Performance of constrained wireless devices in the Internet of Things

Jimmy Assarsson
jimmyassarsson@gmail.com

Linus Karlsson
linus@zozs.se

June 3, 2013

Master's thesis work carried out at *connectBlue*.

Supervisor: Mats Andersson, mats.andersson@connectblue.com
Secondary supervisor: Jonas Jonsson, jonas.jonsson@connectblue.com
Examiner: Roger Henriksson, roger.henriksson@cs.lth.se

Abstract

The Internet of Things is an emerging concept where every device, regardless of size, have their own connection to the Internet. This thesis examines what possible limitations are imposed on the functionality of resource constrained, wireless devices. Several different technologies are evaluated and compared, before a set of them is chosen for inclusion in an implementation, for example: IEEE 802.15.4, 6LoWPAN and CoAP.

The implementation uses the Contiki operating system, and runs on a Texas Instruments CC2530 SoC. We then examine several different performance aspects of our implementation: the amount of data sent, memory usage and energy consumption. The results are discussed together with security aspects applicable to the Internet of things. The memory usage and power consumption were found to be severe issues. Due to the small amount of memory on the chip, all features could not be used at the same time. In addition, the power consumption was found to be too high for battery-powered usage, giving a lifetime of only 27 hours using a button cell battery. The conclusion is that hardware with more memory, and lower power consumption is required. New protocols for radio power-saving should also be developed and implemented in software.

Keywords: Internet of Things, Contiki, CC2530, 6LoWPAN, CoAP

Sammanfattning

Internet of Things – sakernas internet – är ett framväxande koncept där varje enhet, oavsett storlek, har en anslutning till Internet. Detta examensarbete undersöker vilka möjliga begränsningar i funktionalitet detta får på trådlösa enheter med begränsade resurser. Flera olika teknologier undersöks och jämförs, innan ett antal väljs ut för att ingå i en implementation, till exempel: IEEE 802.15.4, 6LoWPAN och CoAP.

Implementationen använder operativsystemet Contiki och körs på ett Texas Instruments CC2530 SoC. Flera prestandaaspekter undersöks: mängden skickad data, minnesanvändning och energiförbrukning. Resultaten diskuteras tillsammans med säkerhetsaspekter att ta hänsyn till i Internet of Things. Minnesanvändningen och energiförbrukningen är de mest problematiska områdena. På grund av chippets begränsade mängd minne kan inte all funktionalitet användas samtidigt. Dessutom är energiförbrukningen för hög för längre tids strömförsörjning med batteri, vilket ger en livslängd på enbart 27 timmar med ett knappcellsbatteri. Slutsatsen är att hårdvara med mer minne och lägre energiförbrukning behövs. Nya protokoll för energibesparande radioanvändning behöver också utvecklas och implementeras i mjukvara.

Acknowledgements

First we would like to thank our examiner Roger Henriksson, who has supported us during the whole thesis.

We would also like to thank our supervisors at connectBlue: Mats Andersson for his valuable support and many interesting discussions about the future of Internet of Things; and Jonas Jonsson for introducing us to wireless technologies and getting us up to speed with the hardware.

Finally we would also like to thank all employees at connectBlue for their feedback, questions and for making us feel welcome.

Contents

Chapter 1

Introduction

On today's Internet, there are lots of connected devices, for example: computers, cell phones, televisions and game consoles. Today most people think it is a requirement that their cell phone has an Internet connection, but still this is quite a recent development. The next step is to connect even more, and even smaller devices to the Internet. Potential use cases could be found in the areas of home automation, healthcare, personal fitness, and many others. This development has already started with for example domestic alarm systems, where home owners can monitor temperatures and turn the alarm off and on using their cell phones [1]. However, since there is a lack of standards, it is complicated to make devices from different vendors communicate with each other.

The interoperability between different devices is a central part of the Internet of Things concept [2]. A house could be equipped with both a domestic alarm system like [1] and also with a remote control heating service like [3]. Instead of manually having to lower the temperature upon leaving the house, the alarm system could notify the heating system that the house is empty, such that the heating system can lower the temperature. Also less hardware would be required, since both systems use temperature readings, but currently they would use separate temperature sensors. If they were conforming to the Internet of Things concept, they would share a single temperature sensor.

In the Internet of Things, every device has a connection to the Internet. This makes it possible for devices to exchange information between each other, but also for the devices to reach, or to be reached from the Internet. This might not sound like it is difficult, but when the devices are very small, have limited power and need to be cheap, special solutions will be required to get a well-functioning system.

1.1 Purpose and goals

The purpose of this master's thesis is to examine what, if any, limitations are imposed on the functionality of small nodes. The nodes should be able to be battery powered, should not require charging, and need battery life in the range of months or ideally years. Several aspects of the implementation must also be examined: packet sizes, memory usage, energy consumption, reliability and security.

To examine this, the nodes should be connected to a wireless network, so that they can be reached from the Internet. There exists several wireless technologies and multiple protocols to make this possible. This means that each technology and protocol need to be examined, and that a decision has to be made on what to use. The decision needs to be based on a set of requirements. After this, the implementation effort can be started.

When an implementation has been completed, it should also be evaluated according to the earlier described aspects. The method and performance metrics to use when evaluating the implementation will be discussed.

1.2 About connectBlue

We have conducted our master's thesis together with connectBlue, a company located in Malmö, Sweden.

ConnectBlue is a company specialised in providing wireless solutions. They develop both complete units for point-to-point communication and wireless modules for embedding in other products. They also provide custom solutions to clients. Currently, the company provides modules for Bluetooth, Bluetooth Low Energy, IEEE 802.15.4 and Wi-Fi (IEEE 802.11).

1.3 Report structure

In Chapter 2 we present a background overview of several different technologies that may be used in the Internet of Things, and finally how they can be combined in a complete solution.

In Chapter 3 we compare the technologies described in the background, and decide which to use in the implementation.

In Chapter 4 our implementation is presented from a hardware and software perspective. We also discuss problems encountered during the implementation phase.

In Chapter 5 we evaluate the implementation. First the methods are discussed, and then the results are presented.

In Chapter 6 the results from the previous chapter are discussed and analysed.

In Chapter 7 the results of this thesis are summarised, and possible future work is discussed.

Chapter 2
Background

There exist several different technologies that can be utilised in the Internet of Things. The purpose of this chapter is to give an overview of some of them. We will start by comparing different wireless technologies, continue by discussing protocols on the network layer and then talk about application level protocols. Finally we will give an overview of how the protocols on the different layers can be combined into complete solutions, and give examples of current standardisation efforts.

2.1 Wireless technologies

The vision with Internet of Things is that every device should have an Internet connection. Connecting everything with cables would not be feasible, so other solutions must be found. There are two major solutions to avoid dedicated network cables: send the data over radio or use power line communication. The latter requires the device to have a mains connection, and cannot be used for battery powered devices. However, wireless techniques can be used by any type of device and thus they are the focus of this report.

There exists a very large amount of wireless technologies today. When choosing which to consider further we have set up a small set of requirements.

- The technologies should use the 2.4 GHz band, which is one of the ISM (Industrial Scientific Medical) bands. The frequency range is 2.400 GHz–2.500 GHz. This frequency band is available for use throughout the whole world, without requiring a license [4].
- The specification of the wireless standard should be *open*. With an open standard we mean a standard that is publicly available for anyone to download and examine.

2.1.1 IEEE 802.15.4

IEEE 802.15.4 is a standard for low-power Wireless Personal Area Networks (WPAN). Its purpose is to provide low-data-rate, low-cost, low-power and short-range wireless communication for embedded devices [5]. The standard only defines the physical layer (PHY) and the media access control layer (MAC). There are several complete communication stack specifications which are built on top of the IEEE 802.15.4 standard, for example ZigBee and WirelessHART [2].

A Personal Area Network (PAN) consists of a PAN coordinator node and several PAN members. The PAN coordinator is responsible for assigning unique addresses to the nodes within the PAN. Nodes can act as a PAN coordinator in one PAN and simultaneously act as a PAN member in another PAN.

A node is either a Full-Function Device (FFD) or a Reduced-Function Device (RFD). FFDs are capable of acting as coordinators and may not enter sleep mode. A RFD is typically a small, battery-powered device for simple applications, which spends most of its time in sleep-mode. An FFD can connect to several FFDs and RFDs, while a RFD can only connect to a single FFD.

The networks can be arranged in three different topologies: star topology, mesh topology and cluster tree topology. In a star network, every PAN member node is directly connected to the PAN coordinator, as shown in Figure 2.1. The area of the network is limited, since all nodes has to be within range of the coordinator.

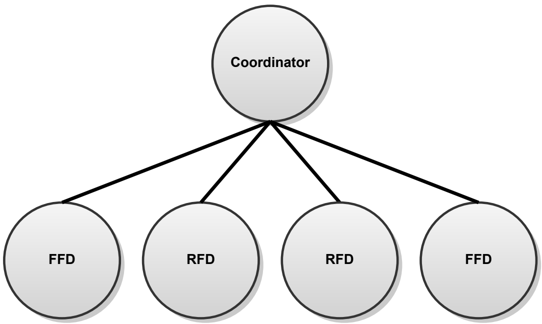


Figure 2.1: IEEE 802.15.4 star topology.

In a mesh network all FFD PAN members can connect to each other. This increases both the area and the robustness of the network. The covered area is increased since nodes do not have to be in the range of the coordinator, and the traffic can be forwarded by intermediate nodes. The robustness increases as well, since it might be possible to find an alternative route even if a connection between two nodes is lost. An example of a mesh structure is shown in Figure 2.2.

In a cluster tree network the nodes are connected in a tree topology. Each connected FFD with children may create its own PAN, where the FFD becomes PAN coordinator within the new PAN [5]. This increases the area of the network, just like in mesh networks, but also increases the maximum number of connected nodes. A disadvantage is that the latency for sending data over the network may also increase. An example of a cluster tree network is shown in Figure 2.3.

It is important to notice that IEEE 802.15.4 does not define how routing should be performed when mesh topology or cluster tree topology are used [5].

There are two kinds of node addresses in an IEEE 802.15.4 network: a short (16-bits) and a long (64-bits) identifier. The PAN coordinator assigns a unique (within the PAN) short identifier to every node. The long identifier is assigned to the node when manufactured, and it is globally unique. The first 24-bits of the long identifier is unique to the manufacturer. Both addresses can be used to address a node. There is also support for broadcast messaging, but not for multicast messaging [6].

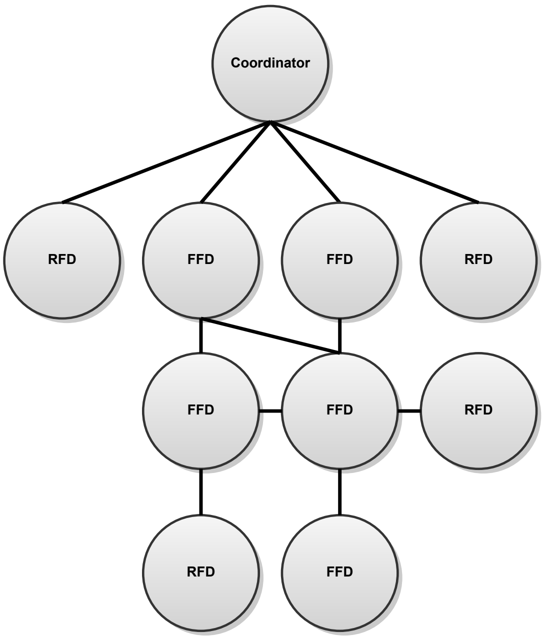


Figure 2.2: IEEE 802.15.4 mesh topology.

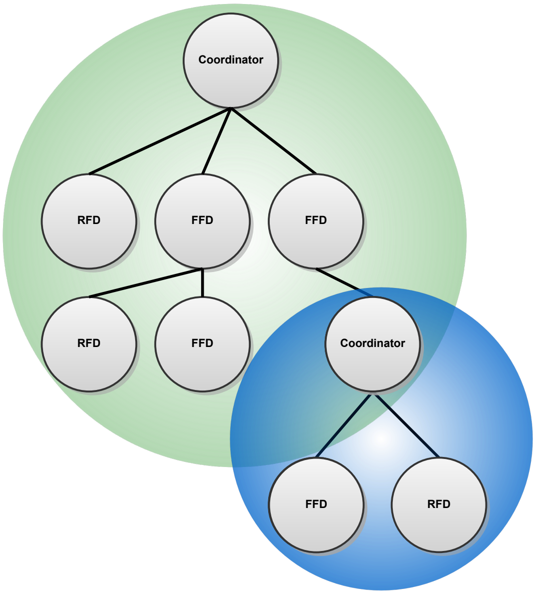


Figure 2.3: IEEE 802.15.4 cluster tree topology.

IEEE 802.15.4 has a maximum transfer rate of 250 kbit/s and a maximum range of a few tens of meters [2]. There is however a sub-gigahertz version of IEEE 802.15.4, which provides a longer reach but with a maximum rate of 20 kbit/s. The maximum payload of an IEEE 802.15.4 physical frame is 127 bytes [6].

A typical low power IEEE 802.15.4 transceiver, like the CC2420 from Texas Instruments, consumes 18.8 mA when receiving data and 17.4 mA when transmitting data, at a voltage of 3.0 V [7].

The frequency of the radio is in the range 2.4000 GHz–2.4835 GHz, divided into 15 different channels with 5 MHz channel spacing. The standard specifies how 128-bit AES can be used for link layer security, to provide confidentiality and integrity [2].

2.1.2 Bluetooth Low Energy

Bluetooth Low Energy (BLE) is a part of the Bluetooth v4.0 standard, adopted in 2010-06-30 [8]. It is meant to be an alternative to classic Bluetooth, but better suited for low power applications. BLE is not compatible with classic Bluetooth, and devices that wish to communicate with both types of devices must implement support for both protocol stacks. However, BLE can be integrated into a current Bluetooth implementation without too much effort [9].

The frequency band is divided into 40 different channels. BLE uses frequency hopping to change channel, which reduces problems with interference. Frequency hopping is performed according to a pattern that the master decides and notifies the slave about. Change of frequency is done for every few packets sent. BLE also uses Adaptive Frequency Hopping, which means that it can detect channels that have a lot of interference, and avoid using them for transmissions [10].

