



**TECHNOLOGICAL EDUCATIONAL
INSTITUTE OF CRETE**

Department of Applied Informatics & Multimedia

CoAP-enabled Sensors for the Internet-of-Things

Evagelia Raptopoulou (AM: 3421)

Afroditi Argiropoulou (AM: 2795)

Supervisor: Mr. Manifavas Charalambos

Crete 2014 ©

Statement

We declare that we have developed this semester project individually under the leadership of Mr.Manifavas Charalambos. We have mentioned all the sources and publications we have used.

.....
Evagelia Raptopoulou, Afroditi Argiropoulou

October 1, 2014

Acknowledgments

We thank everyone that helped us with the elaboration of our project. We would like to specially thank our teachers that provided us with knowledge and motive.

Abstract

Ubiquitous sensing enabled by Wireless Sensor Network (WSN) technologies affects many areas of our modern day living. This offers the ability to measure, interfere and understand environmental statistics, from delicate ecologies and natural resources to smart homes and urban environments. The continuous increment of the number of these devices in an interactive network creates the Internet of Things (IoT) and the information is shared across platforms. Fed by the recent adaptation of a variety of enabling wireless technologies the IoT has made its kick-start and is the next revolutionary technology in transforming the Internet into a fully integrated Future Internet. As we move from www (static pages web) to web2 (social networking web) to web3 (ubiquitous computing web), the need for fast data access increases significantly. This paper presents an implementation of the Constrained Application Protocol (CoAP) on sensor modes using the Contiki OS. Several services are exposed and made available to conventional web browsers.

Keywords: Internet of Things; Wireless sensor networks, Contiki, Cooja;

Σύνοψη

Η έξυπνη ανίχνευση που ενεργοποιείται από ασύρματο δίκτυο αισθητήρων (WSN) επηρεάζει πολλούς τομείς της σύγχρονης ζωής μας. Αυτή προσφέρει τη δυνατότητα να καταγραφούν και να κατανοηθούν περιβαλλοντικοί παράγοντες, από ευαίσθητα οικοσυστήματα και φυσικούς πόρους μέχρι τα έξυπνα σπίτια και τα αστικά περιβάλλοντα. Η συνεχόμενη αύξηση του αριθμού των συσκευών αυτών μέσα σε ένα διαδραστικό δίκτυο αποτελεί το Διαδίκτυο για Αντικείμενα (IoT) και οι πληροφορίες διαμοιράζονται από πλατφόρμα σε πλατφόρμα. Λόγω της πρόσφατης προσαρμογής σε μια γκάμα ασύρματων τεχνολογιών, το Διαδίκτυο για Αντικείμενα έκανε μια εντυπωσιακή αρχή και αποτελεί πλέον την τεχνολογία που θα εξελίξει το διαδίκτυο πέρα από αυτό που ξέρουμε. Καθώς προχωράμε από το www(στατικές ιστοσελίδες) στο web2(κοινωνικός διαδικτυακός ιστότοπος) και στο web3(έξυπνη χρήση διαδικτύου), η ανάγκη για γρήγορη προσπέλαση δεδομένων αυξάνεται σημαντικά. Αυτή η εργασία παρουσιάζει την υλοποίηση του Πρωτοκόλλου Εξαναγκασμένης Εφαρμογής (Constrained Application Protocol-CoAP) σε αισθητήρες, χρησιμοποιώντας το λειτουργικό σύστημα Contiki. Διάφορες υπηρεσίες μελετούνται και είναι προσβάσιμες σε συνήθεις φυλλομετρητές.

Contents

1 Introduction	8
1.1 Background	7
1.2 Ubiquitous computing in the next decade	9
1.3 Motive	11
1.4 Purpose	11
1.5 Structure of Thesis	11
1.6 Definitions	12
1.7 IoT elements.....	13
1.7.1 Radio Frequency Identification (RFID).....	14
1.7.2 Wireless Sensor Networks (WSN).....	14
1.7.3 Addressing schemes.....	16
1.7.4 Data storage and analytics.....	17
1.7.5 Visualization	18
2 COAP.....	19
2.1 Interaction.....	19
2.2 Methods.....	20
3 Technologies Used.....	23
3.1 Introduction	23
3.2 Sensors	23
3.3 Contiki.....	23
3.3.1 General Contiki Program Structure.....	23
3.4 Cooja	25
3.4.1 About Cooja.....	30
3.4.2 Starting Cooja.....	30
4 Implementation	26
4.1 Design Principles.....	26
4.2 TCP/IP	27
4.3 Create Resource.....	28
4.3.1 Create Periodic Resource	29
4.3.2 Create Event Resource	31
4.4 Program Parameters	32
4.4.1 Humidity	32
4.4.2 Temperature	32

4.4.3 LED status.....	33
4.5 Server	33
4.6 Client	35
4.7 Makefile	35
5 Problems encountered and Future directions.....	36
5.1 Problems encountered	36
5.2 Open challenges	36
6 Summary and Conclusions	38
7 References	39
8 Appentix	42

Table with all the document's figures.

Figure 1	Internet of Things schematic showing the end users and application areas
Figure 2	Internet Of Things
Figure 3	The WSN position in Internet
Figure 4	Work flow of Sensors
Figure 5	WSN layers
Figure 6	Abstract Coap layering
Figure 7	Two basic GET transactions, one successful, one not found
Figure 8	An asynchronous GET transaction
Figure 9	An orphaned transaction
Figure 10	The IoT/IP stack
Figure 11	Screenshot while simulate Cooja with two motes
Figure 12	Create new simulation interface
Figure 13	The simulation interface
Figure 14	The create mote type interface
Figure 15	The network window
Figure 16	The RPL route status, as visible to the border router

Figure 17	Server-Client Module
Figure 18	The TCP 3-way handshake
Figure 19	Screenshot from console 1
Figure 20	Cooja
Figure 21	Screenshot from console 2
Figure 22	Cooja on startup
Figure 23	GET method of the humidity resource in COAP
Figure 24	Observing humidity resource
Figure 25	Observing humidity resource moments later
Figure 26	GET method of the temperature resource in COAP
Figure 27	Observing temperature resource
Figure 28	Observing temperature resource moments later
Figure 29	GET method of the LedS resource in COAP
Figure 30	Observing LedS resource
Figure 31	Observing LedS resource moments later

1 Introduction

This chapter gives a short introduction to the background and purpose of this thesis and also the abbreviations used in this report.

1.1 Background

The next wave in the era of computing will be outside the realm of the traditional desktop. In the Internet of Things (IoT) paradigm, many of the objects that surround us will be on the network in one form or another. Sensor network technologies will rise to meet this new challenge, in which information and communication systems are invisibly embedded in the environment around us. These results in the generation of enormous amounts of data which have to be stored processed and presented in a seamless, efficient, and easily interpretable form. This model will consist of services that are commodities and delivered in a manner similar to traditional commodities. Cloud computing is a option that provides the virtual infrastructure for such utility computing which integrates monitoring devices, storage devices, analytics tools, visualization platforms and client delivery.

Smart connectivity with existing networks and context-aware computation using network resources is an indispensable part of IoT. With the growing presence of WiFi and 4G-LTE wireless Internet access, the evolution towards ubiquitous information and communication networks is already evident. However, for the Internet of Things vision to successfully emerge, the computing paradigm will need to go beyond traditional mobile computing scenarios that use smart phones and portables, and evolve into connecting everyday existing objects and embedding intelligence into our environment. For technology to disappear from the consciousness of the user, the Internet of Things demands:

A shared understanding of the situation of its users and their appliances.

Software architectures and pervasive communication networks to process and convey the contextual information to where it is relevant, and the analytics tools in the Internet of Things that aim for autonomous and smart behavior. With these three fundamental grounds in place, smart connectivity and context-aware computation can be accomplished.

The term Internet of Things was first coined by Kevin Ashton in 1999 in the context of supply chain management [1]. However, in the past decade, the definition has been more inclusive covering wide range of applications like healthcare, utilities, transport, etc. Although the definition of ‘Things’ has changed as technology evolved, the main goal of making a computer sense information without the aid of human intervention remains the same. A radical evolution of the current Internet into a Network of interconnected objects that not only harvests information from the environment (sensing) and interacts with the physical world (actuation/command/control), but also uses existing Internet standards to provide services for information transfer, analytics, applications, and communications. Fueled by the prevalence of devices enabled by open wireless technology as well as embedded sensor and actuator nodes, IoT has stepped out of its infancy and is on the verge of transforming the current static Internet into a fully integrated Future Internet [2]. The Internet revolution led to the interconnection between people at an unprecedented scale and pace. The next revolution will be the interconnection between objects to create a smart environment. Only in 2011 did the number of interconnected devices on the planet overtake the actual number of people. Currently there are 9 billion interconnected devices and it is expected to reach 24 billion devices by 2020. According to the GSMA, this amounts to \$1.3 trillion revenue opportunities for mobile network operators alone spanning vertical segments such as health, automotive, utilities and consumer electronics. A schematic of the interconnection of objects is depicted in Fig. 1, where the application domains are chosen based on the scale of the impact of the data generated. The users span from individual to national level organizations addressing wide ranging issues.

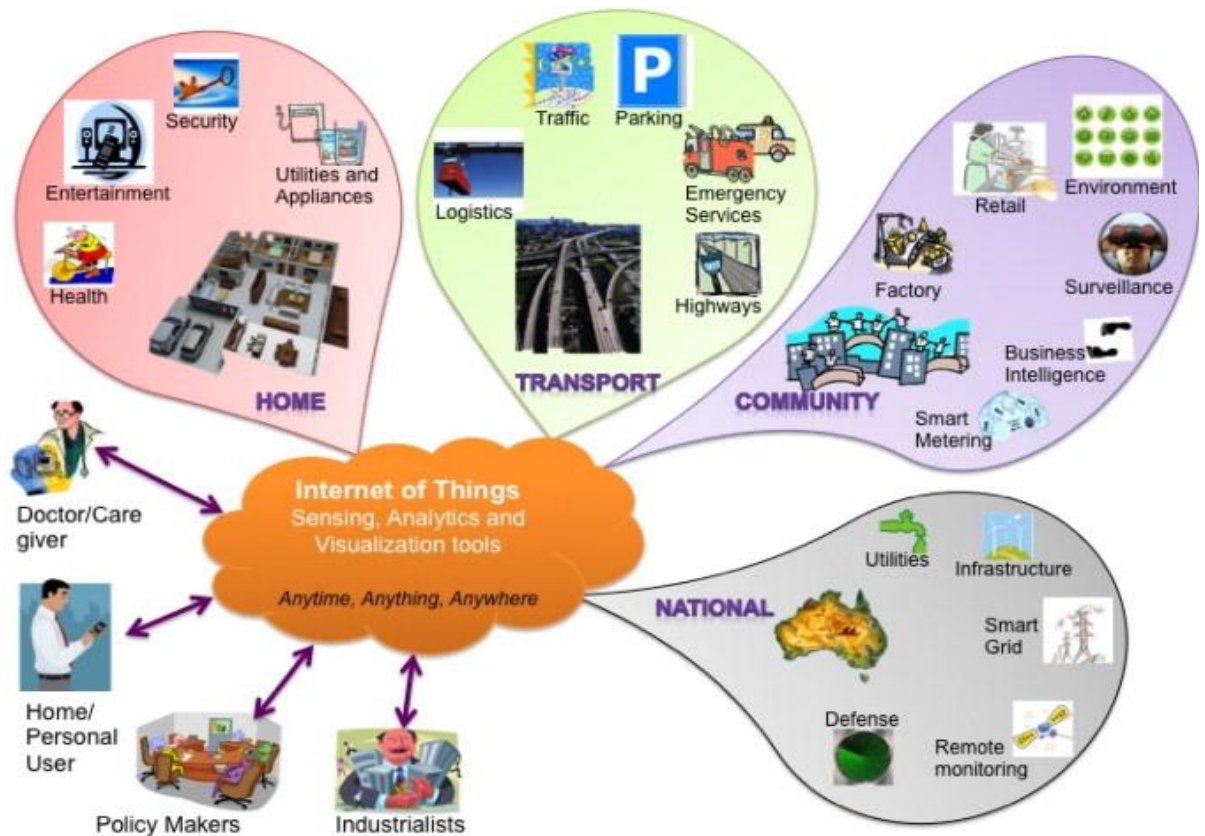


Figure 1, Internet of Things schematic showing the end users and application areas

1.2 Ubiquitous computing in the next decade

The effort by researchers to create a human-to-human interface through technology in the late 1980s resulted in the creation of the ubiquitous computing discipline, whose objective is to embed technology into the background of everyday life. Currently, we are in the post-PC era where smart phones and other hand-held devices are changing our environment by making it more interactive as well as informative. Mark Weiser, the forefather of Ubiquitous Computing, defined a smart environment [3] as “the physical world that is richly and invisibly interwoven with sensors, actuators, displays, and computational elements, embedded seamlessly in the everyday objects of our lives, and connected through a continuous network”.

The creation of the Internet has marked a foremost milestone towards achieving ubicomp’s vision which enables individual devices to communicate with any other device in

the world. The inter-networking reveals the potential of a seemingly endless amount of distributed computing resources and storage owned by various owners.

In contrast to Weiser's Calm computing approach, Rogers proposes a human centric ubicomp which makes use of human creativity in exploiting the environment and extending their capabilities [4]. He proposes a domain specific ubicomp solution when he says—"In terms of who should benefit, it is useful to think of how ubicomp technologies can be developed not for the Sal's of the world, but for particular domains that can be set up and customized by an individual firm or organization, such as for agricultural production, environmental restoration or retailing".

Caceres and Friday discuss the progress, opportunities and challenges during the 20 year anniversary of ubicomp. They discuss the building blocks of ubicomp and the characteristics of the system to adapt to the changing world. More importantly, they identify two critical technologies for growing the ubicomp infrastructure:

Cloud Computing

Internet of Things.

The advancements and convergence of MEMS technology, wireless communications, and digital electronics has resulted in the development of miniature devices having the ability to sense, compute, and communicate wirelessly in short distances. These miniature devices called nodes interconnect to form a WSN and find wide ranging applications in environmental monitoring, infrastructure monitoring, traffic monitoring, retail, etc. [5]. This has the ability to provide a ubiquitous sensing capability which is critical in realizing the overall vision of ubicomp as outlined by Weiser. For the realization of a complete IoT vision, efficient, secure, scalable and market oriented computing and storage resourcing is essential. This platform acts as a receiver of data from the ubiquitous sensors; as a computer to analyze and interpret the data; as well as providing the user with easy to understand web based visualization. The ubiquitous sensing and processing works in the background, hidden from the user.

1.3 Motive

Main motive of ours was to study the possibilities given by the Contiki OS. The use of C programming language was important for us because it is the main language used in Contiki applications. Both core programming and web user interface were important for us.

Contiki especially opens a whole wide ocean of opportunities that someone can just pick and innovate with, as long as he has the proper knowledge. Furthermore, the topic is really interesting for us because IoT is a technology with many fun applications such as smart homes and other.

1.4 Purpose

The main purpose of this project, from the user's point of view, is to suggest an easy web interface that provides information about temperature, humidity and led status of sensors.

The main purposes for us given in the project's specifications were to study the possibilities given by the use of Contiki OS and implement a virtual modes for humidity, temperature and led status. An important part is to study the structure of Contiki programs and satisfy the short memory constrains.

The main features are developed using C programming language. Cooja tool is used.

1.5 Structure of Thesis

This paper presents the current trends in IoT supported by Contiki OS and the development process of this technology. Specifically, in Section 2, we present an overview of CoAP and the methods it is using. In Section 3 we quote about the operating system Contiki and its usability in our thesis and generally in IoT technologies. Also we present Cooja the sensor simulator. In Section 4 we show the technologies that we needed in order to experiment with server-side applications in REST. In Section 5 and we conclude with discussions on open challenges and future trends. Section 6 is a summarize with some conclusions that we came up. In Section 7 we quote our references where we found a lot of useful sources which help us to finish this paper. The last Section is our source code.

1.6 Definitions

As identified by Atzori et al. [6], Internet of Things can be realized in three paradigms—internet-oriented (middleware), things oriented (sensors) and semantic-oriented

(knowledge). Although this type of delineation is required due to the interdisciplinary nature of the subject, the usefulness of IoT can be unleashed only in an application domain where the three paradigms intersect.

The RFID group defines the Internet of Things as– The worldwide network of interconnected objects uniquely addressable based on standard communication protocols.

According to Cluster of European research projects on the Internet of Things– ‘Things’ are active participants in business, information and social processes where they are enabled to interact and communicate among themselves and with the environment by exchanging data and information sensed about the environment, while reacting autonomously to the real/physical world events and influencing it by running processes that trigger actions and create services with or without direct human intervention.

According to Forrester [7], a smart environment– Uses information and communications technologies to make the critical infrastructure components and services of a city’s administration, education, healthcare, public safety, real estate, transportation and utilities more aware, interactive and efficient.

In our definition, we make the definition more user centric and do not restrict it to any standard communication protocol. This will allow long-lasting applications to be developed and deployed using the available state-of-the-art protocols at any given point in time. Our definition of the Internet of Things for smart environments is–

Interconnection of sensing and actuating devices providing the ability to share information across platforms through a unified framework, developing a common operating picture for enabling innovative applications. This is achieved by seamless ubiquitous sensing, data analytics and information representation with Cloud computing as the unifying framework.

CoAP	Constrained Application Protocol
IoT	Internet-of-Things
IETF	Internet Engineering Task Force
REST	Representational state transfer
URI	Uniform Resource Identifier

CoRE	Constrained RESTful Environments
UDP	User Datagram Protocol
UART	Universal asynchronous receiver/transmitter
UbiComp	Ubiquitous Computing
GIS	Geographic Information System
MEMS	Micro-electro-mechanical systems
UDGM	Unit Disk Graph Medium
JNI	Java Native Interface
WSN	Wireless Sensor Networks

Table 1, Abbreviations

1.7 IoT elements

We present a taxonomy that will aid in defining the components required for the Internet of Things from a high level perspective. There are three IoT components which enables seamless UbiComp:

1. Hardware - made up of sensors, actuators and embedded communication hardware
2. Middleware - on demand storage and computing tools for data analytics
3. Presentation - novel easy to understand visualization and interpretation tools which can be widely accessed on different platforms and which can be designed for different applications. In this section, we discuss a few enabling technologies in these categories which will make up the three components stated above.

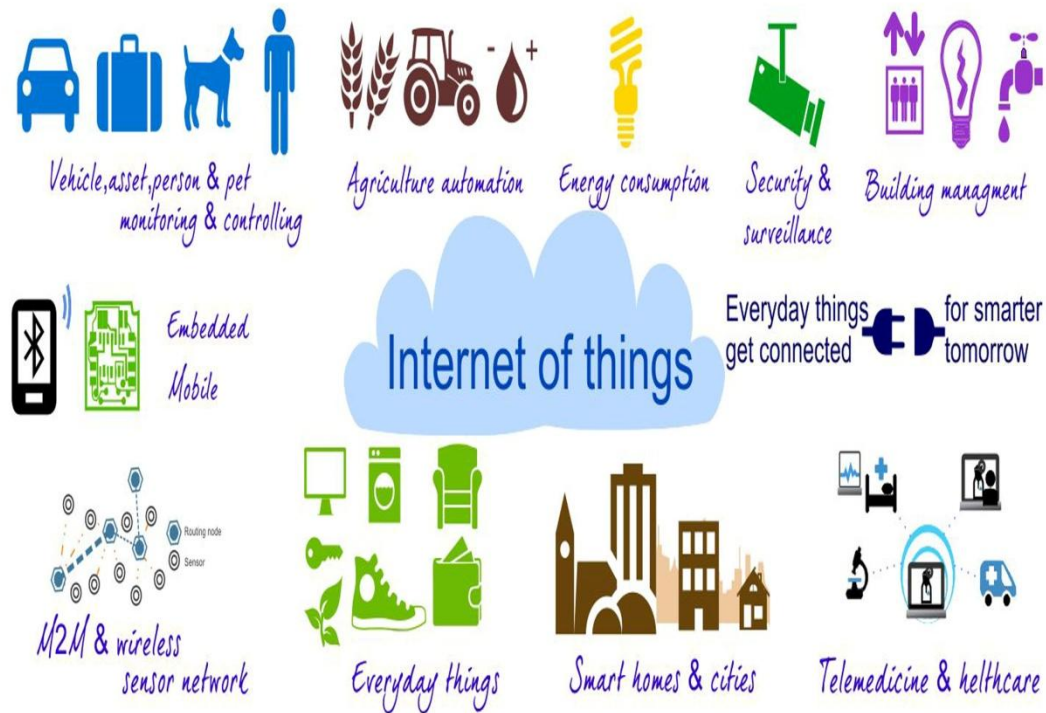


Figure 2,Internet Of Things

1.7.1 Radio Frequency Identification (RFID)

RFID technology is a major breakthrough in the embedded communication paradigm which enables design of microchips for wireless data communication. They help in the automatic identification of anything they are attached to acting as an electronic bar-code. The passive RFID tags are not battery powered and they use the power of the reader's interrogation signal to communicate the ID to the RFID reader. This has resulted in many applications particularly in retail and supply chain management. The applications can be found in transportation (replacement of tickets, registration stickers) and access control applications as well. The passive tags are currently being used in many bank cards and road toll tags which are among the first global deployments. Active RFID readers have their own battery supply and can instantiate the communication. Of the several applications, the main application of active RFID tags is in port containers [8] for monitoring cargo.

