University of Glasgow | School of Computing Science

A lightweight protocol for constrained devices for use in the Internet of Things paradigm

Fergus W. Leahy

School of Computing Science
Sir Alwyn Williams Building
University of Glasgow
G12 8QQ

Level 4 Project — March 22, 2013

Abstract

With the advent of cheap, low-power and small devices, it becomes possible to embed these devices within our everyday surroundings, to create an always connected, sensing and reacting “Internet of Things”, with the purpose of not only simplifying our lives but also saving energy at the same time.

But in order to achieve devices of such size, the resources available is restricted. Because of this, current protocols for connecting traditional devices together simply don’t cater for the constraints of such devices, therefore a new protocol which understands the constraints of these devices and is designed around this “Internet of Things” revolution is necessary.

To design the protocol, a wide variety of existing/developing protocols and systems were reviewed, noting what worked well and what didn’t; from this a new protocol which could not only run on constrained devices but also scale up to more powerful devices, whilst still taking into consideration reliability, fault tolerance and network scalability/performance.

The new protocol was implemented on the TelosB motes, which provided a benchmark for constrained devices, proving it’s possible to create an “Internet of Things” protocol which can suitably run on constrained devices, whilst still providing the functionality, reliability and scalability required.

Acknowledgements

I would like thank to Prof. Joseph Sventek for seeing something in this project from when it was only an seedling of an idea, and for his invaluable support, advice and guidance throughout.

I would also like to thank my family, girlfriend and all the friends I've come to know in Computing for helping keep me sane and on track throughout the past 4 years.

Fergus W. Leahy, 2013

Education Use Consent

I hereby give my permission for this project to be shown to other University of Glasgow students and to be distributed in an electronic format. **Please note that you are under no obligation to sign this declaration, but doing so would help future students.**

Name: _____ Signature: _____

Contents

Chapter 1

Introduction

The “Internet of Things” paradigm, along with the “Smart” prefix, has recently seen a significant rise in interest and popularity with manufacturers, hobbyists and end-users. Everything from your set-top box to your washing machine can now be “Smart” and connect to the Internet to tell you if your favourite TV show has downloaded or that your wash cycle has complete[12, 13]. Some hobbyists have gone so far as to make a plant send a text or tweet when it needs watering[5, 24].

This uptake in interest and development can be largely attributed to the advent of lower power, smaller and cheaper devices. Large companies, such as the mobile phone chip-set manufacturer Qualcomm, recently declared their support at CES 2013 with the announcement of a dedicated development platform for the “Internet of Things”[14].

Similarly there has also been significant enthusiasm from small companies and start-ups trying to create the next hit consumer device. The social funding platform, Kickstarter[21], has seen many attempts at creating the perfect “Thing” for the Internet[20, 25].

Whilst all of these devices from manufacturers, start-ups and hobbyists may very well be great “Things” by themselves, there exists a problem. How does one connect all of these heterogeneous platforms and devices together to create a truely interconnected “Internet of Things”?

This project focuses on just that and endeavours to create a communication protocol that is not only platform agnostic but also lightweight enough to be run on the most constrained devices, such as the TelosB Sky Motes(8MHz CPU,10K ram)[23]. Another core design focus is that the protocol must be able to scale effectively as the network dramatically increases size as can be expected with the rapidly increasing availability of network-connected devices.

Chapter 2

Background and Related Work

2.1 What is the "Internet of Things"

This section describes the concept of the "Internet of Things", its slow development up to the present, and the motivations as to why one would want to create an "Internet of Things".

2.1.1 Past, Present and Future

The standard model of how we use computers and the Internet today revolves around the idea of an interconnected network of servers, routers and data centres around which users connect using their personal computers to access, input, manipulate and retrieve data.

This model heavily relies upon the users to provide the data by which the Internet is powered. Without this data the Internet would be a barren place, with nothing to search for, sell, buy, share, watch, listen to, play, analyse or learn from. Users from all around the world have contributed to make the Internet *the* single biggest resource in the world by snapping photos, capturing videos, writing blogs, creating websites, commenting, discussing, reviewing, buying, selling and inputting data. So much data in fact that the estimated size of the Internet in 2009 was 500 exabytes (that's 500 BILLION gigabytes), of which 70% was contributed to by users.[19]

Within this model there exists two significant problems posed by users; time and accuracy. In terms of time, users only have so many hours in a day to input data; which can only be so accurate, as all users are prone to error through one means or another. These two problems make it difficult to observe our world and represent it in an accurate and reliable digital form.

The idea of taking this responsibility of inputting data away from the user and giving it to the machine is where the "Internet of Things" term initially came from. The term itself was thought to have first been coined by Kevin Ashton[11] in 1999, who at the time was interested in linking up a company's supply chain to the Internet using RFID technology to allow for autonomously monitoring its state. Sadly, at that time the idea progressed little and didn't garner much support.

Fast forward to 2013, the "Year of the Internet of Things" as declared by the MIT Technology Review[1] and observed in the Consumer Electronics Show 2013 (CES) with manufacturers introducing everything from smart fridges to smart weighing scales. Concurrently, low power devices have become increasingly affordable and small enough to embed into other devices, making it much more possible to create a truly autonomous system, either in the home, office or car, wherein the user doesn't have to interact directly with any of the devices but

instead just go about daily life, which, in the background, is enhanced by these ubiquitous devices operating together.

With all the possible "Things" that can exist, the core functionality in the "Internet of Things" can be simply modelled as a closed loop system in which a "Thing" can sense, process and interact with the real world as well as other devices and services. This system is represented at its simplest by an if statement, *if* ***event occurs*** *then* ***do action***, which provides developers and users alike an easy to understand yet powerful model to create rules for autonomous interaction.i.e.,

if ***temp <20C*** *then* ***Turn ON heating***

if ***washing machine done*** *then* ***Text John***

There currently exists a variety of online services and devices which give the end-user this ability to do such tasks, such as the Twine device [25] which is a small low power device with a variety of sensors and a WI-FI radio. The goal of the device was to make it simple and easy to set up and use; the user just puts the device in the desired environment, perhaps on a washing machine, sets up some rules ("When accelerometer is knocked Then tweet"), and when the condition is true, the preset action, in this case tweeting, is performed. Whilst this gives the user great flexibility to perform actions automatically, the limitations show when trying to interact with more than one Twine or with other devices and not just Internet services.

Figure 2.1: Twine IoT device.

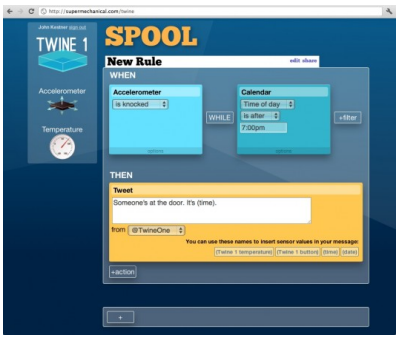

Figure 2.2: Twine rule, When x Then y

The Twine device isn't the only device out there, Smart Things[20] goes one step further than Twine and gives the user a network of devices connected to the Internet through a central hub which the user can then interact with online.

The main problem that exists and will continue to do so is that the digital environment is filled with a huge variety of heterogeneous devices; the key to the future of the "Internet of Things" is to find a way to abstract away the differences between these devices and create a simple platform on which all types of devices can communicate to build a more rich and powerful "Internet of Things".

2.1.2 Motivation

Like any new paradigm of computing, there must be sufficient motivation and interest for it to be picked up, used and developed further. In the case of the "Internet of Things", there are many cases why such a paradigm is of great interest, both to consumers and organisations.

- Workload
  - As mentioned previously, the core principle of the "Internet of Things" is to stop the user from having to consciously input data into systems, instead, replacing them with new devices(sensors, actuators). By doing so, the user's workload is reduced, giving them more time to do other activities, either of more interest or importance.
- Cost and Eco-savings
  - One of the key benefits from collecting data from these devices rather than users is that more accurate and reliable data can be gathered, from which smarter decisions can be made for the user, often resulting in energy savings i.e., turning off unnecessary lights or optimising heating schedules. These energy savings not only benefit the user in terms of monetary cost but also benefit the planet, making it valuable to both the individual and larger community as a whole.
- Security
  - By utilising these devices, not only can data be collected reliably but actions can also be executed reliably, unlike users. This can benefit the user in terms of security, ensuring that an important action is always carried out i.e., automatic door and windows locking upon exiting the house.
- Remote
  - Because of the connectivity of these devices, users can interact with them from anywhere with an Internet connection, giving them access to change, update and ensure the system is working correctly i.e., check oven is turned off and front door locked when out.

2.2 Typical "Internet of Things" examples

In order to better understand the possibilities and typical scenarios surrounding an "Internet of Things" equipped home or office, this section will outline two ideal examples of an "Internet of Things" network embedded in the home and office, discuss their respective benefits as well as the current state of "Internet of Things" networks.