Bluetooth Low Energy only supports a single type of network topology: the star topology [9]. This is because the standard states that: "Physical links between slaves in a piconet are not supported. At this time, slaves are not permitted to have physical links to more than one master at a time." [11]. A network consists of a single master and one or more slaves; together they form a piconet, as can be seen in Figure 2.4. Since no physical links are allowed between slaves, communication between slaves has to pass the master.

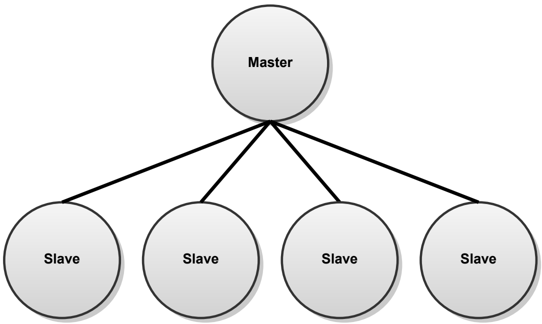


Figure 2.4: A Bluetooth Low Energy piconet.

To save energy, the slave nodes only wake up periodically to listen for data from the master. The master decides this interval, and ensures that each slave has a unique time slot. The medium is thus divided using a Time Division Multiple Access algorithm. If a long interval is chosen, energy consumption can be very low. A study [9] shows that a Bluetooth Low Energy System-on-a-Chip module with a CR2032 button cell battery of

230 mAh can get a lifetime of up to 14.1 years in ideal conditions.

The BLE master uses 32-bit addresses to identify the slaves, but the maximum number of nodes in a piconet is much smaller than 2^{32}. The number of slaves depends on the periodic interval in which the slaves wake up. If the periods between wake-ups are long, more nodes can be a part of the piconet since there are more free time slots to use. The number of slaves a master can handle varies from 2 to 5917 nodes [9].

Bluetooth Low Energy has a maximum bandwidth of 1 Mbit/s on the physical layer, but the maximum application layer throughput is 236.7 kbit/s under ideal circumstances [12]. The throughput is, however, dependent on the radio conditions.

BLE supports link level security: AES-128 is used in CCM mode, which encrypts the data and calculates a MAC. This ensures both confidentiality and integrity for the data, which is important since any device in range can listen to the data sent over the shared radio medium.

Every BLE device has a 48-bit unique device address, which is static and assigned by the manufacturer of the device. Since this address is static, it could be used to track the user of the device, since the device address is used in BLE communication. For privacy reasons, BLE devices can instead generate private addresses, which cannot be resolved by untrusted devices.

BLE does not only specify the physical and link layer, but also higher layers. Generic ATTribute profile (GATT) is an application layer protocol that allows exchange of data in the form of properties between devices. GATT does only exist in BLE.

2.1.3 Wi-Fi

Wi-Fi, also called IEEE 802.11, is a widely used standard for wireless Internet. It is designed to provide high throughput for the connected devices. The maximum bandwidth of Wi-Fi depends on the standard used. For example, 802.11g and 802.11n have a maximum bandwidth of 54 Mbit/s and 600 Mbit/s respectively.

The frequency band is divided into 14 different channels, with a channel width of 20 MHz [13]. This means that the channels partly overlap, because the total width of the frequency band is only 100 MHz. Wi-Fi does not have any built-in functionality to reduce interference by switching channel, so having multiple access points within range of each other may require frequency planning to maximise performance.

There are two different types of network topologies: infrastructure and ad-hoc. In infrastructure mode, there is a central access point which all devices can connect to. Devices that are in range of each other can communicate directly, otherwise the traffic is forwarded by the access point [4].

In ad-hoc mode a connection is established between two or more nodes without an access point. The nodes can only communicate with each other, and cannot reach another network.

There are several different security mechanisms in Wi-Fi. The latest standard, WPA2, uses a protocol called CCMP which uses AES to provide data confidentiality, authentication, integrity and replay protection [13].

Low-power Wi-Fi modules have significantly lower power requirements than ordinary Wi-Fi modules [2]. Still, transmit power is roughly 300 mW, and using a CR2032 button cell battery with a voltage of 3.0 V this will require a current of 100 mA. This peak current would reduce the lifetime of the battery significantly [14].

2.2 Networking

With the exception of Wi-Fi, the wireless techniques described earlier were not initially designed to be used by devices when connecting to the Internet. Instead they have been used for point-to-point communications, only sending small amounts of data. This means that there is no standardised way to take the step out on the Internet.

Today, the Internet Protocol (IP) is used for devices on the Internet to reach and communicate with each other. There are several reasons to also use IP on the devices in the Internet of Things [15]. IP has been used for a long time and is proven to work well on lots of devices. The widespread use means that there is a lot of infrastructure that supports IP, this means that it is easy to connect new devices. Finally, IP is also an open standard, which together with the widespread use means that a lot of people have knowledge of the protocol.

2.2.1 IPv6

Today IPv4 is the main protocol used on the Internet, but a transition to IPv6 is on its way. The reason for this transition is the limited address space in IPv4, which only has a total of 2^{32} addresses. With a world population of 7 billion [16], and a future in which every person has several Internet enabled devices, it is trivial to see that the address space of IPv4 is not enough. IPv6 solves this by enlarging the address space to 2^{128}, which should make it possible for every device to have a unique address.

When connecting many new devices to a network, it is desirable to avoid having to configure every device manually. For some very small devices it may even be impossible due to the lack of an input interface. IPv6 can also solve this problem by *IPv6 Stateless Address Autoconfiguration*. In summary, it generates working IPv6 addresses without manual intervention.

However, all these new features comes at a cost: IPv6 headers are generally larger than IPv4 headers due to the longer addresses. Since wireless technologies like Bluetooth Low Energy and IEEE 802.15.4 have small packet sizes, a large header reduces throughput because there is less available space for the payload. This can be solved by using a protocol called 6LoWPAN. It is used to compress the standard IPv6 header to a smaller size, which reduces the overhead and thus increases the maximum possible payload.

Currently, 6LoWPAN is only available for IEEE 802.15.4, but there is currently an effort by the IETF to adapt it to Bluetooth Low Energy [17].

2.3 Application

The final goal is to actually send useful data using IP over the wireless media. There are a couple of technologies that are useful for the Internet of Things on the application layer.

2.3.1 REST

REpresentational State Transfer (REST) is an abstract model of web architectures, which can be used as a guideline when designing web-based applications [18]. It has become widely used when designing and implementing web services [19].

REST defines a set of constraints, with the purpose "…to minimize latency and network communication, while at the same time maximizing the independence and scalability of component implementations." [18]. These constraints can be summarised into:

Client – Server The architecture is client – server based, where the client requests data from the server. The server processes the request and responds.

Stateless The communication should be stateless, which means that the client itself is responsible for keeping track of its state and has to provide the server with all required data at each request. Therefore the server does not have to store any session data between requests.

Cacheable The response data to a request should be declared, implicitly or explicitly, as cacheable or non-cacheable. If the data is cacheable and the client sends the same request again, it is possible to use the cached data, if a cache is available. This technique reduces the server load.

Uniform interface A uniform interface for accessing and manipulating resources reduces complexity and makes the architecture more scalable. A typical uniform interface is HTTP, which only provides eight basic methods. Four of these: GET, PUT, POST and DELETE, are used to access and modify resources. A resource is reached by its Uniform Resource Identifier (URI). URIs contain either the name of the resource, called Uniform Resource Name (URN); the address of the resource, called Uniform Resource Locator (URL); or both [20, 21].

Layered system Layers are used to make the architecture more scalable. The layers are hierarchical, and communication is only allowed between immediate layers. This allows the system to be distributed among several computers.

Code-on-demand Code-on-demand is used to extend the functionality of the client, by using scripts or applets. This reduces the requirements for the client implementation and makes the client more adaptable.

Architectures that fulfil the REST design criteria are denoted RESTful [20].

2.3.2 SOAP

SOAP, an acronym for Simple Object Access Protocol, is a protocol for communication between a sender and a receiver. Communication is done by sending messages in a well-specified XML-format [22]. The SOAP message is embedded in what is called an envelope, which consists of an optional header and a message body. The XML schema used for all data is defined and specified in the message, so that the sender and receiver know how to parse the data.

SOAP data is generally used in an RPC (Remote Procedure Call) way, where the messages sent represent function calls. This means that the client implementation simply issues a method call. This method call is then translated to SOAP messages by a library. Usually there is just a single URL that implements several methods [6]. Depending on the content of the SOAP message, different responses are returned. This is a major difference compared to a RESTful architecture.

SOAP is not bound to a specific underlying message exchange protocol. It can be used with HTTP, SMTP, POP, raw TCP, etc. It is often used over HTTP, and the SOAP 1.1 specification defines how messaging works over HTTP [22].

Since all SOAP messages are required to conform to a given schema and be valid XML, they have a fairly large size when compared to RESTful techniques. The overhead includes things like `<soap:...>` tags, which can include (long) URLs to schemas. The size of these messages are normally not an issue, but may be worth considering when data is sent over low bandwidth connections such as IEEE 802.15.4 and Bluetooth Low Energy.

2.3.3 CoAP

CoAP, or Constrained Application Protocol, is a web application layer protocol that is tailored for devices with limited resources. It is currently under development by the IETF, and as of this writing the current draft version is `draft-ietf-core-coap-13` [23].

CoAP is similar to the well-known and widespread HTTP, which is used throughout the web. Both protocols use URIs to locate resources, they support Content Types to describe the format of data and the user of the protocol can expect the messages to be reliably transferred. HTTP and CoAP also shares a common set of request methods: GET, POST, PUT and DELETE. All of this makes CoAP easy to understand and integrate into the current architecture of the web.

To make CoAP especially suited for limited devices, it does have several important differences compared to HTTP. First of all, CoAP does not require a reliable transport protocol, which means that it can be used over UDP. Reliable transport protocols, like TCP, increases the complexity, size and resource usage of the software, which may be unwanted or even impossible to use on a constrained device. Instead CoAP implements its own optional, light-weight, and simple, but not fully-featured reliability mechanism.

Another major difference is the format of the header. In HTTP options and request method are transmitted in clear-text, which means that even a basic request consists of several bytes of data. The CoAP header has a binary format instead, where request method and options are encoded into various bits and in a specific order. This reduces the data that has to be sent, received and parsed by the endpoints.

A problem with CoAP is the fact that it is not as widespread as HTTP. This limits the number of devices that can communicate with servers running CoAP. However, because of the similarities between the protocols, it is possible and fairly easy to create a proxy that converts between the protocols. The CoAP standard does even have a dedicated section for “Cross-Protocol Proxying between CoAP and HTTP” [23]. Proxies may even be transparent, such that neither endpoint actually knows about the fact that a protocol translation is being performed by an intermediate party.

Another, albeit temporary, problem with CoAP is that it is currently only a draft. Changes between drafts are sometimes significant, and may be neither forward nor backward compatible. This requires that all CoAP software implement the same version of the protocol.

2.4 Complete solutions

There exist several different approaches of how Internet of things can be accomplished. A gateway will always be required to connect the nodes to the Internet, since the nodes are wirelessly connected. The gateway is equipped with a wireless transceiver for the nodes, and it is connected to the Internet as in Figure 2.5. The communication between Internet and the gateway should be over HTTP, which infers TCP/IP. We have divided the approaches into four different solutions, which differs in the way data is sent between the nodes and the gateway. An overview is found in Table 2.1.

In the first solution the gateway must have knowledge of the type of data sent, because it must repackage the data during communication between the node and the Internet. This

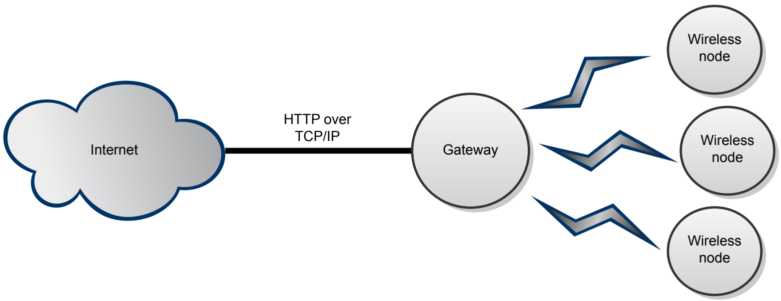


Figure 2.5: Gateway, node and Internet overview.

Table 2.1: Overview of different gateway - node solutions.

	Network layer	Application layer
1	Non-IP	Non-standard
2	IPv6	HTTP
3	6LoWPAN	HTTP
4	6LoWPAN	CoAP

means that the gateway must be extended for future new node types. It must also perform address translations between IP and the addresses used on the wireless network.

In the second solution ordinary IPv6 is used on the nodes, which means that the gateway can simply act as a router and relay packages without having to care about their contents. Because of this the gateway can be fairly simple. However, it requires quite a lot of data to be sent over the wireless link. This is typically used in Wi-Fi networks.

The third solution also uses IPv6 addressing but with header compression, called 6LoWPAN. This reduces the data sent on the network layer, but still has the benefits of the previous solution. The complexity of the gateway is increased somewhat, because it has to perform conversion between IPv6 and 6LoWPAN.

The fourth solution is as well based on 6LoWPAN, but also tries to minimise the data on the application level. By using CoAP the HTTP data can be compressed, so that less data needs to be sent over the wireless link. This means that the gateway needs to perform the conversion between HTTP and CoAP. However, the gateway does not need to care about the actual contents of the HTTP conversation, which means that it can handle new node types.

2.4.1 Current standardisations

There are currently several on-going standardisation efforts on different levels for Internet of Things.

The IP for Smart Objects (IPSO) alliance is a non-profit association promoting the use of IP for devices in the Internet of things. They provide an application framework for accessing data from devices [24]. The framework defines a RESTful design, and only covers the application layer, the underlying transport could be anything. IPSO Alliance does not develop any standards on their own, and their framework is only considered as a set of guidelines.

