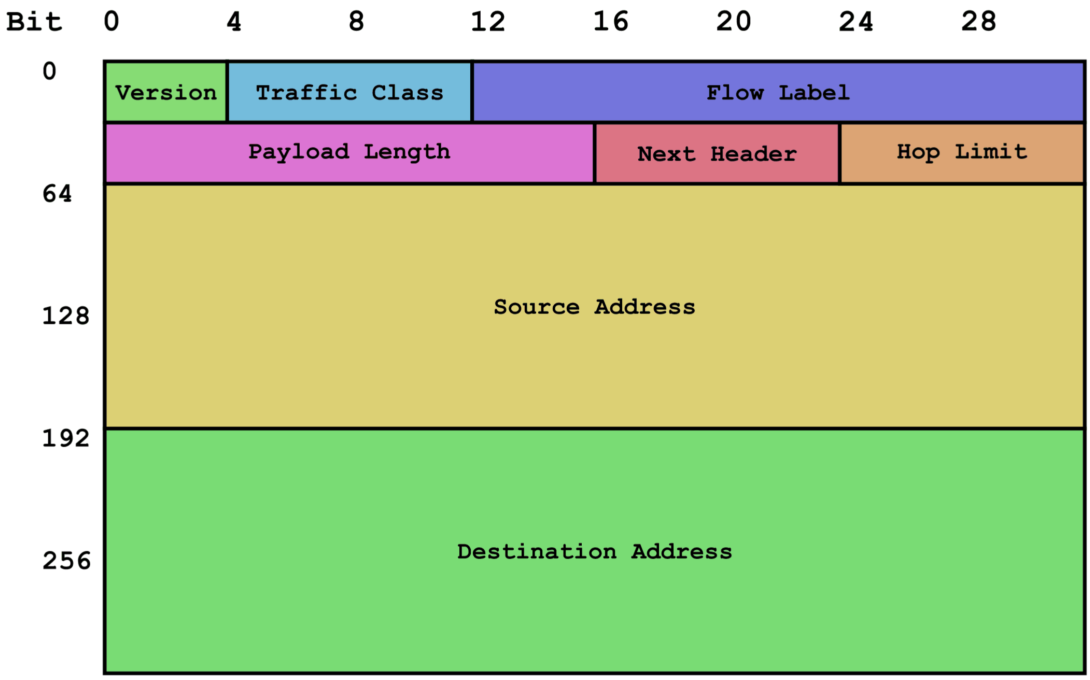


Figure 4.6: IP packet

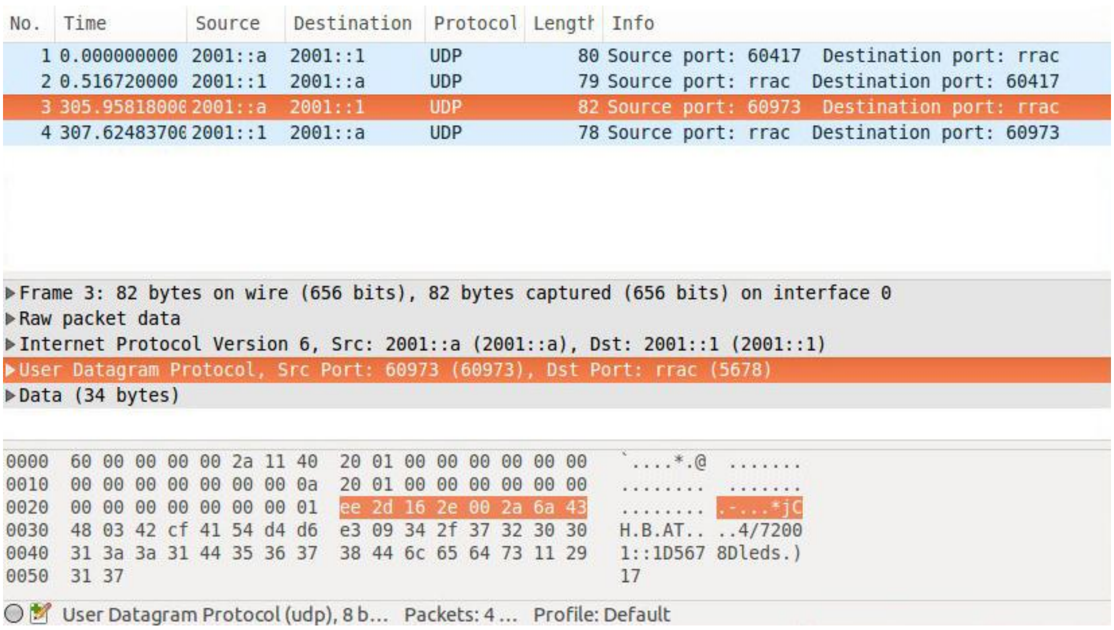
With normal IPv6 UDP communication, we have at least 40 bytes IPv6 header, and 8 bytes UDP header In 6LoWPAN, we have two kind of compress way, stateless compression and flow-independent compression.

For the local link communication, hop limits to 1, flow-independent compression can compress IPv6 header to 2 bytes, UDP header to 4 bytes, then the total header is 6 bytes, but the stateless header compression approach just could compress the header to seven bytes under the best circumstances, and the two bytes checksum could be compressed if there is Complete Verification mechanism in application layer.

If use global address, that means a data packet is transmitted from one LoWPAN to another one, or to other network, Under the best situation, the flow-independent compression approach can compress the header 7 bytes. But the stateless header compression approach, for the address parts, just 64 bits source address could be compressed, there are still 24 address bytes uncompressed.

From the above comparison, we can see flow-independent header compression is significant.

For example, by using the network sniffer wireshark we capture the packets sent from tap0, which is the port of SLIP tunnel. It is a standard IPv6 UDP packet.



From the graph, we can see the total packet size is 82 bytes, the headers of IPv6 and UDP are 40 bytes and 8 bytes respectively. There are a lot of zeros in the header and the most useful data part is only 34 bytes, that means we waste a lot of energy on the header. As we use local link communication, after the flow-independent approach, we achieve a compression of 41 bytes out of 48 bytes, and the total packet size is 41 bytes. As we can see, we almost compress at least 41 bytes of the packet header, which means more than 50% energy is saved.

Additionally, CoAP protocol was also design for constrained nodes and constrained (e.g., low-power, lossy) networks, so this also help to save energy.

5 Related work

This chapter will illustrate existing works that related to CoAP protocol. Include the Internet of Things architectures, a IoT framework for Raspberry Pi which called WebIOPi and finally, we introduce some security mechanisms.

5.1 IoT architectures

It is very important to have an architecture for any kind of applications. For Internet of Things, of course we will need an architecture in the future.

For example, IoT6 project[9] is a 3 years FP7 European research project on the future Internet of Things. It aims at exploiting the potential of IPv6 and related standards (6LoWPAN, CORE, COAP, etc.) to overcome current shortcomings and fragmentation of the Internet of Things. Its main challenges and objectives are to research, design and develop a highly scalable IPv6-based Service-Oriented Architecture to achieve interoperability, mobility, cloud computing integration and intelligence distribution among heterogeneous smart things components, applications and services.

IoT-A project[10] has addressed for three years the Internet-of-Things

Architecture, and created the proposed architectural reference model together with the definition of an initial set of key building blocks. Together they are envisioned as foundations for fostering the emerging Internet of Things. Using an experimental paradigm, IoT-A combined top-down reasoning about architectural principles and design guidelines with simulation and prototyping in exploring the technical consequences of architectural design choices.