2.2.1 "Internet of Things" at Home

The most common association with the "Internet of Things" is the "Home of the Future", which gives the consumer the impression that the static home they are used to will come alive with technology, to make life at home far simpler and more enjoyable. Therefore, this initial example will focus on the "Internet of Things" around the home and demonstrate several uses of it.

The primary goal of an "Internet of Things" network embedded in the home is to create an awareness of the user. By making the home aware of the user, it can observe them in real-time and determine *smart* things to do autonomously within the home. These could relate to security, such as locking and unlocking doors and windows when the user enters or leaves, or relate to convenience, such as turning on and off appropriate lights as the user

moves between rooms, turning on the heating in anticipation of the user's arrival. Over longer periods of time it could keep track of the user's habits and adapt heating schedules or open the blinds when the user usually goes to sleep or wakes, it could even turn on the coffee pot so when the user awakes in the morning a fresh cup of coffee awaits them.

Other examples are less concerned with the user but instead with the safety and upkeep of the home, by embedding sensors within objects in the home autonomous maintenance can be carried out. Some examples of this could be, automatic plant watering when low soil moisture is detected, or autonomous vacuuming at certain periods of time when the user is away.[18] By creating these possibilities, it not only reduces the workload for the user but it also allows the user to do things which matter most to them, and of course it stops plants from being forgotten about.

2.2.2 "Internet of Things" at Work

Much of the benefits from deploying an "Internet of Things" at home also apply when deployed in a work environment, such as energy savings, reducing workload and increasing security.

In the workplace, the concept of the "Internet of Things" being aware of the user could be taken advantage of to a much greater extent to increase productivity, efficiency and independence. This could be possible by allowing workers to be automatically checked in and out of their office and other key locations within the workplace, such as the cafeteria or meeting rooms, with varying granularity based on privacy. By doing so, not only can a worker forget about informing colleagues of their location through punch cards or switchboards, but it can also help keep their colleagues to be more informed about each other, so that instead of finding an empty office after walking from the other side of the building, time can be better spent.

2.3 Open source constrained devices

In the past 5-10 years, there have been many attempts at creating an affordable, low power and approachable electronics platform such as Arduino, Netduino, BeagleBone, Teensy and MSP430 Launchpad. All of which have taken very different approaches, some opting for absolute low cost (MSP430 Launchpad), whilst others aimed to be fully featured and powerful devices (BeagleBone). Other devices such as the TelosB mote, whilst not cheap, has fuelled academic research into new ways of designing and implementing Wireless Sensor Networks. More recently, the Raspberry Pi has created a whole new market of super, low-cost, yet moderately-powerful computers aimed at education and hobbyists.

In the rest of this section a variety of different devices and platforms will be discussed including the Arduino, TelosB mote, Raspberry Pi and other "Internet of Things" related devices.

2.3.1 Arduino

The Arduino was born in 2005 at an Italian university, Interaction Design Institute Ivrea, out of the necessity of creating a cheap and approachable electronics platform which could enable design students to create interaction design projects without the need of an electronics background. The main device which was created and has remained much the same since, is based on an Amtel ATmega328 8bit micro-controller running at 16MHz with 2KB of RAM and 32KB of storage for programmes written in a variation of C/C++. The board itself maps out the micro-controller's mix of 20 digital and analogue input/output pins and supports several standardised protocols for communicating with other devices such as I^2C[1] and UART[2].

But the key to the Arduino Platform is not the micro-controller itself but instead the design, software and support provided by the creators and other developers. The other major factor to its success is that the device, along with the documentation and support, are all open source, thus allowing anyone to learn from, replicate and expand upon the platform in any way they wish.

An example of how these factors have had a hugely positive effect is something which Arduino calls "Shields". These shields plug in on top of the Arduino board and contain standard components which can add additional features such as WI-FI, Ethernet, sound, motor control etc. Rather than users being required to find, purchase and solder the required components to add such features, these pre-built shields provide it in simple and readily available package, made by a variety of manufacturers.

Since its creation, the Arduino platform has created a range of Arduino named devices and shields resulting in a following of over 300,000 users[4] and support from many manufacturers and distributors worldwide.

Due to Arduino's open source licensing policy, many new start-ups have been able to quickly design prototypes and products using the Arduino platform which have been taken to market in various refined forms. Often the same micro-controller which powers the Arduino is kept and the board miniaturised to fit the product's needs. Products such as the Internet "Thing" Twine[25] and the Smart Things "Internet of Things" eco-system have taken this approach[20].

However, whilst there is an abundance of Arduino devices in the wild with many being used as an Internet "Thing", there is yet to be created an open and compatible method to easily network a group of such devices together to form a connected "Internet of Things" network.

[1] I^2C - Inter-Integrated Circuit
[2] UART - Universal Asynchronous Reciever/Transmitter

2.3.2 Raspberry Pi

Launched in 2012 the Raspberry Pi, a $35 credit card sized computer, was eagerly anticipated to change the computing landscape. The charity behind it, the Raspberry Pi foundation, had one main goal; to refresh and promote the teaching of computer science in schools.

Similar to the Arduino Platform, most of the hardware and software for the device is open source with the aim of letting adults and children alike get their hands dirty learning about computing without the worry of breaking an expensive computer.

The Raspberry Pi itself is a moderately well powered, single-board computer capable of running a wide variety of Linux-based operating systems. It's host to a 700MHz ARM CPU, 512MB RAM and a variety of inputs/outputs including an Ethernet port. These features, along with some additional GPIO[3] pins, just like the Arduino, allow it to sense and control the world around it, thus making it a perfect "Thing".

Because of the extremely low price point for such hardware, the Raspberry Pi was a huge success selling, over 500,000 units in the first 8 months, only limited by their production speed.[17]

Even before it's release, the community support and ideas were non-stop, everything from home-media centres[16] to Lego super computers[4][15] were designed and created from Raspberry Pis.

Whilst the support for utilising the GPIO pins is not yet as well developed as the Arduino Platform, the power of the device combined with its low price point and connectivity means that it can be very easily incorporated into a "Internet of Things" network with little additional hardware or cost.

Another considerable point for developing on the Raspberry Pi is that because of its ability to run Linux, a much larger variety of programming languages can be used, including high level languages like Python, C++ and Java.

2.3.3 TelosB Motes

With a significant uptake of devices in academia, TelosB motes are one of the go to platforms for wireless sensor networks research. Whilst they may fit under a different paradigm, the IoT can be seen as a specialisation of a wireless sensor network. Although the are extremely constrained, they are a good, low-end benchmark for which to test and develop software for use on constrained devices.

The device was developed at the University of California, Berkeley, and spun off into a separate company. As a result, these devices have fuelled many academic studies and courses in relation to developing low power wireless sensor networks and researching applications for them. With very limited resources, 8MHz CPU, 10KB RAM, 48KB ROM, light/temperature sensors, battery powered and a low-power 802.15.4 (250 kbps)radio, these devices require new ways of thinking in order to design, program and deploy them. Many scenarios must also take into consideration the environment in which they will be deployed, which are often hostile and unpredictable that can cause nodes to disappear from the network sporadically. Such environments have ranged from forests for fire detection[36] to monitoring livestock [33].

A significant part of the development for these types of devices has surrounded the operating system and programming style. Currently there exist many different operating systems in which TinyOS and Contiki are most used and developed for. Both operating systems take quite different approaches to both operating system design and programming API.

[3] General Purpose Input Output

[4] Even within the School of Computing Science, UoG

TinyOS is designed as a completely modular, event-driven and thread-less operating system which makes for a difficult system to program and reason about; this is due to its unfamiliar execution style and the design of its programming language, nesC, a significant variant of C. Due to its modular design, programs are written in "components" which are comprised of their "configuration"(how they link/wire to other components) and "module"(the implementation of the component).

In contrast, Contiki is designed to fit the middle ground between Event driven programming, like TinyOS, and imperative programming, like C. It achieves this by having both lightweight threads, which can interleave in execution, and events, which allow the program to react to stimuli, written in a slight variation of C [32]. This approach is much more familiar to the experienced programmer of larger systems whilst still providing an efficient environment in which to program albeit with some compromises (stackless threads). Where TinyOS breaks programs down into a complex array of components, modules and configurations, Contiki aligns with standard C, using header and .c files.

In regards to the "Internet of Things", Contiki provides a much more approachable platform to design for, as a significant portion of code written for larger platforms can be ported without much transformations to the code. Combining this with the hardware's array of sensors and low-power radio, the two create an ideal low end platform for developing at the bottom end of the "Internet of Things".

2.3.4 Wireless Sensor Networks

With computers becoming smaller, cheaper and more powerful every year, wireless sensor networks have become an area of rapid research. Wireless sensor networks allow the autonomous collection, aggregation and processing of data from hundreds if not thousands of devices, which before would have only been feasible through simulation. Use cases of such networks range from detecting forest fires[36], to monitoring the health of the Golden Gate bridge[35].