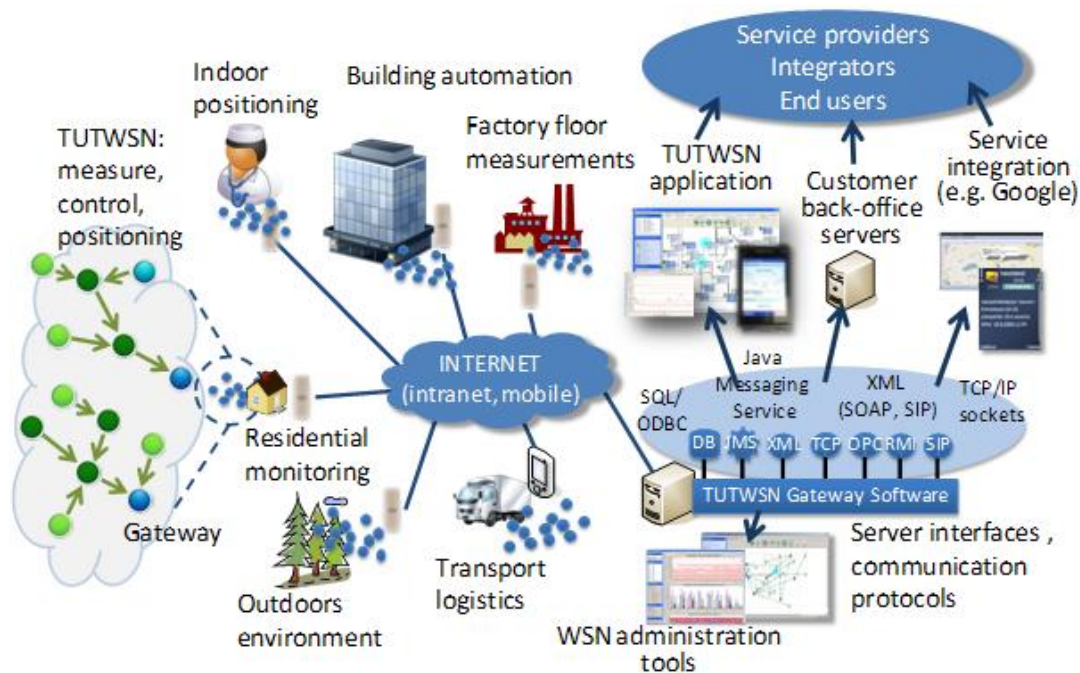


Figure 3, The WSN position in Internet

1.7.2 Wireless Sensor Networks (WSN)

Recent technological advances in low power integrated circuits and wireless communications have made available efficient, low cost, low power miniature devices for use in remote sensing applications. The combination of these factors has improved the viability of utilizing a sensor network consisting of a large number of intelligent sensors, enabling the collection, processing, analysis and dissemination of valuable information, gathered in a variety of environments. Active RFID is nearly the same as the lower end WSN nodes with limited processing capability and storage. The scientific challenges that must be overcome in order to realize the enormous potential of WSNs are substantial and multidisciplinary in nature. Sensor data are shared among sensor nodes and sent to a distributed or centralized system for analytics. The components that make up the WSN monitoring network include:

1. WSN hardware - Typically a node (WSN core hardware) contains sensor interfaces, processing units, transceiver units and power supply. Almost always, they comprise

of multiple A/D converters for sensor interfacing and more modern sensor nodes have the ability to communicate using one frequency band making them more versatile.

2. WSN communication stack - The nodes are expected to be deployed in an ad-hoc manner for most applications. Designing an appropriate topology, routing and MAC layer is critical for the scalability and longevity of the deployed network. Nodes in a WSN need to communicate among themselves to transmit data in single or multi-hop to a base station. Node drop outs, and consequent degraded network lifetimes, are frequent. The communication stack at the sink node should be able to interact with the outside world through the Internet to act as a gateway to the WSN sub-net and the Internet [9].
3. WSN Middleware - A mechanism to combine cyber infrastructure with a Service Oriented Architecture (SOA) and sensor networks to provide access to heterogeneous sensor resources in a deployment independent manner. This is based on the idea of isolating resources that can be used by several applications. A platform-independent middleware for developing sensor applications is required, such as an Open Sensor Web Architecture (OSWA). OSWA is built upon a uniform set of operations and standard data representations as defined in the Sensor Web Enablement Method (SWE) by the Open Geospatial Consortium (OGC).
4. Secure Data aggregation - An efficient and secure data aggregation method is required for extending the lifetime of the network as well as ensuring reliable data collected from sensors. Node failures are a common characteristic of WSNs, the network topology should have the capability to heal itself. Ensuring security is critical as the system is automatically linked to actuators and protecting the systems from intruders becomes very important.

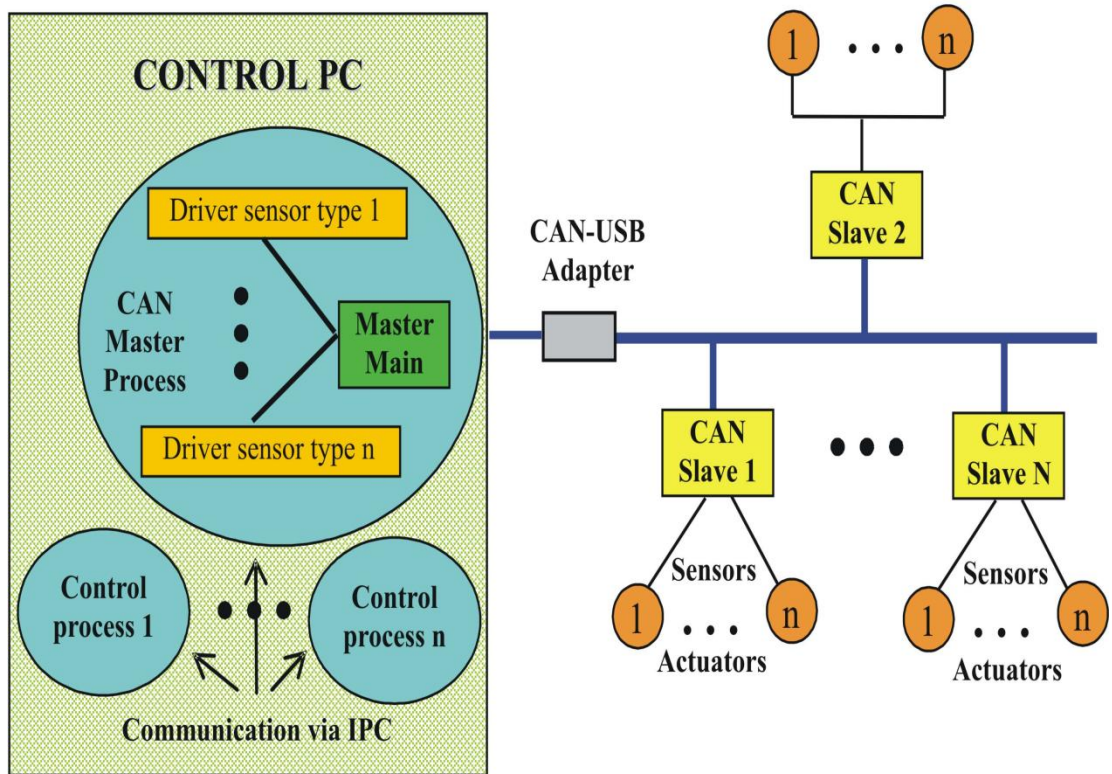


Figure 4, Work flow of Sensors

1.7.3 Addressing schemes

The ability to uniquely identify ‘Things’ is critical for the success of IoT. This will not only allow us to uniquely identify billions of devices but also to control remote devices through the Internet. The few most critical features of creating a unique address are: uniqueness, reliability, persistence and scalability.

Every element that is already connected and those that are going to be connected, must be identified by their unique identification, location and functionalities. The current IPv4 may support to an extent where a group of cohabiting sensor devices can be identified geographically, but not individually. The Internet Mobility attributes in the IPV6 may alleviate some of the device identification problems; however, the heterogeneous nature of wireless nodes, variable data types, concurrent operations and confluence of data from devices exacerbates the problem further [10].

Persistent network functioning to channel the data traffic ubiquitously and relentlessly is another aspect of IoT. Although, the TCP/IP takes care of this mechanism by routing in a

more reliable and efficient way, from source to destination, the IoT faces a bottleneck at the interface between the gateway and wireless sensor devices. Furthermore, the scalability of the device address of the existing network must be sustainable. The addition of networks and devices must not hamper the performance of the network, the functioning of the devices, the reliability of the data over the network or the effective use of the devices from the user interface.

To address these issues, the Uniform Resource Name (URN) system is considered fundamental for the development of IoT. URN creates replicas of the resources that can be accessed through the URL. With large amounts of spatial data being gathered, it is often quite important to take advantage of the benefits of metadata for transferring the information from a database to the user via the Internet [11]. IPv6 also gives a very good option to access the resources uniquely and remotely. Another critical development in addressing is the development of a lightweight IPv6 that will enable addressing home appliances uniquely.

Wireless sensor networks (considering them as building blocks of IoT), which run on a different stack compared to the Internet, cannot possess IPv6 stack to address individually and hence a subnet with a gateway having a URN will be required. With this in mind, we then need a layer for addressing sensor devices by the relevant gateway. At the sub-net level, the URN for the sensor devices could be the unique IDs rather than human-friendly names as in the www, and a lookup table at the gateway to address this device. Further, at the node level each sensor will have a URN (as numbers) for sensors to be addressed by the gateway. The entire network now forms a web of connectivity from users (high-level) to sensors (low-level) that is addressable (through URN), accessible (through URL) and controllable (through URC).

1.7.4 Data storage and analytics

One of the most important outcomes of this emerging field is the creation of an unprecedented amount of data. Storage, ownership and expiry of the data become critical issues. The internet consumes up to 5% of the total energy generated today and with these types of demands, it is sure to go up even further. Hence, data centers that run on harvested energy and are centralized will ensure energy efficiency as well as reliability. The data have to be stored and used intelligently for smart monitoring and actuation. It is important to develop artificial intelligence algorithms which could be centralized or distributed based on

the need. Novel fusion algorithms need to be developed to make sense of the data collected. State-of-the-art non-linear, temporal machine learning methods based on evolutionary algorithms, genetic algorithms, neural networks, and other artificial intelligence techniques are necessary to achieve automated decision making. These systems show characteristics such as interoperability, integration and adaptive communications. They also have a modular architecture both in terms of hardware system design as well as software development and are usually very well-suited for IoT applications. More importantly, a centralized infrastructure to support storage and analytics is required. This forms the IoT middleware layer and there are numerous challenges involved which are discussed in future sections. As of 2012, Cloud based storage solutions are becoming increasingly popular and in the years ahead, Cloud based analytics and visualization platforms are foreseen.

1.7.5 Visualization

Visualization is critical for an IoT application as this allows the interaction of the user with the environment. With recent advances in touch screen technologies, use of smart tablets and phones has become very intuitive. For a lay person to fully benefit from the IoT revolution, attractive and easy to understand visualization has to be created. As we move from 2D to 3D screens, more information can be provided in meaningful ways for consumers. This will also enable policy makers to convert data into knowledge, which is critical in fast decision making. Extraction of meaningful information from raw data is non-trivial. This encompasses both event detection and visualization of the associated raw and modeled data, with information represented according to the needs of the end-user.

2 COAP

Constrained Application Protocol (CoAP) is a specialized web transfer protocol for use with constrained networks and nodes for machine-to-machine applications such as smart energy and building automation. It is particularly targeted for small low power sensors, switches, valves and similar components that need to be controlled or supervised remotely, through standard Internet networks. CoAP provides a method/response interaction model between application end-points, supports built-in resource discovery, and includes key web concepts such as URIs and content-types. CoAP easily translates to HTTP for integration with the web while meeting specialized requirements such as multicast support, very low overhead and simplicity for constrained environments.

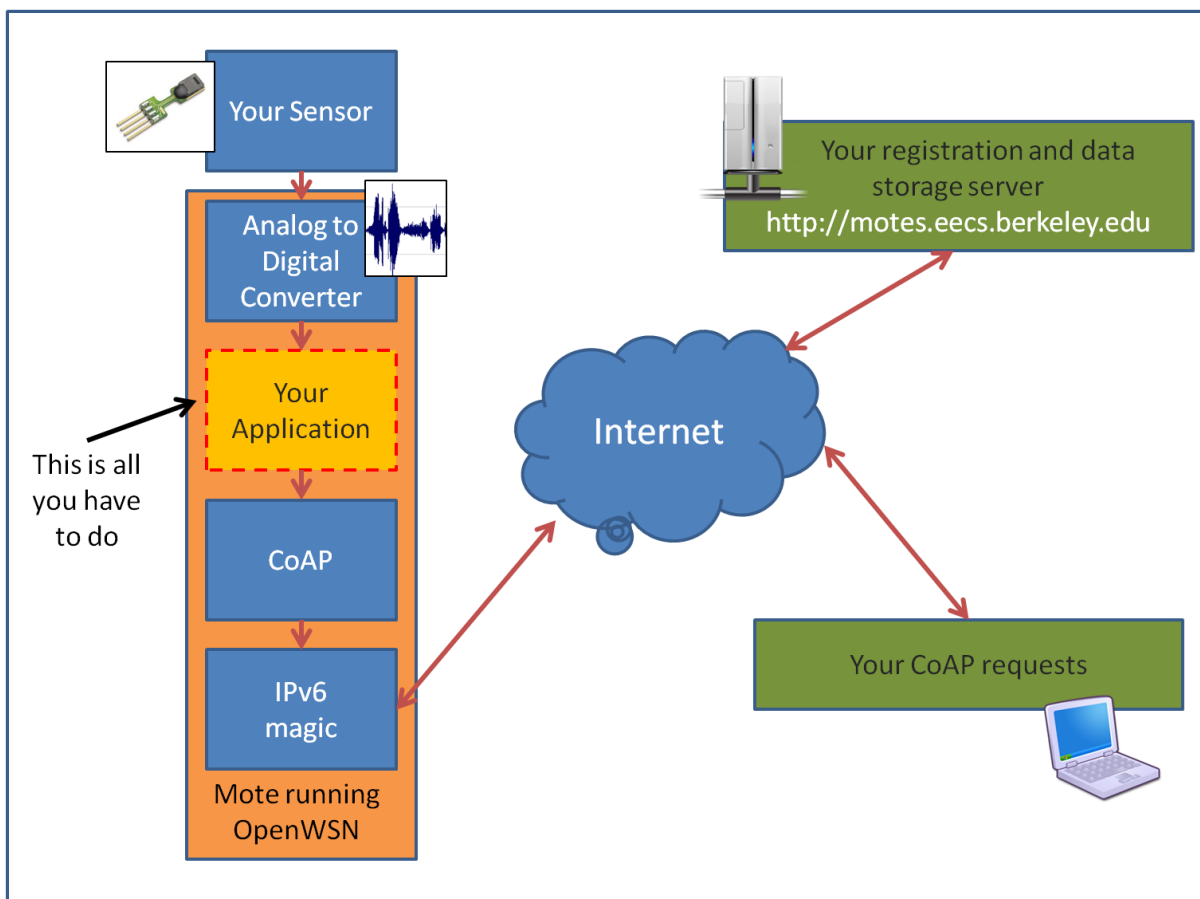


Figure 5,How WSN works on layers

The use of web services (web APIs) on the Internet has become ubiquitous in most applications and depends on the fundamental Representational State Transfer [REST] architecture of the Web. The work on Constrained RESTful Environments (CoRE) aims at realizing the REST architecture in a suitable form for the most constrained nodes (e.g., 8-bit microcontrollers with limited RAM and ROM) and networks (e.g., 6LoWPAN, [RFC4944]). Constrained networks such as 6LoWPAN support the fragmentation of IPv6 packets into small link-layer frames; however, this causes significant reduction in packet delivery probability. One design goal of CoAP has been to keep message overhead small, thus limiting the need for fragmentation.

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. The goal of CoAP is not to blindly compress HTTP [RFC2616], but rather to realize a subset of REST common with HTTP but optimized for M2M applications. Although CoAP could be used for refashioning simple HTTP interfaces into a more compact protocol, more importantly it also offers features for M2M such as built-in discovery, multicast support, and asynchronous message exchanges.

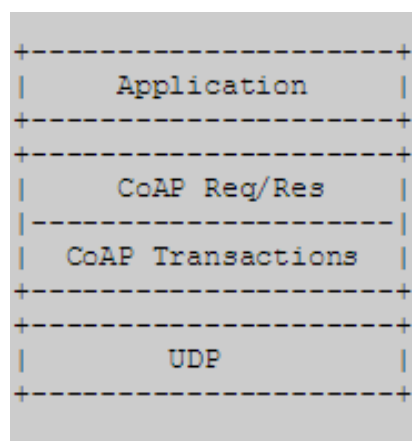


Figure 6, Abstract Coap layering

2.1 Interaction

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 (called an end-point). A CoAP exchange is equivalent to that of HTTP, and is sent by a client to request an action on a resource (identified by a URI) on a server. A response is then sent with a Response Code and resource representation if appropriate.

Unlike HTTP, CoAP deals with these interchanges asynchronously over a UDP transport with support for both unicast and multicast interactions. This is achieved using transaction messages supporting optional reliability (with exponential back-off) and transaction IDs between end-points to carry requests and responses. These transactions are transparent to the request/response interchanges. The only difference being that responses may arrive asynchronously.

One could think of CoAP as using a two-layer approach, a transactional layer used to deal with UDP and the asynchronous nature of the interactions, and the request/response interactions using Method and Response codes. The most basic interaction between the Req/Res and Transaction layers works by sending a request in a confirmable CoAP message and waiting for an acknowledgment message that also carries the response. E.g., two possible interactions for a basic GET are shown in Figure 7.

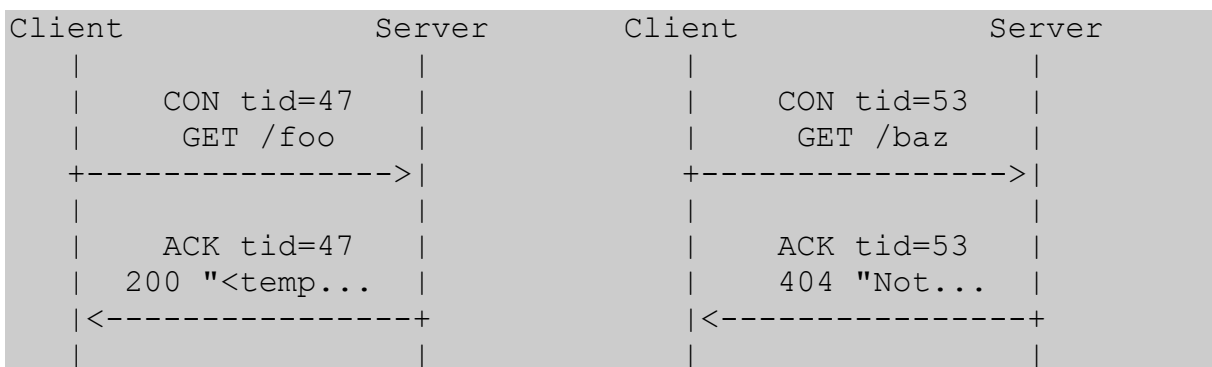


Figure 7, Two basic GET transactions, one successful, one not found

Note that at the transaction layer, the response is returned in an ACK message, independent of whether the request was successful at the Req/Res layer. In effect, the response is piggy-backed on the ACK message, so no separate acknowledgment is required that the GET message was received. The relationship between the confirmable message (CON) and the acknowledgment message (ACK) is indicated by the transaction ID, which is echoed back by the server in the ACK. Transaction IDs are short-lived, they only serve to couple CON and ACK messages. The tight coupling between CON and ACK also relieves the ACK of the need to echo back information from the request, such as the Token Option supplied by the client. We say that a response carried in an ACK pertains to the request in the corresponding CON. Not all interactions are as simple as the basic synchronous exchange shown. For example, a server might need longer to obtain the representation of the resource requested than it can wait sending back the acknowledgment, without risking the client to repeatedly retransmit the request. To handle this case, the response is decoupled from the transaction layer acknowledgment. Actually, the latter does not carry any message at all.

As the client cannot know that this will be the case, it sends exactly the same confirmable message with the same request. The server maybe attempts to obtain the resource (e.g., by acting as a proxy) and times out an ACK timer, or it immediately sends an acknowledgment knowing in advance that there will be no quick answer. The acknowledgment effectively is a promise that the request will be acted upon, see Figure 8 . Since no Token Option was included in the initial request, an "Token Option required by server (CoAP 240)" error will be returned in the ACK. The client would then repeat the request, now including a Token Option. For a request where an asynchronous response is expected, the Token Option can be included in the first request.