Smart Energy Profile 2.0 (SEP 2.0) is a draft standard for energy monitoring and controlling over the Internet. It is defined by the ZigBee Alliance and relies on the ZigBee

IP stack. The ZigBee IP stack uses 6LoWPAN on top of IEEE 802.15.4. On top of this it uses HTTP over TCP. This corresponds to the third solution in Table 2.1.

There are currently on-going negotiations and standardisation efforts regarding how Bluetooth Low Energy devices should be able to connect to the Internet. Two main approaches are discussed. The first approach is to use the existing GATT profile to communicate data between the wireless devices. Data repackaging and translation will be required at the gateway. This corresponds to the first solution in Table 2.1. The second approach is to replace GATT with 6LoWPAN. A recent prototype implementation proved that it is possible and that the power consumption does not increase much compared to GATT [25]. The prototype implementation corresponds to the fourth solution in Table 2.1.

There are also several other organisations who develop and promote the Internet of Things: European Telecommunications Standards Institute, IoT-A, Eclipse M2M and IoT@Work.

Chapter 3

Choosing technologies

We need to combine several of the technologies described earlier. To be more specific, we need a wireless technology, a network layer protocol and finally an application layer protocol. First a set of guidelines are described, and then these guidelines are used in the discussion when choosing the technologies.

3.1 Methods

To choose appropriate technologies, a list of guidelines has been developed.

- The nodes have very limited resources when it comes to RAM, energy supply and ROM. This aspect must be considered during the whole process.
- IP should be used in all communications because of the reasons stated in Section 2.2.
- Nodes should require minimal configuration to work in a network.
- Current standards and guidelines should be taken into consideration, and should be preferred to custom solutions.
- Existing implementations of technologies should be investigated. A well-tested and proven implementation may be better to use.

3.2 Discussion

Starting with the wireless technologies, we have looked at three different options: IEEE 802.15.4, Bluetooth Low Energy and Wi-Fi.

Wi-Fi is a well-tested and widespread wireless technology, which is used together with IP. Networks can be found in homes, schools, universities, workplaces and in public locations. There is, however, a big drawback of Wi-Fi: its power consumption. Wi-Fi is not suitable for the kind of small battery-powered devices that this thesis is aimed towards, as discussed in Section 2.1.3.

Bluetooth Low Energy and IEEE 802.15.4 both have similar, low energy requirements and are well suited for low power devices. BLE does only support the star topology, while IEEE 802.15.4 also supports mesh and cluster tree topologies. The two latter topologies can extend the range of the wireless network, as explained in Section 2.1.1.

Bluetooth Low Energy uses adaptive frequency hopping to reduce problems with interference. This makes it more robust compared to IEEE 802.15.4, which only uses a single channel. This is important when used in areas with many different wireless networks that can interfere with each other.

Another difference between the two wireless technologies is their maximum payloads. IEEE 802.15.4 has a maximum payload of 72–116 bytes, while BLE has 23 bytes. If the data cannot fit into a single packet, it must be fragmented and sent in several packets. This increases the overhead and thus it is preferable if all data can fit into a single packet. Since IEEE 802.15.4 has a larger maximum payload than BLE, it has an advantage. However, BLE has a defined way to fragment packets, while IEEE 802.15.4 has not. This means that support for fragmentation has to be added on top of IEEE 802.15.4.

As described earlier, IPv6 should be used on the wireless network. However, IPv6 is not well suited when the maximum payload is low, since the IPv6 header is 40 bytes large. For BLE, fragmentation will be inevitable, and for IEEE 802.15.4 a large portion of the payload will be consumed by the header.

6LoWPAN is a solution to this problem. It compresses the header such that more data can be fitted into the packets. 6LoWPAN can be used with both wireless technologies. It was first developed for IEEE 802.15.4 as RFC4944 [26] in 2007. It has since then been adopted also for BLE, but it is still a draft [17]. It is advantageous to use a finished standard because it is finalised and has been tested. Another advantage is that there exists open-source implementations of 6LoWPAN for IEEE 802.15.4, which could be used in an implementation [27].

IEEE 802.15.4 and BLE are both suitable to use as a wireless technology, but because writing a new implementation of 6LoWPAN for BLE would be a hard and time-consuming task, out-of-scope of this master's thesis, IEEE 802.15.4 will be used.

The most natural and simple solution would be to simply forward the HTTP traffic from the gateway to the end node. HTTP does require a reliable transport protocol, usually TCP. Using TCP would require the nodes both to send and receive several packets to setup and maintain the connection. This utilises more resources on the node, both ROM and RAM. It also increases the amount of sent data over the wireless link, and increases the likelihood that a packet needs to be fragmented.

CoAP can be used to translate HTTP into a more compact, binary format. This reduces the data that needs to be sent wirelessly. In addition, CoAP does not require the use of a reliable transport protocol, which means that the control data of TCP can be avoided. Because the traffic between the gateway and Internet should be HTTP, a translation between CoAP and HTTP would be required in the gateway. Since the resource usage in the gateway is not as important as on the nodes, this is not a problem and CoAP is preferable.

SOAP is considered more complex and requires more overhead, compared to a RESTful design [19]. SOAP is based on XML, which is verbose and requires parsing. An XML parser requires extra RAM and ROM, and because the nodes are limited it is favourable to avoid it, if possible. The verbosity of XML also requires more data to be sent over the wireless link, which is amplified by the possible fragmentation of packets. A RESTful implementation does not have to use XML, and could thus avoid the problems above.

The IPSO Alliance application framework defines a RESTful design, and the guidelines are supported by several organisations and companies. Because of this, it is suitable as a foundation for the implementation.

In Section 2.4.1 Smart Energy 2.0 and GATT were presented. Neither of them were found to be suitable.

GATT is not IP-based, and can be ruled out immediately.

Smart Energy 2.0 utilises HTTP and TCP, which would increases the demands on the node's performance. In our opinion, this is unnecessary when CoAP and UDP could be used instead. Another disadvantage is that SEP 2.0's primary focus is energy monitoring and control, and is not adapted for other types of data.

3.3 Summary

To summarise, the following techniques will be used in the implementation:

- IEEE 802.15.4 on the physical layer
- 6LoWPAN on the network layer
- CoAP over UDP on the application and transport layer
- IPSO Alliance application framework as a guideline when designing the API

Chapter 4
Implementation

When the various technologies have been chosen, we need to select the various pieces of hardware and software so that we can actually implement our solution.

We did not have many hardware options to choose between. Since connectBlue had modules with IEEE 802.15.4-radios and microcontrollers, we developed our solution based on this hardware.

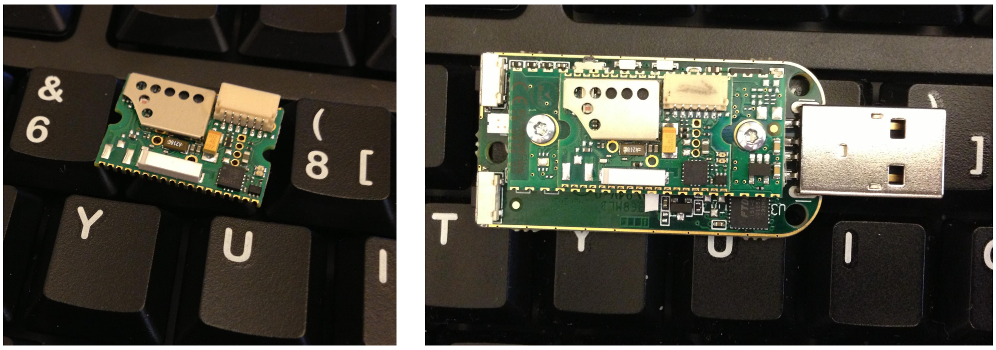
With the hardware on the nodes fixed, the software needed to be chosen such that it works on the given architecture. This does of course limit the available software. Furthermore, it is beneficial to use as much pre-written software as possible. It would not be reasonable to write a solution starting from scratch, since this would require writing radio drivers, a complete network stack, etc. This would require a significant amount of time, and it would probably be inferior to more mature and well-tested software. We have tried to use as many pre-written pieces of software as possible, and worked on integrating them. Using open-source software is a requirement, since we want to be able to adapt the software to our specific needs without any additional costs.

The gateway should be a device that has access to unlimited power and resources, in contrast to the nodes. Therefore it can be practically any device. A requirement is that it should be able to run Linux.

4.1 Hardware

The target hardware for the sensor nodes is a module from connectBlue. The module is equipped with a TI CC2530 SoC, a TI TMP112 temperature sensor, a ST Microelectronics LIS3DH accelerometer and an internal or external antenna. The CC2530 combines an IEEE 802.15.4 transceiver and an 8051 host microcontroller with 256 KB of ROM and 8 KB of RAM. Only 223 B of the RAM is available for the call stack [28]. The node does also include a UART interface, which can be used to communicate over a serial link. A picture of the module can be seen in Figure 4.1(a).

A Texas Instrument CC Debugger is used for programming and to supply the node with power. The node can also be powered by mounting it on a USB-dongle, as seen in Figure 4.1(b). The dongle also makes it possible to access the UART interface over USB.



(a) Stand-alone node.

(b) Node mounted on USB-dongle.

Figure 4.1: The nodes placed on a standard-sized keyboard.

A BeagleBone single-board computer is used as the gateway hardware. The BeagleBone has a 720 MHz ARM processor, 256 MB of RAM and is suitable for running Linux. It is equipped with an Ethernet port and a USB port. This is more than enough for our purposes, and it should not force any restrictions on our solution.

4.2 Software

Our solution consists of several different pieces of software. An overview of how they relate to each other can be seen in Figure 4.2. All software used in the implementation is open-source.

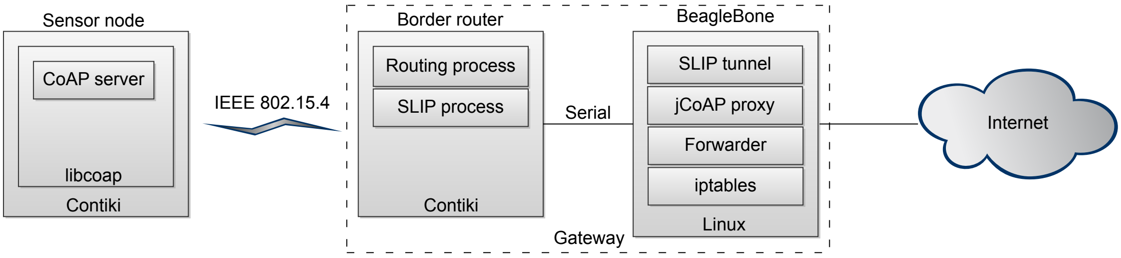


Figure 4.2: Architectural overview of the software.

Contiki [27] was used as foundation for the sensor node implementation. Contiki is a light-weight operating system for devices with limited resources. It has built-in support for several of the technologies we are interested in, for example IPv6, 6LoWPAN, routing, CoAP and an IEEE 802.15.4 radio driver. Contiki has support for several different microcontrollers and platforms, among others: the CC2530 platform.

Libcoap [29] is a light-weight CoAP library which implements the, as of this writing, latest CoAP draft: `draft-ietf-core-coap-13`. It is intended to be used with both Linux and Contiki. Libcoap is used as the CoAP server implementation on the sensor node.

Arch Linux [30] is used as operating system on the BeagleBone gateway. Iptables is used to create a transparent proxy between the Internet and the nodes.

We have also used a Java based CoAP library, jCoAP [31]. It includes an HTTP to CoAP proxy, which is used in the gateway. The library does not conform to the latest CoAP draft (`draft-ietf-core-coap-13`), but we have made some changes to make it compatible with libcoap.

Since Java does only allow a limited subset of socket options, we have written a small forwarder software, that forwards traffic to the jCoAP proxy.

The border router uses the same hardware as the sensor nodes, but is connected to a USB-dongle. It also runs Contiki, but with a routing process and a SLIP process. SLIP is used for IP communication over a serial line. The border router is able to translate between IPv6 and 6LoWPAN. IPv6 is used on the serial line and 6LoWPAN is used on the wireless link.

A SLIP tunnel runs on the BeagleBone, so that the serial interface can be treated as a network interface, which allows routing of traffic to and from the interface.

4.2.1 Contiki

The main idea behind Contiki is to provide a light-weight operating system for small wireless devices in the Internet of things. The operating system is designed in a scalable way, that makes it easy to add and modify components. Contiki contains various useful components, such as: timers, radio drivers, radio duty cycling (RDC) drivers, threads and processes.

By using the well-tested and certified uIP communication stack for IPv6 and 6LoWPAN, which is included in Contiki, we avoid implementing this functionality by ourself, which probably would be very time-consuming. A 6LoWPAN adoption layer for IEEE 802.15.4 is also included in the 6LoWPAN implementation, which enables packet fragmentation, header compression and link layer forwarding [2]. Contiki also supports RPL (Routing Protocol for LLNs), which is an IPv6 routing protocol designed for low power and lossy networks [32].

There are two CoAP libraries included with Contiki: erbium and RESTful contiki. We did not manage to get any of these implementations to run on the nodes. It is also worth mentioning that neither of the libraries supports the latest version of CoAP, which made them less interesting to use in our implementation.

In the Cooja simulation environment for Contiki, it is possible to simulate and debug networks. Unfortunately there is no simulation support for the CC2530 platform. Simulations are however useful for understanding Contiki and its components, especially the network components.

An application to estimate the energy consumption is also included in Contiki. It keeps track of the amount of time the system spends in a certain state. If the energy consumption are known for every state, the total estimated consumption can be calculated.

Another energy related feature of Contiki is its support for Radio Duty Cycling protocols. To reduce the power usage of the node, the radio can be turned off. During this time, the node is unable to receive or send data. To prevent packet loss from occurring, a protocol must be defined so that nodes are able to receive data even though they are not listening all the time.

Contiki supports several different RDC protocols, but there are only two which can be used together with IP: ContikiMAC and nullrdc. The latter is a simple protocol with a duty cycle of 100 % – the radio is turned on all of the time. This means that data can be sent or received at any time.