Many of the situations and environments these networks are deployed in can be extremely harsh, often subjected to adverse weather conditions and even animal wildlife. Because of this, reliability and fault tolerance are significant areas of research, ensuring the network of devices can react and adapt to changes in the environment to ensure correct operation, even in the case of device failures. Due to these often harsh and changing environments, standard architectures of networking aren't feasible, e.g., star topology, because of their reliance on central points or base stations to communicate, which may become out of range or fail entirely; instead a more ad-hoc approach is needed to ensure all devices can be connected to the network, even when devices move in the physical environment. To solve this problem, devices are networked together in a mesh topology, where each device connects to its neighbours and allows traffic to route through one another to reach the final destination.

In some ways the "Internet of Things" can be interpreted as an extension or specialisation of wireless sensor networks, taking the autonomous network of hundreds of sensing devices and creating an autonomous closed-loop system, which can make intelligent decisions based on the environment, and placing it within a home, office or even city. Often the environments are a much less extreme and are more predictable than traditional wireless sensor network deployments, such as the home or office, reducing the concern for reliability. However, because of the pre-existing and often immutable nature of environments such as the home, installing traditional networking topologies ranging from the front door, to the second floor bedroom and all the way down to the bottom of the garden can pose problems; but like in wireless sensor networks, mesh topologies can help solve this by routing traffic through other nearby nodes, rather than trying to connect directly to the base station or router.

2.4 Existing systems/protocols

2.4.1 Java JMS

Although the Java Messaging Service itself isn't directly targeted towards "Internet of Things" implementation, it does provide a standard and well defined framework for communicating and coordinating between both local and remote applications on a network.

It can operate in two modes, either in a publish - subscribe or point-to-point architecture. Using the publish subscribe system, publishers publish to topics, to which subscribers subscribe. The topics bridge the two types of clients together and allow for many to many relationships without either side knowing about the other. With the "Internet of Things" in mind, this type of publish - subscribe systems fits in well with the *if **event occurs** then **do action*** model. Topics map to events/conditions to which sensors can publish and to which other devices can subscribe, performing some action as a result, as shown in figure 2.3 .

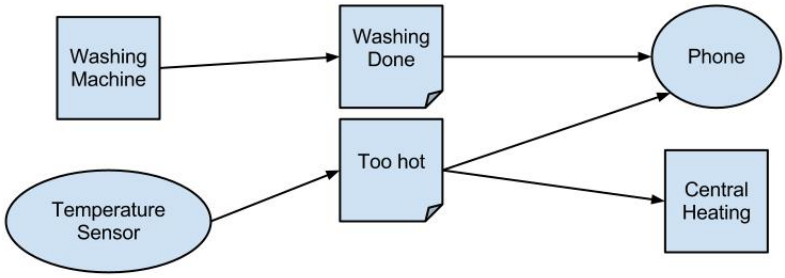

Figure 2.3: JMS Publish - Subscribe modelling IoT

Whilst JMS provides a well fitted framework, it does however come at a cost due to the runtime environment of Java and the additional overhead of having to run a separate server to glue the publishers and subscribers together. Whilst currently running the Java runtime on these constrained devices may not be possible, with the current pace of innovation, especially in optimisation of the Java runtime, and Moore's Law giving devices more power for the same cost and form factor every 18 months, it won't be long before it's really possible to have it running on the constrained devices of tomorrow. But until then, other alternatives and perhaps more suited tools, libraries and frameworks should be engineered starting at the lowest common denominator rather than shoe-horning large scale systems to fit constrained devices.

2.4.2 xAP

In the early 2000's, as an attempt to bring together a variety of technologies developed and used for Home Automation, an open source group decided to create a protocol to bridge the differences and create a unifying platform.[26, 28]

xAP, eXtensible Automation Protocol is designed to be a minimalist, elegant and easy to implement protocol with very basic requirements for the hardware, operating system, network and language on which it can run. The primary implementation is based on UDP/IP with a distributed architecture where no central controller co-ordinates the network. Instead, each device either transmits or listens for data on a broadcast channel. Their core justification for this is that the network becomes fault tolerant and can withstand devices disappearing off the network for one reason or another without having a detrimental effect on the whole network.

In this design there are two key classes of devices, senders and receivers, of which a device can be either or both. The receiver simply attaches to the network and listens for any incoming packets, it then chooses which ones are relevant to it and processes them. The sender attaches to the network and broadcasts packets with payloads containing data relating to its service i.e., sensor data, incoming caller ID, etc. These packets only contain enough information to uniquely identify the source and the payload it wishes to send. There is no constraint on whether or not the packet must have a destination.

This design feature makes it very simple for new devices to attach to the network and start interacting without having to go through the process of setting up connections to other devices. It also makes it very easy to implement certain types of devices like loggers or informational displays which can collect data from all senders with relative ease. However, this also creates a huge strain on the devices attached to the network which all have to receive and process any broadcasted data, especially for those devices with limited power resources. This becomes an increasing problem as more and more connected devices invade the home, from which the network will very quickly deteriorate as the volume of traffic being broadcast increases. The efficient sending of data, by either avoiding broadcast or limiting its use is an ongoing research topic within the Wireless Sensor Networks domain, due to the costly nature of broadcast and the constrained power resources of the WSN devices.[29][34]

The protocol also defines a set schema for how packets should be formatted so that all devices can interoperate correctly. The schema covers both the header and the payload format, which is almost entirely text-based, resembling, albeit pre-dating, JSON, with the main intention of being human readable. Whilst so, this does come at the cost of not only increasing the size of the packet considerably with redundant text, but also the complexity in parsing the data with its varying delimiters.

After the protocol's initial inception it has had limited success, even whilst it has been continuously developed over the years, it has done so in the shadows and not in an explicitly open and collaborative way, with no central repository for code collaboration/review and no single developer's forum for discussion.

2.4.3 Remote Procedure Call - RPC

RPC whilst relatively old in comparison to the other systems described, it does provide an adequate abstraction for communication between devices in an "Internet of Things", via request/response style interactions.

The concept of creating a distributed system, whereby devices in the network request data and services from each on demand, with the relevant response being returned. This fits in well with the idea of an autonomous system sensing, processing and reacting to stimuli, with each device bringing its own services to the system.

However, because of the fixed concept of every request requires a response, the number of packets sent is doubled, even when no response is necessary. In the environment of constrained devices, this increases the load per device and if battery powered, reduces the life of the device significantly.

2.5 Developing systems/protocols

2.5.1 OpenWSN

Currently being developed at the University of California, Berkeley, the Open Wireless Sensor Network project is collection of open-source implementations of protocol stacks implemented against to-be-finalised "Internet of Things" standards, for a range of different software and hardware platforms.

The driving force behind such a project is to create an open source implementation protocols, which academia and industry can use to conduct further research, develop standardised products and contribute to an ecosystem of "Internet of Things" devices, rather than researching and developing proprietary devices. In the long run this can also benefit consumers, increasing their choice of devices across many different and compatible brands/manufacturers.

Whilst the OpenWSN project doesn't actually propose any new research or developments by itself, it does pursue the same idea as this very project, trying to create an open source method in which heterogeneous devices can communicate with one another to form and "Internet of Things".

The OpenWSN "Internet of Things" stack consists of open source implementations of the proposed communications stack, ranging from the physical layer (IEEE802.15.4-2006), to the adaptation of IPv6 (IETF 6LoWPAN) all the way up to the application layer, of which is of most interest. The application layer implements not only HTTP, but also a currently in progress standard called Constrained Application Protocol, with the purpose of bringing RESTful communication to constrained devices.

2.5.2 CoAP

CoAP, Constrained Application Protocol is currently a proposed IETF standard to create a request-response style web protocol, specifically designed to fit the limitations of constrained devices, such as micro-controllers, for use in machine-to-machine communication and the"Internet of Things". The protocol is designed as an asynchronous RESTful protocol, to run on top of UDP, using a subset of the standard commands e.g., GET, PUT, DELETE. The reason is that due to the common commands, it allows for an easy transition to standard HTTP if necessary, whilst reducing the complexity of the protocol. Together with this, the protocol also provides support for service discovery and basic subscribing to other CoAP nodes.

Because the protocol is designed to run on top of UDP, which is a connectionless, unreliable transport layer, CoAP provides both un-reliable and reliable support for requests; which is provided by marking a message as non-confirmable (unreliable) or confirmable (reliable). This allows nodes to send data which is essential, reliably, and to send data un-reliable when data is ephemeral, such as sensor readings.