5.2 WebIOPi[11] on Raspberry Pi

The Raspberry Pi is a low cost, credit-card sized computer, and with a lot of I/O port which can be used to communicate with the outside world, in our thesis, we used it to act as a proxy to connect IPv4 world and IPv6 world. But this is not all of what it can do. It has GPIO ports, UART, I2C and SPI bus, we can fully use them to connect to some sensors and let Raspberry Pi act as a CoAP server.

WebIOPi offers us a Internet of Things framework which developed by Python. It can control, debug, and use Pi's GPIO, sensors and converters from a web browser or any app. With this Swiss-knife like library, it includes REST APIs, a lot of ADC/DAC and sensors driver. It not only support CoAP protocol but also support HTTP protocol.

Because of Raspberry Pi is a Linux based computer, it will be very easy to develop applications no matter using C, Python, Java, Erlang and so on, especially for programmers who is familiar with Linux system. In particular, Linux has integrated a lot of low level drivers, programmers do not need to waste of time to develop these things, they just need to focus on how to implement the high-level applications.

5.3 Securing CoAP

Like what we have in HTTP protocol, the security is needed by the Internet of Things. But we still do not know how to use the existing security protocols to makes our low-power, lossy network protocol safe. In the Internet, the de facto standard to achieve security is Transport Layer Security (TLS). But this kind of security protocol is not suit for our Internet of Things network, because our constrained network environment, because it requires a lot of hardware performance. The CoAP specification suggests to use of Datagram Transport Layer Security(DTLS) protocol[7].

DTLS is a protocol for securing network traffic which has been specified by the IETF in RFC 6347[12]. This is not like the TLS module that depend on reliable message transfer(like TCP), but can be used with unreliable datagram transfer, e.g., UDP.

Except secure the data on the transport layer, we can also secure them on the link layer, it means that in the 6LoWPAN network environment, we need to secure the 802.15.4 link layer. 802.15.4 link-layer security is the current state-of-the-art security solution for the IP-connected IoT[13]. It offers data encryption and integrity verification, achieved by a single pre-shared key used for symmetric cryptography applied to all outgoing packets while message integrity is realized by including a Message Authentication Code (MAC) in the packets.

6 Conclusions

This thesis has introduced CoAP protocol, which can be used in the low-power, lossy network(e.g., wireless sensor network), and implement it with ELIoT framework. The protocol is designed for machine-to-machine (M2M) applications such as smart energy and building automation, and Erlang is a high level programming language which scalable, safe, and efficient. Erlang separate processes with no shared memory but communicating via message passing，naturally transfers to multicore processors in a way that is largely transparent to the programmer, so that you can run your Erlang programs on more powerful hardware without having to redesign them.

6.1 What we have done

The work started from using cantcoap C++ library to build a CoAP APIs for Erlang language. These APIs provide us a simple but powerful way to implement CoAP protocol with Erlang language. Using these Erlang APIs we can encapsulate and resolve CoAP packet very easy. Also because of the virtual machine mechanism, just like Java, we could transplant Erlang programs to different platforms without change code. This is much convenient for programmer to develop and debug programs on different platforms.

Then, we were going to use the APIs which we had mentioned above to build a CoAP client, CoAP proxy and CoAP server with Erlang language. CoAP client and server can be used directly in the environment of tradition network (e.g., IPv4), and if in the future, the Internet evolve into IPv6, it will be very easy to modify the code to adapt the demand. However, nowadays, when we use the traditional IPv4 network mostly on our PCs or smart phones and the wireless sensor network has already used IPv6 network, we need a proxy to convent the network protocol between IPv4 and IPv6. As a result, we then develop a CoAP proxy which also using Erlang so that help devices from different network environment to communicate with each other.

After this, we develop a CoAP server based on Contiki Operating system with tmote sky platform. This platform has temperature sensor, light sensor and humidity sensor, so it is very suitable for environment monitoring. When we get a CoAP request, according to the request information, we can send back a corresponding response.

All of these, we now have a entire CoAP environment, include client, server, proxy. So we also would test them to see if they work fine and the system performance too.

6.2 Future work

As we all know that Erlang is design for distributed programming, but we do not fully take the advantage of this. During our research, from client to proxy and finally to the server, we always use UDP to transmit packet. Of course, using UDP to transmit messages will be more friendly compatible to the other applications which are not programmed by Erlang. However, if all of the CoAP nodes are programmed by Erlang, we suggest that maybe we can use Erlang message operator "!" to transmitted message directly, just keep Erlang epmd(Erlang Port Mapper Daemon) running and TCP, UDP 4396 port open and let Erlang VM to handle the message passing messy affairs.

What's more. If we want to implement all of the Internet of Things nodes running Erlang program, and finally reach ELIoT. The most important obstacle is that Erlang is now only support Linux based and Windows based devices, but not support low-power devices operating system such like Contiki, TinyOS, even for the most popular mobile device systems, like Android, IOS are not supported formally. So in the future, we need to adapt Erlang to these devices.

6.3 Final remarks

Nowadays, IoT(Internet Of Things) technology is still in its primary stage, many technologies are still evolving, in order to achieve large-scale industrial applications deployment， it still have a long way to go. But just because of this, ELIoT technology has a broad development space and considerable prospects. This thesis has achieved initial CoAP applications with ELIoT technology, although there are many imperfect drawbacks, It provides a basic reference to the IoT applications and as the direction of future research, other researchers can enrich and improve the functionality and performance on the other the platforms, allowing more platforms to play a bigger role of ELIoT.