ContikiMAC can reduce the radio duty cycle to well below 5 %, depending on settings [33]. Nodes wake up periodically with a frequency of typically 2 Hz to 16 Hz. When a node wants to send data, it sends the same packet repeatedly such that the receiving node is guaranteed to be awake sometime during the transmission. As soon as the receiver has woken up and has received the packet, an acknowledgement is sent, and the sender can stop the strobing. For example, if the wake up frequency is 2 Hz, the sender may need to send for 0.5 s to match the receiver's wake up interval. If a sender wants to broadcast a packet to all nodes within range, it will always have to send during the whole interval.

Contiki uses a hardware watchdog timer to prevent system hangs. The timer has to be cleared periodically before it expires, otherwise something probably went wrong and the system resets.

The RPL routing protocol has functionality for detecting if a neighbour node becomes unreachable, and will then automatically repair and reroute the network. This improves the reliability for networks that contains movable nodes or nodes that are periodically unreachable.

Contiki does not provide any security features. There are however external security libraries that can be used with Contiki [34]. The CC2530 has hardware accelerated AES-128 encryption and decryption, which is suitable for security functionality.

Even though Contiki supports the CC2530 platform, the standard code base is not very suitable for the platform because of its limited call stack size. Therefore a patched version of Contiki has been used [35]. A more detailed discussion regarding this can be found in Section 4.3.1.

4.2.2 Sensor node

Contiki is used on the sensor nodes for handling communication and interfacing hardware. A CoAP server process runs in Contiki, where the libcoap library is used as the server implementation.

A temperature resource is registered at the start up of the CoAP server process. The resource's URI, the type of the resource and an interface description are set when a resource registration is performed. A handler function is also registered for each associated method that the resource supports. The handler functions are used for executing the requested method and for generating a proper response.

The CoAP server process waits until a message is received. When a message is received, the request handler processes its content. The request handler first checks that the data forms a CoAP message, if it does not, the message is dropped and the process waits for a new message. If the requested resource is not present, the server responds with code 404 (Not found). The server responds with code 405 (Method Not Allowed), if the requested resource exists but there is no handler registered for the specific method. If the request is for a valid resource and there is a valid handler function registered for the method, the corresponding handler function is called and finally the response is sent. A flowchart of the server request-response cycle can be found in Figure 4.3.

The different handler functions contains application specific code, like reading temperature data, which is used for the GET request for the temperature resource.

A typical temperature GET response is shown below. The response is in JavaScript Object Notation (JSON) format, which is a way to serialise objects to text, similar to XML. The response conforms to the Sensor Markup Language (SenML), which the IPSO Alliance application framework refers to [36].

```
{"e":[{"n":"urn:dev:mac:1234567890123456:temp","v":30.6250}]}
```

It is an element (`e`), with two fields, name (`n`) and value (`v`). The name is a URN (`urn`), that specifies a device (`dev`) that is identified by its MAC address (`mac`) `1234567890123456` and the resource is a temperature sensor (`temp`). The measured value of the resource is `30.6250`.

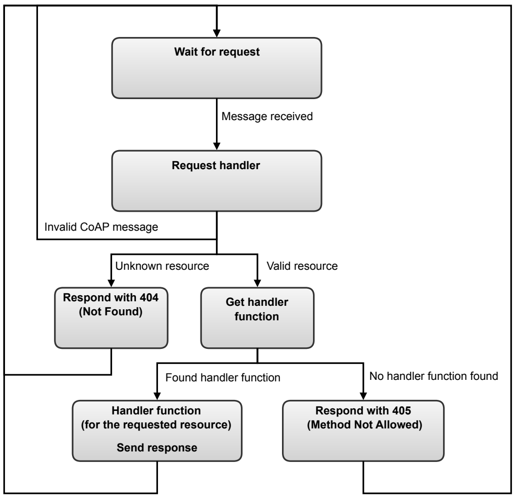


Figure 4.3: Flowchart of server request handling.

The server does also respond to GET requests of the `/.well-known/core` URI. It is a standardised URI for service discovery, which can be used to discover and retrieve information about available resources [37, 38]. The response returned to the client includes all resources and their attributes. The resources and capabilities of the node are described according to the IPSO Alliance application framework. This makes it possible for other devices that confirm to the same guidelines to interface the resources hosted on our node.

4.2.3 Gateway

The gateway consists of two parts: the border router and the BeagleBone. The path a request takes through the gateway is shown in Figure 4.4.

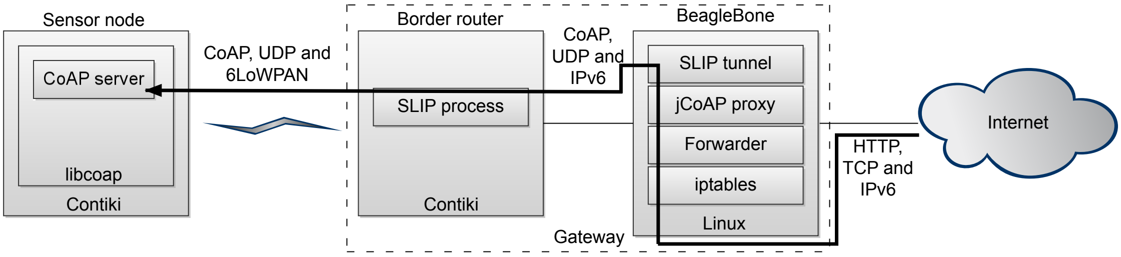


Figure 4.4: Path of a request through the gateway.

As mentioned previously, the BeagleBone part of the gateway runs Arch Linux as its operating system. It was chosen because the authors have previous experience with it and because it offered pre-compiled code for all kernel modules required. More specifically, we used the Linux iptables to route traffic between the connection to the Internet and the wireless sensor nodes. The gateway also runs a transparent proxy that translates between CoAP and HTTP. When traffic arrives on the network interface connected to the Internet, iptables will redirect all traffic on HTTP port 80 to the Forwarder. All other traffic will be handled as normal.

The Forwarder is a small C-program written by us. It simply forwards all connections from one socket to another. In this case it forwards all traffic on port 1234 to port 8080, the latter is the port used by the jCoAP proxy. This might seem like an unnecessary redirection, but it is required. We want our proxy to be transparent; it should accept packets that has another destination than the host the proxy is running at. Normally, sockets in Linux will silently drop such packets, but it can be told to accept them by setting a socket option called `IP_TRANSPARENT`. This option can, however, not be set on sockets from within Java. By using our forwarder we can accept all traffic, and then open a local connection to the proxy, which allows it to receive packets using the standard socket options.

jCoAP is a Java library for CoAP, which also includes an implementation of a proxy that translates between HTTP and CoAP. When an HTTP request arrives, it parses it and creates a new CoAP request which it sends to the node over the SLIP tunnel interface. When the sensor node sends back a response, a translation in the opposite direction, from CoAP to HTTP, is performed. jCoAP does not support the latest CoAP version, but we have modified the source so it is compatible with `draft-ietf-core-coap-13`, the same version libcoap uses.

The SLIP (Serial Line over IP) tunnel is a part of Contiki. SLIP is used to encapsulate IP packets and send them over a serial link. The application will connect to a serial port and create a virtual network interface which can be used by Linux as any other network interface.

The serial interface is the connection between the two parts of the gateway. The border router runs on identical hardware as the sensor nodes, but are connected with a USB-dongle such that a serial interface can be accessed.

On the border router Contiki is used as operating system. The border router has two interfaces: a serial interface and a wireless radio. By running a special SLIP process, packets can be routed between the two interfaces. The outcome is that packets received over the SLIP tunnel can be forwarded and sent wirelessly. The opposite direction is also possible: data received on the wireless interface will be forwarded and sent over the SLIP tunnel to the BeagleBone.

The border router is also responsible for converting between IPv6 and 6LoWPAN. IPv6 is used in almost every communication link, but the final step from the border router to the sensor node uses 6LoWPAN. The conversion is handled automatically by Contiki.

4.2.4 Development tools

During the development we have compiled Contiki on Ubuntu Linux 12.10. Compilation of Contiki for the CC2530 has been done with `SDCC ncs51 3.1.1 #7100` using the `huge-model` for the 8051 architecture. SDCC has been built according to the instructions in [39].

4.3 Difficulties during implementation

All implementation has not been straightforward. In this section we discuss some issues that we had and how we dealt with them.

4.3.1 Call stack

The lack of call stack space has caused a lot of trouble during the whole project. The reason is that the 8051 architecture uses an 8-bit stack pointer, which results in a maximum stack size of 256 bytes. However, the first 33 bytes of the stack are reserved and cannot be used. This reduces the maximum stack size even more, down to 223 bytes.

The standard code base of Contiki is not very well suited for the CC2530 platform, due to the limited stack size. It causes very frequent stack overflows, which result in crashes and unexpected behaviour.

There is however a branch of Contiki, contiki-sensionde, with stack usage optimisations for the 8051 architecture [35], which we have used instead. These optimisations reduces the stack usage a lot, and makes it possible to run Contiki on 8051 based platforms [28].

Whenever a function call is made, memory is allocated at the stack and is later regained when the function returns. The amount of memory that becomes allocated when a function is called depends on: where the function is located in the ROM; the number of arguments and the size of their data types; and local variables within the function and their sizes. This gives us several methods to reduce the stack pressure:

Avoid function calls The stack usage for passing arguments and for executing a function call, can be saved by defining the function as inline and thereby avoiding a function call. Even though some stack space can be saved by declaring functions as inline, it may also increase the total stack usage. The local variables from the inlined function will be moved to the calling function, which will increase the stack allocation of the calling function. Therefore this method should be used with care and preferably only be applied for nested function calls. The ROM memory footprint will increase if an inline declared function, is called from more than one location in the code.

Use as few function arguments as possible Another method for saving stack space is to reduce the number of arguments passed to a function. This can save a lot of stack space, especially if the arguments are of large data types. Sometimes an argument is passed along several nested function calls. Then it might be possible to declare a global variable that can be assigned to the argument, instead of passing it to each new function call. This can result in code that is harder to understand and maintain, and should therefore be used with care.

Use as few local variables as possible Variables can be allocated with either automatic storage duration (on the stack) or with static storage duration (in RAM). There is no support for dynamic allocations in our implementation. The stack usage can be reduced by declaring local variables as static. Since the RAM is also limited (7936 bytes), it is not possible to declare all local variables as static. The build tools will generate errors if we are using more RAM than available. This is in contrast to the stack usage, which the build tools does not analyse at all. Since we cannot fit all local variables into RAM, we have to decide which to store in RAM and which to store on the stack. First off all, variables in functions that has to be reentrant cannot be moved to RAM. Variables that end up near the bottom of the stack, variables that are used in functions that are frequently called and large variables or large arrays, should preferable be stored in RAM.

Do not use larger data types than necessary By not use larger data types than necessary, unused memory resources wont be wasted. This advice is applicable for all parts of the code, but will affect different memory areas depending of where the variables are declared. As told before, the stack is affected by local non-static variables and by function arguments.

In summary it is possible to reduce the stack usage, but some of the optimisations require consideration and should be used with certain caution. Some of the improvements may cause reduced stack usage at the cost of increased usage of other memory resources. The code can also become harder to understand and maintain. It is always important to be restrained with the memory resources and not to use more memory than required.

4.3.2 Development tools

SDCC has been used to compile and link the source code. Usually connectBlue uses the IAR embedded workbench for Microsoft Windows as development environment for the CC2530 chip. Therefore we made some efforts in building Contiki with the tools from IAR embedded workbench, without any success.

The use of SDCC implies several drawbacks. It has no support for hardware debuggers, which is very desirable. Another missing feature is support for mixed declarations and statements, since some parts of Contiki, as well as code from other projects, expects the compiler to support this.

We also experienced that bitfields caused strange behaviour and did not work as intended. This forced us to rewrite some parts of libcoap that used bitfields.

Using the `inline` keyword does not produce the expected outcome either. If a static function, used only at a single place in the code, is declared as inline, not only will the code of the function be inlined, but a callable function will also be generated. This almost doubled the required RAM and ROM usage of the function. Instead of inlining functions we made macros out of some functions, and we also made copy-and-paste of some other functions.

We believe that some of these problems would have been avoided if we were able to use IAR embedded workbench.

Chapter 5

Evaluation of the implementation

As described in the introduction, there are several aspects of the implementation that must be examined. In this chapter we will examine memory usage, energy consumption and packet size.

We will first discuss the evaluation methods chosen, and later present the actual results.

5.1 Methods

Before any measurements are performed, we have to decide and discuss what methods to use to measure the performance. This section will discuss our choice of methods.

5.1.1 Memory usage

The memory usage is a central part of this master's thesis. We want to measure the memory cost of the Internet of things functionality. The results can be used not only to compare with other systems, but also to specify memory requirements when designing a new product.

There are mainly three different memory types that we want to measure: RAM, ROM and call stack. The stack is actually a reserved part of RAM, thus we will further on refer to RAM as the area of RAM excluding the reserved stack area.

We want to measure the memory requirements for the CoAP server and also for both of the RDC drivers: nullrdc and ContikiMAC. A minimal, non-functional, RDC driver implementation, called stubrdc, will be used as a reference to measure nullrdc and ContikiMAC.

ROM usage

We want to measure the ROM usage of Contiki without any RDC driver or additional applications. The ROM usage will also be measured for the CoAP server application, the nullrdc RDC driver and the ContikiMAC RDC driver. SDCC will be used to measure the ROM usage. When the project is compiled with SDCC, the total ROM usage is presented. All debugging functionality such as printouts and logging will be disabled during the measurements.

RAM usage

The RAM usage will also be measured. We will use the same method for measuring RAM usage as described for ROM usage above, since SDCC also presents the total RAM usage.

Stack usage

As described in Section 4.3.1, the stack has caused a lot of problems. We want to evaluate the stack usage of two different versions of the CoAP server, one without any optimisations and one that has stack optimisations.

It is a lot harder to measure the call stack usage, compared to measuring ROM and RAM usage. We will select certain functions in the code where the stack pointer will be printed to the UART. The checkpoints will be placed in functions that are used when a CoAP GET request is received and handled. We can compare the results between the different versions to get an indication of how much the stack usage differs between the versions. However, this is not a very accurate and deterministic method for measuring the call stack usage, since the results are affected by other processes and events.