As mentioned previously, the protocol also supports service discovery, to allow nodes to find new sources to request resources from, without necessarily using a centralised server. If a node's IP address is known a priori, then a unicast discovery can be used to locate the entry point of the resource of interest, otherwise a multicast discovery must be used; this is done by making a GET request to a standardised location, from which receiving nodes respond based on whether the request matches any of the resources it holds (by type or description).[37]

To combat one of the problems normally associated with constrained devices, power, CoAP supports the ability for nodes to subscribe to one another. This means that instead of a node being repeatedly polled for any changes, the node instead notifies all subscribed nodes when the resource of interest has changed. This reduces the amount of packets received by the receiving node, in turn helping to reduce its power consumption.

Lastly, because CoAP is so similar to HTTP in the commands that it uses (GET, PUT, DELETE, etc.), it makes it simple to translate messages between protocols, and therefore allow these constrained devices to send and receive data to and from the WWW.[30]

At the time of writing, the protocol is still being developed and finalised, although there have been a variety of implementations (of some degree) [8][6][7], it's yet to see if the protocol will be fully adopted both commercially and in the open source community.

2.5.3 SmartThings

Launched in 2012, the SmartThings platform aims to allow the user to turn any ordinary object in the home into a "Smart" object by giving it the ability to connect to the Internet. The platform consists of a central hub connected to the Internet, containing a low-power Zigbee radio, from which it connects to an array of SmartThings accessories as well as many pre-existing third party Zigbee devices. Some of these accessories include motion detectors, moisture sensors, vibration detectors, power-plug switches as well as many others. The user can then connect to the SmartThings service through an Internet-connected pc or a smartphone, allowing them to either control the devices directly or set-up rules/schedules for devices, similar to the previously mentioned Twine device(2.1.1).

Whilst providing easy to set-up pre-made SmartThings, the developer also provides the open-source Arduino kits to let hobbyists create their own SmartThings utilising the already well supported Arduino platform.

Currently, there is no information on how the underlying protocol for connecting the SmartThings to the hub works, but at the time of writing, the current implementation uses a "cloud first" approach. This means that rather than the hub wiring all devices together based on the rules and schedules set up by the user, all the intelligence of the network is being handled in the cloud. This brings about the problem of Internet connectivity, with two points of failure, either the user or the cloud. From the user's standpoint, an Internet connection might not be available where the hub is located, or the user could have a faulty, slow or non-permanent connection, which renders all the SmartThings devices into dumb devices. In contrast to this, because of the reliance on the cloud, if the cloud service provider experiences downtime then all SmartThings users devices become dumb devices. In cases where these devices are used for security and safety, dire consequences could result.

2.5.4 Qualcomm

As an example of a major hardware manufacturer acknowledging the "Internet of Things", at CES 2013, Qualcomm announced a new platform for developing "Things", named the "Internet of Everything", based on their mobile chipsets, providing 3G network connectivity and utilising the Java Micro Edition runtime [14]. So far it seems the concept of connectivity is not dissimilar to existing devices, such as the Twine, where the device is essentially a single device connecting to the Internet, in contrast to a network of interconnected "Things".

The platform has yet to be released at the time of writing, but will prove to be an interesting start to hopefully more manufacturers creating purposely designed hardware, enabling developers to create "Internet of Things" applications.

Chapter 3

Requirements gathering

As discussed in the previous chapter, many different platforms and protocols exist to create an "Internet of Things" and most take very different approaches. This chapter will highlight several drawbacks of some the previously mentioned systems, explain how such problems can be resolved with this new approach and discuss the requirements for developing a new system, which is better suited for constrained devices and the environment of the "Internet of Things".

3.1 Primary Requirements

Due to the nature of the target hardware platforms, constrained devices such as Arduino and TelosB, heavyweight approaches become infeasible i.e., Java & JMS. Therefore an extremely lightweight and scalable protocol, both in terms of the protocol data unit (pdu) and the complexity in processing and managing the runtime, is a primary requirement.

Secondary to this, because of the often unstructured and ad-hoc design of a typical "Internet of Things" network, building a centralised system i.e., a name server for device search, is an unnatural fit with the network. A single failure in the server could bring the entire network down, whilst all other devices are fully operational and no way to communicate with each other. Instead, creating a distributed system which gives all the devices in the network the power to discover and communicate with each other, relieves the network of a single point of failure and can help spread the load.

Taking into consideration the typical types of devices connected to the "Internet of Things", three general roles of devices can be discerned; a sensor, an actuator and a controller. Sensors, anything from a temperature sensor to an open door sensor, provide a continuous service to other devices in the form of real-time data, usually from the outside world. Similarly, actuators also provide a service to other devices in the form of an interaction with the outside world, such as a speaker, light switch or thermostat. Lastly, controllers are devices which orchestrate the "Things" in the network, forming relationships between devices and creating useful interactions between the digital and real world i.e., connecting to both a door sensor and a light switch, when the door opens the light turns on. Whilst there are these distinct roles, it is often the case that one device could take on more than more role i.e., a light switch can both sense its state, and provide an action, turn on or off.

3.2 Typical Use Cases

This section will demonstrate several typical use case scenarios of different combinations of devices that an "Internet of Things" around the home may be composed of, with some necessary requirements in order to operate correctly.

- Logging
  - The logging application runs on a controller device interacting with one or more sensors. The controller requests data from a temperature sensor and logs it to a file locally. The purpose of such an application can be to help with understanding the state of an area, room or home over a larger period of time, from which some assessment can be made e.g., one room is always warmer than the other during the afternoon.
- Presence detection lighting
  - The presence detection application runs on a controller device interacting with both a sensor and actuator devices. Within certain areas of the environment motion detectors are placed to detect the presence of a person in a particular space. This data is then communicated back to the controller and used to determine which lights to command to turn on or off. The purpose of this application would be to reduce the energy costs associated with lights unnecessarily left on.
- Wash cycle complete
  - The wash cycle complete application runs on a controller interacting with a sensor and an actuator. The actuator could be any form of communication medium that can alert the user i.e., sending a text message or tweet. This application would enable a washing machine to act as a sensor, relaying information back to the user(time left) and when a wash cycle has completed or requires further attention the user could be alerted in some form so that they can attend to it.
- Direct control and observation
  - A direct control application would allow a user to interact with the "Internet of Things" directly through a web browser or smartphone application. Such an application would operate on a controller device and allow a user to dynamically create new rules and relationships between existing devices as well as directly command devices such as lights, thermostats etc. This would allow a user to easily create and destroy relationships between devices without reconfiguring or restarting the network of devices.
- Multi-decision heating
  - A multi-decision application, such as a heating system, would take advantage of an arrangement of multiple different types of sensors and existing services. These sensors could include: temperature sensors to sense the current temperature of different spaces, presence sensors to detect where a person in the environment, location sensors to detect if someone is actually home. The combination of these sensors could allow the system to detect whether the environment should be heated, and if so when and by how much. Building upon the initial use case of logging, a logger could be used to create a basic schedule of when people are in the environment so that heating schedules can be created, this would allow the environment to be heated upon the arrival of someone.

3.3 Functional Requirements

This section describes the functional requirements which were gathered and are based upon the needs of the use cases discussed in the previous section.

1. Support 3 device roles; Sensor, Actuator, Controller
    - These 3 roles of devices are needed to form the basic closed loop interaction model necessary for the "Internet of Things".
2. Configuration of device attributes both at compile time and run time i.e., name, type, frequency, etc.
    - This is necessary in order make each device distinct from another, as well as customise it based on its needs, either in terms of performance, power constraints or otherwise. By enabling run time configuration it simplifies updating the devices on the network without the need to recompile.
3. Discovery/querying of devices
    - Discovery of other devices on the network is a core feature of a distributed "Internet of Things" network, without this the network can't dynamically grow by finding new devices. Querying of devices is to determine compatibility of devices before making a connection, by filtering out requests which don't match a devices attributes network congestion can be reduced.
4. Negotiation and connection to devices
    - Negotiation is necessary to determine the characteristics of the relationship between two devices, which is based on the demands of the device initiating the connection and whether or not the recipient can or wants to meet that demand i.e., a device can request a certain frequency of readings from another device which may or may not be able to match it.
5. Closing connections to devices
    - Devices need to be able to gracefully close any formed connections, which could be due to limited power constraints or no longer needing the resources that the connection offers. These connections also must be closed gracefully, in contrast to just not responding/dissappearing off the network, in order to ensure that a device's resources are not wasted by keeping state for a terminated connection, which could be confused for transient communication loss.
6. Sensors must be able to send data to controllers
    - A sensor's core functionality is collecting data through sensing the environment, this data may be of interest to other devices on the network and therefore must able to to send it to them. These data transmissions also need to be sent at a set frequency, either configured or negotiated, so that a receiving device knows when to expect data.
7. Actuators must be able to receive commands from controllers
    - An actuator's core functionality is interacting with the environment, without informed control of the device it has no use by itself. Therefore the device must be able to accept commands from other devices, such as controllers, in order to interact with the environment with a purpose.
8. Controllers must be able to send commands to actuators and receive data from sensors
    - A controller's purpose in the network is to form connections to other devices to create the closed loop system, which in order to do this, must be able to receive data from sensors and send commands to actuators.