Appendix

#COAP-CLIENT

```
-module(coap_client).
-export([get/2,get/3,get/4,put/3,delete/3]).
-include("coap.hel").

get(Host, URI, Para) ->
    Token = make_token(),
    ID = make_message_id(),
    {ok, PDU} = pdu:make_pdu(0, ?COAP_GET, Token, ID, URI),
    {ok, Newpdu} = getpara(PDU, Para),
    {ok, Destaddr} = inet_parse:address(Host),
    case gen_udp:open(?TMP_PORT, [binary, inet, {active, false}]) of
        {ok, Socket} ->
            gen_udp:send(Socket, Destaddr, ?PORT, Newpdu),
            Res =  case gen_udp:recv(Socket, 0, 6000) of
                {ok, {Destaddr, ?PORT, Packet}} ->
                    {ok, Ver, Type, Tkl, Code, MID} = pdu:get_header(Packet),
                    io:format("~p~n", [{ok, Ver, Type, Tkl, Code, MID}]),
                    {ok, Content} = pdu:get_content(Packet),
                    io:format("value is: ~p~n", [Content]);
                {error, Reason} ->
                    {error, Reason}
            end,
            gen_udp:close(Socket),
            Res;
        {error, Reason} ->
            {error, Reason}
    end.

put(Host, URI, Para) ->
```

```
    Token = make_token(),
    ID = make_message_id(),
    {ok, PDU} = pdu:make_pdu(0, ?COAP_PUT, Token, ID, URI),
    {ok, Newpdu} = getpara(PDU, Para),
    {ok, Destaddr} = inet_parse:address(Host),
    case gen_udp:open(?TMP_PORT, [binary, inet, {active, false}]) of
        {ok, Socket} ->
            gen_udp:send(Socket, Destaddr, ?PORT, Newpdu),
            Res =  case gen_udp:recv(Socket, 0, 6000) of
                {ok, {Destaddr, ?PORT, Packet}} ->
                    {ok, Ver, Type, Tkl, Code, MID} = pdu:get_header(Packet),
                    io:format("~p~n", [{ok, Ver, Type, Tkl, Code, MID}]),
                    {ok, Content} = pdu:get_content(Packet),
                    io:format("value is: ~p~n", [Content]);
                {error, Reason} ->
                    {error, Reason}
            end,
            gen_udp:close(Socket),
            Res;
        {error, Reason} ->
            {error, Reason}
    end.

delete(Host, URI, Para) ->
    Token = make_token(),
    ID = make_message_id(),
    {ok, PDU} = pdu:make_pdu(0, ?COAP_DELETE, Token, ID, URI),
    {ok, Newpdu} = getpara(PDU, Para),
    {ok, Destaddr} = inet_parse:address(Host),
    case gen_udp:open(?TMP_PORT, [binary, inet, {active, false}]) of
        {ok, Socket} ->
            gen_udp:send(Socket, Destaddr, ?PORT, Newpdu),
            Res =  case gen_udp:recv(Socket, 0, 6000) of
                {ok, {Destaddr, ?PORT, Packet}} ->
```

```
                    {ok, Ver, Type, Tkl, Code, MID} = pdu:get_header(Packet),
                    io:format("~p~n", [{ok, Ver, Type, Tkl, Code, MID}]),
                    {ok, Content} = pdu:get_content(Packet),
                    io:format("value is: ~p~n", [Content]);
                {error, Reason} ->
                    {error, Reason}
            end,
            gen_udp:close(Socket),
            Res;
        {error, Reason} ->
            {error, Reason}
    end.

get(Host, URI) ->
    Token = make_token(),
    ID = make_message_id(),
    {ok, PDU} = pdu:make_pdu(0, ?COAP_GET, Token, ID, URI), %type,method,token,id,URI; method:COAP_CODE_GET
    {ok, Address} = inet_parse:address(Host),
    case gen_udp:open(?TMP_PORT, [binary, inet, {active, false}]) of
        {ok, Socket} ->
            gen_udp:send(Socket, Address, ?PORT, PDU),
            Res = case gen_udp:recv(Socket, 0, 3000) of                  %time-out 3000 ms
                {ok, {Address, ?PORT, Packet}} ->
                    pdu:get_content(Packet);
                {error, Reason} ->
                    {error, Reason}
            end,
            gen_udp:close(Socket),
            Res;
        {error, Reason} ->
            {error, Reason}
    end.
```

```

get(Host, URI, URI_HOST, URI_PORT) ->
    Token = make_token(),
    ID = make_message_id(),
    {ok, PDU} = pdu:make_pdu(0, ?COAP_GET, Token, ID, URI),
    {ok, PDU1} = pdu:add_option(PDU, ?COAP_OPTION_URI_HOST, length(URI_HOST), URI_HOST),
    {ok, PDU2} = pdu:add_option(PDU1, ?COAP_OPTION_URI_PORT, length(URI_PORT), URI_PORT),
    {ok, Address} = inet_parse:address(Host),
    case gen_udp:open(?TMP_PORT, [binary, inet, {active, false}]) of
        {ok, Socket} ->
            gen_udp:send(Socket, Address, ?PORT, PDU2),
            Res = case gen_udp:recv(Socket, 0, 7000) of
            {ok, {Address, ?PORT, Packet}} ->
                pdu:get_header(Packet);
            {error, Reason} ->
                {error, Reason}
            end,
            gen_udp:close(Socket),
            Res;
        {error, Reason} ->
            {error, Reason}
    end.


make_token() ->
    make_token(?TOKEN_LENGTH, []).

make_token(Remaining, Acc) when Remaining == 0 ->
    Acc;
make_token(Remaining, Acc) ->
    make_token(Remaining - 1, [random:uniform(256 - 1)|Acc]).

```

```
make_message_id() ->
    random:uniform(?MAX_ID) - 1.

recv_subscribe() ->
    receive
        {From, HostAddress, NewPDU} ->
            io:format("i am in fun recv_subscribe~n"),
            case gen_udp:open(0, [binary, inet, {active, true}]) of        %use random UDP port number
            {ok,Socket} ->
                gen_udp:send(Socket, HostAddress, ?PORT, NewPDU),
                io:format("i have sent the packet~n"),
                recv_subscribe_loop(Socket, HostAddress, From);
            {error, Reason} ->
                io:format("{error, ~p}~n", [Reason])
            end
    end.

recv_subscribe_loop(Socket, Host, From) ->
    io:format("i am waitting for packet~n"),
    receive
        {udp, Socket, Host, ?PORT, Bin} ->
            {ok, Content} = pdu:get_content(Bin),
            {ok, Ver, Type, TKL, Code, MID} = pdu:get_header(Bin),
            io:format("{ok, ~p}~n", [Content]),
            io:format("Recever packet with version: ~p~nType: ~p~nTKL:~p~nCode:~p~nMessageID:~p~n", [Ver,Type,TKL,Code,MID]),
            From ! {self(), {ok, Content}},
            recv_subscribe_loop(Socket, Host, From);
        {error, Reason} ->
            From ! {self(), {error, Reason}},
            io:format("{recv_subscribe_loop get error, ~p}~n", [Reason]),
            recv_subscribe_loop(Socket, Host, From);
        {stop, subscribe} ->
```

```
                stop,
                gen_udp:close(Socket),
                io:format("sub loop is stoped.~n")
        after 5000 ->
            From ! {self(), {error, no_message}},
            recv_subscribe_loop(Socket, Host, From)
        end.

loop() ->
    receive
        {Pid, {ok, Content}} ->
            io:format("{ok, ~p}~n", [Content]),
            loop();
        {Pid, {error, Content}} ->
            io:format("{error, ~p}~n", [Content]),
            loop();
        Other ->
            io:format("{error, recvother}~n"),
            loop()
    end.

getpara(PDU, []) ->
    {ok, PDU};
getpara(PDU, Allpara) ->
    [H|L] = Allpara,
    {ok, Newpdu} = case H of
        {urihost, Val} ->
            pdu:add_option(PDU, ?COAP_OPTION_URI_HOST, length(Val), Val);
        {uriport, Val} ->
            pdu:add_option(PDU, ?COAP_OPTION_URI_PORT, length(Val), Val);
        {uriquery, Val} ->
            pdu:add_option(PDU, ?COAP_OPTION_URI_QUERY, length(Val), Val);
        {value, Val} ->
            pdu:add_payload(PDU, Val, length(Val))
```

```
    end,
    getpara(Newpdu, L).
```

PDU

```
-module(pdu).
-export([init/0, make_pdu/5, get_content/1, add_option/4, add_payload/3, get_header/1]).
-on_load(init/0).

-define(APPNAME, erl_coap).

init() ->
    case code:priv_dir(?APPNAME) of
        {error, _} ->
            error_logger:format("~w priv dir not found~n", [?APPNAME]),
            exit(error);
        PrivDir ->
            erlang:load_nif(filename:join([PrivDir, "pdu_drv"]), 0)
    end.

make_pdu(_Type, _Method, _Token, _ID, _URI) ->
    erlang:nif_error(nif_not_loaded).

get_content(_Buffer) ->
    erlang:nif_error(nif_not_loaded).

add_option(_PDU, _Optnum, _Optlen, _Opnval) ->
     erlang:nif_error(nif_not_loaded).

add_payload(_PDU, _Payloadvalue, _Payloadlen) ->
     erlang:nif_error(nif_not_loaded).