The maximum reached stack depth will also be printed whenever the stack pointer is printed, to be able detect possible stack overflows. The non-optimised version of the CoAP server is only able to handle GET requests for unknown resources and respond with code 404 (Not found). Because of this we will send a GET request for an unknown resource when the measurements are performed.

A better method for measuring stack usage would be to use a hardware debugger for monitoring and debugging the call stack. This is not possible, since we have not been able to use a debugger together with the development environment.

It is also possible to analyse the code and to calculate a theoretical call stack usage, or even better to do this for the generated instructions. Since the code is very complex and hard to analyse manually, we decide to not use this method.

5.1.2 Energy consumption

The energy consumption of the nodes is a very important result. We have previously discussed that nodes should be able to be battery powered, with a battery life in the range of months or even years. To calculate the estimated battery life, we want to measure the average current consumption.

The nodes should be connected as in Figure 5.1. The power source is a USB-powered debugger device by Texas Instruments. A shunt with a resistance of 1.1 Ω is used, and an oscilloscope and a voltmeter can be connected over the shunt to measure the voltage over the shunt. Using Ohm's law, the current used by the node can be calculated as: $I = U_{shunt}/R$.

There are two main sources of power consumption in the hardware: the radio transceiver and the CPU. The hardware also has some other components, such as temperature sensors and voltage regulators. We choose to include these in the same category as the CPU, since it is hard to measure them independently. We expect the radio transceiver to affect the energy consumption the most, so we want to measure the radio's part of the total power consumption.

To do this we have set up a number of different test cases:

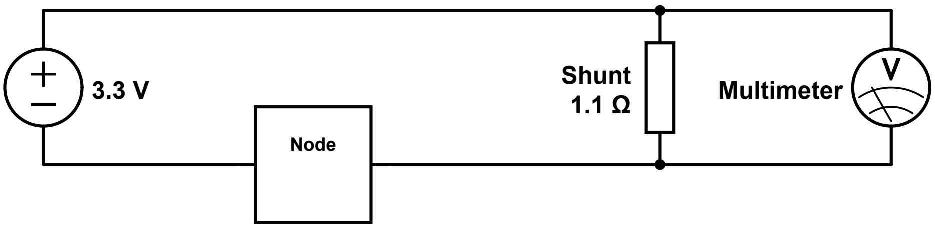


Figure 5.1: Energy measurement

1. CPU active, radio listening 100 % of the time (using Contiki's nullrdc RDC driver)
2. CPU active, radio turned off completely (using Contiki's nullrdc RDC driver, but disable the turn-on function of the radio)
3. CPU active, radio using ContikiMAC with a check rate of 2 Hz

Using test cases 1 and 2 we can calculate the radio's effect on the total consumption. We expect the radio's share to be a very significant part. Using test case 3 we can verify that the power consumption is lowered by using a radio duty cycling protocol.

The actual radio duty cycle (percentage of time the radio is turned on) can be estimated by using Contiki's built-in energy estimation features [40]. It works by keeping track of for how long the radio has been turned on, in terms of CPU clock ticks. By counting the total number of clock ticks the node has been turned on, an estimate of the radio duty cycle can be calculated. The current number of ticks can be sampled periodically, so that the radio duty cycle can be measured over time.

In the default configuration, the energy estimation data is printed over the UART. However, if we wish to measure the real energy consumption we cannot keep the node connected to the computer all the time. The USB and FTDI components require extra power, and our measurements would be more inexact. This means that the node must run independently with only a power source attached. Since we still need to get the samples from the node, they need to be sent wirelessly. Sending data at every sample is impossible, since we also want to be able to measure on a completely idle node. Our solution is to store all sampled data on the node until the test is finished. We then request each sample using a simple protocol based on UDP. This solution requires a significant amount of RAM on the node, which limits the maximum length of a test to around 10 min.

When the radio is constant off, the node cannot send any energy estimation data. Therefore we have performed this test in two parts: first we measure the average current consumption, and then we connected the node to the computer with USB and performed the test once again, this time saving the energy estimation data.

Even if the node does not send any application level data, the underlying network layer send routing information periodically. Therefore we want to measure the energy consumption both when the node is idle (no application layer data is sent) and when some data is sent. Due to the memory restrictions on the node, we cannot make the energy measurements using libcoap together with ContikiMAC. Instead we choose to send periodic ICMP PING messages to the node to simulate application level data. Our different tests are:

Idle test The node and the border router are restarted simultaneously. A sample is taken every 5 s for a total of 10 min (120 samples). No application level data is sent from or to the node.

Ping test The node and the border router are restarted simultaneously. A sample is taken every 5 s for a total of 10 min (120 samples). After waiting 2 minutes, 60 ping messages are sent to the node from the computer, passing the border router. One message is sent every second, with a packet size of 18 bytes. After this we wait one minute before we send the exact same ping sequence again. Finally, during the remaining five minutes of the test, no application data is sent.

To make the tests as reproducible as possible the sequences are scripted using Makefiles. However, the reboot of the border router and the node is done manually by pressing the reset buttons at the same time as the script is started.

The CPU's energy usage should also be possible to lower by using low-power modes supported by the CPU. We have not been able to get them working correctly together with Contiki, which makes it impossible for us to measure potential savings.

5.1.3 Packet size

The main reason for choosing CoAP over HTTP was the reduced overhead in both the application and transport layer. Furthermore, we chose to use 6LoWPAN instead of regular IPv6 to further reduce the overhead. We want to evaluate this choice to see if the overhead really is reduced.

We will do this by looking at a single request-response cycle, using two computers and a node. To perform the measurements, we connect the border router and a wireless IEEE 802.15.4 sniffer to one of the computers. On the same computer we also run the CoAP-HTTP proxy, the forwarder and a packet capturing application. We will use Wireshark to capture packets from the three different interfaces: the SLIP tunnel, the wired Internet connection and the wireless sniffer. In this way, we can follow the same request and response traversing the three different interfaces, and examine the packets' contents. Finally, we use the second computer to generate the HTTP request, using Mozilla Firefox as the client.

This will give us the following protocol combinations:

- IPv6, TCP and HTTP. A GET request is sent from the web browser to jCoAP.
- IPv6, UDP and CoAP. jCoAP translates the request from HTTP to CoAP and forwards it to the border router.
- 6LoWPAN, UDP and CoAP. The border router sends the request to the sensor node over the wireless link, compressing the IPv6 header using 6LoWPAN.
- 6LoWPAN, UDP and CoAP. The node handles the request and sends the response wirelessly to the border router.
- IPv6, UDP and CoAP. The border router decodes the 6LoWPAN header and sends the response to jCoAP.
- IPv6, TCP and HTTP. jCoAP translates the response from CoAP to HTTP and forwards it to the web browser.

In this measurement we are only interested in comparing the sizes of the network, transport and application layers. Therefore we have removed any link layer headers to get comparable results.

Since TCP is a connection-oriented protocol, it will also require a connection to be setup, closed and that all data received is acknowledged. Some CoAP requests are also acknowledged. We choose to ignore the connection setup and tear-down of the TCP connection when comparing packet sizes. We also ignore stand-alone acknowledgement packets. It is however important to remember that there are some more data sent not included in our comparison.

5.2 Results

This section presents all the results from our measurements, according to the methods described earlier. All results come from our modified version of Contiki and libcoap.

5.2.1 Memory usage

ROM usage

The ROM usage of CoAP server is shown in Table 5.1. Table 5.2 displays the ROM usage of the different RDC drivers. The tables also contains the percentage of the 262 144 bytes of total ROM available.

Table 5.1: ROM usage, with and without CoAP server.

	ROM	Δ ROM	of total ROM
Without CoAP server	113 209 B	0 B	43.2 %
With CoAP server	191 850 B	78 641 B	73.2 %

Table 5.2: ROM usage of the different RDC drivers.

	ROM	Δ ROM	of total ROM
stubrdc	113 209 B	0 B	43.2 %
nullrdc	115 100 B	1891 B	43.9 %
ContikiMAC	126 847 B	13 638 B	48.4 %

RAM usage

The RAM usage of CoAP server is shown in Table 5.3. The RAM usage requirements for the different RDC drivers can be found in Table 5.4. The tables also displays the percentage of the 7936 bytes of total RAM available.

Table 5.3: RAM usage, with and without CoAP server.

	RAM	Δ RAM	of total RAM
Without CoAP server	5525 B	0 B	69.6 %
With CoAP server	7763 B	2238 B	97.8 %

Table 5.4: RAM usage of the different RDC drivers.

	RAM	Δ RAM	of total RAM
stubrdc	5525 B	0 B	69.6 %
nullrdc	5605 B	80 B	70.6 %
ContikiMAC	5775 B	250 B	72.7 %

Stack usage

We measured the call stack usage for two different implementations of the CoAP server: one without any optimisations, referred to as version 1, and the other one with call stack optimisations, referred to as version 2. We inserted checkpoints at ten different locations in the code, in both versions. The measured stack pointer values are shown in Table 5.5 and plotted in Figure 5.2. Note that the stack grows from lower addresses towards higher addresses, in the range 0–255. The maximum stack pointer value for version 1 was 255, which occurred between checkpoint 7 and 8. For version 2 the maximum stack pointer value was 221, which also occurred between checkpoint 7 and 8.

Table 5.5: Stack pointer values at different measurement points.

	Checkpoint									
	1	2	3	4	5	6	7	8	9	10
CoAP server v1	74	119	123	169	172	208	220	236	232	136
CoAP server v2	74	135	125	133	136	172	184	200	196	136

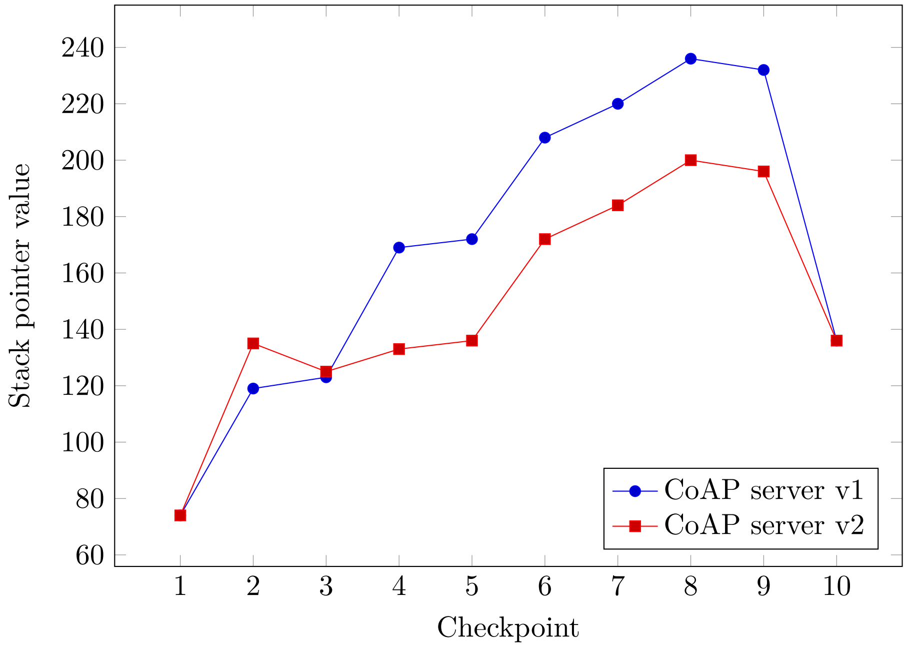


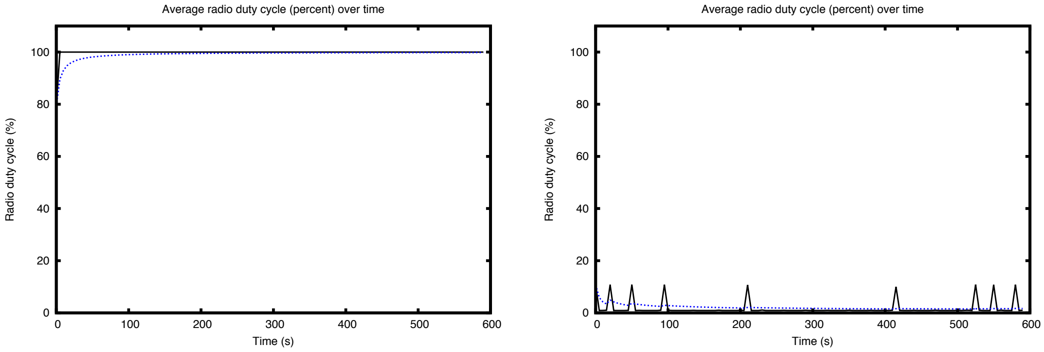
Figure 5.2: Graph showing the stack usage for two different CoAP server implementations.

5.2.2 Energy consumption

In Table 5.6 the average current consumption of a node is shown for the Idle test. The node is running and the border router is within range. This means that the only data being sent and being received by the node is routing data required to create and maintain the connection. When the radio is constant off, the node will not be able to communicate and cannot create a connection at all. The average RDC have been calculated using the data from Contiki's energy estimation feature. The graphs of the corresponding energy estimation can be seen in Figure 5.3.

Table 5.6: Energy consumption and average RDC for different radio duty cycle protocols running the Idle test.

RDC protocol	Average RDC	Average current consumption
Radio constant on	99.8 %	26.9 mA
Radio constant off	0 %	8.3 mA
ContikiMAC	1.7 %	8.5 mA



(a) Radio constant on

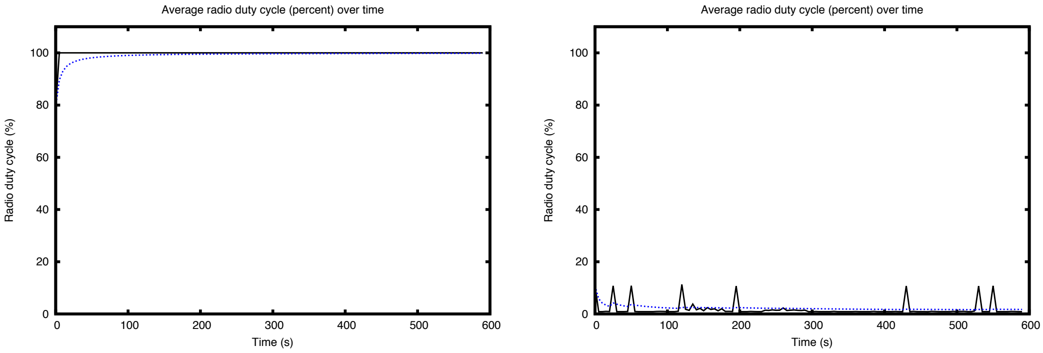
(b) ContikiMAC

Figure 5.3: Energy estimation graphs when running the Idle test. Solid black line is RDC since the last sample. Blue dashed line is the moving average RDC.