9. All device roles must support connections to multiple devices

    - This is necessary in order to allow a device to form relationships to several other devices, which not makes better use of the available resources within each device but also allows for devices to form multiple, more complex relationships between different devices i.e., a temperature sensor connecting to one controller which logs its data and to another controller which is connected to an actuator that controls the thermostat.

10. Liveness checks through pings

    - Due to the distributed and unreliable nature of the network environment, devices need to be able to check the liveness of the connections to ensure devices are still active, and for those that aren't, react gracefully by cleaning up the expired connection and perhaps try to discover new devices.

11. Application Protocol Interface

    - This is necessary for applications to be built on top of the device roles, which will utilise the capabilities of the underlying network of "Things" to create rich and powerful systems in the "Internet of Things".

3.4 Non-Functional Requirements

This section describes the non-functional requirements based on the expectations of the system, its performance characteristics and the constraints surrounding the development the project.

3.4.1 System Requirements

- Closed loop system
    - The system must be able to sense, process and react within the network itself, to create a fully reactive closed loop system.
- Lightweight
    - Because of the target systems, constrained devices, the system must be lightweight enough, both in terms of the protocol data unit and use of system resources, so that devices such as the TelosB mote can run it.
- Scalable, from small to large networks of devices
    - Because of the rapid adoption of technology within our environment, the system must be able to scale from small to large networks in order to support the use of 10's if not 100's of devices with a single network, without causing unnecessary congestion and still ensuring the same performance of smaller networks.
- Cross platform compatibility
    - Due to the heterogeneous nature of hardware and software platforms available today, the system must be designed in such a way so that it is both hardware and software independent, enabling it to be implemented on these platforms as well as across a variety of networks.
- Simple to extend

  - Due to the constantly evolving nature of technology the system must be extensible in order to support new types of devices, and also be able to support new payloads specific to these devices.

- Reliability of essential communications
  - Because of the reliability concerns regarding the underlying network, building in some reliability is necessary to ensure that critical communications are successful, such as forming connections between devices.

- Basic assurance of correct data
  - Due to the reliance upon data within the network in order to sense, process and react to the environment, all data transferred must be reliable, therefore the need for some checks to ensure it is correct when it reaches the destination are needed.

- Network resilience to device failures
  - Because of the unpredictable nature of the environment and the limited capabilities of some devices (battery powered), the system must react gracefully to device failures, without the entire system being rendered in-operational.

- Liveness monitoring
  - The system must be able to monitor and react to the liveness of devices in the network i.e., if a device becomes unreachable, close connection gracefully and clean up.

- System performance
  - Because of the real-time nature of the "Internet of Things", the system must perform suitably, so that data is received and commands are carried out within acceptable real-time bounds i.e., receiving temperature readings for an hour ago are of no use to a device which shows the user the current temperature.

- Device resources
  - The system must consume as little device resources as possible in order to support the development of user applications on top.

3.4.2 Development Constraints

- The system must be researched, designed and developed within a limited time frame, about 20 weeks.
- The choice of platforms for implementing the design is based on the availability of constrained devices between the School of Computing Science and the author.
- The choice of operating system for the constrained device is based on prior experience with similar systems and platform languages available to the OS. The Contiki OS uses a minor variation of the C programming language, making it quicker and simpler to get started based on the author's experience with C, as well as simpler to keep the system (mostly) platform independent.

3.4.3 Development Assumption

In order to design and implement the proposed system, the scope of the problem needs to the limited, therefore it becomes necessary to define several assumptions.

- The number of possible devices types is limited, no more than 255.
- Sensor data is not critically important, so packet loss is acceptable.
- The system will be used in an environment where security is a non-issue (but MUST be considered as future work before deployment).

Chapter 4

Design

This chapter describes and illustrates the design of the new protocol, independent of any hardware and software platforms, as set out by the requirements gathered in the previous chapter. This chapter will discuss the different roles of devices to be used, the fundamental features necessary to meet the requirements, the types and flow of messages between the different roles, the protocol data unit and command formats defined and finally some of the key design choices for this protocol will be compared to decisions made by other pre-existing protocols, to highlight and discuss their differences and/or similarities.

4.1 The three device roles

In order to understand and design the system correctly, the three basic device roles in the "Internet of Things" must be explicitly defined; these roles will ultimately dictate the core communication model between devices in the network. However, whilst these roles are distinct in nature, a single device may take on one or more roles, e.g., a light sensing its state, and offering a command to turn it on/off.

By separating out functionality into distinct roles, it enables each to device to offer individual, context-less services which can be offered to many different devices and combined to create a powerful, rich and adaptable "Internet of Things".

4.1.1 Sensor

The sensor is *the* key component of any "Internet of Things" network; without the ability to sense, either on demand, periodically or in real time, the user is forced to consciously interact with the system, in turn defeating the purpose of the "Internet of Things".

The sensor in the "Internet of Things"can be seen as a simple device which does little to no computation, except that of formatting or preprocessing the data it senses, and simply forwards it to one or more devices in the network that have expressed interest in its data, at set time intervals. These intervals can be decided either by the sensor or the interested device, based upon the needs and resources available to both.

Because data from sensors is usually of an ephemeral nature, recovery/retransmission of damaged/lost packets is usually not of interest after the next one in the sequence has arrived, e.g., no need for previous temperature readings when interested in the current temperature now. However, knowing that the data has arrived correctly and undamaged is of importance, as data must be correct for the system to operate correctly and as expected.

4.1.2 Actuator

Actuators, like sensors, are a key component in the "Internet of things", performing actions to interact with the real world, e.g., turn on light or display the current temperature. Without the use of actuators, the system would only be able to record and process the environment without having anyway of reacting to any events.

By themselves actuators are primitive devices and are only able to perform context-less actions by themselves, e.g., turn on and off a light. In order to create meaningful actions, such as turning on a light when someone enters the room, actuators need to be able to offer their actions, or functions, as a service to other devices in the network whom are interested in performing such actions.

Because of the on-demand semantics of performing an action, actuators need to be always available and ready to respond to incoming commands at any time.

4.1.3 Controller

Controllers, unlike sensors and actuators, are the brains of the "Internet of Things"; by themselves, they can do nothing, but by utilising other devices in tandem, they can create a powerful, sensing and reactive system.

By connecting to sensors, controllers can give the data sensed a purpose and meaning, and by connecting to actuators it can give actions a reason to be performed, e.g., when the temperature sensor readings are >26C, the room is too warm, therefore turn down the thermostat, in order to reduce the temperature to a more preferable setting.

Within a single network it is possible to have several controllers all connected to the same or different devices as well as connected to each other, and each could perform different functions in the network i.e., one controller logs all devices in the network, one controller controls living room devices and another controls bedroom devices. Extending this, allowing controllers to connect to one another enables the network to form more complex relationships to perform actions that one controller alone could not, such as one controller having Internet connectivity providing resources such as cloud connectivity to the other controllers.

4.2 Protocol Design

This section discusses the design of the protocol based on the formal requirements, from the basic sequence of messages passed between the different roles, to deriving specific state machines for each role.

4.2.1 Features

The protocol is designed to fulfil several key requirements:

- Minimalistic protocol, reliability for critical data only, no fragmenting
- Minimal header overhead, reduce transmission and receiving cost of data
- Distributed device discovery, use of broadcast to discover devices
- Reliability, reliable set-up of connections and integrity of data
- Fault tolerant, liveness checks and no central point of failure

4.2.2 Transport Layer

Due to the requirements of the protocol, allowing it to run on a variety of networks, and reducing necessary transmissions to save power, the design of the protocol is such that it only requires packets to be sent over a best effort link with no reliability guarantees; instead the protocol itself handles any problems with corrupt data or dropped packets, as discussed later in this section,4.2.5, 4.2.7.

The one significant requirement of the transport layer is that it must provide the capability to send both broadcast and unicast packets, due to the device discovery requirement. Therefore, a simple datagram transport layer can be used to provide the required functionality, such as the IPv4/6 User Datagram Protocol (UDP) or Contiki Rime stack[31].

4.2.3 Multiplexing devices, channels

In order to enable devices to communicate with more than one other device, a method for multiplexing a device's resources must be created. Within this system, each connection between two devices is called a channel, and each device can serve multiple channels, depending on its own resources, i.e., the more resources available, the more channels a device can serve, the more devices a single device can connect to.

Each device will contain at least two channels, one home channel (default) and one or more ephemeral channels. The purpose of the home channel is to enable devices to have a well known location at which they can be contacted; this will be used for device discovery and to request to start a connection on one of the ephemeral channels.