```

```
get_header(_PDU) ->
```

#COAP-PROXY

```
    erlang:nif_error(nif_not_loaded).
-module(coapproxy).
-export([main/0, main/1]).
-include("coap.hel").

%% this is a proxy demon
main() ->
    Pid0 = spawn(fun openport/0),
    register(proxy, Pid0),
    Pid1 = spawn(fun openport6/0),
    register(proxy6, Pid1).

%% this is a proxy demon can be assign a special udp prot
main(P) ->
    Pid2 = spawn(fun() -> openport(P) end),
    register(proxy, Pid2),
    Pid3 = spawn(fun() -> openport6(P) end),
    register(proxy6, Pid3).

openport() ->
    case gen_udp:open(?PORT, [binary, inet, {active, true}]) of
        {ok, Socket} ->
            io:format("IPv4 Port:~p opened, wait for messages.~n",[?PORT]),
            % 4 means IPv4
            proxy_recv(Socket, 4);
        {error, Reason} ->
            io:format("{error, ~p}~n", [Reason])
    end.
openport(Port) ->
    case gen_udp:open(Port, [binary, inet, {active, true}]) of
```

```
        {ok, Socket} ->
            io:format("IPv4 Port:~p opened, wait for messages.~n",[Port]),
            proxy_recv(Socket, 4);
        {error, Reason} ->
            io:format("{error, ~p}~n", [Reason])
    end.

openport6() ->
    case gen_udp:open(?PORT + 1, [binary, inet6, {active, true}]) of
        {ok, Socket} ->
            io:format("IPv6 Port:~p opened, wait for messages.~n",[?PORT + 1]),
            % 6 means IPv6
            proxy_recv(Socket, 6);
        {error, Reason} ->
            io:format("{error, ~p}~n", [Reason])
    end.
openport6(Port) ->
    case gen_udp:open(Port + 1, [binary, inet6, {active, true}]) of
        {ok, Socket} ->
            io:format("IPv6 Port:~p opened, wait for messages.~n",[Port+1]),
            proxy_recv(Socket, 6);
        {error, Reason} ->
            io:format("{error, ~p}~n", [Reason])
    end.
proxy_recv(S, FromIPvN) ->
    receive
        {udp, S, FromIP, FromPort, Bin} ->
            spawn(fun() -> wait_response(FromIP, FromPort, S, Bin, FromIPvN) end),
            proxy_recv(S, FromIPvN);
        {error, Reason} ->
            io:format("{get error message, ~p}~n", [Reason]),
            proxy_recv(S, FromIPvN);
        stop ->
        gen_udp:close(S),
```

```
        io:format("UDP port closed~n")
    end.

wait_response(FromIP, FromPort, Socket, Bin, FromIPvN) ->
    {_, A1, A2} = now(),
    case pdu:get_URI(Bin) of
        {ok, URI} ->
            io:format("Get request message from:~p:~p with URI: ~p~n",[FromIP, FromPort, URI]);
        {error, URI} ->
            io:format("get_URI error:~p~n",[{error, URI}]),
             exit(no_URI)
    end,
    case pdu:get_option(Bin, ?COAP_OPTION_URI_HOST) of
        {ok, URI_HOST} ->
            io:format("Get URI_HOST: ~p~n",[URI_HOST]);
        {error, URI_HOST} ->
            io:format("get_option error ~p~n", [{error, URI_HOST}])
    end,
    {ok, RemoteIP} = inet_parse:address(URI_HOST),
    case pdu:get_option(Bin, ?COAP_OPTION_URI_PORT) of
        {ok, URI_PORT} ->
            io:format("Get URI_PORT: ~p~n",[URI_PORT]);
        {error, URI_PORT} ->
            io:format("get_option error ~p~n", [{error, URI_PORT}])
    end,
    case FromIPvN of
        4 ->
            RemoteSort = list_to_integer(URI_PORT);
        6 ->
            Temp = [0|URI_PORT],
            [X1,X2,X3] = Temp,
            RemotePort = X2 *256 + X3
    end,
```

```
case pdu:get_header(Bin) of
    {ok, _Version, _Type, _Tkl, _Code, _MessageID} ->
        io:format("Get Version: ~p, Type: ~p, Tkl: ~p, Code: ~p, MessageID: ~p~n",[_Version, _Type, _Tkl, _Code, _MessageID]);
    {error, H_reason} ->
        io:format("get_header error ~p~n", [{error, H_reason}]),
          {_Type, _Code, _MessageID} = {0, 0, 0}, %Unsafe case
          exit(no_Header)
end,
case pdu:get_token(Bin) of
    {ok, Token} ->
        io:format("Get Token: ~p~n",[Token]);
    {error, Token} ->
        io:format("get_token error ~p~n", [{error, Token}]),
          exit(no_Token)
end,
 {ok, Socket_to} = case FromIPvN of
     4 ->
          gen_udp:open(0, [binary, inet6, {active, true}]);
     6 ->
          gen_udp:open(0, [binary, inet, {active, true}])
 end,
 gen_udp:send(Socket_to, RemoteIP, RemotePort, Bin),
 %% send the packet and wait for the response
 {_, A3, A4} = now(),
 io:format("From client at: ~p seconds, ~p microseconds\n", [A1, A2]),
 io:format("Leave proxy at: ~p seconds, ~p microseconds\n", [A3, A4]),
 io:format("Process in proxy use: ~p seconds\n",[((A3*1000000+A4)-(A1*1000000+A2))/1000000]),
 receive
     {udp, Socket_to, RemoteIP, RemotePort, B} ->
          {_, A5, A6} = now(),
          io:format("Round trip time from proxy to sensor: ~p seconds\n",
```

```
[((A5*1000000+A6)-(A3*1000000+A4))/1000000]),
            gen_udp:send(Socket, FromIP, FromPort, B),
            {_, A7, A8} = now(),
            io:format("Process time to forward packet: ~p seconds\n", [((A7*1000000+A8)-(A5*1000000+A6))/1000000])
        %% after 5 seconds the gateway send back timeout information
        after 5000 ->
            case pdu:make_pdu(?COAP_ACKNOWLEDGEMENT, ?COAP_GATEWAY_TIMEOUT, binary_to_list(Token), _MessageID, atom_to_list(URI)) of
                {ok, TimeoutResponse} ->
                    gen_udp:send(Socket, FromIP, FromPort, TimeoutResponse),
                    io:format("Not received response in 5 seconds.~n");
                {error, Make_pdu} ->
                    io:format("Make pdu failed with reason: ~p~n",[Make_pdu])
            end
    end.
```