We have also measured the energy usage and radio duty cycle when data is being sent over the link using the Ping test. There is no point in sending data when the radio is constantly turned off, so this case has been ignored. The results can be seen in Table 5.7 and Figure 5.4.

Table 5.7: Energy consumption and average RDC for different radio duty cycle protocols running the Ping test.

RDC protocol	Average RDC	Average current consumption
Radio constant on	99.8 %	26.9 mA
ContikiMAC	1.7 %	8.5 mA



(a) Radio constant on

(b) ContikiMAC

Figure 5.4: Energy estimation graphs when running the Ping test. Solid black line is RDC since the last sample. Blue dashed line is the moving average RDC.

5.2.3 Packet size

We have extracted the data from Wireshark to give a visual representation of the different parts in the packets and how their sizes are affected by the compression achieved by the various protocols. The different parts and their respective colour encodings are: network layer, transport layer and application layer. The request packets can be seen in Figure 5.5 and the response packets in Figure 5.6. The size of the packets and the layers can be seen in Table 5.8.

Table 5.8: The size of the packets and their parts, expressed in bytes.

	Request			Response		
	HTTP	CoAP IPv6	CoAP 6LoWPAN	CoAP 6LoWPAN	CoAP IPv6	HTTP
Network	40	40	21	12	40	80
Transport	32	8	8	6	8	64
Application	303	15	15	69	69	227
Total	375	63	44	87	117	371

```
|‘....O.@........|
|................|
|..K..:y....P1...|
|{..w.. ..6......|
|..*~...]GET /sen|
|/temp HTTP/1.1..|
|Host: small-node|
|..User-Agent: Mo|
|zilla/5.0 (Macin|
|tosh; Intel Mac |
|OS X 10.8; rv:15|
|.0) Gecko/201001|
|01 Firefox/15.0.|
|1..Accept: text/|
|html,application|
|/xhtml+xml,appli|
|cation/xml;q=0.9|
|,*/*;q=0.8..Acce|
|pt-Language: en-|
|us,en;q=0.5..Acc|
|ept-Encoding: gz|
|ip, deflate..Con|
|nection: keep-al|
|ive....|
```

```
|‘......@........|
|................|
|..K..:y....3....|
|@....sen.tempQ)|
```

```
|x...?..........c|
|........3....@..|
|..sen.tempQ)|
```

Figure 5.5: The request part of the request-response cycle. From the left: HTTP request, CoAP and IPv6 request, and finally CoAP and 6LoWPAN request.

```
|~............3..|
| A‘E...!..{"e":[|
|{"n":"urn:dev:ma|
|c:12345678901234|
|56:temp","v":30.|
|6250}]}|
```

```
|‘....M.?........|
|..K..:y.........|
|.........3...M A|
|‘E...!..{"e":[{"|
|n":"urn:dev:mac:|
|1234567890123456|
|:temp","v":30.62|
|50}]}|
```

```
|‘......@........|
|..K..:y.........|
|.........P..{..w|
|1......x........|
|......*~HTTP/1.1|
| 200 OK..Content|
|-Type: text/plai|
|n..Content-lengt|
|h: 61..Date: Wed|
|, 13 Mar 2013 10|
|:44:51 CET..Expi|
|res: Wed, 13 Mar|
| 2013 10:44:52 C|
|ET..Connection: |
|keep-alive....|
|‘....].@........|
|..K..:y.........|
|.........P..{...|
|1......x........|
|......+.{"e":[{"|
|n":"urn:dev:mac:|
|1234567890123456|
|:temp","v":30.62|
|50}]}|
```

Figure 5.6: The response part of the request-response cycle. From the left: CoAP and 6LoWPAN response, CoAP and IPv6 response, and finally HTTP response.

Chapter 6
Discussion

We will first discuss and analyse the results of the measurements from the previous section. Later our implementation's conformance to the standards we have chosen will be analysed, and finally the security aspects of our implementation will be discussed.

6.1 Memory usage

As described in Section 4.3, memory usage has been an issue during the implementation. In this section we evaluate the various memory figures: ROM, RAM and call stack usage.

6.1.1 ROM usage

The results of the ROM usage measurements can be seen in Table 5.1 and Table 5.2. From the first table, we can see an increase in ROM usage of 78 641 bytes when the CoAP server is enabled. This corresponds to an increase of 69.5 %. The footprint of the compiled code consumes 73.2 % of the total amount of ROM.

The second table shows an increase in ROM usage of 1891 bytes for nullrdc, and an increase of 13 638 bytes for ContikiMAC. None of the RDC protocols requires a lot of ROM when compared to the rest of Contiki. The amount of occupied ROM is however about 7 times more for ContikiMAC compared to nullrdc.

We can conclude that 205 488 bytes of ROM is required to be able to fit Contiki with both ContikiMAC and CoAP. This means that 56 656 bytes of ROM will left for user specific applications.

6.1.2 RAM usage

The results from the RAM usage measurements with and without the CoAP server can be found in Table 5.3. It shows that the RAM used by the base functionality of Contiki is 5525 bytes, that corresponds to 69.6 % of the total available RAM. When the CoAP server is enabled, the RAM usage increases with 2238 bytes, which results in 97.8 % of the total RAM being occupied.

In Table 5.4 the RAM usage for the different RDC drivers is presented. Enabling nullrdc caused an increase of the RAM usage with 80 bytes. For ContikiMAC the RAM usage increased with 250 bytes, which is slightly more than 3 times that of nullrdc.

As we can see, a lot of RAM is being used by Contiki, but the CoAP server does also consume a lot of RAM. This can be derived to stack optimisation, where local variables in functions has been moved to RAM to reduce stack pressure. Therefore a much lower RAM usage can be expected for implementations with more stack space.

To run Contiki with both ContikiMAC and CoAP server, 8013 bytes of RAM is required. Since the chip is only equipped with 7936 bytes of RAM, this is not possible. Even if it would fit into RAM, it would probably require more stack than available, since both CoAP server and ContikMAC requires a lot of stack space.

6.1.3 Stack usage

In order to measure the impact of stack optimisations, we made some measurements. We used two different versions of the CoAP server, one without any optimisations and one with stack usage optimisations.

When we first integrated the CoAP server into our project, it could only handle CoAP GET requests for unknown resources, due to stack overflows. We then applied the stack optimisations described in Section 4.3.1, and were able to get it functioning properly.

The results of the measurements can be found in Section 5.2.1. It shows that more stack memory was used in the optimised version at the early checkpoints (2 and 3) where the stack usage was low, and later when the stack usage increased even more, the non-optimised code used more stack. This shows that the applied optimisations improved the stack usage for the optimised implementation.

The maximum stack value of the optimised version was 221, which implies that no stack overflow has occurred. However in the non-optimised version, the maximum possible stack value 255, was reached. We cannot tell if the stack overflowed or not, since the node did not crash. We did however observe strange behaviours from the node, causing the response to be resent, even though it was successfully received the first time. This indicates that a stack overflow probably occurred.

The ContikiMAC RDC driver required a lot of stack space, which at first made it impossible to use without getting stack overflows. We had to make several stack optimisations to be able to use ContikiMAC without any stack overflows.

We have not made any approximation of how much stack that would be required for this kind of system. To make the code run with the current stack space, a great deal of optimisations are required. These optimisations can make the code harder to understand and to maintain. In addition, it is most probably very hard to fit both a good RDC protocol and CoAP server so that they can work together. Our conclusion is that 223 bytes of stack is not enough due to these reasons.

6.2 Energy consumption

In Section 5.2.2 we presented the results of the energy consumption measurements. Starting at the Idle test, looking at Table 5.6 and Figure 5.3 we can see that the Contiki energy estimation feature seems to be working. When the radio was constantly turned off, a radio duty cycle of 0 % was reported, just as expected. With the radio constantly turned on, a duty cycle of 99.8 % was reported. Looking at Figure 5.3(a) we can see that the first sample does not have a duty cycle of 100 %, which causes the average RDC to be slightly below 100 %. This is because the radio is not turned on until other initialisations have been performed in Contiki.

Assuming the current consumption of the radio is proportional to the duty cycle, we can calculate the expected average current consumption $\overline{i_D}$ for a duty cycle of D as:

$$\overline{i_D} = \overline{i_0} + \frac{\overline{i_{0.998}} - \overline{i_0}}{0.998 - 0} \times D$$

where $\overline{i_{0.998}}$ is the average current consumption of an RDC of 99.8 %, $\overline{i_0}$ the average current consumption of an RDC of 0 %, and 0.998 and 0 are duty cycles.

Calculating the expected average current consumption for an RDC of 1.7 % gives that $\overline{i_{0.017}} = 8.6\,\text{mA}$. Our measured value was 8.5 mA, which is slightly lower, but still the energy estimation mechanism seems to be reliable.

Continuing with the Ping test, we see from Table 5.7 and Figure 5.4 that the average RDC and current consumption remain the same as for the Idle test. This is not surprising when the radio is constantly turned on, since in this case the current consumption should be independent of the radio utilisation. In the case of ContikiMAC the RDC and current consumption should increase with a higher radio utilisation. Looking at Figure 5.4(b) we can clearly see that the RDC is higher during the ping intervals 120 s–180 s and 240 s–300 s. The RDC during these intervals is quite low, rarely over 2 %, which does not affect the total average RDC much.

In Figure 5.4(b) there is a significant difference in RDC during the sending of ping responses and the sending of RPL broadcast messages (the spikes with RDC ≈ 11 %). The reason is that broadcast messages need to be sent during the whole 0.5 s interval, so that all possible receivers' wake-up intervals are covered. When sending a ping response, the message is destined for the border router, which is constantly awake. As soon as it receives the ping response and acknowledges it, the node can stop sending and turn off the radio again. As we can see, this reduces the radio utilisation significantly.

After verifying that the current consumptions seems to be correct, we can estimate the lifetime of our node assuming they are running on batteries. As mentioned earlier, a typical CR2032 button cell battery has a capacity of 230 mAh. Using ContikiMAC, this would give a disappointing lifetime of 27 h. Even if the button cell battery is replaced with AA batteries with a capacity of 2300 mAh, the lifetime would be in the range of 1.5 weeks. Since these kind of nodes need a lifetime of months, or even years, this makes our current implementation unsuitable.

Even with the radio completely turned off, we had a current consumption of 8.3 mA. This means that the radio uses only 2 % of the total current in the case of ContikiMAC. The CC2530 chip has support for three different levels of power saving, which reduces the current consumption to 0.2 mA, 1 µA and 0.4 µA respectively [41]. During sleep the radio and the CPU are turned off and can not be used. We have not managed to get these power saving modes to work correctly. Also, these values do only indicate the current consumption of the SoC. However, as mentioned in the hardware section, our node consists of more hardware than the CC2530, all of which consume some extra current.

In an ideal world, the only current drawn from our node should be the current drawn by the use of the radio. This means that the theoretical lowest possible average current consumption we could get from our node would be 0.2 mA. This is the ContikiMAC consumption with the zero percent duty cycle consumption subtracted. However, even under these ideal conditions, the standard CR2032 battery of 230 mAh would only have a lifetime of 48 days. This means that either a redesigned radio duty cycling protocol, or a chip with lower radio current usage is needed.

6.3 Packet size

The results from Table 5.8 shows that CoAP with 6LoWPAN reduces the overhead significantly. The request is reduced from 375 B to 44 B, a compression ratio of 0.12. The response has a compression ratio of 0.23.

There are a couple of things to notice about the data. First, we see that the CoAP-HTTP proxy reduces the application layer data with 95 % for the request. CoAP does compress HTTP headers by using a binary format, instead of a textual representation used in HTTP. This is only a part of the explanation. A lot of data is saved by simply ignoring some HTTP headers, which are of no importance in CoAP. As an example, the following headers from the HTTP request in Figure 5.5 are not present in the CoAP request: `User-Agent`, `Accept-Language`, `Accept-Encoding` and `Connection`. The remaining headers are compressed to the binary CoAP format.

Looking at the request, we can also see that 6LoWPAN can compress the network layer to almost half the size of IPv6, a significant improvement. Moving from TCP to UDP also lower the size required on the transport layer by a factor four.

Looking at the response, we can see that both the network and transport layers have an even lower size for CoAP/6LoWPAN compared to the request, even though the exact same protocols are used. This is because 6LoWPAN uses UDP compression in the response, which means that UDP information is included in the 6LoWPAN header. This also removes the UDP length field, which can be calculated from the link-layer size instead. This reduces the transport layer data from 8 to 6 bytes. Note that there is no real UDP header in the data, but we have counted the bytes used for UDP header compression in the 6LoWPAN header as the transport layer size.

We do not fully understand why the UDP header compression is not performed on the request. We believe the reason is that the border router only act as a router, and that Contiki does not compress the network layer when packets are routed from the SLIP interface to the wireless interface. When the response is created on the node, a completely new packet is generated, which then triggers the UDP header compression.

Looking at the HTTP response, we see that the network and transport sizes are doubled compared to the HTTP request. This is because the response is sent in two separate packets, each having one IPv6 header and one TCP header. The application layer data is actually smaller in the response than in the request, so there should be no need to split the packet because of this. Looking at the packet dump in Figure 5.6 we see that the application data is split right between the HTTP headers and the HTTP body. This seems to be done by the HTTP library used by the jCoAP proxy, for reasons unknown to us. Since the HTTP request is only sent on the Internet, these extra network layer and transport layer headers should not be a problem.