In order to uniquely identify each connection, a triple of identifiers is required:

- transport layer address of each host the system is running on (IP address or Rime address).
- transport layer multiplexing identifier associated with the running system process (UDP port or Rime channel).
- a multiplexing identifier for each connection to a device, called a channel.

4.2.4 Selective Multicast/Unicast

The use of multicast in a network can be extremely expensive, both in terms of the bandwidth consumed on the network as well as the time and energy cost of every device being forced to receive and process the packet. Therefore multicast transmissions must be kept to a minimum and where they are necessary, ensure that it is not used excessively.

In the case of device discovery, multicasting is required due to the distributed nature of the system, where no central name server exists and therefore the network needs to be queried in order to locate devices. In order for this to work, all devices on the network need to listen on a well known address (e.g. UDP port), to which the discoverer can send multicast discovery packets. Once a device receives a multicast discovery packet, it can decide whether or not to reply, based on whether the query information, device type or name, matches its own. If it does match, then it replies using a unicast message to notify the sender of an available device matching its request. Because filtering is used at the receiving end, this further reduces the strain on the network resources, by stopping unnecessary packets from being transmitted.

4.2.5 Selective Reliability

In an effort to both reduce the cost of data transmission for devices and bandwidth consumption on the network, reliability will only be ensured for critical data transmissions, including connection set-up, tear-down and actuator commands.

Because of the unreliable nature of the underlying network, ensuring that connections are formed correctly is extremely important, otherwise a connection forming packet might be dropped, causing one device to believe the connection has been fully set-up whilst the other waits for the dropped packet which will never arrive. Another case where this is important is when issuing commands to actuators, as it is necessary to ensure actions are in fact carried out and the command hasn't been lost on the network. This can achieved through at-least-once semantics, ensuring that the actuator receives at least one command, so that it performs the action, and any repeat requests can be ignored.

Whereas, when sending data such as sensor readings the reliability of it arriving is not as important, this is largely attributed to the ephemeral nature of sensor readings, especially in a real-time environment like the "Internet of Things". For example, in a scenario where the current temperature is being displayed to the user, the need for old, out of sequence data, which may have been lost and retransmitted becomes unnecessary, as the user is only interested in the current temperature.

4.2.6 Liveness response

In order to ensure the connections between devices are kept alive, the liveness of a connection needs to be tracked, this is done differently depending on the device.

Sensor - Because sensors transmit data back to a controller at a set frequency, the controller can easily keep track of the liveness by simply monitoring incoming packets. If an arbitrary number of packets is not received after a multiple of the frequency period, then a time-out will occur and the connection can then be probed using a PING message to check if the packet loss is transient. If no responses are received in the form of PACKs, then the connection can be closed gracefully.

Actuator - Because actuators just listen on the connection for any commands from a controller, it becomes necessary to regularly check at an arbitrary frequency that the connection is still alive using PINGs. Without PINGs, if a command was sent, it would initially be difficult to tell whether the packet was lost or the connection is dead. By using PINGs, the status of the connection to the actuator can be regularly monitored, and if necessary, the associated channel closed when the actuator is persistently unresponsive.

Controller - Because controllers actively keep track of actuators, actuators therefore implicitly know that the controller is still alive. However, this is not the case for sensors, which are only liveness checked passively unless there is packet loss. Because of this, the sensor must therefore regularly send PINGs at an arbitrary frequency to check to see if the controller is still alive to receive the packets of data being transmitted. If this was not checked, then the sensor would be not only wasting its own resources by needlessly sending data, but it would also consume the resources of other devices if this was deployed on a low-power multi-hop mesh network, typical for "Internet of Things" networks.

In order to ensure the network does not become congested by the retransmission of PINGs within a short period of time, an exponential back-off will be used. This will ensure that if a device were to disappear of the network, the rest of the network would not suffer. In a large network where multiple devices could fail at once, perhaps due to a common link, the network could become heavily congested if exponential back-off were not used.

If after an arbitrary number of PINGs, a PING ACK is not received, the connection is deemed dead and the associated channel closed.

4.2.7 Data integrity

Whilst most data sent in the "Internet of Things" network does not need to be sent reliably, due to it's ephemeral nature, it is important however that when it arrives it is consistent and accurate; if sensor readings were to become corrupt in transit, then unintended actions could result from inaccurate data. For example, a temperature sensor transmits the current temperature to a controller, which becomes corrupt in transit, as a result of the corrupt sensor reading, the controller believes the current temperature is too hot and the thermostat is commanded to turn off, when in fact the temperature was at an acceptable level.

Therefore, the use of a mechanism to ensure the integrity of the data is necessary, such as a Cyclic Redundancy Check, CRC. In this protocol, a 16bit CRC will be used to ensure that when data is sent from one device to another, that it is possible to discern whether or not it is accurate and consistent. If the integrity of the data is found to be compromised, then the packet will simply be dropped. If the packet was sent as reliable data (connection request or command), a timeout will occur in the sender device expecting an acknowledgement, and the packet will be resent.

4.2.8 Flow of messages

Based upon the use cases described in the Requirements section, a set of messages are described and demonstrated in the sequence diagram contained in figure 4.1, which illustrates the flow of messages between the different roles of devices within the network.

Device discovery – Controller → Sensors/Acutators

In order for a controller to locate devices on the network, a device query must be broadcast across the network; any devices which match the request and have resources available can reply with an acknowledgement, otherwise they ignore the message, implicitly signalling no. If a controller receives no responses from the initial query, it could be a result of either, no devices on the network matching the request, or a response packet was lost/corrupted on the network. Because it's not possible to distinguish between an implicit no and a network failure, it may be necessary to retransmit a query multiple times, perhaps with an exponential back-off, so as not to congest the network whilst still achieving a confident scan of the available devices on the network.

Set up channel – Controller → Sensors/Actuators

In order to ensure that a channel is fully set up, it becomes necessary to introduce reliability, via acknowledgement messages. Similar to TCP, a three-way handshake is used to ensure the channel is fully opened on both sides.

- This covers most scenarios and allows for devices to recover from dropped packets during the exchange. For example, device A initiates setting up a channel,
  - if the initial message is dropped, an acknowledgement timeout occurs on A, A then resends message.
  - if B's ACK is dropped, an acknowledgement timeout occurs on A, A then resends the message with the same sequence number, when B receives the initial message again, its ACK message must have been dropped, so B resends the ACK message.

- if A's ACK message is dropped, an acknowledgement timeout occurs on B, then B resends the second message with the same sequence number, when A receives B's ACK message again, its ACK message must have been dropped, A then resends the ACK message.
- if A receives no more ACK messages from B, it knows the channel has been fully set up.

Send sensor data – Sensor → Controller

Once a channel has been set up to a controller, a sensor then needs to send its sensor readings at the previously agreed upon frequency. Due to the high frequency requirements and transient nature of the data, using reliability becomes an expensive overhead without providing much benefit, i.e., consuming twice the bandwidth, and if a packet is dropped and resent out of sequence it becomes useless in a real-time environment, like the "Internet of Things". Without reliability, packets may be dropped, but because of the timeliness of the data it will usually go unnoticed to the user. For example, a dropped packet won't have noticeable effect on a real-time temperature display with a frequency granularity of 5 seconds.

Send command – Controller → Actuator

Once a channel has been set up to an actuator, a controller needs to be able to send it commands to carry out. To ensure that an action is carried out, it becomes necessary to acknowledge each command. By using acknowledgements in combination with message sequence numbers, it is possible to ensure that a command is carried out exactly once per request.

- The following cases are covered via the use of acknowledgements and sequence numbers
  - if command message is dropped, an acknowledgement timeout occurs on the controller, controller then resends command.
  - if ACK message is dropped, an acknowledgement timeout occurs on the controller, controller then resends command with the same sequence number, when the actuator receives repeat commmand, it ignores command and resends ACK message.

Ping – All roles

Once a channel has been set up between two devices, it is possible for one to disappear without closing the channel gracefully, which could be due to various reasons e.g., loss of battery power, failed network link, device failure, etc. Therefore it becomes necessary to test the channel and see if the other participant is still alive, terminating it and cleaning up the channel if not.

Other cases for its use in this protocol are the following: for the controller to check that the channel to the actuator is still alive, and for the sensor to check that the controller is still alive to receive its data, ensuring network resources are not being wasted sending data to a disconnected controller.

To test the channel a PING is sent from one device to the other, with the expectation of receiving a PING ACK to signify the channel is alive.

Close channel – All roles

Once a channel has been set up between two devices, one of the devices may decide at a later time that it either no longer needs the services provided by the channel, or is unable to support it, perhaps due to limited power.

Therefore, it becomes necessary to close the channel gracefully, ensuring the both ends understand the channel is closed and all associated state destroyed. In order to ensure the channel is closed on both sides correctly, an acknowledgement must be used to signify the other side received the command.