#PDU

```
-module(pdu).
-export([init/0, make_pdu/5, get_content/1, add_option/4, add_payload/3, get_header/1, get_token/1, get_URI/1, get_option/2]).
-on_load(init/0).

-define(APPNAME, wsngateway).

init() ->
    case code:priv_dir(?APPNAME) of
        {error, _} ->
            error_logger:format("~w priv dir not found~n", [?APPNAME]),
            exit(error);
        PrivDir ->
            erlang:load_nif(filename:join([PrivDir, "pdu_drv"]), 0)
    end.
```

```

MID::integer(), URI::list()) ->
make_pdu(_Type, _Method, _Token, _ID, _URI) ->
    erlang:nif_error(nif_not_loaded).

get_content(_Buffer) ->
    erlang:nif_error(nif_not_loaded).

add_option(_PDU, _Optnum, _Optlen, _Opnval) ->
     erlang:nif_error(nif_not_loaded).

add_payload(_PDU, _Payloadvalue, _Payloadlen) ->
     erlang:nif_error(nif_not_loaded).

Code::integer(), message::integer()} |
get_header(_PDU) ->
     erlang:nif_error(nif_not_loaded).

get_token(_PDU) ->
     erlang:nif_error(nif_not_loaded).

get_URI(_PUD) ->
     erlang:nif_error(nif_not_loaded).

get_option(_PDU, _Optnum) ->
     erlang:nif_error(nif_not_loaded).
```

#SERIAL

```
-module(serial).
-export([slipinit/0]).

%% This function initialize the SLIP port
slipinit() ->
```

```
    {ok, LocalIP} = inet_parse:address("2001::a"),
    gen_udp:open(0, [binary, inet6, {active, true}, {ifaddr, LocalIP}]).

%% this is not used in the project.
serialloop(Socket) ->
    {ok, Local} = inet_parse:address("127.0.0.1"),
    receive
        {udp, Socket, Local, 12345, Packet} ->
            {ok, _, _, _, _, Mid} = pdu:get_header(Packet),
%           IntMid = binary_to_integer(BinMid),
            io:format("Mid is: ~p~n",[Mid]),
            case get_keys(Mid) of
                undefined ->
                    io:format("undefined MessageID\n"),
                    undefined;
                [PID] ->
                    erase(PID),
                    PID ! {back, self(), Packet}
            end,
            serialloop(Socket);
        {delete, PID} ->
            erase(PID),
            serialloop(Socket);
        {FromPid, Mid, URI_HOST, URI_PORT, Coap} ->
            put(FromPid, Mid),
            Host_bin = list_to_binary(URI_HOST),
            Port_int = list_to_integer(URI_PORT),
            Packet = <<Host_bin/binary, 255:8, Port_int:16, 255:8, Coap/binary>>,
            gen_udp:send(Socket, Local, 12345, Packet),
            serialloop(Socket)
    end.
```

UDP-SERVER

```c
#include "contiki.h"
#include "contiki-net.h"
#include "sys/ctimer.h"
#include "net/uip.h"
#include "net/hc.h"
#include "net/uip-ds6.h"
#include "net/uip-udp-packet.h"
#include "net/netstack.h"
#include "dev/leds.h"
#include "sys/clock.h"
#include "dev/sht11-sensor.h"
#include "dev/light-sensor.h"
#include "er-coap-13.h"
#include "buffer.h"
#include "erbium.h"

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

//#include <netinet/in.h>

#define UDP_SERVER_PORT 5678
#define UIP_UDPIP_BUF   ((struct uip_udpip_hdr *)&uip_buf[UIP_LLH_LEN])

#define DEBUG DEBUG_PRINT
#include "net/uip-debug.h"

#define MAX(a, b) ((a) >= (b)? (a) : (b))


static struct uip_udp_conn *conn;
char val[20];
coap_packet_t* response;
unsigned short s,r;
```

```
int p;

PROCESS(udp_server_process, "UDP Server process");
AUTOSTART_PROCESSES(&udp_server_process);

void
init_message(coap_packet_t *packet, uint8_t type, uint8_t code, uint16_t mid)
// , uint8_t code, uint16_t mid, uint8_t version)
{
    coap_packet_t * coap_pkt = (coap_packet_t *) packet;
    memset(coap_pkt, 0, sizeof(coap_packet_t));

    coap_pkt->version=1;
    coap_pkt->type = type;
    coap_pkt->code = code;
    coap_pkt->mid = mid;
}

static void
tcpip_handler(void)
{
    int value=0;
    //printf("%.*x    %d\n",uip_datalen(),uip_appdata,UIP_UDPIP_BUF->len);
    if (init_buffer(300)) {
        if(uip_newdata()) {
            coap_packet_t* request = (coap_packet_t*)allocate_buffer(sizeof(coap_packet_t));
        //      coap_packet_t* response;
            if (request) {
                coap_parse_message(request, uip_appdata, uip_datalen());
                response = (coap_packet_t*)allocate_buffer(sizeof(coap_packet_t));
                init_message(response,COAP_TYPE_ACK,0,request->mid);

```

```
if(!memcmp("liph",request->uri_path,MAX(request->uri_path_len,4)))
        {
            if (request->code==1)//get
            {
                s=clock_time();
                value=get_light_photosynthetic();
                r=clock_time();
                printf("%u,%u,%u\n",s,r,(r-s));
                sprintf(val,"%d",value);
                //printf("%.*s\n",strlen(val),val);
                coap_set_status_code(response,CONTENT_2_05);
                coap_set_payload(response,&val,strlen(val));
//              printf("%d",value);
            }
            else
                coap_set_status_code(response,METHOD_NOT_ALLOWED_4_05); //Not Allowed
        }
        else if(!memcmp("lito",request->uri_path,MAX(request->uri_path_len,4)))
        {
            if (request->code==1)//get
            {
                s=clock_time();
                value=get_light_total_solar();
                r=clock_time();
                printf("%u,%u,%u\n",s,r,(r-s));
                sprintf(val,"%d",value);
                //printf("%.*s\n",strlen(val),val);
                coap_set_status_code(response,CONTENT_2_05);
                coap_set_payload(response,&val,strlen(val));
//              printf("%d",value);
            }
            else
```

```
                        coap_set_status_code(response,METHOD_NOT_ALLOWED_4_05);

                }
                else if(!memcmp("temp",request->uri_path,MAX(request->uri_path_len,4)))
                {
                    if (request->code==1)//get
                    {
                        s=clock_time();
                        value=get_temperature();
                        r=clock_time();
                        printf("%u,%u,%u\n",s,r,(r-s));
                        sprintf(val,"%d",value);
                        //printf("%.*s\n",strlen(val),val);
                        coap_set_status_code(response,CONTENT_2_05);
                        coap_set_payload(response,&val,strlen(val));
            //          printf("%d",value);
                    }
                    else
                        coap_set_status_code(response,METHOD_NOT_ALLOWED_4_05);

                }
                else if(!memcmp("humi",request->uri_path,MAX(request->uri_path_len,4)))
                {
                    if (request->code==1)//get
                    {
                        value=get_humidity();
                        sprintf(val,"%d",value);
                        //printf("%.*s\n",strlen(val),val);
                        coap_set_status_code(response,CONTENT_2_05);
                        coap_set_payload(response,&val,strlen(val));
            //          printf("%d",value);
                    }
```

```
                else
                    coap_set_status_code(response,METHOD_NOT_ALLOWED_4_05);

            }
            else if(!memcmp("batt",request->uri_path,MAX(request->uri_path_len,4)))
            {
                if (request->code==1)//get
                {
                    value=get_battery();
                    sprintf(val,"%d",value);
                    //printf("%.*s\n",strlen(val),val);
                    coap_set_status_code(response,CONTENT_2_05);
                    coap_set_payload(response,&val,strlen(val));
    //          printf("%d",value);
                }
            else
                coap_set_status_code(response,METHOD_NOT_ALLOWED_4_05);
            }
            else if(!memcmp("leds",request->uri_path,MAX(request->uri_path_len,4)))
            {
                uint8_t ld;
                if (request->code==1)//get
                {
                    ld=leds_get();
                    //printf("%x\n",ld);
                    coap_set_status_code(response,CONTENT_2_05);
                    coap_set_payload(response,&ld,1);
                }
                else if (request->code==3)//put
                {
                    memcpy(&ld,request->uri_query,1);
                    if(48<=ld && ld<=55)
```

```
                        {
                            leds_init();
                            leds_on(ld-48);
                        }
                        else
                            coap_set_status_code(response,BAD_REQUEST_4_00);
                    }
                    else if (request->code==4)//delete
                    {
                        memcpy(&ld,request->uri_query,2);
                        if (48<=ld && ld<=55)
                            leds_off(ld);
                        else
                            coap_set_status_code(response,BAD_REQUEST_4_00);
                    }
                }
                else
                {
                    //strcpy(val,"\0");
                    coap_set_status_code(response,NOT_FOUND_4_04);

                }


    coap_set_header_token(response,request->token,request->token_len);
                coap_set_header_uri_path(response,request->uri_path);

            }
        }
        delete_buffer();
    }
}
```

```

int
get_temperature(void)
{
    return ((sht11_sensor.value(SHT11_SENSOR_TEMP) / 10) - 396) / 10;
}

int
get_light_photosynthetic(void)
{
    return 10 * light_sensor.value(LIGHT_SENSOR_PHOTOSYNTHETIC) / 7;
}

int
get_light_total_solar(void)
{
    return 10 * light_sensor.value(LIGHT_SENSOR_TOTAL_SOLAR) / 7;
}

int
get_humidity(void)
{
    int h=sht11_sensor.value(SHT11_SENSOR_HUMIDITY);
    return -4+405*h/10000-28*h*h/10000000;
}

int
get_humidity_true(void)
{
    int t=get_temp();
    int h=sht11_sensor.value(SHT11_SENSOR_HUMIDITY);
    return (t-25)*(1/100+8*h/10000)+get_humi();
}