As mentioned in the methods, extra TCP overhead is not included in these figures. For every HTTP connection a TCP connection has to be established and later terminated, and the packets in the connection have to be acknowledged. As an example, a TCP connection establishment requires three TCP packets, all of which would require their corresponding network layer headers. If a connection has to be established for every request, this incurs a lot of extra data having to be sent and received.

We think that the CoAP conversion is a significant improvement, since it reduces the load on the nodes. The amount of data that the node needs to receive and send is much lower than if HTTP would have been used. Using UDP also reduces the memory consumption on the node, since no TCP state has to be recorded in memory.

6.4 Conformance to standards

In our solution we use several different standards and RFCs. Since interoperability is an important aspect in the Internet of Things, it is important that an implementation conforms to the standards used. This section describes features that are missing from our implementation, which makes it incompatible with different standards. Some standards, particularly CoAP, includes a wide range of optional features. Such missing optional features are not included here.

The maximum packet size for IEEE 802.15.4 is 127 bytes. However, IPv6 requires that the underlying layers support packets of at least 1280 bytes [42]. This means that link-layer fragmentation support must exist. Contiki does support fragmentation of packets, but due to memory limitations, we cannot support both CoAP and fragmentation at the same time.

Apart from being being non-standard, this also limits the use cases of the nodes. A maximum packet size of 127 bytes means that we can send a maximum of 71 bytes of CoAP data in one packet. This is especially an issue for service discovery, since it is only possible to fit a single resource in the `/.well-known/core` resource listing. If every node should be able to supply different resources, for example one temperature sensor and one humidity sensor, fragmentation support would be required to be able to use the service discovery features.

6.5 Security

Our solution has no built-in security at all. This means that anyone could eavesdrop, forge or access the node's resources. One of the main reasons for this lack of security is that there is no built-in support for any security in the Contiki operating system. This section is therefore devoted to describing possible attacks, and possible solutions, together with their feasibility for resource limited nodes.

Traditionally, sensor networks and other small items have never had a direct connection to the Internet. The wireless networks have been isolated, without contact with the outside world. When building the Internet of things it may be easy to move rapidly, but we must also take a step back and consider the security of the devices.

There are several situations where security of the data can be important. For example, in the area of healthcare the privacy is important. A patient using medical equipment that connects to the Internet may not want his or her data to be accessed by unauthorised people. It is important that the developers of the system make the users aware of possible limitations in their implementation, such that users can make informed decisions about the data sent on the network.

There are several security issues that must be considered when connecting new devices to the Internet. Considering a small sensor node, one might want to ensure that only certain people have access to the sensor's value over the Internet. This of course requires support for authentication and authorisation, but it will most certainly also require confidentiality. There is limited use of requiring authorisation if the data can be heard by an eavesdropper. If we instead consider an actuator, the user will almost certainly want authorisation to avoid that anyone can make changes. However, confidentiality might not be required in this case.

Looking from the other direction: how can the user be sure that the value returned comes from the correct node and that is has not been tampered with during the transmission? This is solved by requiring authentication of the other party and message integrity.

Depending on the security level required, this problem can be solved differently. For some appliances, it may be enough to secure the wireless link between the node and the border router. Others may require end-to-end security.

6.5.1 End-to-end security

A solution to all these problem could be DTLS, Datagram Transport Layer Security [43]. It is an adaption of TLS, Transport Layer Security, to work with unreliable transport protocols such as UDP. It is designed to be as similar as possible to TLS, and supports mutual authentication, confidentiality and message integrity. Authorisation would have to be handled by the application on the node.

There exists previous implementations of DTLS for Contiki, but they are not at all adapted for our platform, and are not officially included in Contiki. In [34] the maximum stack usage is over 2400 bytes, and the amount of RAM required is 1961 bytes. The exact figures would not be the same if ported to CC2530, but it is clear that DTLS is unsuitable for our platform. Furthermore, the implementation in [34] required pre-shared keys, which requires manual key management. A certificate-based solution may be even more costly in terms of resources.

Since DTLS is an end-to-end security protocol, it also requires that CoAP is used all the way from the client to the node. If a CoAP-HTTP proxy is used on the gateway, some sort of mixed solution is required, as described later in Section 6.5.3.

6.5.2 Wireless link security only

So far only end-to-end security protocols have been discussed. If only the wireless link needs to be secured, it can be done on several different layers: the transport layer, the network layer or on the link layer. IEEE 802.15.4 has defined link layer message encryption and integrity using AES-CCM. Unfortunately, there is no defined key management protocol for link-layer security, and one would have to be defined for secure operation. The use of link layer encryption will encrypt the whole MAC payload, which includes all data and IP headers. All traffic from the node will be affected. This may or may not be positive. Some applications on the node might not require encryption or authentication, while some does it. At the link layer there is no way to distinguish these cases.

IPsec could also be used to provide encryption and authentication. Just like on the link layer it will affect all traffic from the node. It is also important to note that there are currently no specific adaptions for IPsec in 6LoWPAN, which means that the IPsec headers cannot be compressed. The implementation [44] proposes header compression for IPsec in 6LoWPAN, and shows that authentication and encryption in IPsec only adds an extra 3 bytes of data compared to link layer authentication and encryption.

Finally transport layer security is also a possibility, using DTLS as described above. This would make it possible for the node to have different services requiring different security features. However, using encryption at this layer just to provide security for the wireless link would require the gateway to act as a proxy, decrypting data at the transport layer and then repackage it and send it to the Internet. This increases the complexity of the gateway.

6.5.3 Mixed

A possible scenario is that we ideally would like end-to-end security, but because we want to use the CoAP-HTTP proxy this is impossible. The reason this is not possible is because the proxy must read and transform the contents of the HTTP and CoAP packets, which is only possible for packets that are not encrypted or authenticated. However, if we already trust the gateway to convert the data, we might as well trust it as a party in our secure communications. If the gateway is trusted, we could use DTLS between the gateway and the node, and ordinary TLS between the gateway and the client.

As previously discussed, DTLS requires significant resources, which may not be available. In the solution of the previous paragraph, DTLS could easily be replaced by another protocol, since this does not affect the client at all. This makes it possible to use a more light-weight encryption and authentication scheme, for example on the link layer or network layer. This reduces the resources required on the node, and still provides security on the Internet as well as on the wireless link.

6.5.4 Other security-related problems

Because of the nodes' limited resources in terms of CPU, memory and network bandwidth, Denial of Service (DoS) attacks are an issue if the nodes are to be access directly from the Internet. The best location to counter these attacks is in the gateway, since it it has unlimited power, and in general better performance [45].

DoS attacks can also be performed by jamming the radio channel. This can disturb the transmissions significantly [46]. This affects IEEE 802.15.4 since it does not use frequency hopping. The jamming attack requires the attacker to be within fairly close range of the wireless network, which makes it more difficult. However, if the network is located in a public space it may still pose a significant threat.

Access control to the wireless network is also an important feature, so that only authorised nodes are able to be a part of the wireless network. IEEE 802.15.4 have some support for this through the use of access control lists (ACLs). The ACLs are used to set the keys used for encryption and authentication, and by requiring all packets to be authenticated non-authorised devices can be kept out from the network.

Chapter 7
Conclusions

The goal of this thesis was to examine possible limitations in the functionality of small, constrained, wireless devices. Our study shows that with our choice of hardware, an implementation will be severely limited in terms of both features and power consumption.

It was not possible to fit both the CoAP server and the energy saving features on the chip at the same time, and even if it were possible, the energy saving protocols are not efficient enough to use small button cell batteries. This is an important result, since it shows that hardware with more memory is not enough; better energy saving protocols or more power-efficient hardware is also required.

The use of CoAP together with 6LoWPAN seems to work well, especially when combined with the proxy at the gateway. It significantly reduces the amount of data sent, which is important for wireless transmissions. An HTTP and TCP implementation would not have been possible to fit on our device, but even if it was possible, the extra resources used by these two protocols may be better spent on the actual user application.

Choosing Contiki as the operating system has simplified the implementation a lot. It includes support for most of the protocols required on every layer. Implementing a network stack on our own would have been a very time-consuming task. Contiki is well-supported, and there is a lot of activity on the mailing list and in their code repository. This is good because it shows that the project is actively developed, and that it is actually used and tested.

To solve the issues described earlier, there are two main solutions: Better, more power-efficient hardware with more memory, need to be used; and more efficient power-saving radio duty cycling protocols need to be developed. Additional future work is required to investigate or develop such hardware and protocols.

7.1 Future of the Internet of things

We think that the future of the Internet of things is bright, even though our implementation did not show any impressive performance. During the project we have seen a lot of activity regarding the Internet of things, and we expect even more activity in the future, which will result in new standards and technologies.

We think that several different communication technologies will be used to achieve the visions of the Internet of things, both wired and wireless. We have used IEEE 802.15.4, but as mentioned in the Background, it exists several other different technologies like BLE and Wi-Fi. BLE can be found in many new smartphones, which may make it suitable for devices you carry with you. Wi-Fi may be used in appliances where the power consumption is not restricted. New technologies will probably also be developed. An example is the emerging standard Weightless, which relies on a dedicated infrastructure of base stations, much like cellular networks, without the need of a border router [47].

We believe that IP will be used by the majority of all devices in the Internet of Things, and that the majority off all wireless technologies will support IP. IEEE 802.15.4 and Wi-Fi already supports IPv6, and there is an on-going effort to support it in BLE as well. In this way the wireless technology can be chosen freely, to suit a specific use-case, without requiring application layer modifications.

The adoption of the Internet of things allows better system integration and reduces the amount of independent parallel systems. More systems can share resources and data, and thereby benefit from each other.

A potential use case for Internet of Things-enabled devices is to reduce energy costs. Equipment that consumes a lot of power could contact the energy supplier and find out at which time the energy price is the lowest. In this way the society utilises the power grid more efficiently. There are probably a lot of other potential areas where Internet of Things will benefit people.

7.2 Future work

There is a lot of work that could be done in possible future implementations. Using our current hardware, there are some improvements that can be made:

- Try to optimise the stack usage even further. This can make it possible to add new features. It should be noted that this probably requires widespread changes in code, with patches that are unsuitable for other platforms. If open-source code is used, maintaining code with lots of patches may be a lot of work.
- Run the border router directly on the BeagleBone by connecting an IEEE 802.15.4-radio directly to it. That way we are not restricted by the limited resources of the current border router, which for example affects the number of maximum connected nodes and maximum number of routes.

Another option is to use different hardware, presumably hardware that has more resources available, especially more stack memory. Texas Instruments has a new chip called CC2538, which has a more powerful CPU with more stack and RAM [48]. There is already a port of Contiki to this platform, and because of the increase of stack and RAM we believe that it would solve several issues with our current implementation.

Our current solution requires clients to connect to the node every time it want to check whether some data has been updated or not. An alternative solution would be a kind of subscription solution, where clients subscribe to different resources and get notified by the server when these resources change. There is an observe extension to CoAP [49], which allows a client to request that the server sends an updated value to the client when such a new value exists. This could be implemented to avoid the client doing periodic polling. The solution could also be combined with a new HTML5 feature, still in draft form, called Server-Sent Events [50], which would allow the subscription based service to be used with HTTP and TCP. This would require such functionality to be implemented in the CoAP-HTTP proxy.

There is also potential to make the nodes cloud-enabled, so that they push data to the cloud, which can then later be fetched by clients. This could potentially reduce the workload on the node, simplify security schemes and reduce power consumption, since the node does only have to send data at its own discretion.

Future work could also focus on other wireless techniques, like Bluetooth Low-Energy or Weightless. They have no support in Contiki, which is adapted for IEEE 802.15.4. This would require some other operating system, or even writing your own.

Bibliography

[1] Verisure. App. [Online]. Available: http://www.verisure.se/en/verisure/products-and-technology/app.html

[2] J.-P. Vasseur and A. Dunkels, *Interconnecting Smart Objects with IP: The Next Internet*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2010.

[3] Passivsystems. Hello smart home. [Online]. Available: http://www.passivsystems.com/Homes/PassivEnergy/HelloSmartHome.aspx

[4] B. A. Forouzan and S. C. Fegan, *Data communications and networking*, ser. McGraw-Hill Forouzan networking series. Boston: McGraw-Hill, 2007.

[5] *IEEE Standard for Local and metropolitan area networks - Part 15.4: Low-Rate Wireless Personal Area Networks (LR-WPANs)*, IEEE Computer Society Std. 802.15.4-2011, 06 2011.

[6] Z. Shelby and C. Bormann, *6LoWPAN: The Wireless Embedded Internet*. Wiley Publishing, 2010.

[7] "CC2420 2.4 GHz IEEE 802.15.4 / ZigBee-ready RF transceiver," Chipcon, 06 2004.

[8] Specification: Adopted documents. Bluetooth SIG. [Online]. Available: https://www.bluetooth.org/Technical/Specifications/adopted.htm

[9] C. Gomez, J. Oller, and J. Paradells, "Overview and evaluation of bluetooth low energy: An emerging low-power wireless technology." *Sensors (Switzerland)*, vol. 12, no. 9, pp. 11 734–11 753, 2012.

[10] *Part 15.2: Coexistence of Wireless Personal Area Networks with Other Wireless Devices Operating in Unlicensed Frequency Bands*, IEEE Std. 802.15.2, 08 2003.

[11] *Specification of the Bluetooth System, Covered Core Package, Version: 4.0*, Bluetooth SIG Std., 06 2010.

[12] C. Gomez, I. Demirkol, and J. Paradells, "Modeling the maximum throughput of Bluetooth Low Energy in an error-prone link," *Communications Letters, IEEE*, vol. 15, no. 11, pp. 1187 –1189, november 2011.

[13] *Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications*, IEEE Std. 802.11, Rev. 2012, 03 2012.