- The following cases are covered via the use of acknowledgements between devices A and B:
  - if the close channel message sent from A is dropped, an acknowledgement timeout occurs on A, A then resends the message.
  - if B's ACK message is dropped, an acknowledgement timeout occurs on A, A resends the close channel message, when B receives the message it checks for any channel associated to A and closes it (if not done already), then resends the ACK message to A.

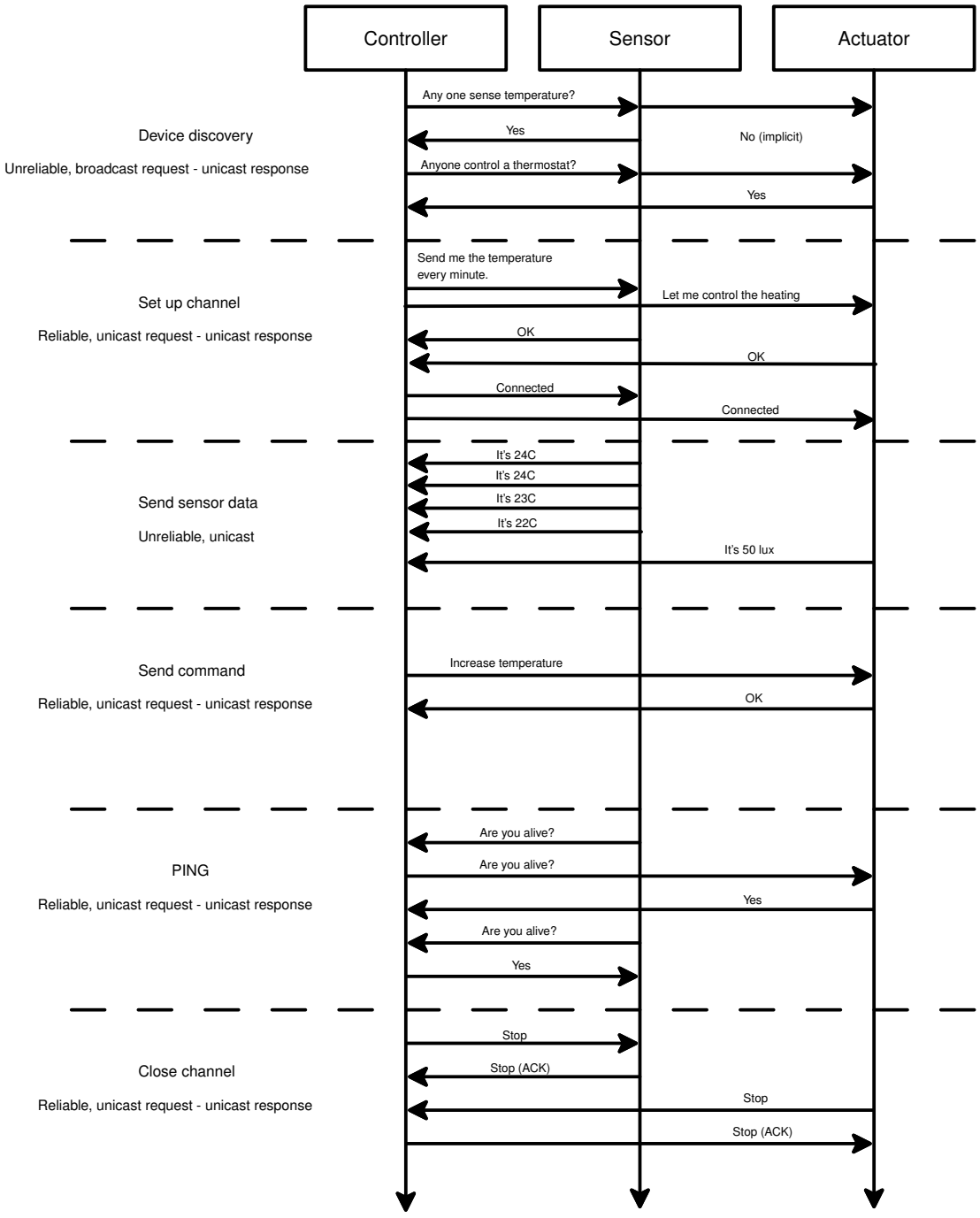

Figure 4.1: Controller - Sensor message diagram

4.2.9 Protocol state machines

With the use of the message sequence diagrams to help understand the flow of messages within the network, the sequence of states each connection within a role can transition between can be discerned, and is shown in the following figures.

Within these diagrams, a number of message types are show as labels on the transitions between the various states within each system, these message types are later described in the next section, 4.3