```

```
int
get_battery(void)
{
    return sht11_sensor.value(SHT11_SENSOR_BATTERY_INDICATOR);
}

void
send_packet(void* packet)
{
    //int i;
    char buf[300];
    int size=0;
//      printf("mid: %u\n",UIP_HTONS(response->mid));
//      printf("type: %u\n",response->type);
//      printf("code: %u\n",response->code);
 //     printf("payload: %s\n",*response->payload);
//      printf("token:");
//      for (i=0;i<response->token_len;i++)
//     printf("%d,",response->token[i]);
 //     printf("\n");

    size=coap_serialize_message(packet,buf);
    printf("Sending packet... MessageID: %u Packet size: %d\n",UIP_HTONS(response->mid),size);
    uip_udp_packet_sendto(conn, buf, size,&UIP_UDPIP_BUF->srcipaddr, UIP_UDPIP_BUF->srcport);
}

static void
print_local_addresses(void)
{
    int i;
    uint8_t state;

```

```
    PRINTF("IPv6 addresses: ");
    for(i = 0; i < UIP_DS6_ADDR_NB; i++) {
        state = uip_ds6_if.addr_list[i].state;
        if(uip_ds6_if.addr_list[i].isused &&(state == ADDR_TENTATIVE || state == ADDR_PREFERRED)) {
            PRINT6ADDR(&uip_ds6_if.addr_list[i].ipaddr);
            PRINTF("\n");
        }
    }
}

static void
set_global_address(void)
{
    uip_ipaddr_t ipaddr;

    uip_ip6addr(&ipaddr, 0x2001, 0, 0, 0, 0, 0, 0, 0);
    uip_ds6_set_addr_iid(&ipaddr, &uip_lladdr);
    uip_ds6_addr_add(&ipaddr, 0, ADDR_AUTOCONF);
    uip_ip6addr(&ipaddr, 0x2001, 0, 0, 0, 0, 0, 0,1);
    uip_ds6_addr_add(&ipaddr, 0, ADDR_MANUAL);

}

PROCESS_THREAD(udp_server_process, ev, data)
{
    int i;

    PROCESS_BEGIN();

    PROCESS_PAUSE();

    set_global_address();

```

```
    PRINTF("UDP server process started\n");

    print_local_addresses();

    SENSORS_ACTIVATE(light_sensor);
    SENSORS_ACTIVATE(sht11_sensor);
    conn = udp_new(NULL, NULL, NULL);
    if(conn == NULL) {
        PRINTF("No connection!\n");
        PROCESS_EXIT();
    }
    udp_bind(conn, UIP_HTONS(UDP_SERVER_PORT));

    while(1) {
        PROCESS_YIELD();
        if(ev == tcpip_event) {
            tcpip_handler();
           for(i=1;i<sizeof(uip_buf);i++)
           printf("%x", uip_buf[i]);
           printf("\n");
        send_packet(response);
        }

    }

    PROCESS_END();
}
```

#UDP-CLIENT

```
#include "contiki.h"
#include "contiki-net.h"
#include "sys/clock.h"
```

```
#include "net/uip.h"
#include "net/uip-ds6.h"
#include "net/uip-udp-packet.h"
#include "net/netstack.h"
#include "dev/leds.h"
#include "dev/button-sensor.h"
#include "dev/sht11-sensor.h"
#include "dev/light-sensor.h"
#include "er-coap-13.h"
#include "buffer.h"
#include "erbium.h"


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

#define UDP_SERVER_PORT 5678
#define TAP_PORT 5684
#define UIP_UDPIP_BUF   ((struct uip_udpip_hdr *)&uip_buf[UIP_LLH_LEN])

#define DEBUG DEBUG_PRINT
#include "net/uip-debug.h"

#define MAX(a, b) ((a) >= (b)? (a) : (b))


static struct uip_udp_conn *conn;
static uip_ipaddr_t tap_ipaddr;
char val[20];
unsigned short s,r;
int p;


PROCESS(udp_client_process, "UDP Server process");
```

```
AUTOSTART_PROCESSES(&udp_client_process);

uint8_t
int8_rand(void)
{
    return 1+(int)(255.0*rand()/(32767+1.0));
}

void
init_message(coap_packet_t *packet, uint8_t type, uint8_t code, uint16_t mid)
{
    coap_packet_t * coap_pkt = (coap_packet_t *) packet;
    memset(coap_pkt, 0, sizeof(coap_packet_t));

    coap_pkt->version=1;
    coap_pkt->type = type;
    coap_pkt->code = code;
    coap_pkt->mid = mid;
}

static void
tcpip_handler(void)
{
    int value=0;
    if (init_buffer(300)) {
        if(uip_newdata()) {
            coap_packet_t* request = (coap_packet_t*)allocate_buffer(sizeof(coap_packet_t));
            if (request) {
                coap_parse_message(request, uip_appdata, uip_datalen());
                printf("Value is: %.*s\n",request->payload_len,request->payload);
            }
        }
```

```
        delete_buffer();
    }
}


void
send_packet(void)
{
    int i=0;
   uint8_t token[8];
    char buf[300];
    int size=0;
    if (init_buffer(300)) {
        coap_packet_t* request = (coap_packet_t*)allocate_buffer(sizeof(coap_packet_t));
        init_message(request,COAP_TYPE_CON,1,random_rand());
        //printf("%u    %u\n",int8_rand(),random_rand());
        for(i;i<8;i++)
            token[i]=int8_rand();
        coap_set_header_token(request,token,8);
        coap_set_header_uri_path(request,"time");
        coap_set_header_uri_host(request,"127.0.0.1");
        request->uri_port=5678;
        SET_OPTION(request, COAP_OPTION_URI_PORT);
        size=coap_serialize_message(request,buf);
        printf("Sending Request... Message ID: %u\n",UIP_HTONS(request->mid));
        uip_udp_packet_sendto(conn, buf, size,&tap_ipaddr, UIP_HTONS(TAP_PORT));
        delete_buffer();
    }
}

static void
print_local_addresses(void)
{
```

```
    int i;
    uint8_t state;

    PRINTF("IPv6 addresses: ");
    for(i = 0; i < UIP_DS6_ADDR_NB; i++) {
        state = uip_ds6_if.addr_list[i].state;
        if(uip_ds6_if.addr_list[i].isused &&(state == ADDR_TENTATIVE || state ==
ADDR_PREFERRED)) {
            PRINT6ADDR(&uip_ds6_if.addr_list[i].ipaddr);
            PRINTF("\n");
        }
    }
}

static void
set_global_address(void)
{
    uip_ipaddr_t ipaddr;

    uip_ip6addr(&ipaddr, 0x2001, 0, 0, 0, 0, 0, 1, 0);
    uip_ds6_set_addr_iid(&ipaddr, &uip_lladdr);
    uip_ds6_addr_add(&ipaddr, 0, ADDR_AUTOCONF);
    uip_ip6addr(&ipaddr, 0x2001, 0, 0, 0, 0, 0, 1,1);
    uip_ds6_addr_add(&ipaddr, 0, ADDR_MANUAL);

    uip_ip6addr(&tap_ipaddr, 0x2001, 0, 0, 0, 0, 0, 0, 0xa);

}

PROCESS_THREAD(udp_client_process, ev, data)
{

    PROCESS_BEGIN();