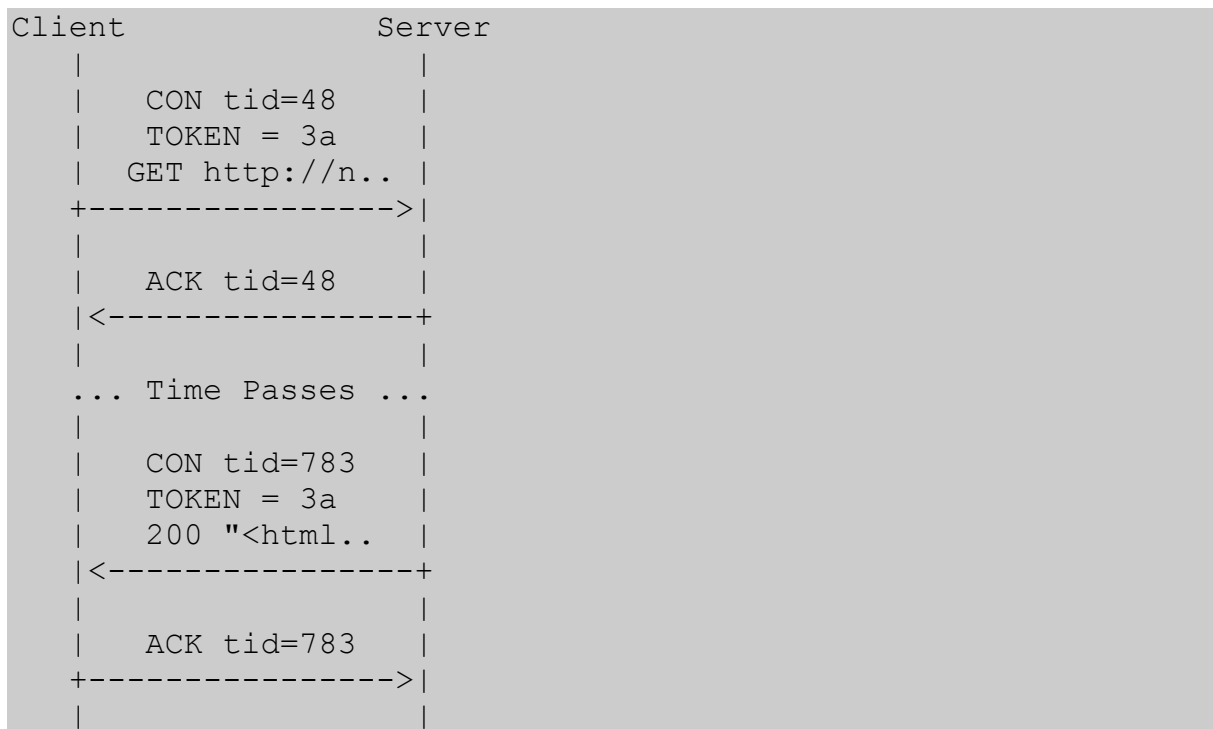


Figure 8, An asynchronous GET transaction

When the server finally has obtained the resource representation and is ready to send the response, it initiates a transaction to the client. This new transaction has its own transaction ID, so there is no automatic coupling of the response to the request. Instead, the Token Option is echoed back to the client in order to associate the response to the original request. To ensure that this message is not lost, it is again sent as a confirmable message and answered by the client with an ACK, citing the new TID chosen by the server. As a special failure situation, a client may no longer be aware that it sent a request, e.g., if it does not have stable storage and was rebooted in the meantime. This can be indicated by a special "Reset" message, as shown in Figure 9.

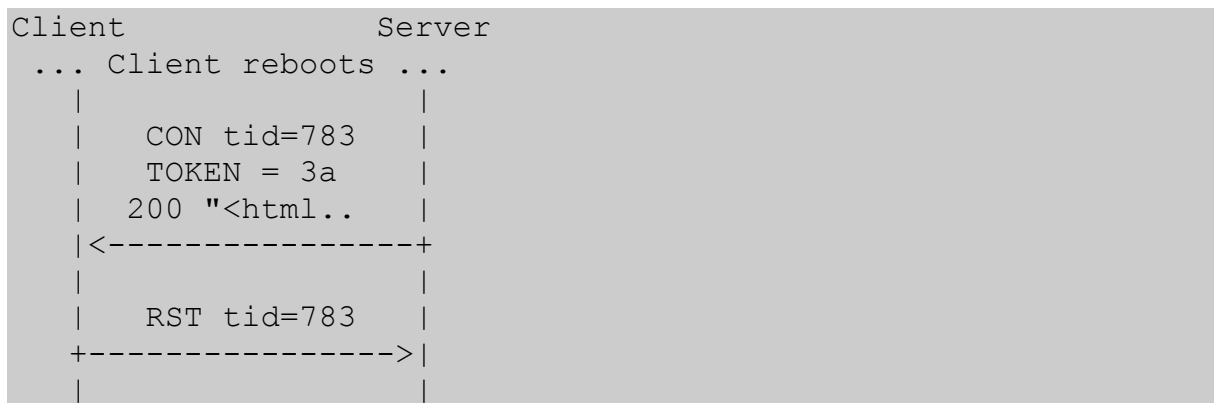


Figure 9, An orphaned transaction

2.2 Methods

CoAP supports the basic methods of GET, POST, PUT, DELETE, which are easily mapped to HTTP. As CoAP methods manipulate resources, they have the same properties of safe (only retrieval) and idempotent (you can invoke it multiple times with the same effects) as HTTP Section 9.1 [RFC2616]. The GET method is safe, therefore it MUST NOT take any other action on a resource other than retrieval. The GET, PUT and DELETE methods MUST be performed in such a way that they are idempotent. Unlike PUT, POST is not idempotent because the URI in the request indicates the resource that will handle the enclosed body. This resource indicated by the POST may be used for data processing, a gateway to other protocols and it may create a new resource as a result of the POST.

The basic methods are :

GET

The GET method retrieves the information of the resource identified by the request URI. Upon success a 200 (OK) response SHOULD be sent.

POST

The POST method is used to request the server to create a new subordinate resource under the requested parent URI. If a resource has been created on the server, the response SHOULD be 201 (Created) including the URI of the new resource in a Location Option with any possible status in the message body. If the POST succeeds but does not result in a new

resource being created on the server, a 200 (OK) response code SHOULD be returned. Responses to this method are not cacheable.

PUT

The PUT method requests that the resource identified by the request URI be updated or created with the enclosed message body. If a resource exists at that URI the message body SHOULD be considered a modified version of that resource, and a 200 (OK) response SHOULD be returned. If no resource exists then the server MAY create a new resource with that URI, resulting in a 201 (Created) response. If the resource could not be created or modified, then an appropriate error response code SHOULD be sent.

Responses to this method are not cacheable.

DELETE

The DELETE method requests that the resource identified by the request URI be deleted. The response 200 (OK) SHOULD be sent on success.

Responses to this method are not cacheable.

3 Related Technologies

In order to assimilate this report a brief description of important technologies is given.

3.1 Introduction

Several complementary technologies are associated with this IoT application. As previously discussed, the most common technique to start is the sensors, so influences on sensors are likewise influences on Contiki OS itself. Both of them are going to be extended in detail later. However, other specifications and technologies also impact on, with most important Cooja tool(chapter 3.4).

3.2 Sensors

There is a wide range of application areas in which sensor networks can be used. Military applications include surveillance and reconnaissance, and in the health area sensor nodes could help monitor and aid patients. As deployment time can be very short, sensor networks can be used for monitoring disaster areas. Another example is to use sensor nodes in smart homes as alarms controlling devices.

Advances in wireless communication and electronic enable development of cheaper and smaller sensor nodes. Basically, a sensor node has processing power, wireless communication abilities and sensing devices. But to minimize costs they are often very limited, a typical sensor node has a short communication range, low initial amount of energy source, as that often is the main factor corresponding to the length of the sensor node life.

A sensor network may consist of up to the thousands or even millions of sensor nodes densely deployed and without pre-determined positions. The network has to be fault tolerant, scalable and of low production costs. These circumstances demand a new set of ad-hoc protocols and algorithms to be developed. Those used in traditional networks are often not well suited for sensor networks, for reasons such as the larger amount of nodes in communication range, high failure rates and limited resources of individual nodes.

Sensor networks are very application-specific, different networks' ideal nodes may differ in both hardware and algorithms. And of course, sensor node hardware can be constructed in several different ways. A simulator should therefore be adjustable to easily simulate different software as well as deferent underlying hardware platforms.

Also, similar wireless sensor network may include deferent kinds of sensor nodes with different purposes. As an example consider a network with two different types of nodes, a cheap type and an expensive type. The cheap nodes are simple, the only gather temperature data and forwards it to the nearest expensive and present the average temperature on the Internet. Note that these sensor node type differ not only in their running software, but also in their hardware platforms. To simulate the above example, a simulator must support such heterogeneous networks, with several node types differing in both hardware and software.

3.3 Contiki

Contiki is an operating system designed for memory constrained environments, such as the nodes used in WSN. It is built around an event-driven kernel, and features include dynamic loading and unloading of individual programs and services, and optional preprocess pre-emptive multi-threading. It also supports a full TCP/IP stack via the uIP library, as well as the programming abstraction Protothreads. Contiki is implemented in the C language and has been designed to be easily portable to new platforms. It has been ported to more than 20 deferent platforms since its release 2003.

In a purely event-based system, a process is implemented as an event handler, letting different blocks of code execute depending on which event is given. These blocks are always allowed to run to completion once called. Since a single code block will never be interrupted, these blocks can be designed so that they may all share the same stack. Compared to a multi-threaded model this requires less memory and computation overhead when having several concurrent processes. In Contiki, a process consists of an event handler and an optional poll handler function. The Contiki kernel holds the event scheduler that dispatches events to processes and periodically polls processes that registered a poll handler function. It uses a single stack for all processes, which is rewound between each invocation of an event handler.

To better understand Contiki one should take a look at its system architecture.

Typically, a running Contiki system consists of the kernel, libraries, the program loader, and a set of processes. Communication between the processes always goes through the kernel, which does not provide a hardware abstraction layer, but lets device drivers and applications communicate directly with the hardware. A process is defined by an event handler function and an optional poll handler function. The process state is held in the process' private memory and the kernel only keeps a pointer to the process state. All processes share the same address space and do not run in different protection domains. Interprocess communication is done by posting events. Looking at it from a higher perspective, the Contiki system is made up of two parts: the core and the loaded programs. Typically, the core consists of the kernel, the program loader, the most commonly used parts of the language runtime, and a communication stack with device drivers for the communication hardware. The core is compiled into a single binary image and is usually not modified after deployment (although it is possible to use a special boot loader to overwrite or patch the core). The program loader is in charge of loading/unloading the programs into the system either by using the communication stack or directly attached storage (such as EEPROM). It is also important to note here the abstraction model:

Programs know the core; the core does not know the program.

Using Contiki as the operating system of the motes offers many advantages. As mentioned before, Contiki's kernel is event based, which means that the block wait abstraction is not supported. For this reason, when programming such a system one will have to use a state machine to implement the control flow for high level logic that cannot be expressed as a single event handler. However this approach is cumbersome and programmers usually have problems with it. To overcome this shortcoming and simplify the programs, Contiki implements a programming abstraction called protothreads. One has to remember though, that as the name implies, they are merely tools for creating a conditional blocking wait statement [14].

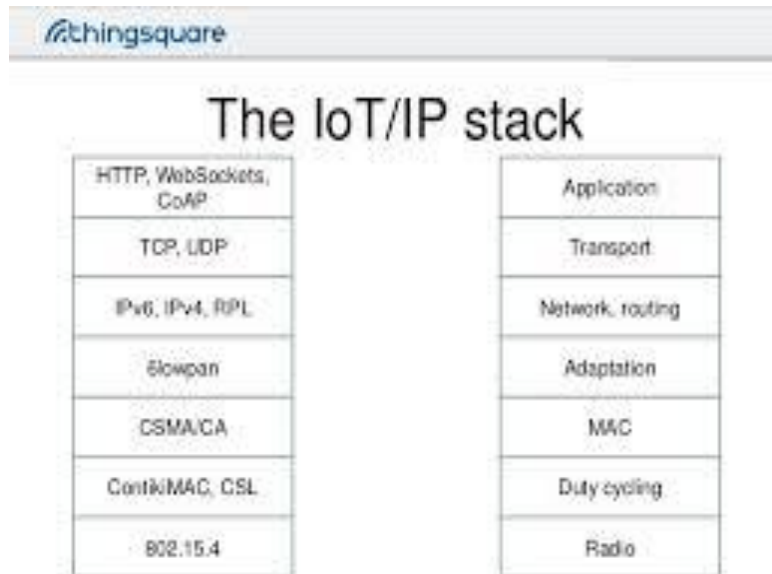


Figure 10,The IoT/IP stack

When developing software for large sensor networks it is very important to be able to dynamically download program code into the network. One possible advantage of this is the ability to patch bugs in operational networks. As opposed to most operating systems for embedded systems, such as TinyOS, which require a complete binary image of the entire system to be flushed into each device, Contiki has the ability to load and unload individual applications or services at runtime. Of course, this greatly reduces the actual time required to do the modifications. As an example just think of a network of 30 motes at the time when some routing protocol is tested and modifications to the code are bound to be numerous. If it takes us around 2 minutes to program one mote than we would need an hour to program the entire network. With Contiki, this reprogramming can be done in less than 5 minutes! This is why dynamically loading programs into the system and programming over the radio is one of the key features of Contiki. Dynamic loading of programs is done using ELF object files and more information can be found in.

The support of a native TCP/IP protocol stack is very important in the context of embedded systems because it provides interoperability with the existing systems and makes it easy to integrate Contiki into the existing IP network infrastructure. As mentioned in one of the goals of future research in WSN should be the consideration of the Internet side of them, i.e. the exchange of data between different WSN and organizations.

Contiki uses two communication stacks uIP and RIME. uIP is a minimal RFC compliant implementation that has a 5kB footprint. It is important to remember that uIP has several limitations: no IP options, no sliding window, can handle only one network interface, uses a single buffer for both incoming and outgoing packets and does not buffer sent packets. A more detailed description of these can be found in [1]. Note however that this paper is a bit old and UDP is now implemented by Contiki uIP (contrary to what is written there). The other communication stack used by Contiki is RIME. Rime is a new lightweight communication stack designed for low-power radios. Rime provides a wide range of communication primitives, from best-effort local area broadcast, to reliable multi-hop bulk data flooding.

Another noticeable feature of Contiki is the ability to support multi-threading by implementing it as a library that can be optionally linked with programs that explicitly require it. This feature is important whenever a lengthy computation is performed, which given the event-driven kernel, will monopolize the CPU and will make the system unresponsive to external events.

Other differences between Contiki and the more popular embedded operating systems, like TinyOS include the code footprint and also in the underlying architecture, such as scheduling. While the native TCP/IP support offered by Contiki is nice, it requires more resources from the system (and might not be essential for all WSN deployments). Concerning scheduling, Contiki uses FIFO event queue and pool handlers with priority, as opposed to only a FIFO event queue, in TinyOS.

The ability to do simulations for WSNs is incredibly important for system development. Although a variety of simulators exist, they allow simulations only at a fixed level, such as application, operating system or hardware level. One more advantage of Contiki is that it has a specially designed cross-level simulator, named COOJA, which has been shown to perform well. This means that developers of Contiki based WSNs have a powerful tool on their hands for thoroughly testing different system ideas before they are to be implemented in practice, thus enabling them to save time and money.

One last advantage of Contiki is that once the porting for a platform is done, there are a number of very useful applications available, such as Telnet. These applications can come in very handy for developers. The Contiki repository is well maintained by the small community involved in this project and the feedback received so far by those who adopted this OS is positive.

3.3.1 General Contiki Program Structure

A Contiki program has a predefined structure. Simple instructions on how to program WSN nodes on the Contiki operating system platform is illustrated here. The most exciting thing we can find as a WSN programmer is the proto-threads in Contiki. Proto-threads allow us to write multi-threaded applications on top of the Contiki operating system. Therefore we can get rid of writing codes to implement state machines to run on the event driven kernel. The below program is the simplest Contiki program you can write which contains most of the necessary components that should be included in any Contiki program. The following example is the simple hello world novice program:

```
#include "contiki.h"
#include <stdio.h> /* For printf() */

PROCESS(hello_world_process, "Hello world process");
AUTOSTART_PROCESSES(&hello_world_process);

PROCESS_THREAD(hello_world_process, ev, data)
{
    PROCESS_BEGIN();

    printf("Hello, world\n");

    PROCESS_END();
}
```

The header file “contiki.h” which is included at the very first contains all the declarations of the Contiki operating systems abstractions. Stdio.h is included only because the used printf () function inside the program. The Macro in third line declares a new Contiki process. First parameter of it is the variable for the process and the second parameter is a string name for the process. Fourth line, specify that this process should be started in the

start-up of the Contiki operating system. Therefore, when the hardware device which will be the destination of compiled code switches on, our process also begins running.

Fifth line of the code opens the definition of our process. As the first parameter, we pass the process' variable which is `my_first_process` in this scenario. The second parameter is `'ev'` which is the event parameter. It can be used to make the program responding to events that occur in the system. The third parameter `'data'` can be used to receive some data which are coming with those events.

The rest of the instructions of the Contiki process will be included inside two important lines which are `PROCESS_BEGIN()` and `PROCESS_END()`. In this Contiki process only a string is printed. This output goes to the standard output in the platform where our code will be running. If you compile this code to native platform and run in the terminal, the output will be printed in the terminal. If you run this in a mote, the output most probably will go to the UART port of the mote. However this behavior depends on the design of a particular WSN mote.

Protothreads have a low memory overhead and are also stackless, i.e. they all run on the same stack and context switching is done by stack rewinding. Since processes in Contiki are nothing more than protothreads it makes sense to take a closer look at this abstraction. Protothreads are written in C and no compiler changes are required. The only problem with this abstraction is the following restrictions to the programmers: automatic variables not stored across a blocking wait and no switch statements are allowed. To overcome this one has to use static local variables and avoid switch statements completely below we show the simple implementation of protothreads:

```
#define PT_END (pt)
#define PT_WAIT UNTIL
#define PT_EXIT (pt)
#define PT_BEGIN (pt)
#define PT_INIT (pt)

struct pt {
    unsigned short lc ;
} ;
```

```

pt->lc = 0
switch (pt->lc) {
    case 0:
        pt->lc = 0;
        return 2
    case LINE:
        (pt, c) pt->lc = LINE;
        if (!(c))
            return 0;
}
pt->lc = 0 ;
return 1;

```

Analyzing this implementation it is easy to see why the restrictions mentioned above exist. As previously mentioned, Contiki processes are just protothreads. To better illustrate how this works we present the following piece of code which gets light sensor readings and prints it to the screen when the mote is connected to the computer via USB and a special program, `tunslip`, is running (it creates the `tun` and `SLIP` interface for communication over the serial line).

```

/ *declare the process */
PROCESS (light_process, " light_process " ) ;

/ *make the process start when the module is loaded */
AUTOSTART_PROCESS (&light_process);

```

```

/ *define the actual process code */
PROCESS_THREAD (light_process, ev, data)
{
    PROCESS_BEGIN ( ); / *must begin with this */

    / *note that we have to make the variable static */
    static struct etimer etimer ;

    / *Photosynthetically Active Radiation. */
    unsigned static reading1 ;

    / *Total Solar Radiation. */
    unsigned static reading2 ;
    printf ( " light process starting \n " ) ;

    / *initialize the light sensors */
    sensors_light_init ( ) ;

    while ( 1 ) { / *do this forever */
        / *set active timer */
        etimer_set (&etimer , CLOCK_SECOND) ;

        / *take readings at discrete time intervals */
        PROCESS_WAIT_UNTIL( etimer_expired (&etimer ) ) ;
        reading1 = sensors_light1 ( ) ;

        reading2 = sensors_light2 ( ) ;
        printf("READING1:%2d\nREADING2:%u\n",reading1 ,reading2);
    }
    PROCESS_END ( ); / * finish process * /
}

```

In the example above we also saw how to use timers. There are basically four types of timers that can be used depending on the needs of the programmer:

- struct timer : passive and only keeps track of its expiration time
- struct etimer: active and sends an event when it expires
- struct ctimer: active and calls a function when it expires
- struct rtimer: real-time timer and calls a function when it expires

There are two ways to make a process run. You can post an event and asynchronously run the process: `process post(process ptr, eventno, ptr)`, or run it immediately: `process post synch(process ptr, eventno, ptr)`. However if you post a process you should not call it from an interrupt. The other way to make a process run is to poll the process using: `process poll(process ptr)`.

3.4 Cooja

With the rapid increase in the amount of wireless sensor nodes and other wireless devices forming heterogeneous networks, it becomes unfeasible to test real setups using physical hardware. While one can test systems and protocols on an abstract level by simulating wireless phenomena, such simulation alone is insufficient because software can be prone to bugs and unexpected interrelations. Simulating complicated wireless setups using exactly the same firmware image that will later be used on real wireless nodes is therefore crucial.

The Cooja wireless simulator [6] is widely used, but the range of hardware it can emulate has been limited. The original approach, in which MSPSim [2] was tightly integrated into the Cooja source code, limited the ability of running unmodified firmware images to nodes based on either 16-bit TI MSP430 CPUs using MSPSim or 8-bit Atmel AVR CPUs using Avrora.

Taking into consideration the growing popularity of low-power 32-bit CPUs,

which are gaining a widespread adoption in the wireless sensor field [5], we need a way to integrate Cooja with an emulation framework that can support Cortex-M3. This was the reason for creating EmuLink, an abstract solution for interconnecting Cooja and hardware emulation software.

Although Cooja has proven to be useful for interoperability tests of heterogeneous software stacks before creating EmuLink [3, 4], the emulated hardware was limited to a single platform. By using EmuLink it becomes possible to take interoperability testing one step further and have both heterogeneous software and hardware.

3.4.1 About Cooja

COOJA is a flexible Java-based simulator designed for simulating networks of sensors running the Contiki operating system. COOJA simulates networks of sensor nodes where each node can be of a different type; differing not only in on-board software, but also in the simulated hardware. COOJA is flexible in that many parts of the simulator can be easily replaced or extended with additional functionality. Example parts that can be extended include the simulated radio medium, simulated node hardware, and plug-ins for simulated input/output.

A simulated node in COOJA has three basic properties: its data memory, the node type, and its hardware peripherals. The node type may be shared between several nodes and determines properties common to all these nodes. For example, nodes of the same type run the same program code on the same simulated hardware peripherals. And nodes of the same type are initialized with the same data memory. During execution, however, nodes' data memories will eventually differ due to e.g. different external inputs.

COOJA currently is able to execute Contiki programs in two different ways. Either by running the program code as compiled native code directly on the host CPU, or by running compiled program code in an instruction-level TI MSP430 emulator. COOJA is also able to simulate non-Contiki nodes, such as nodes implemented in Java or even nodes running another operating system. All different approaches have advantages as well as disadvantages. Javabased nodes enable much faster simulations but do not run deployable code. Hence, they are useful for the development of e.g. distributed algorithms. Emulating nodes provides more fine-grained execution details compared to Javabased nodes or nodes running native code. Finally, native code simulations are more efficient than node emulations and still simulate

deployable code. Since the need of abstraction in a heterogeneous simulated network may differ between the different simulated nodes, there are advantages in combining several different abstraction levels in one simulation. For example, in a large simulated network a few nodes may be simulated at the hardware level while the rest are implemented at the pure Java level. Using this approach combines the advantages of the different levels. The simulation is faster than when emulating all nodes, but at the same time enables a user to receive fine-grained execution details from the few emulated nodes.