[14] K. Furset and P. Hoffman. (2011) High pulse drain impact on CR2032 coin cell battery capacity. [Online]. Available: `http://www.eetimes.com/ContentEETimes/Documents/Schweber/C0924/C0924post.pdf`

[15] *IPv6 over Low-Power Wireless Personal Area Networks (6LoWPANs): Overview, Assumptions, Problem Statement, and Goals*, IETF RFC 4919, 8 2007. [Online]. Available: `http://tools.ietf.org/html/rfc4919`

[16] U. C. Bureau. (2013) World popclock projection. [Online]. Available: `http://www.census.gov/population/popclockworld.html`

[17] *Transmission of IPv6 Packets over BLUETOOTH Low Energy*, IETF Internet-Draft draft-ietf-6lowpan-btle, Rev. 12, 02 2013. [Online]. Available: `http://tools.ietf.org/html/draft-ietf-6lowpan-btle-12`

[18] R. T. Fielding and R. N. Taylor, "Principled design of the modern web architecture," *ACM Trans. Internet Technol.*, vol. 2, no. 2, pp. 115–150, May 2002.

[19] P. A. Castillo, J. L. Bernier, M. G. Arenas, J. J. M. Guervós, and P. García-Sánchez, "SOAP vs REST: Comparing a master-slave GA implementation," *CoRR*, vol. abs/1105.4978, 2011.

[20] L. Richardson and S. Ruby, *RESTful web services*, 1st ed. O'Reilly, 2007.

[21] *Uniform Resource Identifier (URI): Generic Syntax*, IETF RFC 3986, 1 2005. [Online]. Available: `http://tools.ietf.org/html/rfc3986`

[22] M. E. S. James McGovern, Sameer Tyagi and S. Mathew, *Java Web Services Architecture.* Morgan Kaufmann, 2003.

[23] *Constrained Application Protocol (CoAP)*, IETF Internet-Draft draft-ietf-core-coap, Rev. 13, Dec 2012. [Online]. Available: `http://tools.ietf.org/html/draft-ietf-core-coap-13`

[24] *The IPSO Application Framework*, IPSO Std. draft-ipso-app-framework, Rev. 04, 8 2012. [Online]. Available: `http://www.ipso-alliance.org/downloads/Application+Framework`

[25] T. Savolainen and M. Xi, "IPv6 over Bluetooth Low-Energy Prototype," presented at the Aalto University Workshop on Wireless Sensor Systems, Aalto, Finland, December 11, 2012. [Online]. Available: `http://wsn.aalto.fi/en/activities/wowss2012/btle_paper_aalto_wowss2012.pdf`

[26] *Transmission of IPv6 Packets over IEEE 802.15.4 Networks*, IETF RFC 4944, 9 2007. [Online]. Available: `http://tools.ietf.org/html/rfc4944`

[27] Contiki. [Online]. Available: `http://www.contiki-os.org/`

[28] G. Oikonomou and I. Phillips, "Experiences from porting the contiki operating system to a popular hardware platform," in *Distributed Computing in Sensor Systems and Workshops (DCOSS), 2011 International Conference on*, June, pp. 1–6.

[29] libcoap. [Online]. Available: `http://sourceforge.net/projects/libcoap/`

[30] Arch Linux ARM. [Online]. Available: `http://archlinuxarm.org/`

[31] jCoAP. [Online]. Available: `https://code.google.com/p/jcoap/`

[32] *RPL: IPv6 Routing Protocol for Low-Power and Lossy Networks*, IETF RFC 6550, 3 2012. [Online]. Available: `http://tools.ietf.org/html/rfc6550`

[33] A. Dunkels, "The contikimac radio duty cycling protocol," SICS, Tech. Rep. T2011:13, December 2011. [Online]. Available: `http://soda.swedish-ict.se/5128/1/contikimac-report.pdf`

[34] V. Perelman, "Security in IPv6-enabled wireless sensor networks: An implementation of TLS/DTLS for the Contiki operating system," Master's thesis, Jacobs University, June 2012.

[35] contiki-sensinode. [Online]. Available: `https://github.com/g-oikonomou/contiki-sensinode/`

[36] *Media Types for Sensor Markup Language (SENML)*, IETF Internet-Draft draft-jennings-senml, Rev. 10, Oct 2012. [Online]. Available: `http://tools.ietf.org/html/draft-jennings-senml-10`

[37] *Constrained RESTful Environments (CoRE) Link Format*, IETF RFC 6690, 8 2012. [Online]. Available: `http://tools.ietf.org/html/rfc6690`

[38] *Defining Well-Known Uniform Resource Identifiers (URIs)*, IETF RFC 5785, 4 2010. [Online]. Available: `http://tools.ietf.org/html/rfc5785`

[39] G. Oikonomou. 8051 requirements. [Online]. Available: `https://github.com/contiki-os/contiki/wiki/8051-Requirements`

[40] A. Dunkels, F. Osterlind, N. Tsiftes, and Z. He, "Software-based on-line energy estimation for sensor nodes," in *Proceedings of the 4th workshop on Embedded networked sensors*, ser. EmNets '07. New York, NY, USA: ACM, 2007, pp. 28–32. [Online]. Available: `http://doi.acm.org/10.1145/1278972.1278979`

[41] "CC2530F32, CC2530F64, CC2530F128, CC2530F256," Texas Instruments, February 2011. [Online]. Available: `http://www.ti.com/lit/ds/symlink/cc2530.pdf`

[42] *Internet Protocol, Version 6 (IPv6) Specification*, IETF RFC 2460, Dec 1998. [Online]. Available: `http://tools.ietf.org/html/rfc2460`

[43] *Datagram Transport Layer Security Version 1.2*, IETF RFC 6347, Jan 2012. [Online]. Available: `http://tools.ietf.org/html/rfc6347`

[44] S. Raza, S. Duquennoy, T. Chung, D. Yazar, T. Voigt, and U. Roedig, "Securing communication in 6lowpan with compressed ipsec." *2011 International Conference on Distributed Computing in Sensor Systems & Workshops (DCOSS)*, p. 1, 2011.

[45] L. M. L. Oliveira, J. J. P. C. Rodrigues, A. F. de Sousa, and J. Lloret, "Denial of service mitigation approach for ipv6-enabled smart object networks," *Concurr. Comput. : Pract. Exper.*, vol. 25, no. 1, pp. 129–142, Jan. 2013. [Online]. Available: `http://dx.doi.org.ludwig.lub.lu.se/10.1002/cpe.2850`

[46] K. Pelechrinis, M. Iliofotou, and S. Krishnamurthy, "Denial of service attacks in wireless networks: The case of jammers," *Communications Surveys Tutorials, IEEE*, vol. 13, no. 2, pp. 245–257, 2011.

[47] W. Webb, *Understanding Weightless: Technology, Equipment, and Network Deployment for M2M Communications in White Space*, ser. Understanding Weightless. Cambridge University Press, 2012.

[48] "CC2538," Texas Instruments, April 2013. [Online]. Available: `http://www.ti.com/lit/ds/symlink/cc2538.pdf`

[49] *Observing Resources in CoAP*, IETF Internet-Draft draft-ietf-core-observe, Rev. 08, Feb 2013. [Online]. Available: `http://tools.ietf.org/html/draft-ietf-core-observe-08`

[50] *Server-Sent Events*, W3C Candidate Recommendation, Dec 2012. [Online]. Available: `http://www.w3.org/TR/2012/CR-eventsource-20121211/`

Appendix A

Glossary

6LoWPAN IPv6 over Low power Wireless PAN

ACL Access Control List

AES Advanced Encryption Standard

BLE Bluetooth Low Energy

CBC-MAC Cipher Block Chaining Message Authentication Code

CCM Counter with CBC-MAC

CCMP CCM Protocol

CoAP Constrained Application Protocol

DoS Denial of Service

DTLS Datagram Transport Layer Security

FFD Full-Function Device

GATT Generic ATTribute

IPSO IP for Smart Objects

JSON JavaScript Object Notation

LLN Low power and Lossy Network

MAC Media Access Control *or* Message Authentication Code

PAN Personal Area Network

RDC Radio Duty Cycling

REST REpresentational State Transfer

RFD Reduced-Function Device

RPC Remote Procedure Call

RPL Routing Protocol for LLNs

SenML Sensor Markup Language

SEP Smart Energy Profile

SLIP Serial Line Internet Protocol

SOAP Simple Object Access Protocol

SoC System On a Chip

TI Texas Instruments

TLS Transport Layer Security

UART Universal Asynchronous Receiver/Transmitter

URI Uniform Resource Identifier

URL Uniform Resource Locator

URN Uniform Resource Name

WPA2 Wi-Fi Protected Access 2

Appendix B
Division of work

Most of the practical implementation work has been done jointly, because of limited hardware resources both in terms of sensor nodes and workstations.

The following sections have been written jointly:

- Chapter 1 (Introduction)
- Section 2.2 (Networking)
- Section 2.4 (Complete solutions)
- Chapter 3 (Choosing technologies)
- Section 4.1 (Hardware)
- Section 5.1.3 (Packet size)
- Section 5.2.3 (Packet size)

The following sections have been written by Jimmy:

- Section 2.3.1 (REST)
- Section 4.2.1 (Contiki)
- Section 4.2.2 (Sensor node)
- Section 4.2.4 (Development tools)
- Section 4.3 (Difficulties during implementation)
- Section 5.1.1 (Memory usage)
- Section 5.2.1 (Memory usage)
- Section 6.1 (Memory usage)
- Section 7.1 (Future of the Internet of things)

The following sections have been written by Linus:

- Section 2.1.3 (Wi-Fi)
- Section 2.3.2 (SOAP)
- Section 2.3.3 (CoAP)
- Section 4.2.3 (Gateway)
- Section 5.1.2 (Energy consumption)
- Section 5.2.2 (Energy consumption)
- Section 6.2 (Energy consumption)
- Section 6.3 (Packet size)
- Section 6.4 (Conformance to standards)
- Section 6.5 (Security)
- Chapter 7 (Conclusions)
- Section 7.2 (Future work)

Appendix C

Popular science article

Performance of devices in the Internet of Things

Jimmy Assarsson
Linus Karlsson

June 3, 2013

On today's Internet there are lots of connected devices, like cell phones and computers. The next step is to connect more and more of our things to the Internet. This thesis examines if and how small, wireless, battery-powered devices can be connected, and how it affects them.

Getting a device to work for a long time using only battery power is hard. It is even harder when the device communicates wirelessly, since radio transmitters consume a lot of power when they are used. Finally, since we do not want to change batteries in every small gadget too often, we expect a battery lifetime of months, or even years.

One of the most common wireless technologies today are Wi-Fi, which can be found in essentially all laptops and smartphones today. Unfortunately, Wi-Fi consumes too much power if you want long battery lifetime. Therefore we have examined other wireless technologies, and decided to use one called IEEE 802.15.4 instead. It is also important to minimise the amount of data sent, because more data means that the radio transceiver has to be turned on for a longer period of time.

We have developed our solution using modules from connectBlue, a picture of a node can be seen in Figure C.1.

To minimise the data sent we have used various compression techniques that reduces the data. A protocol called 6LoWPAN is used to reduce the size of IPv6 headers, and CoAP is used instead of HTTP on the application layer.

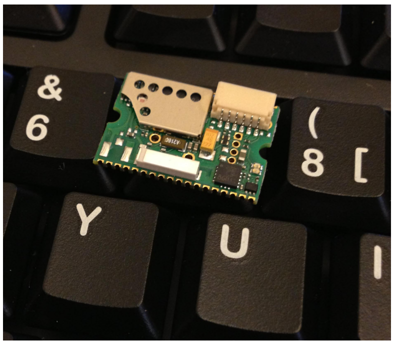


Figure C.1: Module on a standard-size keyboard.

Since these are protocols that are not very common, there is in general no support for them on ordinary devices connected to the Internet. Because of this, we have also developed an intermediate gateway which translates between CoAP and HTTP. This way, any ordinary web browser can communicate with our modules, even though they do not talk the same protocol. The gateway has two interfaces: one wireless interface which communicates with the modules, and one wired interface which communicates with the rest of the Internet. In this way, the gateway act as a bridge between the small modules and the big Internet.

We have used only open-source software to create our solution. As operating system we have used Contiki, which is specifically designed with the Internet of Things in mind. It supports a great deal of the techniques we are interested in. Furthermore we have used libcoap as server on the modules, and jCoAP as the software performing the protocol translation on the gateway.

During implementation, we quickly

found that the hardware available had very limited memory resources. This made it very hard to fit all features on the device at the same time. This required us to use a special version of Contiki, adapted for our hardware. Still, we could not fit all functionality we wanted in memory at the same time.

We investigated several different aspects of our solution: amount of data sent, memory usage, energy consumption and security.

We found that the amount of data sent was reduced significantly by using CoAP and 6LoWPAN. It was only 12 % to 23 % of the equivalent data in HTTP and IPv6 format.

As mentioned earlier, we had difficulties with the modules' memory usage. Depending on which memory that was exhausted, we got issues like sporadic resets or compilation errors. Some of the errors could be fixed by patching the code, while in some cases we had to remove the functionality completely.

We did also measure the energy consumption and compared two different options: one where the radio was turned on all the time, and one power-saving radio protocol where the radio was turned on only 1.7 % of the time. The latter reduced the total power consumption with 69 %. Still we found that the total power usage was too high, giving a battery life of only 27 hours with a button cell battery. This is far too low for any practical purposes.

Our solution has not focused on any security issues, but if it were to be used in production, encryption and access control should be implemented. This is important to avoid unauthorised people to read the data, or to manipulate it. The main reason our solution did not include any security features was because there is no built-in support for this in the open-source software we have used.

After evaluating the performance of our implementation we found that memory and power usage of our solution are the main issues. The problems can be attributed to both hardware and software. The amount of memory is very limited on our modules, and our conclusion is that hardware with more available memory is required to create fully-featured solutions. New hardware can reduce power consumption, since other processors and radio transceivers may have lower power requirements. Power consumption can also be reduced by software, for example by creating a new power-saving radio protocol. In this way, the radio could be turned on for a smaller amount of time to reduce the energy used.

Even though our implementation is far from perfect, we believe that the Internet of things has a bright future. We see possible future application areas in for example: home automation, healthcare, personal fitness, and many others. In this way, Internet of Things will become an integrated parts of our life — but not as long as battery replacements have to be done frequently.