```

```
    set_global_address();

    PRINTF("UDP client process started\n");

    print_local_addresses();

    SENSORS_ACTIVATE(button_sensor);
    SENSORS_ACTIVATE(light_sensor);
    SENSORS_ACTIVATE(sht11_sensor);
    conn = udp_new(NULL, NULL, NULL);
    if(conn == NULL) {
        PRINTF("No connection!\n");
        PROCESS_EXIT();
    }
    udp_bind(conn, UIP_HTONS(UDP_SERVER_PORT));

    while(1) {
        PROCESS_WAIT_EVENT();
        if(ev == tcpip_event) {
            tcpip_handler();

            r=clock_time();
            printf("æ”¶%d\n",r);
            p=(r-s)*1000000/128;
            printf("time %u\n",p);
        }
      if(ev == sensors_event && data == &button_sensor){
            send_packet();
            s=clock_time();
            printf("å ‘%d\n",s);
        }

    }

```

```
    PROCESS_END();
}
```

#BORDER_ROUTER

```
#include "contiki.h"
#include "contiki-lib.h"
#include "contiki-net.h"
#include "net/uip.h"
#include "net/uip-ds6.h"
#include "net/rpl/rpl.h"

#include "net/netstack.h"
#include "dev/button-sensor.h"
#include "dev/slip.h"
#include "dev/leds.h"

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

#define DEBUG 0
#include "net/uip-debug.h"

uint16_t dag_id[] = {0x1111, 0x1100, 0, 0, 0, 0, 0, 0x0011};

static uip_ipaddr_t prefix;
static uint8_t prefix_set;

PROCESS(border_router_process, "Border router process");

#if WEBSERVER==0
```

```
AUTOSTART_PROCESSES(&border_router_process);
#elif WEBSERVER>1

#include "webserver-nogui.h"
AUTOSTART_PROCESSES(&border_router_process,&webserver_nogui_process);
#else

#include "httpd-simple.h"

#define WEBSERVER_CONF_LOADTIME 0
#define WEBSERVER_CONF_FILESTATS 0
#define WEBSERVER_CONF_NEIGHBOR_STATUS 0

#define WEBSERVER_CONF_ROUTE_LINKS 0
#if WEBSERVER_CONF_ROUTE_LINKS
#define BUF_USES_STACK 1
#endif

PROCESS(webserver_nogui_process, "Web server");
PROCESS_THREAD(webserver_nogui_process, ev, data)
{
  PROCESS_BEGIN();

  httpd_init();

  while(1) {
    PROCESS_WAIT_EVENT_UNTIL(ev == tcpip_event);
    httpd_appcall(data);
  }

  PROCESS_END();
}
AUTOSTART_PROCESSES(&border_router_process,&webserver_nogui_process);