COOJA executes native code by making Java Native Interface (JNI) calls from the Java environment to a compiled Contiki system. The Contiki system consists of the entire Contiki core, pre-selected user processes, and a set of special simulation glue drivers. This makes it possible to deploy and simulate the same code without any modifications, minimizing the delay between simulation and deployment.

The Java simulator has full control over the memory of simulated nodes. Hence the simulator may at all times view or change Contiki process variables, enabling very dynamic interaction possibilities from the simulator. Another interesting consequence of using JNI is the ability to debug Contiki code using any regular debugger, such as gdb, by attaching it to the entire Java simulator and breaking when the JNI call is performed. Also entire simulation states may be saved and later restored, skipping back simulations over time.

The hardware peripherals of simulated nodes are called interfaces, and enable the Java simulator to detect and trigger events such as incoming radio traffic or a LED being lit. Interfaces also represent properties of simulated nodes such as positions that the actual node is not aware of.

All interactions with simulations and simulated nodes are performed via plugins. An example of a plugin is a simulation control that enables a user to start or pause a simulation. Both interfaces and plugins can easily be added to the simulator, enabling users to quickly add custom functionality for specific simulations.

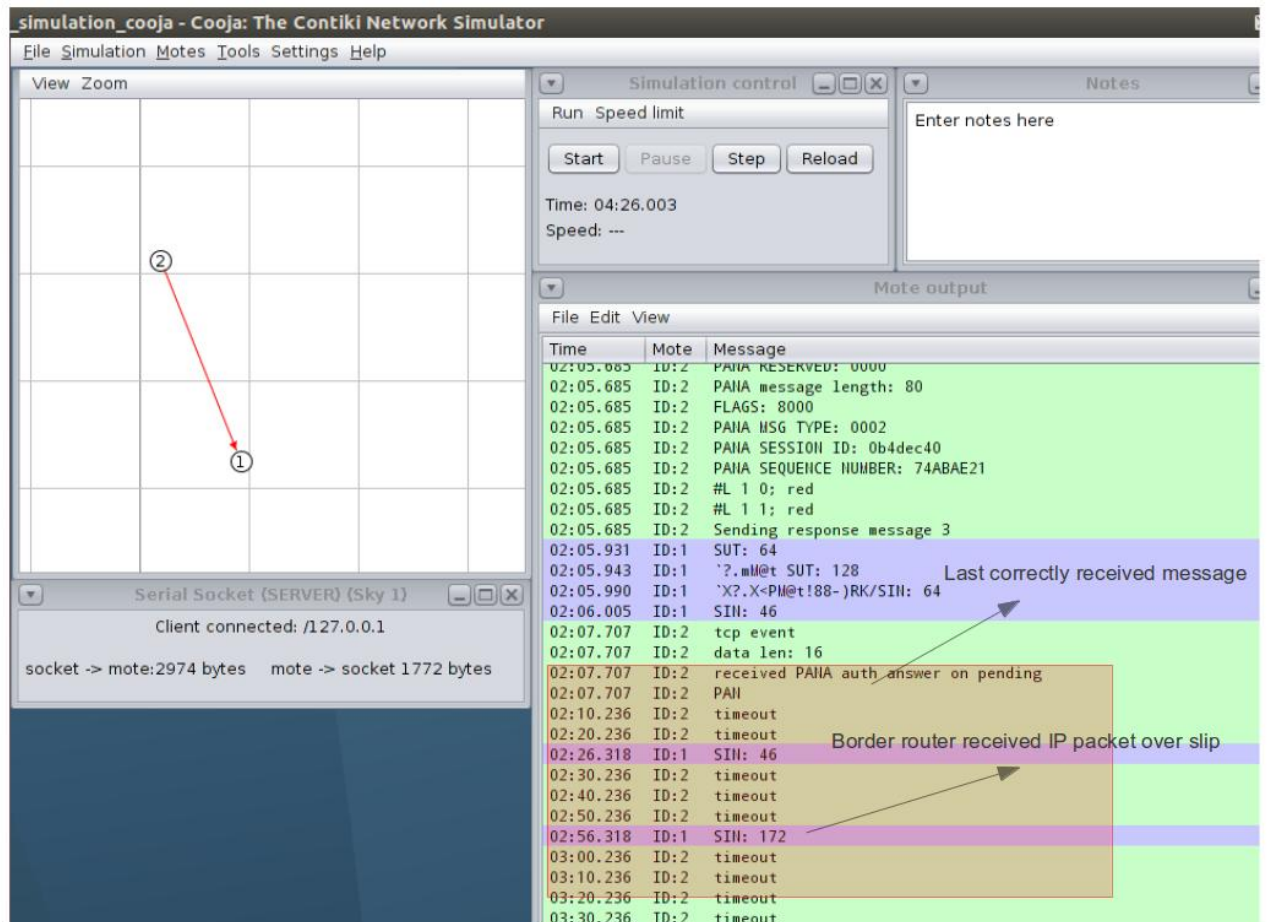


Figure 11, Screenshot while simulate Cooja with two motes

3.4.2 Starting Cooja

You can start the Cooja simulator with the following commands:

- `cd contiki -2.7/ tools/cooja`
- `and run`



Figure 12, Create new simulation interface

You will see lots of text go past as the Cooja environment is compiled, built and started

Firstly you will need to create a new simulation. To do this, click “File >New Simulation”. This will present a dialog box, shown in Figure 11. Give your simulation a more suitable title, and then select the radio medium that best suits your simulation type. For most simulations, Unit Disk Graph Medium (UDGM) is quite suitable. Turning radio simulation off by selecting “No Radio Traffic”, when not needed saves CPU time and makes the simulation run quicker. Random start-up delays the booting of each mote simulated, simulated randomly so they don’t all start exactly at the same time. The main random seed is a seed for the random number generator. You can tick the box at the end to get a random seed. When you’re ready click "Create". You will now see the Cooja desktop once more, but with the simulation environment presented, as shown in Figure 12.

The simulation interface, shown in Figure 12, consists of five windows. The Network window shows the physical layout of the network, i.e. you will be able to physically place motes here and move them around, as needed, in order to form the topology and layout you are interested in. The Simulation Control window lets you start, stop and reload the simulation. It also lets you control the rate at which the simulation proceeds. The Mote

Output window shows any serial output generated by all the motes, i.e. the output from the printf command. You may filter the output shown based on the string you enter into the “Filter” field. For example, if you wish to filter the output such that it only shows output from mote 2, then you can enter “ID:2” in this field. The Timeline window shows events that occur on each mote over the timeline of simulation. These events can be radio traffic, LED activity or anything else. The Notes window can be used to take temporary notes in the simulation.

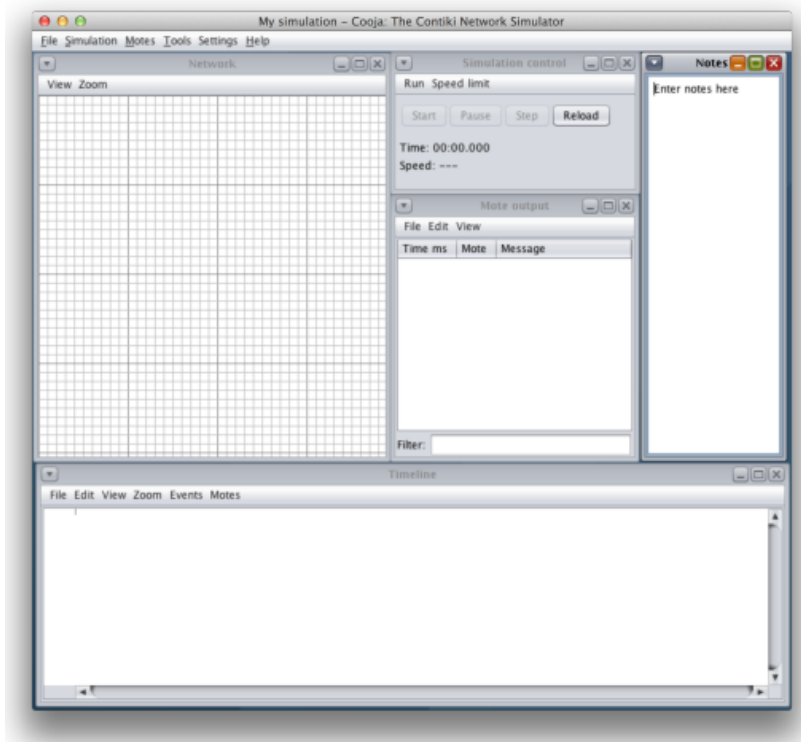


Figure 13, The simulation interface

The next stage is to set the mote types. In Cooja, you create a list of mote types, define their parameters and assign their program code/binary. You then, as a separate task, create instances of them to simulate. Here we address the first part: creating the mote types. Click “Motes > Add Motes > Create New Mote Type > Sky Mote”. Give the mote type a useful description. Let the first mote type be the border router, so we name this mote Border Router, as shown in Figure 13. In the “Contiki Process / Firmware” field, you should specify your source file (the .c file) or the binary file (the .sky file). If you specify the binary file, you

won't see any compile instructions (binary files are the result of compilation – there is no need to compile). If you specify the source code, then the compile commands field becomes active, and you can specify specific instructions for compilation. In this case, since we have already compiled our binaries before, we will just use the Browse button to select the border-router.sky file. If you have specified source code, you need to press Compile before you can create the mote type. You can see the compilation output/results in the compilation output tab. If successful, the Create button becomes available. You may now use this button to create a mote of the Border Router type.

Similarly, you may create the UDP Server mote by using the udp-server.

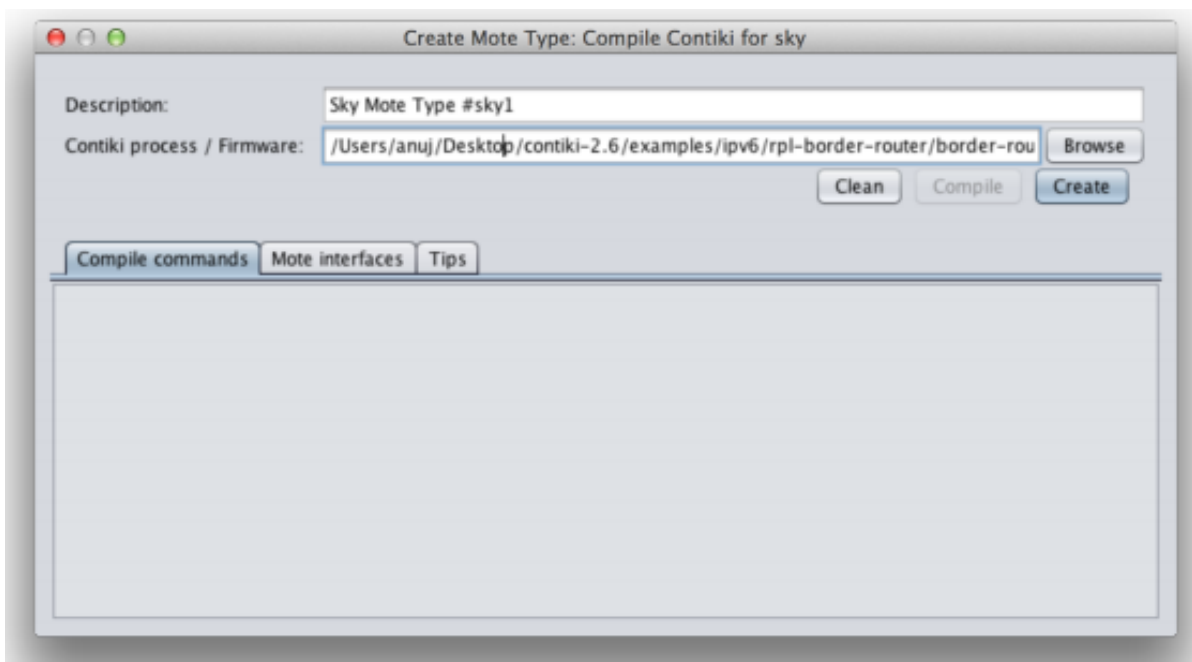


Figure 14, The create mote type interface

Now that you have created the Border Router and UDP Server type motes, we can start adding physical nodes to the simulation. The “Add motes” dialogue allows you to set the number and configuration of these new motes. This can be accessed by clicking “Motes > Add Motes”. Add one mote of the type Border Router and five mote of type UDP Server. You may use random positioning for adding the nodes. Once you do this, you will notice that a total of six randomly placed nodes will appear in the Network window. One possible random arrangement can be seen in Figure 14(a). Amongst these, node 1 is the Border Router and the rest are nodes which will execute the UDP Server code. You can interact with each of

the nodes by clicking and dragging them around. You may re-organize the layout of the nodes by dragging them around, such that they will form a topology that you are interested in. When you click on a node, it also shows the radio environment in green and grey color. The green circle represents the area within which the signal is successfully received by other nodes and grey represents the area where radio interference from the current node exists. An example of this radio environment and a new layout such that only a maximum of two nodes are within range of each node, can be seen in Figure 14(b).

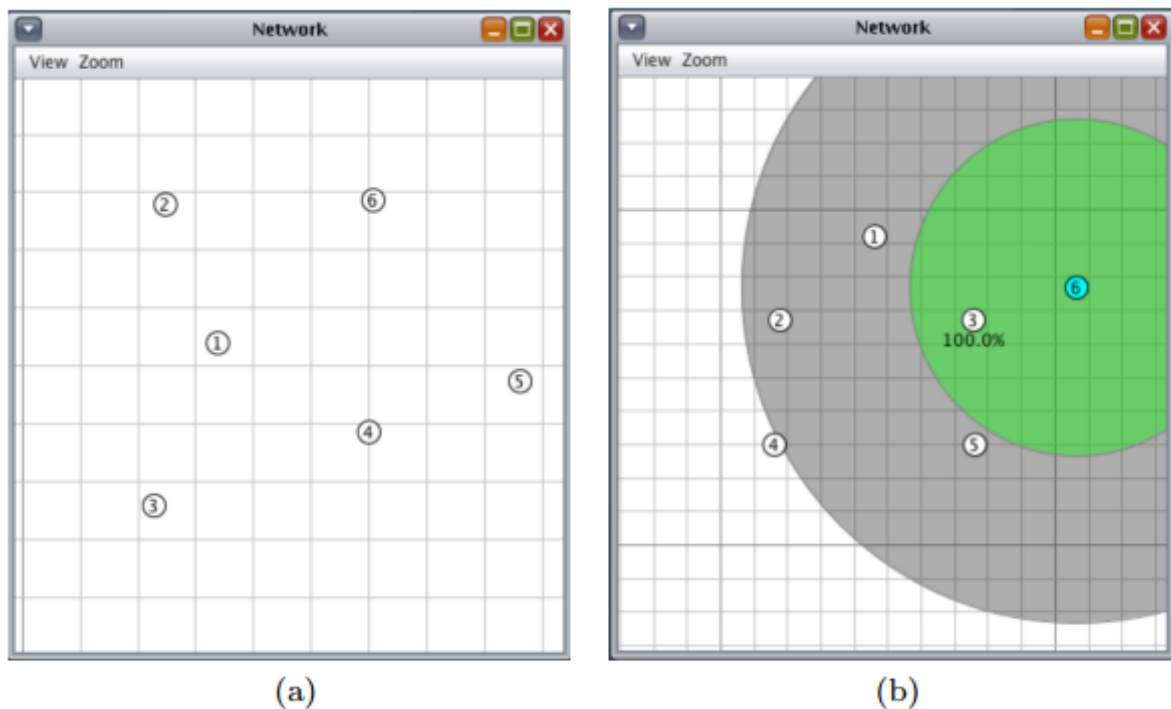


Figure 15, The network window

Even though the network layout is now setup, before starting the simulation a bridge with the RPL network must be created. Since node 1 is the Border Router, the bridge will be created with this node. Right click node 1 and then select “More tools for Border Router > Serial Socket (SERVER)”. This creates a serial port on the simulated node 1, which is accessible via UDP port number 60001 on the local machine.

You can now start the simulation by clicking the “Start” button in the Simulation Control window.

Now, in a terminal window, enter the following commands in order to setup the bridge:

- `cd contiki -2.6/ tools`
- `make tunslip6`
- `sudo ./tunslip6 -a 127.0.0.1 aaaa::1/64`

This will now setup a bridge into the RPL network, via node 1 of the simulation. The prefix of all nodes in the network will be `aaaa:: /64`. Once you enter this command, you will see output that looks similar to the following:

- `slip connected to "127.0.0.1:60001"`
- `opened tun device "/dev/tun0 "`
- `ifconfig tun0 inet 'hostname ' up`
- `ifconfig tun0 add aaaa::1/64`
- `ifconfig tun0 add fe80::0:0:0:1/64`
- `ifconfig tun0`
-
- `tun0 Link encap:UNSPEC Hwaddr`
- `00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00`
- `inet addr:127.0.1.1 P-t-P:127.0.1.1 Mask:255.255.255.255`
- `inet6 addr: fe80::1/64 Scope:Link`
- `inet6 addr: aaaa::1/64 Scope:Global`
- `*** Address:aaaa::1 => aaaa:0000:0000:0000`
- `Got configuration message of type P`
- `Setting prefix aaaa::`
- `Server IPv6 addresses:`
- `aaaa::212:7401:1:101`
- `fe80::212:7401:1:101`

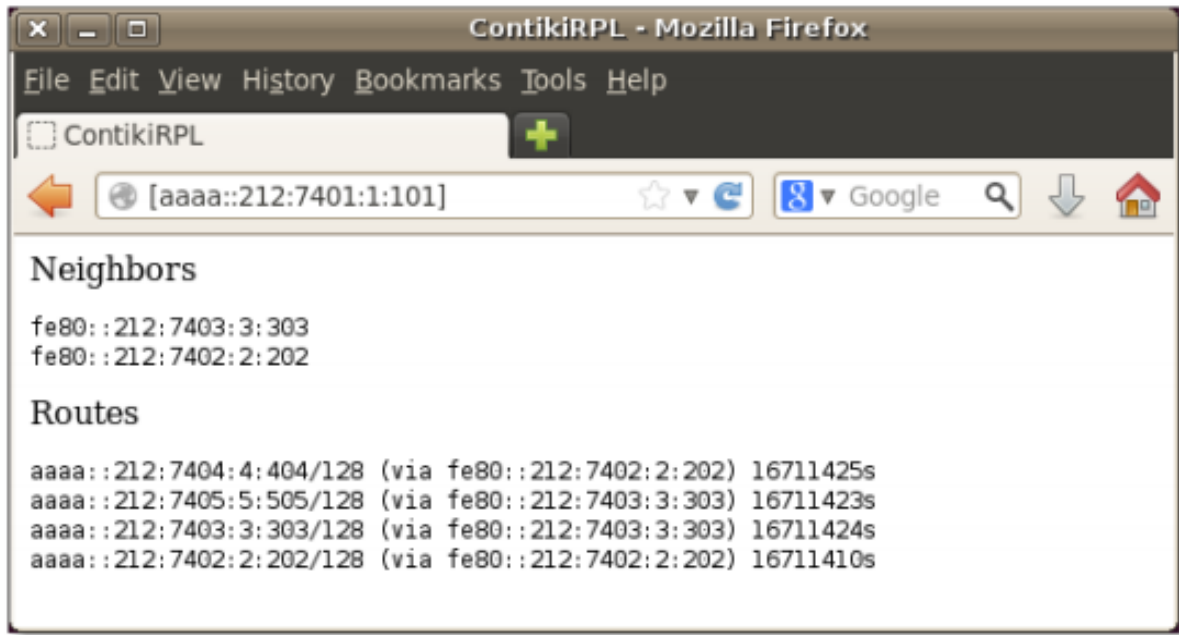


Figure 16, The RPL route status, as visible to the border router

This output confirms that the bridging is setup and that the border router has the address `aaaa::212:7401:1:101` setup. We can verify this by pinging this address using the `ping6` command. You can also see the current RPL network status by going to the web interface located at the border router, as shown in Figure 15. The IP address for node 2 is `aaaa::212:7402:2:202`. You now know how to work with the Contiki Cooja simulator.

3.4.3 Cooja Architecture/MSPSim/Prisma

Cooja is a cross-level network simulator implemented in Java. Cooja handles the simulation of radio mediums and communication between nodes. The simulated nodes are implemented using a plugin architecture enabling various types of nodes. Node types range from abstract nodes implemented in Java to nodes executing firmware images using cycle-accurate hardware emulators.

MSPSim is a Java-based emulator of the TI MSP430 microprocessor series. MSPSim is tightly integrated in Cooja and provides cycle-accurate emulation, which enables development of timing-sensitive applications in Cooja. Exactly the same firmware that runs on

a real sensor node can be loaded and executed in MSPSim, thereby providing a highly valuable development and debugging capability as well. Furthermore, we have earlier shown that Cooja/MSPSim enables accurate network-scale power profiling of sensor networks [2].

The Prisma Emulator is partly based on QEMU and supports emulation of a wide variety of ARM microprocessors. The Prisma Emulator is extensible through Python plug-ins that make it possible to easily set up emulation of specific boards and System on Chip. For the Cooja integration we implement emulation of the STM32W platform, which is a Cortex-M3 with an integrated IEEE 802.15.4 radio.

The Prisma Emulator is implemented partly in native code, partly for the .NET framework. It executes in the Mono platform, which provides a crossplatform .NET development framework [1]. The different run-time environment used by Prisma Emulator makes it difficult to integrate into the Java-based Cooja platform. Hence, this initial hurdle in creating heterogeneous simulation support for Cooja is what motivated us to create the EmuLink component to connect emulators implemented in any language.

EmuLink consists of an EmuLink component in Cooja that connects external emulators into Cooja and EmuLink-enabled emulators that act as emulation servers. When running a simulation, Cooja will schedule the emulators for execution and deliver radio messages as well as serial data to and from the nodes that execute in the external emulators. By using the concept of abstract emulation servers, EmuLink enables the use of emulators written in any programming language with Cooja and also makes it possible to distribute the hardware emulators on multiple computers to be able to scale the simulations to large networks.

4 Implementation

The first purpose of this chapter is to describe the implementation process, follow the developer's steps and expand the interesting parts of the source code. Various screenshots are showing basic steps and outputs of the program.

In the course of this project, a Server and a Client have been implemented for the Contiki Operating system. The implementation includes the support of GET operation and have been accomplished in the C programming language.

Section 4.1 describes the implementation design principles. In Section 4.2 and overview of the Server is presented. Section 4.3 provides an overview of the Client.

4.1 Design Principles

The main principle is to have a small program to satisfy the low memory constraints (10 kilobytes of RAM and 30 kilobytes of ROM). Besides that a Server-Client model needs to be followed in order to implement this project on IoT.

In Computer science server-client is a software architecture model consisting of two parts, client systems and server systems, both communicate over a computer network or on the same computer. A client-server application is a distributed system consisting of both client and server software. The client process always initiates a connection to the server, while the server process always waits for requests from any client. Especially, a TCP/IP protocol is followed in this project.

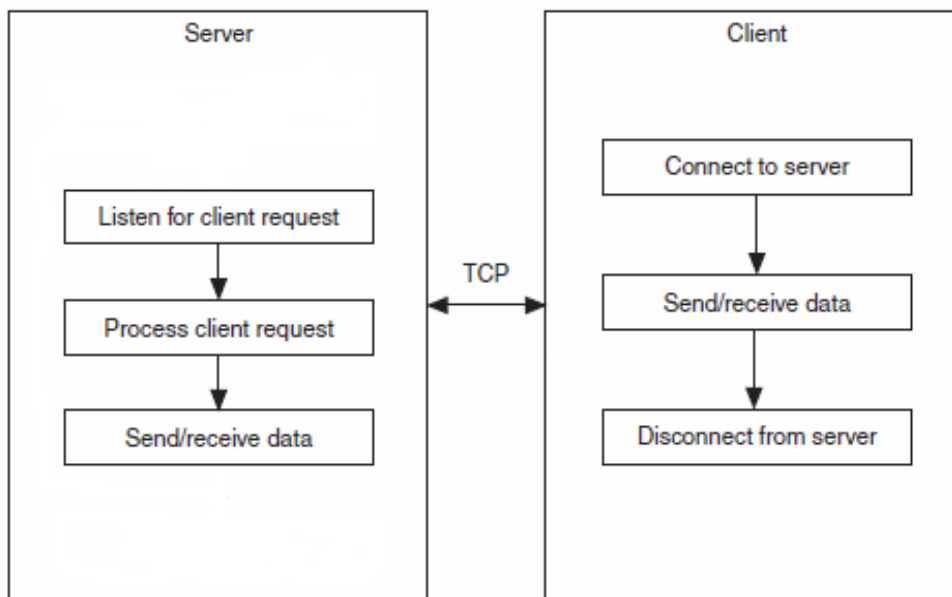


Figure 17, Server-Client Module

4.2 TCP/IP

The Internet protocol suite is the computer networking model and communications protocols used in the Internet and similar computer networks. It is commonly known as TCP/IP, because its most important protocols, the Transmission Control Protocol (TCP) and the Internet Protocol (IP), were the first networking protocols defined in this standard.

TCP/IP provides end-to-end connectivity specifying how data should be packetized, addressed, transmitted, routed and received at the destination. This functionality is organized into four abstraction layers which are used to sort all related protocols according to the scope of networking involved. From lowest to highest, the layers are the link layer, containing communication technologies for a single network segment (link), the internet layer, connecting hosts across independent networks, thus establishing internetworking, the transport layer handling host-to-host communication, and the application layer, which provides process-to-process application data exchange.

The TCP/IP model and related protocols are maintained by the Internet Engineering Task Force (IETF).

4.5.1 TCP 3-Way Handshake/Protocols

The TCP three-way handshake in Transmission Control Protocol (also called the TCP-handshake; three message handshake and/or SYN-SYN-ACK) is the method used by TCP set up a TCP/IP connection over an Internet Protocol based network. TCP's three way handshaking technique is often referred to as "SYN-SYN-ACK" (or more accurately SYN, SYN-ACK, ACK) because there are three messages transmitted by TCP to negotiate and start a TCP session between two computers. The TCP handshaking mechanism is designed so that two computers attempting to communicate can negotiate the parameters of the network TCP socket connection before transmitting data such as SSH and HTTP web browser requests.

This 3-way handshake process is also designed so that both ends can initiate and negotiate separate TCP socket connections at the same time. Being able to negotiate multiple TCP socket connections in both directions at the same time allows a single physical network interface, such as ethernet, to be multiplexed to transfer multiple streams of TCP data simultaneously.

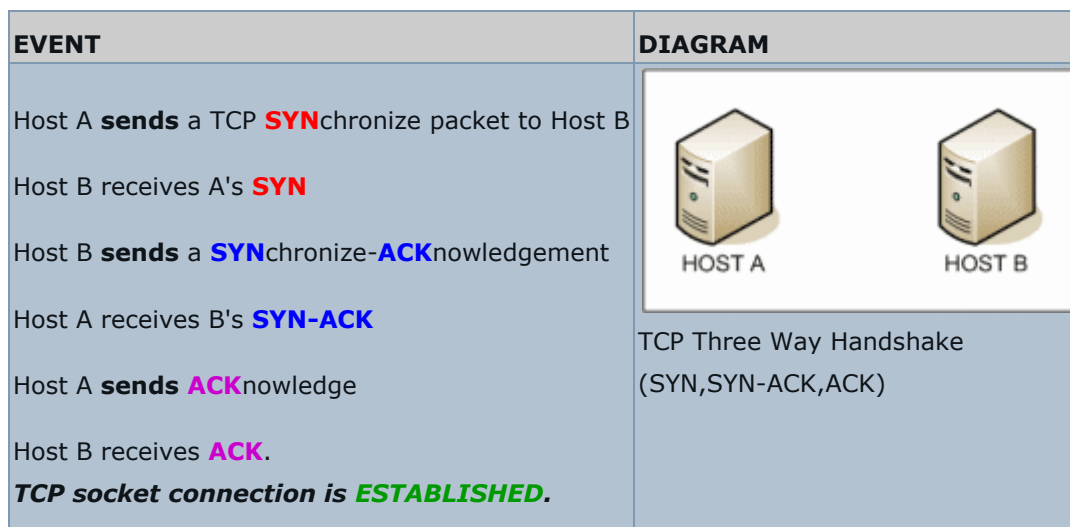


Figure 18, This is a (very) simplified diagram of the TCP 3-way handshake process

Have a look at the diagram on the right as you examine the list of events on the left.

SYNchronize and ACKnowledge messages are indicated by either the SYN bit, or the ACK bit inside the TCP header, and the SYN-ACK message has both the SYN and the ACK bits turned on (set to 1) in the TCP header.

TCP knows whether the network TCP socket connection is opening, synchronizing, established by using the SYNchronize and ACKnowledge messages when establishing a network TCP socket connection.

When the communication between two computers ends, another 3-way communication is performed to tear down the TCP socket connection. This setup and teardown of a TCP socket connection is part of what qualifies TCP a reliable protocol. TCP also acknowledges that data is successfully received and guarantees the data is reassembled in the correct order.

Note that UDP is connectionless. That means UDP doesn't establish connections as TCP does, so UDP does not perform this 3-way handshake and for this reason, it is referred to as an unreliable protocol. That doesn't mean UDP can't transfer data, it just doesn't negotiate how the connection will work, UDP just transmits and hopes for the best.

Note also that FTP, Telnet, HTTP, HTTPS, SMTP, POP3, IMAP, SSH and any other protocol that rides over TCP also has a three way handshake performed as connection is opened. HTTP web requests, SMTP emails, FTP file transfers all manage the messages they each send. TCP handles the transmission of those messages. TCP 'rides' on top of Internet Protocol (IP) in the protocol stack, which is why the combined pair of Internet protocols is called TCP/IP (TCP over IP). TCP segments are passed inside the payload section of the IP packets. IP handles IP addressing and routing and gets the packets from one place to another, but TCP manages the actual communication sockets between endpoints (computers at either end of the network or internet connection).

4.6 Create Resource

In order to create a Resource, a standart format is followed. A Resource is defined by the RESOURCE macro. The attributes of a Resource are: resource name, the RESTful methods it handles, and its URI path (omitting the leading slash). Besides the definition a function is needed to implement a Resource.

A handler function named [resource name]_handler must be implemented for each Resource. A buffer for the response payload is provided through the buffer pointer. Simple resources can ignore preferred_size and offset, but must respect the REST_MAX_CHUNK_SIZE limit for the buffer. If a smaller block size is requested for CoAP, the REST framework automatically splits the data. For example:

```
RESOURCE(helloworld, METHOD_GET, "hello", "title=\"Hello world:
?len=0..\";rt=\"Text\"");
```

```
void helloworld_handler(void* request, void* response, uint8_t
*buffer, uint16_t preferred_size, int32_t *offset)
{
    const char *len = NULL;
    char const * const message = "Hello World ABCDEFGHIJKL
MNOPQRSTUVWXYZab cdefghijklmnopqrstuvwxyz";
    int length = 12;
    if (REST.get_query_variable(request, "len", &len)){
        length = atoi(len);
        if (length<0)
            length = 0;
        if(length>REST_MAX_CHUNK_SIZE)
            length = REST_MAX_CHUNK_SIZE;
        memcpy(buffer, message, length);
    }
}
```

```

else {
    memcpy(buffer, message, length);
}
REST.set_header_content_type(response, REST.type.TEXT_PLAIN);
REST.set_header_etag(response, (uint8_t *) &length, 1);
REST.set_response_payload(response, buffer, length);
}

```

A “helloworld” resource is implemented. Although the variable message is assigned a big string, because the length variable is set to 12, only the first 12 characters of the message are copied to the buffer. Finally, the message is send throught the last instructions of the handler.

The query string can be retrieved by `rest_get_query()` or parsed for its key-value pairs. Also, at `set_header_content_type`, `text/plain` is the default, hence this option could be omitted.

4.6.1 Create Periodic Resource

In order to create a Periodic Resource, a format similar to the simple Resource is followed. A Periodic Resource is defined by the `PERIODIC_RESOURCE` macro. The attributes of a Periodic Resource are: resource name, the RESTful methods it handles, and its URI path (omitting the leading slash), as seen at the simple Resource. Furthermore, it takes an additional period parameter, which defines the interval to call `[name]_periodic_handler()`.

A default `post_handler` takes care of subscriptions by managing a list of subscribers to notify. Again, a handler function named `[resource name]_handler` must be implemented for each Resource. A buffer for the response payload is provided through the buffer pointer. Simple resources can ignore `preferred_size` and `offset`, but must respect the `REST_MAX_CHUNK_SIZE` limit for the buffer. If a smaller block size is requested for CoAP, the REST framework automatically splits the data.

Additionally, a handler function named [resource name]_handler must be implemented for each PERIODIC_RESOURCE and it will be called by the REST manager process with the defined period.

For example:

```
PERIODIC_RESOURCE(pushing,METHOD_GET, "test/push", "title=\"Periodic demo\";obs", 5*CLOCK_SECOND);
```

```
void pushing_handler(void* request, void* response, uint8_t *buffer, uint16_t preferred_size, int32_t *offset)
```

```
{
    REST.set_header_content_type(response, REST.type.TEXT_PLAIN);
    const char *msg = "It's periodic!";
    REST.set_response_payload(response, msg, strlen(msg));
}
```

```
void pushing_periodic_handler(resource_t *r){
    static uint16_t obs_counter = 0;
    static char content[11];
    ++obs_counter;
    PRINTF("TICK %u for %s\n", obs_counter, r->url);
    coap_packet_t notification[1];
    coap_init_message(notification,COAP_TYPE_NON,REST.status.OK,0);
    coap_set_payload(notification, content, sprintf(content, sizeof(content), "TICK %u", obs_counter));
    REST.notify_subscribers(r, obs_counter, notification);
}
```

In this example, the period is set to 5*CLOCK_SECOND. A CLOCK_SECOND equals second/10 so the period is not 5 seconds but 5/10 seconds. Usually, a CoAP server would response with the resource representation matching the periodic_handler, thus the msg is set to “It's periodic!”. Also a post_handler that handles subscriptions will be called for periodic resources by the REST framework.

The lines:

```
coap_packet_t notification[1];
coap_init_message(notification, COAP_TYPE_NON, REST.status.OK, 0 );
coap_set_payload(notification, content, sprintf(content, sizeof(content), "TICK %u",
obs_counter));
```

Are used to build a notification and:

```
REST.notify_subscribers(r, obs_counter, notification);
```

Is to notify the registered observers with the given message type, observe option, and payload.

4.6.2 Create Event Resource

In order to create an Event Resource, a format similar to the Periodic Resource is followed. An Event Resource is defined by the `EVENT_RESOURCE` macro. The attributes of an Event Resource are: resource name, the RESTful methods it handles, and its URI path (omitting the leading slash), as seen at the simple Resource.

A default `post_handler` takes care of subscriptions by managing a list of subscribers to notify. Again, a handler function named `[resource name]_handler` must be implemented for each Resource. A buffer for the response payload is provided through the buffer pointer. Simple resources can ignore `preferred_size` and `offset`, but must respect the `REST_MAX_CHUNK_SIZE` limit for the buffer. If a smaller block size is requested for CoAP, the REST framework automatically splits the data.

Additionally, a handler function named `[resource name]_handler` must be implemented for each `EVENT_RESOURCE` and it will be called by the REST manager process with the defined period.

For example:


```

EVENT_RESOURCE(event, METHOD_GET, "sensors/button", "title=\"Event
demo\";obs");
void event_handler(void* request, void* response, uint8_t *buffer,
uint16_t preferred_size, int32_t *offset)
{
    REST.set_header_content_type(response, REST.type.TEXT_PLAIN);
    const char *msg = "It's eventful!";
    REST.set_response_payload(response, (uint8_t *)msg,
    strlen(msg));
}

void event_event_handler(resource_t *r)
{
    static uint16_t event_counter = 0
    static char content[12];
    ++event_counter;

    PRINTF("TICK %u for /%s\n", event_counter, r->url);
    coap_packet_t notification[1];
    coap_init_message(notification, COAP_TYPE_CON, REST.status.OK, 0);
    coap_set_payload(notification, content, sprintf(content,
    sizeof(content), "EVENT %u", event_counter));
    REST.notify_subscribers(r, event_counter, notification);
}

```

4.7 Program Parameters

In this chapter we quote our RESTful example. We create three resources: humidity, temperature and led status. Each resource has two servers which interact with a CoAP client.

A prerequisite to run correctly the program is to define the Cooja addresses in /etc/hosts file that we can find in contiki root folder.

These addresses are :

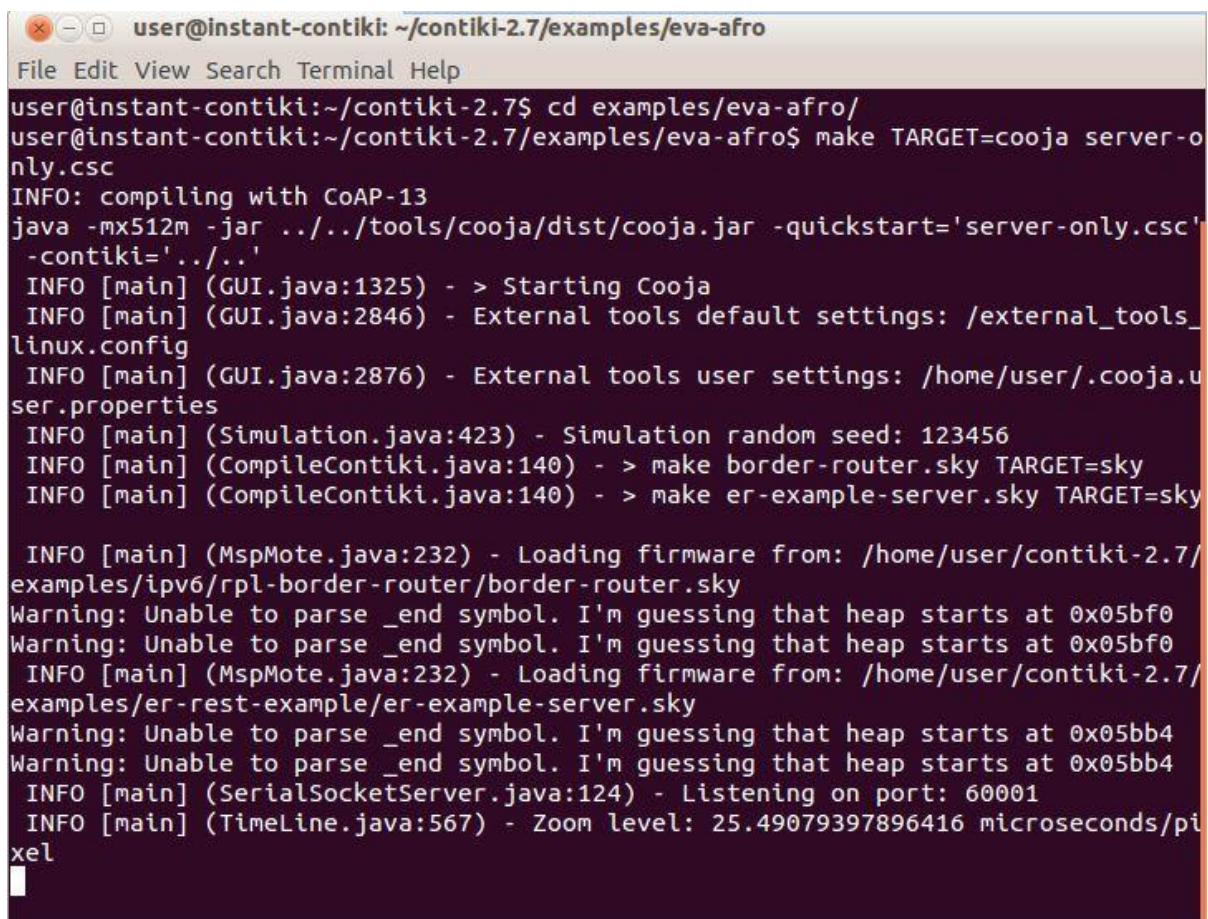
```
aaaa::0212:7401:0001:0101 cooja1
```

```
aaaa::0212:7402:0002:0202 cooja2
```

```
aaaa::0212:7403:0003:0303 cooja3
```

First we have to compile the application. So we open an terminal console and type:

```
make TARGET=cooja server-only.csc
```



```
user@instant-contiki: ~/contiki-2.7/examples/eva-afro
File Edit View Search Terminal Help
user@instant-contiki:~/contiki-2.7$ cd examples/eva-afro/
user@instant-contiki:~/contiki-2.7/examples/eva-afro$ make TARGET=cooja server-only.csc
INFO: compiling with CoAP-13
java -mx512m -jar ../../tools/cooja/dist/cooja.jar -quickstart='server-only.csc'
-contiki='../../..'
INFO [main] (GUI.java:1325) - > Starting Cooja
INFO [main] (GUI.java:2846) - External tools default settings: /external_tools_linux.config
INFO [main] (GUI.java:2876) - External tools user settings: /home/user/.cooja.user.properties
INFO [main] (Simulation.java:423) - Simulation random seed: 123456
INFO [main] (CompileContiki.java:140) - > make border-router.sky TARGET=sky
INFO [main] (CompileContiki.java:140) - > make er-example-server.sky TARGET=sky

INFO [main] (MspMote.java:232) - Loading firmware from: /home/user/contiki-2.7/examples/ipv6/rpl-border-router/border-router.sky
Warning: Unable to parse _end symbol. I'm guessing that heap starts at 0x05bf0
Warning: Unable to parse _end symbol. I'm guessing that heap starts at 0x05bf0
INFO [main] (MspMote.java:232) - Loading firmware from: /home/user/contiki-2.7/examples/er-rest-example/er-example-server.sky
Warning: Unable to parse _end symbol. I'm guessing that heap starts at 0x05bb4
Warning: Unable to parse _end symbol. I'm guessing that heap starts at 0x05bb4
INFO [main] (SerialSocketServer.java:124) - Listening on port: 60001
INFO [main] (TimeLine.java:567) - Zoom level: 25.49079397896416 microseconds/pixel
```

Figure 19, Screenshot from the console after we typed the make command. The program starts to run

In order to add motes servers we have to add also the corresponding address in the /etc/hosts file.