```

```
static const char *TOP = "<html><head><title>ContikiRPL</title></head><body>\n";
static const char *BOTTOM = "</body></html>\n";
#if BUF_USES_STACK
static char *bufptr, *bufend;
#define ADD(...) do {                                               \
    bufptr += snprintf(bufptr, bufend - bufptr, __VA_ARGS__);      \
  } while(0)
#else
static char buf[256];
static int blen;
#define ADD(...) do {                                               \
    blen += snprintf(&buf[blen], sizeof(buf) - blen, __VA_ARGS__);      \
  } while(0)
#endif

/*---------------------------------------------------------------------------*/
static void
ipaddr_add(const uip_ipaddr_t *addr)
{
  uint16_t a;
  int i, f;
  for(i = 0, f = 0; i < sizeof(uip_ipaddr_t); i += 2) {
    a = (addr->u8[i] << 8) + addr->u8[i + 1];
    if(a == 0 && f >= 0) {
      if(f++ == 0) ADD("::");
    } else {
      if(f > 0) {
        f = -1;
      } else if(i > 0) {
        ADD(":");
      }
      ADD("%x", a);
    }
  }
```

```
}
/*---------------------------------------------------------------------------*/
static
PT_THREAD(generate_routes(struct httpd_state *s))
{
  static int i;
  static uip_ds6_route_t *r;
  static uip_ds6_nbr_t *nbr;

#if BUF_USES_STACK
  char buf[256];
#endif
#if WEBSERVER_CONF_LOADTIME
  static clock_time_t numticks;
  numticks = clock_time();
#endif

  PSOCK_BEGIN(&s->sout);

  SEND_STRING(&s->sout, TOP);
#if BUF_USES_STACK
  bufptr = buf;bufend=bufptr+sizeof(buf);
#else
  blen = 0;
#endif
  ADD("Neighbors<pre>");

  for(nbr = nbr_table_head(ds6_neighbors);
      nbr != NULL;
      nbr = nbr_table_next(ds6_neighbors, nbr)) {

#if WEBSERVER_CONF_NEIGHBOR_STATUS
#if BUF_USES_STACK
{char* j=bufptr+25;
```

```
     ipaddr_add(&nbr->ipaddr);
     while (bufptr < j) ADD(" ");
     switch (nbr->state) {
     case NBR_INCOMPLETE: ADD(" INCOMPLETE");break;
     case NBR_REACHABLE: ADD(" REACHABLE");break;
     case NBR_STALE: ADD(" STALE");break;
     case NBR_DELAY: ADD(" DELAY");break;
     case NBR_PROBE: ADD(" NBR_PROBE");break;
     }
}
#else
{uint8_t j=blen+25;
     ipaddr_add(&nbr->ipaddr);
     while (blen < j) ADD(" ");
     switch (nbr->state) {
     case NBR_INCOMPLETE: ADD(" INCOMPLETE");break;
     case NBR_REACHABLE: ADD(" REACHABLE");break;
     case NBR_STALE: ADD(" STALE");break;
     case NBR_DELAY: ADD(" DELAY");break;
     case NBR_PROBE: ADD(" NBR_PROBE");break;
     }
}
#endif
#else
     ipaddr_add(&nbr->ipaddr);
#endif

     ADD("\n");
#if BUF_USES_STACK
     if(bufptr > bufend - 45) {
       SEND_STRING(&s->sout, buf);
       bufptr = buf; bufend = bufptr + sizeof(buf);
     }
#else
```

```
      if(blen > sizeof(buf) - 45) {
        SEND_STRING(&s->sout, buf);
        blen = 0;
      }
#endif
  }
  ADD("</pre>Routes<pre>");
  SEND_STRING(&s->sout, buf);
#if BUF_USES_STACK
  bufptr = buf; bufend = bufptr + sizeof(buf);
#else
  blen = 0;
#endif

  for(r = uip_ds6_route_head(); r != NULL; r = uip_ds6_route_next(r)) {

#if BUF_USES_STACK
#if WEBSERVER_CONF_ROUTE_LINKS
    ADD("<a href=http://[");
    ipaddr_add(&r->ipaddr);
    ADD("]/status.shtml>");
    ipaddr_add(&r->ipaddr);
    ADD("</a>");
#else
    ipaddr_add(&r->ipaddr);
#endif
#else
#if WEBSERVER_CONF_ROUTE_LINKS
    ADD("<a href=http://[");
    ipaddr_add(&r->ipaddr);
    ADD("]/status.shtml>");
    SEND_STRING(&s->sout, buf);
    blen = 0;
    ipaddr_add(&r->ipaddr);
```

```
    ADD("</a>");
#else
    ipaddr_add(&r->ipaddr);
#endif
#endif
    ADD("/%u (via ", r->length);
    ipaddr_add(uip_ds6_route_nexthop(r));
    if(1 || (r->state.lifetime < 600)) {
      ADD(") %lus\n", r->state.lifetime);
    } else {
      ADD(")\n");
    }
    SEND_STRING(&s->sout, buf);
#if BUF_USES_STACK
    bufptr = buf; bufend = bufptr + sizeof(buf);
#else
    blen = 0;
#endif
  }
  ADD("</pre>");

#if WEBSERVER_CONF_FILESTATS
  static uint16_t numtimes;
  ADD("<br><i>This page sent %u times</i>",++numtimes);
#endif

#if WEBSERVER_CONF_LOADTIME
  numticks = clock_time() - numticks + 1;
  ADD(" <i>(%u.%02u sec)</i>",numticks/CLOCK_SECOND,(100*(numticks%CLOCK_SECOND))/CLOCK_SECOND));
#endif

  SEND_STRING(&s->sout, buf);
  SEND_STRING(&s->sout, BOTTOM);
```

```

  PSOCK_END(&s->sout);
}
/*---------------------------------------------------------------------------*/
httpd_simple_script_t
httpd_simple_get_script(const char *name)
{

  return generate_routes;
}

#endif /* WEBSERVER */

/*---------------------------------------------------------------------------*/
static void
print_local_addresses(void)
{
  int i;
  uint8_t state;

  PRINTA("Server IPv6 addresses:\n");
  for(i = 0; i < UIP_DS6_ADDR_NB; i++) {
    state = uip_ds6_if.addr_list[i].state;
    if(uip_ds6_if.addr_list[i].isused &&
       (state == ADDR_TENTATIVE || state == ADDR_PREFERRED)) {
      PRINTA(" ");
      uip_debug_ipaddr_print(&uip_ds6_if.addr_list[i].ipaddr);
      PRINTA("\n");
    }
  }
}
/*---------------------------------------------------------------------------*/
void
request_prefix(void)
```

```
{
  uip_buf[0] = '?';
  uip_buf[1] = 'P';
  uip_len = 2;
  slip_send();
  uip_len = 0;
}
/*---------------------------------------------------------------------------*/
void
set_prefix_64(uip_ipaddr_t *prefix_64)
{
  uip_ipaddr_t ipaddr;
  memcpy(&prefix, prefix_64, 16);
  memcpy(&ipaddr, prefix_64, 16);
  prefix_set = 1;
  uip_ds6_set_addr_iid(&ipaddr, &uip_lladdr);
  uip_ds6_addr_add(&ipaddr, 0, ADDR_AUTOCONF);
}
/*---------------------------------------------------------------------------*/
PROCESS_THREAD(border_router_process, ev, data)
{
  static struct etimer et;
  rpl_dag_t *dag;

  PROCESS_BEGIN();

  prefix_set = 0;
  NETSTACK_MAC.off(0);
  PROCESS_PAUSE();
  SENSORS_ACTIVATE(button_sensor);

  PRINTF("RPL-Border router started\n");

  while(!prefix_set) {
```

```
      etimer_set(&et, CLOCK_SECOND);
      request_prefix();
      PROCESS_WAIT_EVENT_UNTIL(etimer_expired(&et));
    }

    dag = rpl_set_root(RPL_DEFAULT_INSTANCE,(uip_ip6addr_t *)dag_id);
    if(dag != NULL) {
      rpl_set_prefix(dag, &prefix, 64);
      PRINTF("created a new RPL dag\n");
    }

    NETSTACK_MAC.off(1);

  #if DEBUG || 1
    print_local_addresses();
  #endif

    while(1) {
      PROCESS_YIELD();
      if (ev == sensors_event && data == &button_sensor) {
        PRINTF("Initiating global repair\n");
        rpl_repair_root(RPL_DEFAULT_INSTANCE);
      }

    }

    PROCESS_END();
  }
  /*---------------------------------------------------------------------------*/
```

Bibliographic references

[1] http://en.wikipedia.org/wiki/Main_Page

[2] Mahmood Ali, Sai Kumar Ravula. Real-time support and energy efficiency in wireless sensor networks. Technical report, IDE0805, January 2008

[3] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo,D. Gay, J. Hill, M. Welsh, E. Brewer, and D. Culler. TinyOS: An Operating System for Sensor Networks

[4] Tobias Reusing，Christoph Söllner,Comparison of Operating Systems TinyOS and Contiki: 2012-08

[5] Francesco Cesarini and Simon Thompson. Erlang Programming. June 2009: First Edition.

[6] Alessandro Sivieri, A Programming Framework for the Internet of Things

[7] Z. Shelby, K. Hartke, C. Bormann. draft-ietf-core-coap-18. June 28, 2013

[8] tmote-sky-datasheet

[9] IoT6 project at: http://www.iot6.eu/

[10] IoT-A project at: http://www.iot-a.eu

[11] WebIOPi Internet of Things framework at: https://code.google.com/p/webiopi/

[12] E. Rescorla and N. Modadugu. Datagram Transport Layer Security Version 1.2. RFC 6347 (Proposed Standard), Jan. 2012.

[13] S. Raza, S. Duquennoy, J. H¨oglund, U. Roedig, and T. Voigt. Secure Communication for the Internet of Things - A Comparison of Link-Layer Security and IPsec for 6LoWPAN. Security and Communication Networks, Jan. 2012.