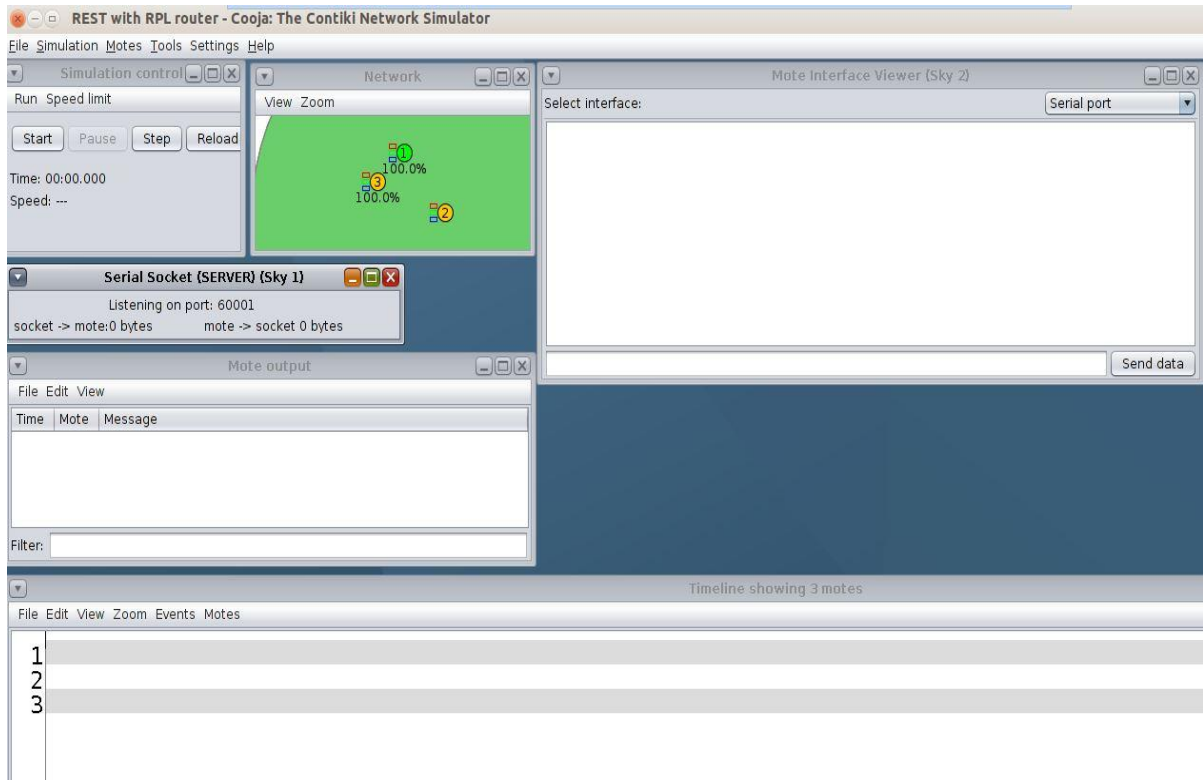


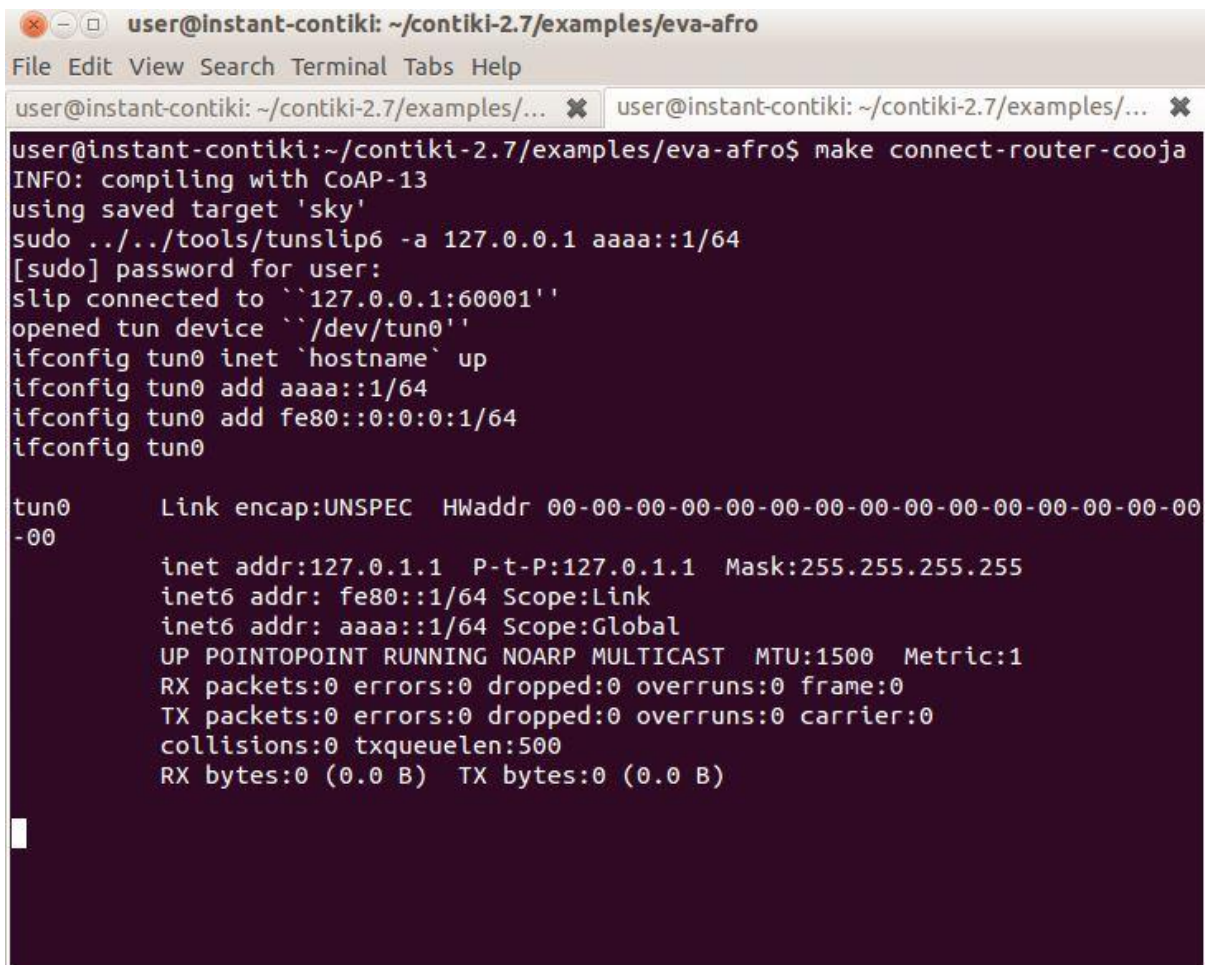
Figure 20, Screenshot after we run Cooja

As we can see in Network window we have 2 erbium servers the motes 2 and 3.

Before we start Cooja we to type another one make command to connect the router and the cooja.

So we open a new terminal and type:

```
make connect-router-cooja
```



```
user@instant-contiki: ~/contiki-2.7/examples/eva-afro
File Edit View Search Terminal Tabs Help
user@instant-contiki: ~/contiki-2.7/examples/... user@instant-contiki: ~/contiki-2.7/examples/...
user@instant-contiki:~/contiki-2.7/examples/eva-afro$ make connect-router-cooja
INFO: compiling with CoAP-13
using saved target 'sky'
sudo ../../tools/tunslip6 -a 127.0.0.1 aaaa::1/64
[sudo] password for user:
slip connected to ``127.0.0.1:60001''
opened tun device ``/dev/tun0''
ifconfig tun0 inet `hostname` up
ifconfig tun0 add aaaa::1/64
ifconfig tun0 add fe80::0:0:0:0/64
ifconfig tun0

tun0      Link encap:UNSPEC  HWaddr 00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00
-00
          inet addr:127.0.1.1  P-t-P:127.0.1.1  Mask:255.255.255.255
          inet6 addr: fe80::1/64 Scope:Link
          inet6 addr: aaaa::1/64 Scope:Global
          UP POINTOPOINT RUNNING NOARP MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:500
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
```

Figure 21, Screenshot after the second command

4.7.1 Humidity

Humidity is a periodic resource and is defined by the RESOURCE macro:

```
PERIODIC_RESOURCE(humidity, METHOD_GET, "humidity", "title=\"Hello  
humidity: ?len=0..\";rt=\"Text\"", 30*CLOCK_SECOND);
```

The first parameter (humidity) is the resource name, the second (METHOD_GET) is the RESTful method it handles, the third ("humidity") is a string name for this resource, the fourth ("title=\"Hello humidity: ?len=0..\";rt=\"Text\"") is its URI path and the last one (30*CLOCK_SECOND) is the period time which defines the dead time between two responses.

The default handler for this resource and its implement:

```
void  
humidity_handler (void* request, void* response, uint8_t  
*buffer, uint16_t preferred_size, int32_t *offset)  
{  
    REST.set_header_content_type(response,  
    REST.type.TEXT_PLAIN);  
  
    const char *msg = "Observe Periodic Humidity!";  
  
    REST.set_response_payload(response, (uint8_t*)msg, strlen(  
    msg));  
}
```

This handler will print the string "Observe Periodic Humidity!" if we choose the method get.

The first line is a function call which sets the type of the response parameter as a text message.

The second line creates a const char and initializes it with a string.

The third line in the handler's implantation is a function call which sets the payload that will be printed.

The `REST.set_header_content_type()` and the `REST.set_response_payload()` are both functions which have been defined and implemented in the `rest.h` and `rest.c` files which we must include.

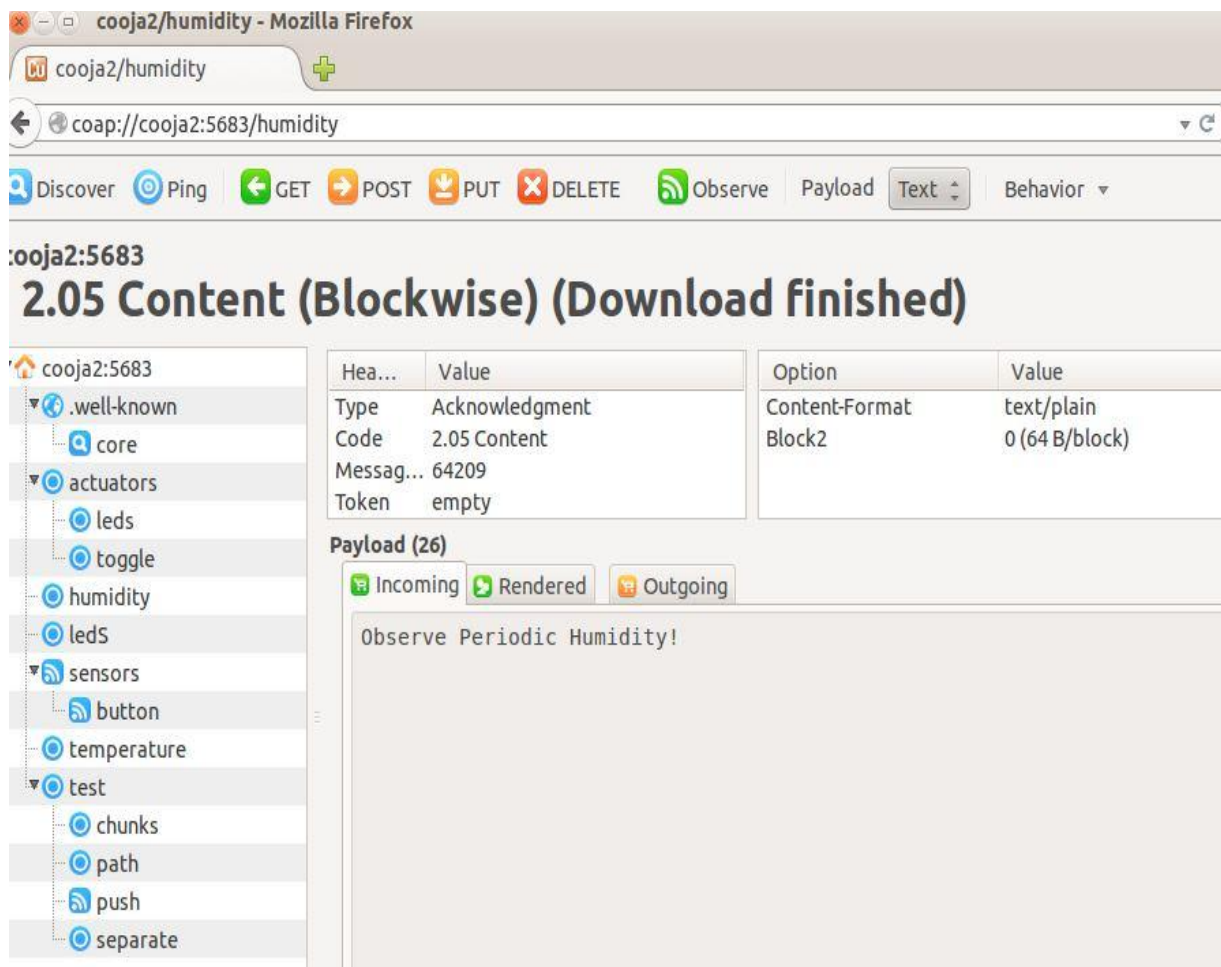


Figure 23, Screenshot while we choose the GET method of the humidity resource in COAP

The payload appears in the incoming.

In order to make this resource periodic we have to create another handler function which will be called by the REST manager process with the defined period:

```

void
humidity_periodic_handler (resource_t *r)
{
    static uint16_t humidityVal = 0;
    static char content[30];

    humidityVal=(uint16_t)rand()%100;

    coap_packet_t notification[1];
    coap_init_message(notification,COAP_TYPE_NON,REST.st
atus.OK, 0 );
    coap_set_payload(notification,content,snprintf(conte
nt,sizeof(content),"Humidity: %u %%", humidityVal));

    REST.notify_subscribers(r,humidityVal, otification);
}

```

In this handler we create a humidity variable which takes random values.

Also we need a char table that we will use as a payload. The following lines needed to build a notification so the packet can be treated as a usual pointer. So we need the `coap_init_message()` call and the `coap_set_payload()` which defines the format and merges the message that we want to be printed.

These two functions are defined and implemented in the `coap-server.h` and `coap-server.c` which we have to include. The function `REST.notify.subscribers()` is required to notify the registered observers with the given message type, the periodic value and the payload .

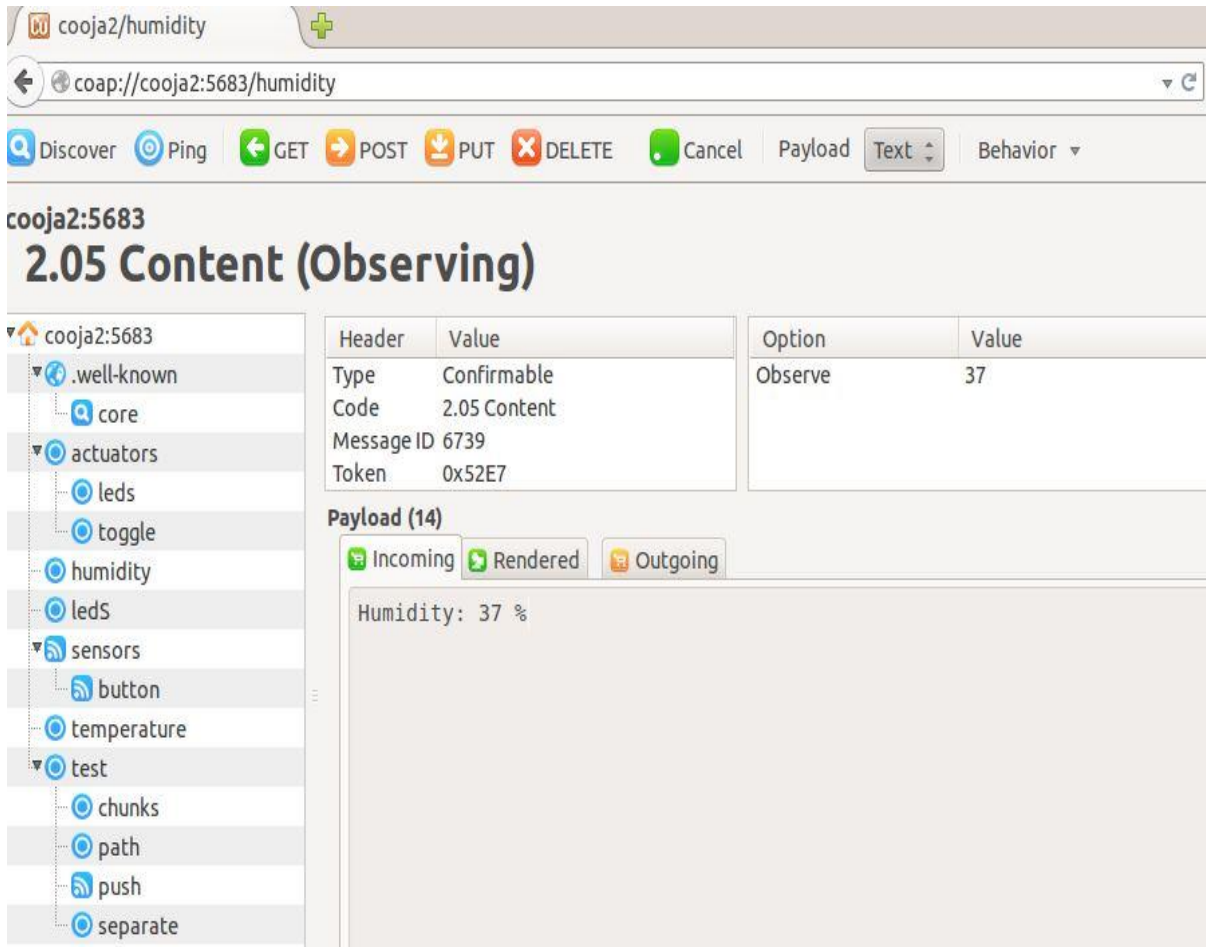


Figure 24, Screenshot while observe the humidity resource

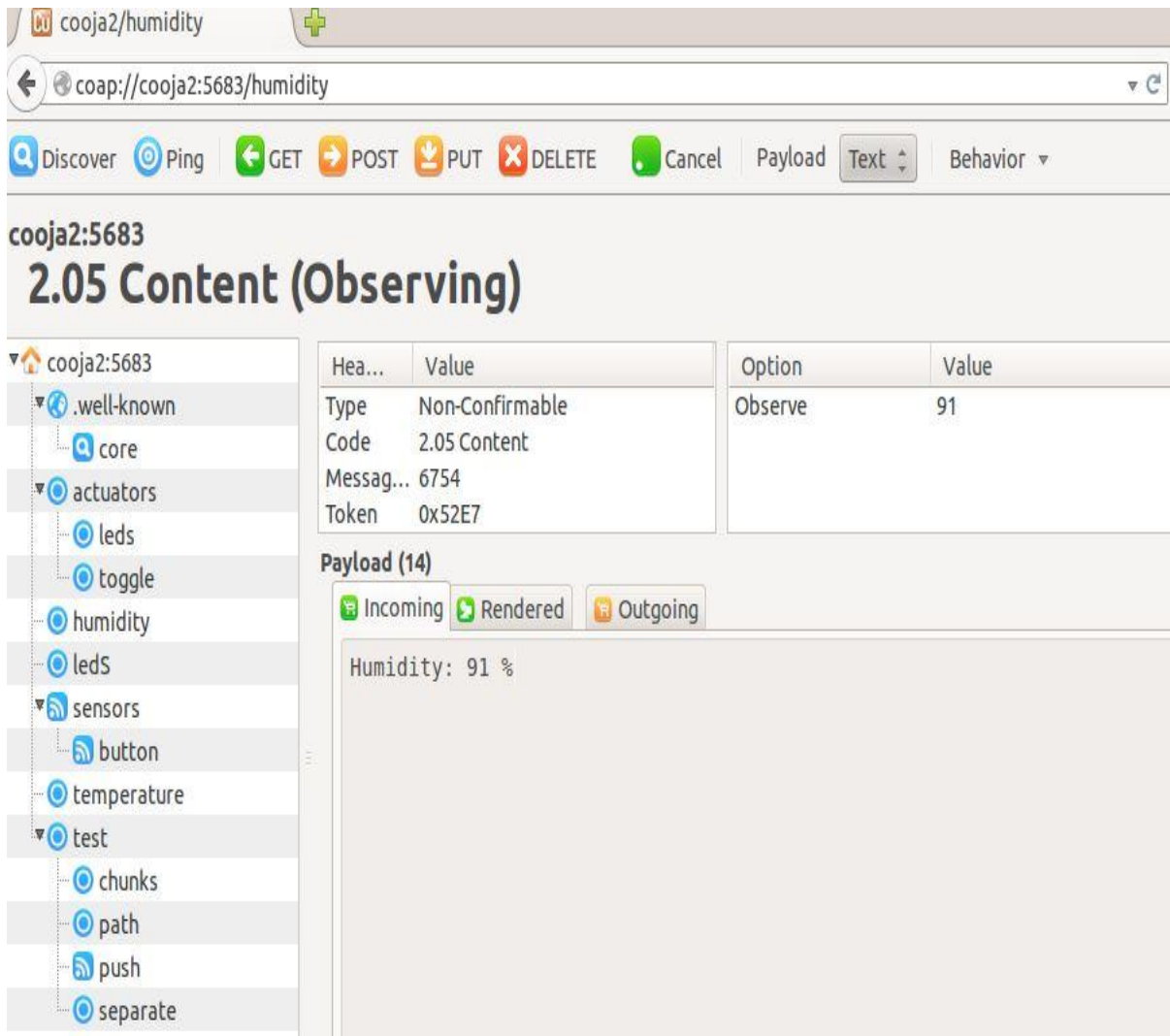


Figure 25, Screenshot while observing the same resource some periods after

The message ID now is 6754 . So, 15 messages after the first screenshot.

4.7.2 Temperature

Temperature is a periodic resource and is defined by the RESOURCE macro:

```
PERIODIC_RESOURCE(temperature,METHOD_GET,"temperature","title=\
>Hello temperature: ?len=0..\";rt=\"Text\"", 30*CLOCK_SECOND);
```

The first parameter (temperature) is the resource name, the second (METHOD_GET) is the RESTful method it handles, the third ("temperature") is a string name for this resource, the fourth ("title=\"Hello temperature: ?len=0..\";rt=\"Text\"") is its URI path and the last one (30*CLOCK_SECOND) is the period time which defines the dead time between two responses.

The default handler for this resource and its implement:

```
void
temperature_handler (void* request, void* response, uint8_t *buffer,
uint16_t preferred_size, int32_t *offset)
{
    REST.set_header_content_type(response,REST.type.TEXT_PLAIN);

    const char *msg = "Observe Periodic Temperature!";

    REST.set_response_payload(response,(uint8_t*)msg,strlen(msg));
}
```

This handler will print the string "Observe Periodic Temperature!" if we choose the method get.

The first line is a function call which sets the type of the response parameter as a text message.

The second line creates a const char and initializes it with a string.

The third line in the handler's implantation is a function call which sets the payload that will be printed.

The REST.set_header_content_type() and the REST.set_response_payload() are both functions which have been defined and implemented in the rest.h and rest.c files which we must include.

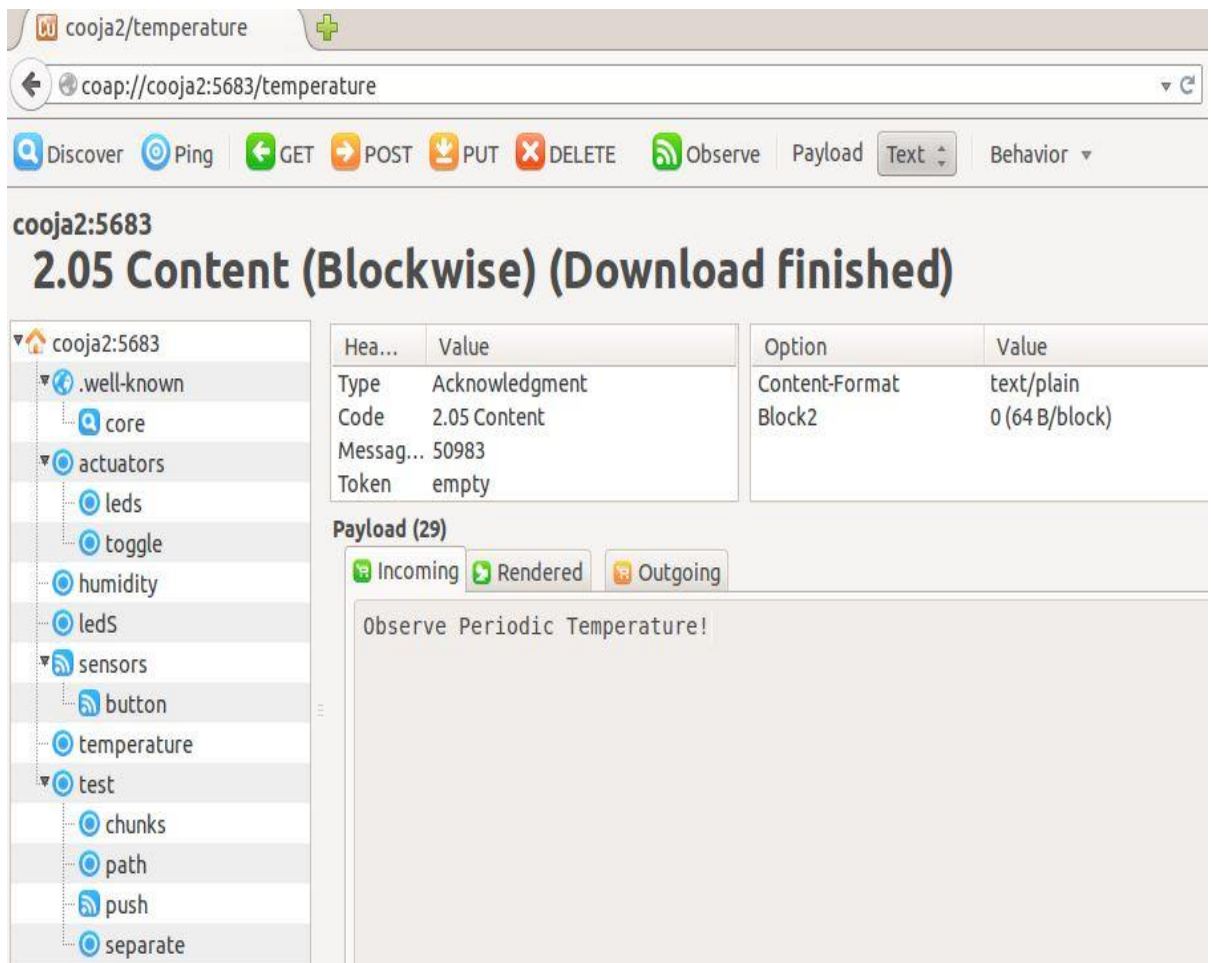


Figure 26, Screenshot while we choose the GET method of the temperature resource in COAP

The payload appears in the incoming.

In order to make this resource periodic we have to create another handler function which will be called by the REST manager process with the defined period:

```
void
temperature_periodic_handler (resource_t *r)
{
    static uint16_t temperatureVal = 0;
    static char content[30];

    temperatureVal=(uint16_t)rand()%100;

    coap_packet_t notification[1];

    coap_init_message(notification,COAP_TYPE_NON,REST.status.OK,0);
    coap_set_payload(notification,content,snprintf(content,sizeof(c
ontent), "Temperature: %u F", temperatureVal));

    REST.notify_subscribers(r, temperatureVal, notification);
}
```

In this handler we create a temperature variable which takes random values.

Also we need a char table that we will use as a payload.

The following lines needed to build a notification so the packet can be treated as a usual pointer. So we need the `coap_init_message()` call and the `coap_set_payload()` which defines the format and merges the message that we want to be printed.

These two functions are defined and implemented in the `coap-server.h` and `coap-server.c` which we have to include.

The function `REST.notify.subscribers()` is required to notify the registered observers with the given message type, the periodic value and the payload .

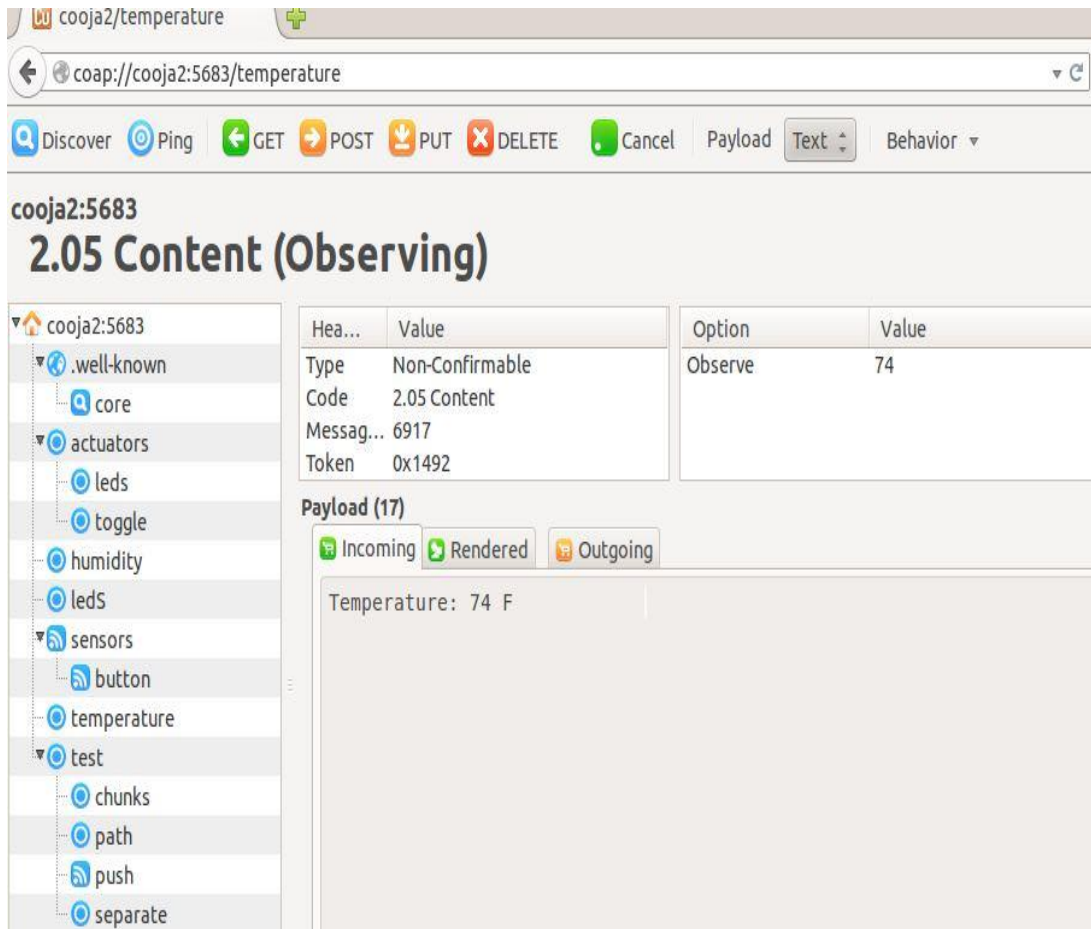


Figure 27, Screenshot while observe the temperature resource

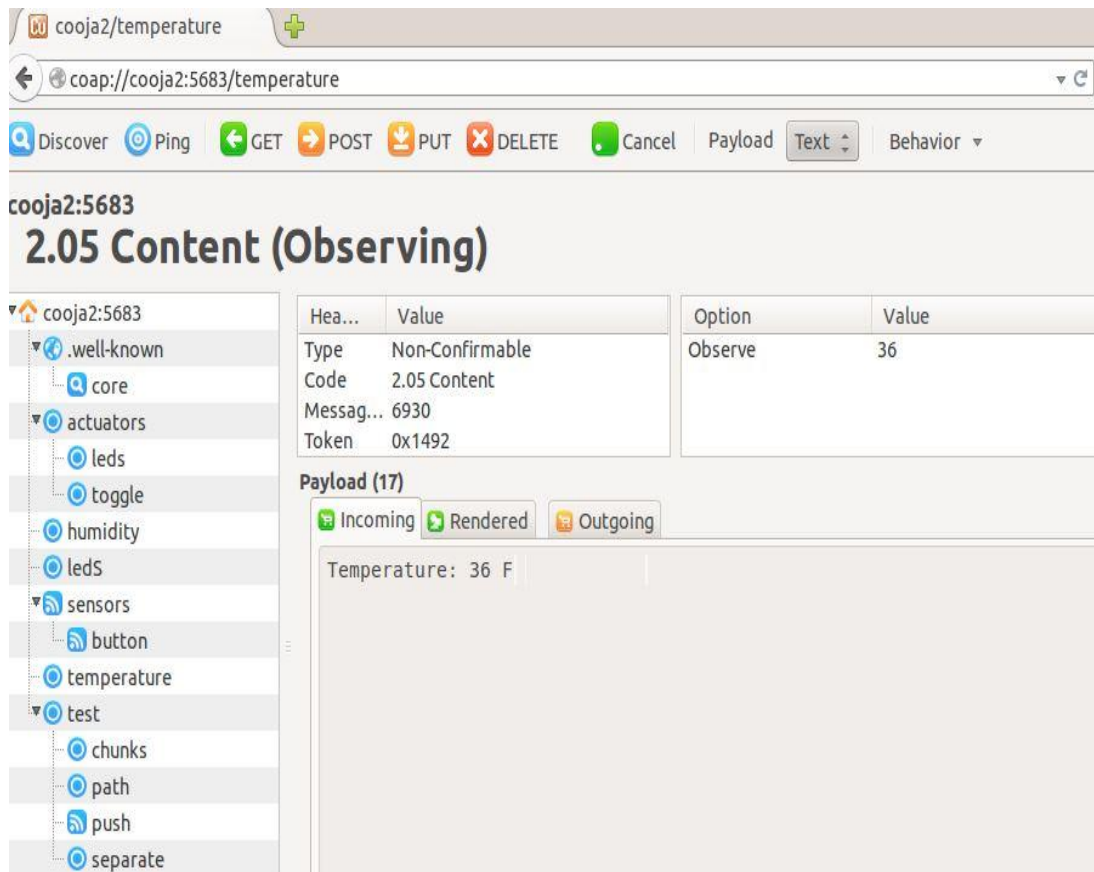


Figure 28, Screenshot while observing the same resource some periods after

The message ID now is 6930. So, 13 messages after the first screenshot.

4.7.3 LED status

Led status is a periodic resource and is defined by the RESOURCE macro:

```
PERIODIC_RESOURCE (ledS, METHOD_GET, "ledS", "title=\"Hello  
ledS: ?len=0..\";rt=\"Text\"", 30*CLOCK_SECOND);
```

The first parameter (ledS) is the resource name, the second (METHOD_GET) is the RESTful method it handles, the third ("ledS ") is a string name for this resource, the fourth ("title=\"Hello ledS: ?len=0..\";rt=\"Text\"") is its URI path and the last one (30*CLOCK_SECOND) is the period time which defines the dead time between two responses.

The default handler for this resource and its implement:

```
void  
ledS _handler (void* request, void* response, uint8_t *buffer,  
uint16_t preferred_size, int32_t *offset)  
{  
    REST.set_header_content_type(response, REST.type.TEXT_PLAIN);  
  
    const char *msg = "LedS!";  
  
    REST.set_response_payload(response, (uint8_t*)msg, strlen(msg));  
}
```

This handler will print the string "LedS!" if we choose the method get.

The first line is a function call which sets the type of the response parameter as a text message.

The second line creates a const char and initializes it with a string.

The third line in the handler's implementation is a function call which sets the payload that will be printed.

The `REST.set_header_content_type()` and the `REST.set_response_payload()` are both functions which have been defined and implemented in the `rest.h` and `rest.c` files which we must include.

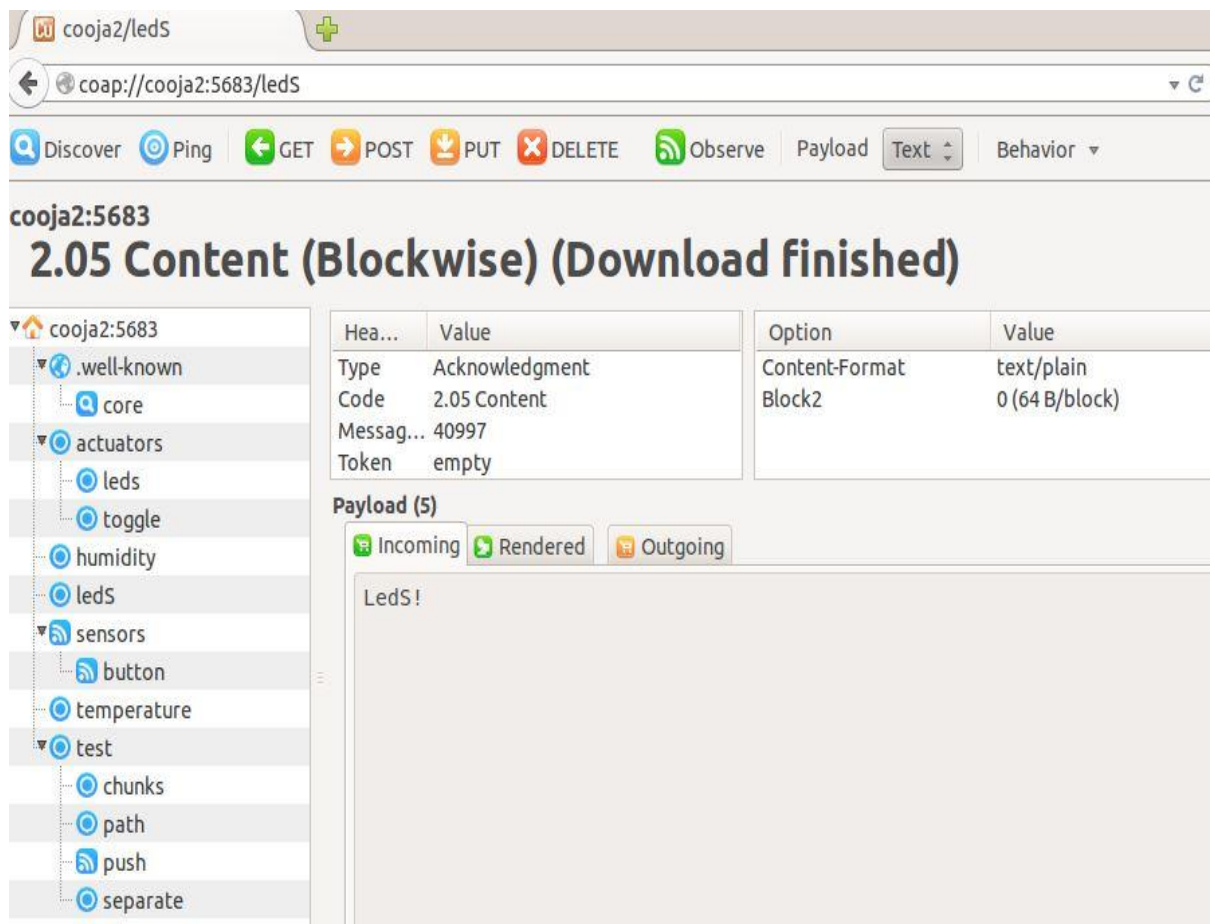


Figure 29, Screenshot while we choose the GET method of the LedS resource in COAP

The payload appears in the incoming.

In order to make this resource periodic we have to create another handler function which will be called by the REST manager process with the defined period:

```
void
temperature_periodic_handler (resource_t *r)
{
    static uint16_t ledSVal = 0;
    static char content[30];

    ledSVal=(uint16_t)rand()%120;

    coap_packet_t notification[1];

    coap_init_message(notification,COAP_TYPE_NON,REST.status.OK,0);

    coap_set_payload(notification,content,snprintf(content,sizeof(c
ontent), "LedS: %u ", ledSVal));

    REST.notify_subscribers(r, ledSVal, notification);
}
```

In this handler we create a ledS variable which takes random values.

Also we need a char table that we will use as a payload.

The following lines needed to build a notification so the packet can be treated as a usual pointer. So we need the `coap_init_message()` call and the `coap_set_payload()` which defines the format and merges the message that we want to be printed.

These two functions are defined and implemented in the `coap-server.h` and `coap-server.c` which we have to include.

The function `REST.notify.subscribers()` is required to notify the registered observers with the given message type, the periodic value and the payload .

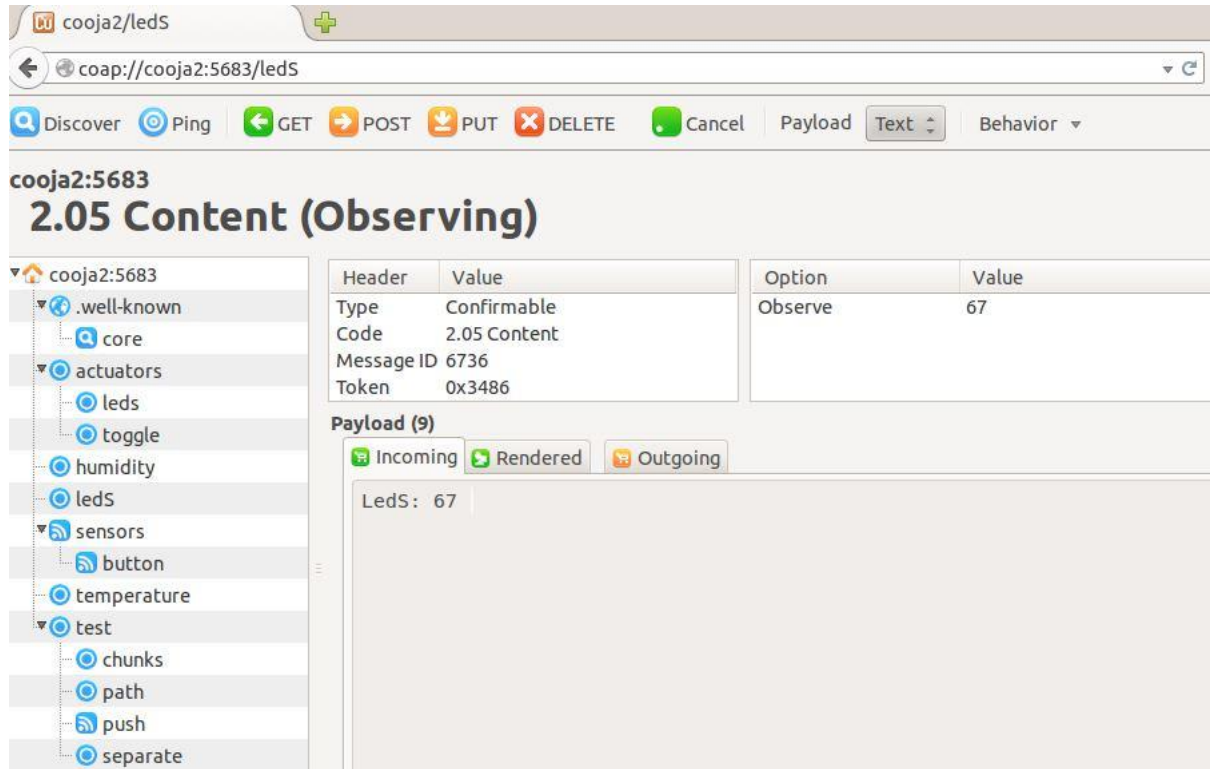


Figure 30, Screenshot while observe the LedS resource

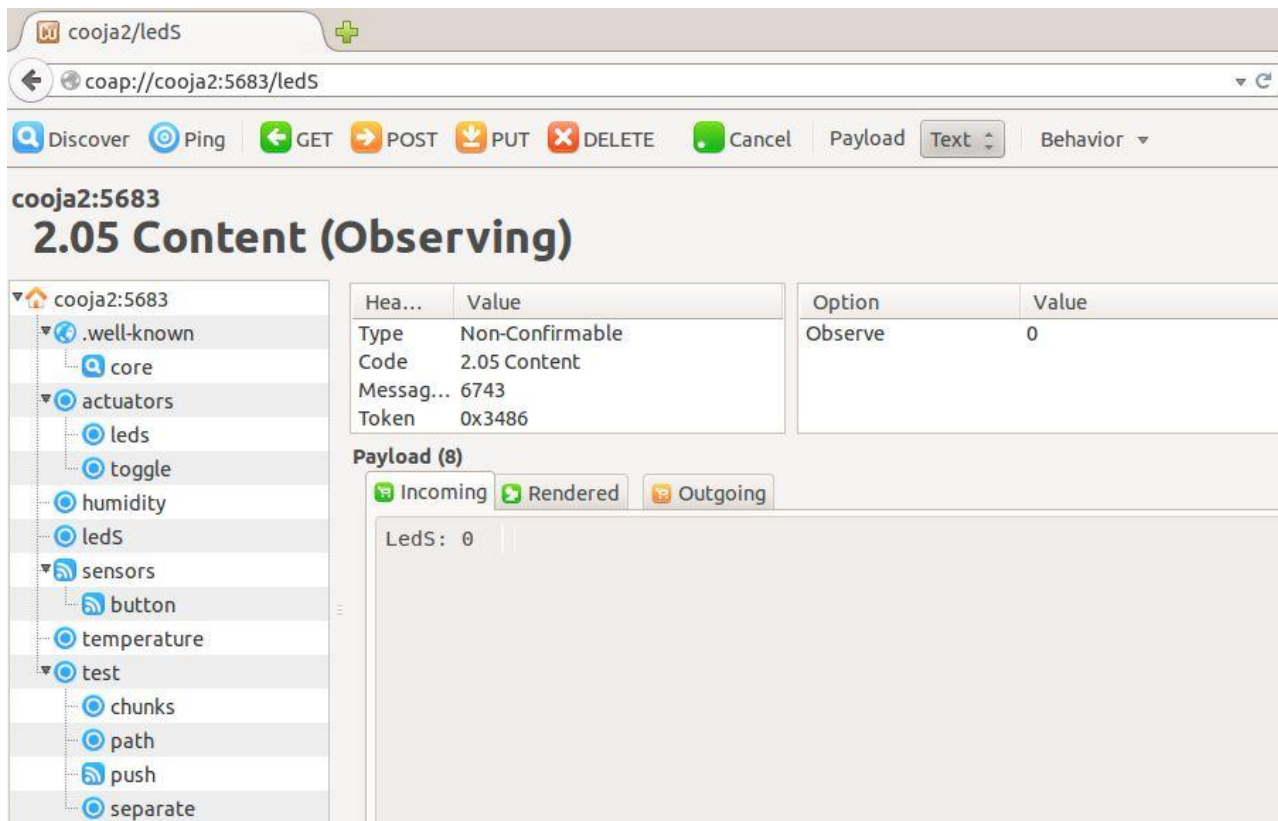


Figure 31, Screenshot while observing the same resource some periods after

The message ID now is 6743. So, 7 messages after the first screenshot.

4.8 Server

This chapter explains the implementation of a Server in our REST example. This is a RESTful server showing how we used the REST layer to develop this server-side application.

These lines are both in client and server and are libraries that are included.

```
> #include "contiki.h"
> #include "contiki-net.h"
```

Specifically, "contiki.h" is a sum of various libraries listed below:

```
> #include "contiki-version.h"
> #include "contiki-conf.h"
> #include "contiki-default-conf.h"
> #include "sys/process.h"
> #include "sys/autostart.h"
> #include "sys/timer.h"
> #include "sys/ctimer.h"
> #include "sys/etimer.h"
> #include "sys/rtimer.h"
> #include "sys/pt.h"
> #include "sys/procinit.h"
> #include "sys/loader.h"
> #include "sys/clock.h"
> #include "sys/energest.h"
```

The first three lines define the version of Contiki in a string and configure some types. For example:

```
> typedef uint8_t  u8_t;
> typedef uint16_t u16_t;
```

```
> typedef uint32_t u32_t;
> typedef int32_t s32_t;
> typedef unsigned short uip_stats_t;
```

The rest of the “contiki.h” library sets core features of Contiki. For example, implements kernel, timer, necessary features of process hierarchy model and other system core features.

These lines of code define which resources to include to meet memory constraints:

```
> #define REST_RES_HELLO 0
> #define REST_RES_CHUNKS 1
> #define REST_RES_SEPARATE 1
> #define REST_RES_PUSHING 1
> #define REST_RES_EVENT 1
> #define REST_RES_SUB 1
> #define REST_RES_LEDS 0
> #define REST_RES_TOGGLE 1
> #define REST_RES_LIGHT 0
> #define REST_RES_BATTERY 0
> #define REST_RES_RADIO 0
> #define REST_RES_MIRROR 0
```

The lines above, include chunks, separate, pushing, event, sub and toggle resources and skip the rest.

The line “#include "erbiun.h"” includes and implements the Erbium engine. Erbium is a low-power REST engine.

This block of code is adding the correct library for according to the use of the program:

```
> #if defined (PLATFORM_HAS_BUTTON)
> #include "dev/button-sensor.h"
> #endif
> #if defined (PLATFORM_HAS_LEDS)
> #include "dev/leds.h"
> #endif
> #if defined (PLATFORM_HAS_LIGHT)
> #include "dev/light-sensor.h"
> #endif
> #if defined (PLATFORM_HAS_BATTERY)
> #include "dev/battery-sensor.h"
> #endif
> #if defined (PLATFORM_HAS_SHT11)
> #include "dev/sht11-sensor.h"
> #endif
> #if defined (PLATFORM_HAS_RADIO)
> #include "dev/radio-sensor.h"
> #endif
```

For example, if PLATFORM_HAS_LEDS then „dev/leds.h“ library is included. That protects the code from over sizing and having more than one occurrences of the same code.

The rest of the server's source code is the code that implements the resources (previous chapters) and the auto start process that runs the program (appendix).

4.9 Client

The first phase of our project was to develop a REST server in one node and the client in another node so the client periodically accesses resources of server and prints the payload. The protocol that we follow is CoAP and the libraries below are the necessities in order to include all the needed structs.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "contiki.h"
#include "contiki-net.h"
#include "rest.h"
#include "buffer.h"

#define SERVER_NODE(ipaddr)  uip_ip6addr(ipaddr, 0xfe80, 0,
0, 0, 0x0212, 0x7401, 0x0001, 0x0101)
#define LOCAL_PORT 61617
#define REMOTE_PORT 61616
```

We have to inform client about the number of services, or resources, that we implement:

```
#define NUMBER_OF_URLS 3
char*service_urls[NUMBER_OF_URLS]={"led_status","temperature","humidity"}
```

Client now is ready to response to server's requests and for that we have to add three functions to handle the responses, to send data when we have a request and one function to

handle incoming data formalize the packet tha we recivied. These functions are defined and implement in Appentix section.

5 Problems encountered and Future directions

This thesis takes an exploratory approach to the Web integration of smart things and experiments with a simple implementation. Rather than focusing on one particular problem we looked at the bigger picture of this integration and tried to understand and experience its implications. As a consequence, the thesis provides a holistic view of this emerging domain but also emphasizes on several challenges instrumental to the realization of the Internet-of-Things.

5.5 Problems encountered

Not significant obstacles were faced through this project. The main problem was to familiarize ourselves with the new technologies and understand the concept of machine-to-machine internet. Studying the libraries and understand the flow of the code (functions inheritance) was the first and most difficult step to make. The fact that many libraries with even more functions were calling each other and via versa led us to fully understand the source of Contiki OS and furthermore make the rest of the project flow without unfortunate events.

5.6 Open challenges

Some open challenges are discussed based on the IoT elements presented earlier. The challenges include IoT specific challenges such as privacy, participatory sensing, data analytics, GIS based visualization and Cloud computing apart from the standard WSN challenges including architecture, energy efficiency, security, protocols, and Quality of Service. The end goal is to have Plug n' Play smart objects which can be deployed in any environment with an interoperable backbone allowing them to blend with other smart objects

around them. Standardization of frequency bands and protocols plays a pivotal role in accomplishing this goal. The section ends with a few international initiatives in the domain which could play a vital role in the success of this rapidly emerging technology.

Furthermore, although this thesis illustrates the suitability of Web standards and protocols for communicating real-world objects it also reveals their shortcomings. HTTP was designed as an architecture where clients initiate interactions and this model works fine for control-oriented IoT applications. However, monitoring-oriented applications are often event-based and thus smart things should also be able to push data to clients (rather than being continuously polled). Using syndication protocols improves the model for monitoring applications, since devices can publish asynchronously data on an intermediate server.

In this thesis particular care was given to experiment with Contiki OS and implement a core code for simple sensors. However these prototypes were not tested in real sensors, but only as virtual ones. More generally, there is a significant lack of large-scale real-world deployment of these sensors. However, the vision behind IoT is to implement a global world of smart things. Hence, future work should also focus on larger deployments of the developed concepts and technologies that will certainly raise challenging issues but also perhaps make an even stronger point for using Web standards. Efforts should also be made to bring these technologies closer to real-world use-cases and to the business. Towards this aim we open-sourced all of our software that was presented in this thesis hoping to help third-parties to implement their particular use-cases.

6 Summary and Conclusions

The general field of this thesis is the Internet of things technologies and some particular systems that included in the wireless sensor networks. These technologies are open to discover and this is something attract to someone who search for something new and revolutionary in the internet area.

In this paper also, we have presented COOJA, a cross-level simulator for the Contiki operating system. COOJA enables simultaneous simulations at the network, operating system and machine code instruction set level. We have shown that cross-level simulation has advantages in terms of effectiveness and memory usage. It allows a user to combine simulated nodes from several different abstraction levels. This is especially useful in heterogeneous networks where fine-grained execution details are only needed for a subset of the simulated nodes.

Additionally, this document gives an introduction to Contiki OS and provides the basics for someone who wants to use this operating system.

EmuLink enables heterogeneous simulations in Cooja. This is shown using the Prisma Emulator and connecting it to Cooja with EmuLink. With EmuLink Cooja becomes a platform for interoperability testing on multiple hardware and software platforms. With the availability of Prisma Emulator, Cooja also becomes a simulation framework for the 32-bit Cortex-M3.

A limitation with the current implementation is that Prisma Emulator is driven by the system clock and runs in real time, thereby requiring the simulations to also run in real time. Another effect of this design is that the simulations become non-deterministic. We plan to improve this by driving the EmuLink-connected emulators with Cooja's simulation clock to enable the simulations to run at arbitrary speeds.

7 References

- [1]K. Ashton, That “Internet of Things” thing, RFID Journal (2009).
- [2]J. Buckley (Ed.), The Internet of Things: From RFID to the Next-Generation Pervasive Networked Systems, Auerbach Publications, New York, 2006.
- [3] M. Weiser, R. Gold The origins of ubiquitous computing research at PARC in the late 1980s IBM Systems Journal (1999)
- [4]Y. Rogers, Moving on from Weiser’s vision of calm computing: engaging ubicomp experiences, in: UbiComp 2006: Ubiquitous Computing, 2006.
- [5]I.F. Akyildiz, W. Su, Y. Sankarasubramaniam, E. Cayirci Wireless sensor networks: a survey Computer Networks, 38 (2002), pp. 393–422
- [6]L. Atzori, A. Iera, G. Morabito The Internet of Things: a survey Computer Networks, 54 (2010), pp. 2787–2805
- [7]J. Belissent, Getting clever about smart cities: new opportunities require new business models, Forrester Research, 2010.
- [8]A. Juels RFID security and privacy: a research survey IEEE Journal on Selected Areas in Communications, 24 (2006), pp. 381–394
- [9]A. Ghosh, S.K. Das Coverage and connectivity issues in wireless sensor networks: a survey Pervasive and Mobile Computing, 4 (2008), pp. 303–334
- [10]M. Zorzi, A. Gluhak, S. Lange, A. Bassi From today’s Intranet of Things to a future Internet of Things: a wireless- and mobility-related view IEEE Wireless Communications, 17 (2010), pp. 43–51
- [11]N. Honle, U.P. Kappeler, D. Nicklas, T. Schwarz, M. Grossmann, Benefits of integrating meta data into a context model, 2005, pp. 25–29.
- [12]-Z. Shelby. Embedded web services. IEEE Wireless Communications, 17(6):52–57, December 2010.
- [13]-IPSO alliance: Enabling the internet of things. <http://ipso-alliance.org/>.
- [14]-The contiki operating system - instant contiki. <http://www.sics.se/contiki/instant-contiki.html>.
- [15]-Internet of things - ThingSpeak. <https://www.thingspeak.com/>.
- [16]URIs, URLs, and URNs: clarifications and recommendations 1.0. <http://www.w3.org/TR/uri-clarification/#contemporary>

8 Appendix

Source code from the server file:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "contiki.h"
#include "contiki-net.h"

/* Define which resources to include to meet memory constraints. */
#define REST_RES_HELLO 0
#define REST_RES_CHUNKS 1
#define REST_RES_SEPARATE 1
#define REST_RES_PUSHING 1
#define REST_RES_EVENT 1
#define REST_RES_SUB 1
#define REST_RES_LEDS 1
#define REST_RES_TOGGLE 1
#define REST_RES_LIGHT 0
#define REST_RES_BATTERY 0
#define REST_RES_RADIO 0
#define REST_RES_MIRROR 0 /* causes largest code size */

#include "erbium.h"
```

```
#if defined (PLATFORM_HAS_BUTTON)
#include "dev/button-sensor.h"
#endif

#if defined (PLATFORM_HAS_LEDS)
#include "dev/leds.h"
#endif

#if defined (PLATFORM_HAS_LIGHT)
#include "dev/light-sensor.h"
#endif

#if defined (PLATFORM_HAS_BATTERY)
#include "dev/battery-sensor.h"
#endif

#if defined (PLATFORM_HAS_SHT11)
#include "dev/sht11-sensor.h"
#endif

#if defined (PLATFORM_HAS_RADIO)
#include "dev/radio-sensor.h"
#endif
```

```
/* For CoAP-specific example: not required for normal RESTful Web
service. */
```

```
#if WITH_COAP == 3
#include "er-coap-03.h"
#elif WITH_COAP == 7
#include "er-coap-07.h"
#elif WITH_COAP == 12
#include "er-coap-12.h"
#elif WITH_COAP == 13
#include "er-coap-13.h"
#else
```

```

#warning "Erbium example without CoAP-specific functionality"
#endif /* CoAP-specific example */

/*****
*****/

PERIODIC_RESOURCE(humidity, METHOD_GET, "humidity", "title=\"Hello
humidity: ?len=0..\";rt=\"Text\"", 30*CLOCK_SECOND);

void
humidity_handler(void* request, void* response, uint8_t *buffer,
uint16_t preferred_size, int32_t *offset)
{
    REST.set_header_content_type(response, REST.type.TEXT_PLAIN);

    const char *msg = "Observe Periodic Humidity!";

    REST.set_response_payload(response, (uint8_t *)msg, strlen(msg));
}

void
humidity_periodic_handler(resource_t *r)
{
    static uint16_t humidityVal = 0;
    static char content[30];

    humidityVal=(uint16_t)rand()%100;

    coap_packet_t notification[1];
    coap_init_message(notification, COAP_TYPE_NON, REST.status.OK, 0
);

```



```
    coap_set_payload(notification, content, snprintf(content,
sizeof(content), "Humidity: %u %%", humidityVal));
```

```
    REST.notify_subscribers(r, humidityVal, notification);
}
```

```
//temperature
```

```
PERIODIC_RESOURCE(temperature, METHOD_GET,
"temperature", "title=\"Hello temperature: ?len=0..\";rt=\"Text\"",
30*CLOCK_SECOND);
```

```
void
```

```
temperature_handler(void* request, void* response, uint8_t *buffer,
uint16_t preferred_size, int32_t *offset)
```

```
{
    REST.set_header_content_type(response, REST.type.TEXT_PLAIN);
    const char *msg = "Observe Periodic Temperature!";
    REST.set_response_payload(response, (uint8_t *)msg, strlen(msg));
}
```

```
void
```

```
temperature_periodic_handler(resource_t *r)
```

```
{
    static uint16_t temperatureVal = 0;
    static char content[30];

    temperatureVal=(uint16_t)rand()%120;
```

```
    coap_packet_t notification[1];
```

```
    coap_init_message(notification, COAP_TYPE_NON, REST.status.OK, 0
);
```

```
    coap_set_payload(notification, content, snprintf(content,
sizeof(content), "Temperature: %u F", temperatureVal));
```

```
    REST.notify_subscribers(r, temperatureVal, notification);
}
```

```
//ledstatus
```

```
PERIODIC_RESOURCE(ledS, METHOD_GET, "ledS", "title=\"Hello ledS:
?len=0..\"";rt=\"Text\"", 30*CLOCK_SECOND);
```

```
void
```

```
ledS_handler(void* request, void* response, uint8_t *buffer,
uint16_t preferred_size, int32_t *offset)
```

```
{
    REST.set_header_content_type(response, REST.type.TEXT_PLAIN);
    const char *msg = "LedS!";
    REST.set_response_payload(response, (uint8_t *)msg, strlen(msg));
}
```

```
void
```

```
ledS_periodic_handler(resource_t *r)
```

```
{
    static uint16_t ledSVal = 0;
    static char content[30];

    ledSVal=(uint16_t)rand()%120;

    PRINTF("LedS: %d/% for /%s\n", ledSVal, r->url);

    coap_packet_t notification[1];
```

```

    coap_init_message(notification, COAP_TYPE_NON, REST.status.OK, 0
);
    coap_set_payload(notification, content, snprintf(content,
sizeof(content), "LedS: %u ", ledSVal));

    REST.notify_subscribers(r, ledSVal, notification);
}

/*****
*****/

PROCESS(rest_server_example, "Erbium Example Server");
AUTOSTART_PROCESSES(&rest_server_example);

PROCESS_THREAD(rest_server_example, ev, data)
{
    PROCESS_BEGIN();

    PRINTF("Starting Erbium Example Server\n");

#ifdef RF_CHANNEL
    PRINTF("RF channel: %u\n", RF_CHANNEL);
#endif
#ifdef IEEE802154_PANID
    PRINTF("PAN ID: 0x%04X\n", IEEE802154_PANID);
#endif

    PRINTF("uIP buffer: %u\n", UIP_BUFSIZE);
    PRINTF("LL header: %u\n", UIP_LLH_LEN);
    PRINTF("IP+UDP header: %u\n", UIP_IPUDPH_LEN);
    PRINTF("REST max chunk: %u\n", REST_MAX_CHUNK_SIZE);

```

```

/* Initialize the REST engine. */
rest_init_engine();

/* Activate the application-specific resources. */

rest_activate_periodic_resource(&periodic_resource_humidity);
rest_activate_periodic_resource(&periodic_resource_temperature);
rest_activate_periodic_resource(&periodic_resource_ledS);

/* Define application-specific events here. */
while(1) {
    PROCESS_WAIT_EVENT();
    #if defined (PLATFORM_HAS_BUTTON)
    if (ev == sensors_event && data == &button_sensor) {
        PRINTF("BUTTON\n");
    #if REST_RES_EVENT
        /* Call the event_handler for this application-specific
event. */
        event_event_handler(&resource_event);
    #endif
    #if REST_RES_SEPARATE && WITH_COAP>3
        /* Also call the separate response example handler. */
        Separate_finalize_handler();
    #endif
    }
    #endif /* PLATFORM_HAS_BUTTON */
} /* while (1) */

    PROCESS_END();
}

```

Coap-client file for the first phase of this project:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "contiki.h"
#include "contiki-net.h"
#include "rest.h"
#include "buffer.h"

#define SERVER_NODE(ipaddr)    uip_ip6addr(ipaddr, 0xfe80, 0, 0, 0,
0x0212, 0x7401, 0x0001, 0x0101)
#define LOCAL_PORT 61617
#define REMOTE_PORT 61616

char temp[100];
int xact_id;
static uip_ipaddr_t server_ipaddr;
static struct uip_udp_conn *client_conn;
static struct etimer et;
#define MAX_PAYLOAD_LEN    100

#define NUMBER_OF_URLS 3
char* service_urls[NUMBER_OF_URLS] =
{"led_status", "temperature", "humidity"};
```

```

static void
response_handler(coap_packet_t* response)
{
    uint16_t payload_len = 0;
    uint8_t* payload = NULL;
    payload_len = coap_get_payload(response, &payload);

    PRINTF("Response transaction id: %u", response->tid);
    if (payload) {
        memcpy(temp, payload, payload_len);
        temp[payload_len] = 0;
        PRINTF(" payload: %s\n", temp);
    }
}

static void
handle_incoming_data()
{
    PRINTF("Incoming packet size: %u \n",
(uint16_t)uip_datalen());
    if (init_buffer(COAP_DATA_BUFF_SIZE)) {
        if (uip_newdata()) {
            coap_packet_t* response =
            (coap_packet_t*)allocate_buffer(sizeof(coap_packet_t
));
            if (response) {
                parse_message(response, uip_appdata,
uip_datalen());
                response_handler(response);
            }
        }
    }
}

```

```

        }
    }
    delete_buffer();
}

static void
send_data(void)
{
    char buf[MAX_PAYLOAD_LEN];

    if (init_buffer(COAP_DATA_BUFF_SIZE)) {
        int data_size = 0;
        int service_id = random_rand() % NUMBER_OF_URLS;
        coap_packet_t* request =
        (coap_packet_t*)allocate_buffer(sizeof(coap_packet_t));
        init_packet(request);
        coap_set_method(request, COAP_GET);
        request->tid = xact_id++;
        request->type = MESSAGE_TYPE_CON;
        coap_set_header_uri(request, service_urls[service_id]);

        data_size = serialize_packet(request, buf);

        PRINTF("Client sending request to:");
        PRINT6ADDR(&client_conn->ripaddr);
        PRINTF("]:%u/%s\n", (uint16_t)REMOTE_PORT,
service_urls[service_id]);
        uip_udp_packet_send(client_conn, buf, data_size);
        delete_buffer();
    }
}

```

```

PROCESS(coap_client_example, "COAP Client Example");
AUTOSTART_PROCESSES(&coap_client_example);

PROCESS_THREAD(coap_client_example, ev, data)
{
    PROCESS_BEGIN();

    SERVER_NODE(&server_ipaddr);

    /* new connection with server */
    client_conn = udp_new(&server_ipaddr, UIP_HTONS(REMOTE_PORT),
NULL);
    udp_bind(client_conn, UIP_HTONS(LOCAL_PORT));

    PRINTF("Created a connection with the server ");
    PRINT6ADDR(&client_conn->ripaddr);
    PRINTF(" local/remote port %u/%u\n",
    UIP_HTONS(client_conn->lport),          UIP_HTONS(client_conn-
>rport));
    etimer_set(&et, 5 * CLOCK_SECOND);
    while(1) {
        PROCESS_YIELD();
        if (etimer_expired(&et)) {
            send_data();
            etimer_reset(&et);
        } else if (ev == tcpip_event) {
            handle_incoming_data();
        }
    }
    PROCESS_END();
}

```