Sensor

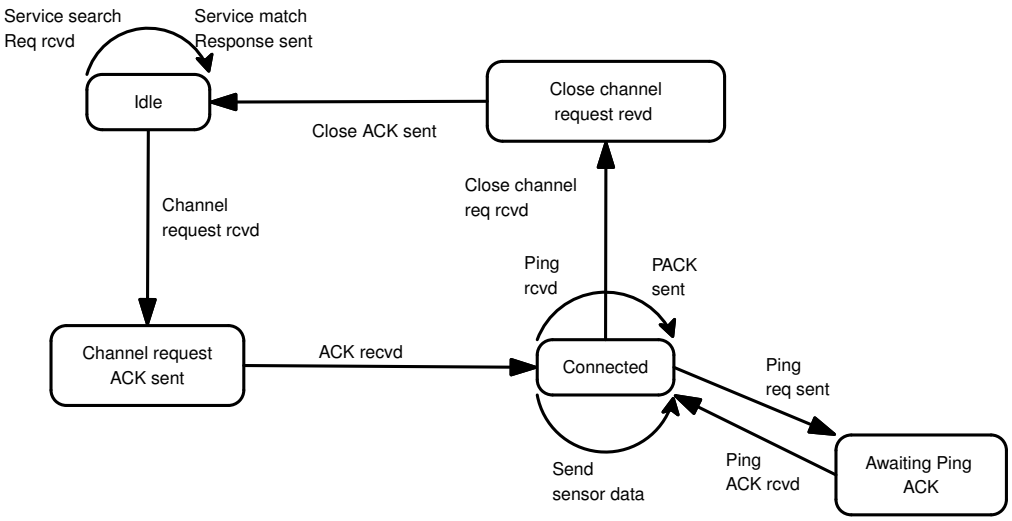

Figure 4.2: Sensor life cycle diagram

Actuator

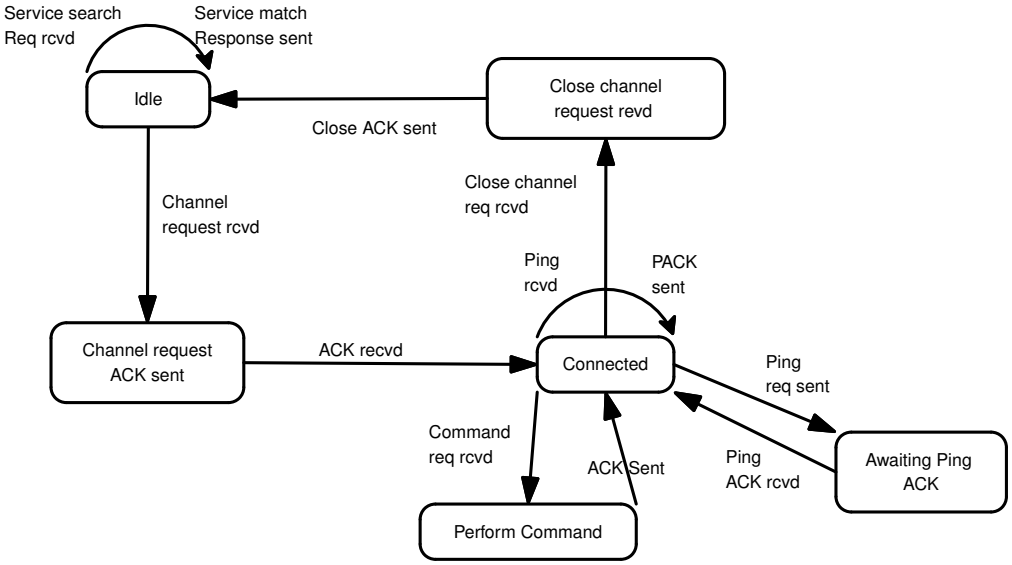

Figure 4.3: Actuator life cycle diagram

Controller

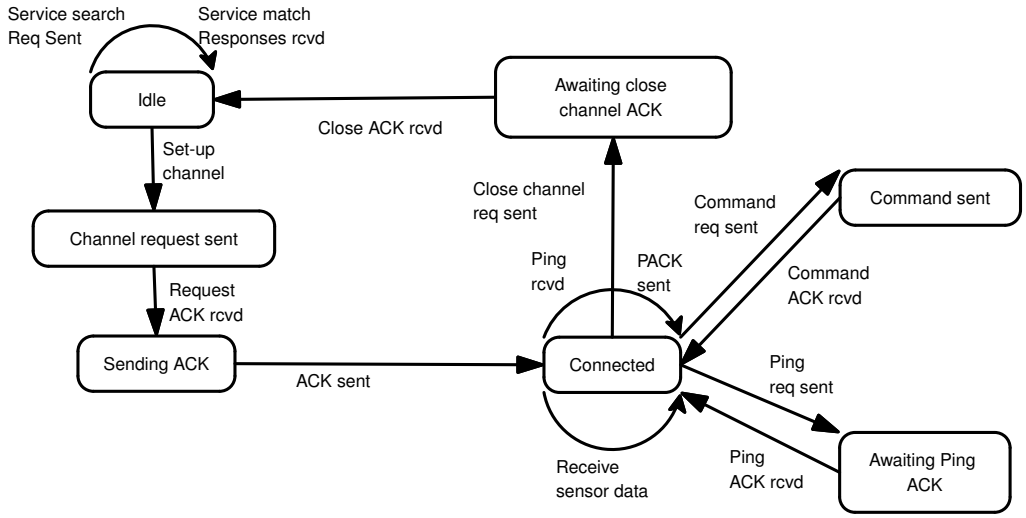

Figure 4.4: Controller life cycle diagram

4.3 Protocol Data Unit

This section discusses and illustrates the design of the Protocol Data Unit (PDU), starting with the protocol header format and followed by the various message types and their respective payload formats. Where appropriate, for each message type, a short pseudo code analysis will be discussed to illustrate the various scenarios that could occur in the relevant state machine, as a result of receiving the respective messages.

4.3.1 Protocol Header

The design of the protocol header tries to minimise the overall size by only containing a minimal set of non-textual data fields, with the purpose of reducing the power consumption for transmitting and receiving packets. The included fields are essential for identifying channels, distinguishing message types and ensuring the integrity of the data transmitted.

The protocol header contains 6 data fields:

- source and destination channel fields to identify the connection (used with lower layer addressing to uniquely identify each connection),
- a sequence number field to detect duplicates and out of order packets,
- a command field to distinguish what type the payload is,
- a payload length field which contains the size of the payload, maximum size = 32 bytes
- and a checksum is computed over the entire packet to detect any errors that might have occurred during in transmission.
  - When initially computed for the packet, the checksum field is filled with 0's and then filled with the result. Once received, the checksum can then be recomputed and the result should be 0 if the packet is correct.

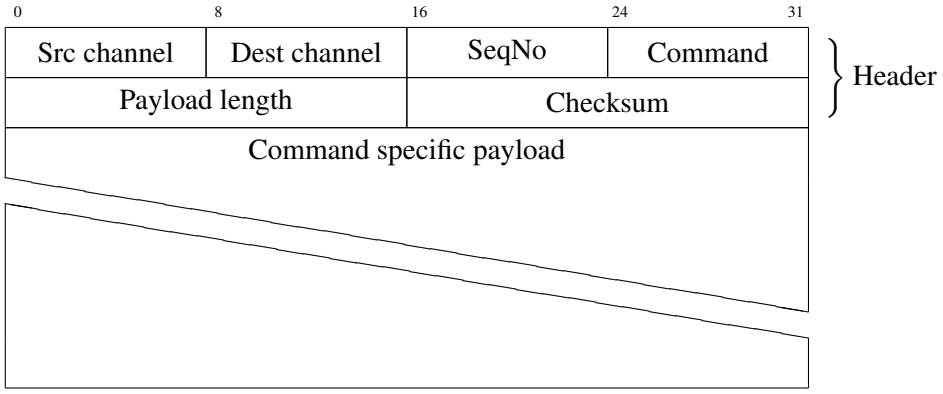

Figure 4.5: Protocol header format

To further illustrate the different classes of messages and how they relate to the header, a packet class digram is shown in figure 4.6.

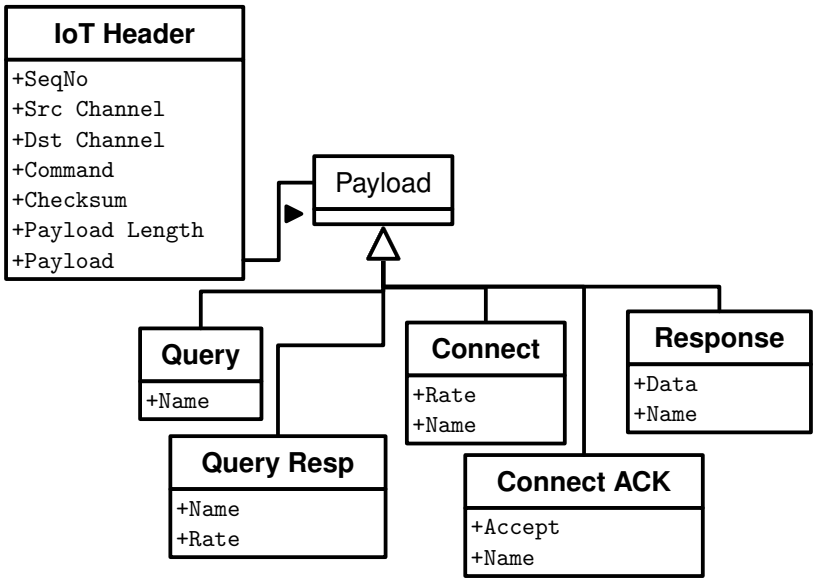

Figure 4.6: Message hierarchy

4.4 Payload Message Types and Formats

This section discusses the various payload message types which can be sent, and the associated actions taken upon receiving a message. For each message type the payload format is described and illustrated.

4.4.1 QUERY

Before a controller can connect to any devices, it must first try to discover any matching devices on the local network. To do this, the device's home channel sends a QUERY message to the home channel on every other device in the network.

The receiving state machine does the following on receipt of a QUERY message:

- if the destined channel is the home channel, it checks the type field,

    - if it matches its own type then transmits a unicast QUERY Response message back to the sender.
    - if the type does not match, message is dropped and device does not respond.
  - if the destined channel is not the home channel, the message is dropped as the requesting channel is confused.

The QUERY message contains two data fields: the type of device the controller is searching for, light sensor, temperature sensor, informational display or otherwise, and the name of the device sending the QUERY message. This packet is sent as a broadcast transmission to all devices on the network.

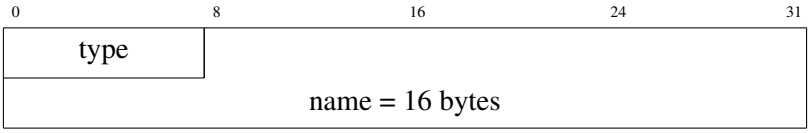

Figure 4.7: QUERY message payload format

4.4.2 QUERY Response

After a device receives a QUERY message, it can decide whether or not to reply to it, based on a type match and if there are resources available, if so it replies with the QUERY Response message.

The receiving state machine does the following on receipt of a QUERY Response message:

- if destined for the home channel, it accepts all messages and passes them up to any application on top.
- if not destined for the home channel, then the message is dropped as the requesting channel is confused.

The QUERY Response message contains 3 data fields: the type and name of the device which is responding and the rate, which either indicates the default rate that a sensor transmits readings at, or the rate at which an actuator will expect PING messages from a connected controller to confirm the connection is still alive.

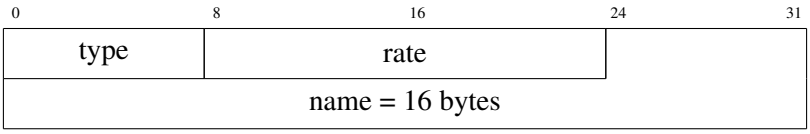

Figure 4.8: QUERY Response message payload format

4.4.3 CONNECT

Once a device has been located, either a priori or through querying, a controller needs to send a CONNECT message to start the process of creating a channel with the desired device. As shown in the message sequence diagrams, this uses a three-way handshake, of which this is the initial message.

The receiving state machine does the following on receipt of a CONNECT message:

- if destined for the home channel, and the device has not previously allocated a channel for the requesting device, it checks if the rate is compatible with its own,

    - if compatible, it creates a new channel for this connection, sets it to the CONNECT sent state and responds with an accepting CONNECT ACK from the new channel.
    - if not, respond with a declining CONNECT ACK.
  - if it has already allocated a channel for the requesting device and is in the CONNECT sent state, then the CONNECT ACK must have been lost, CONNECT ACK is resent.
  - if it has allocated a channel for the requesting channel and is in the CONNECTED state, then the requesting device state machine is confused.
  - if not destined for the home channel, then the requesting channel state machine is confused.

The CONNECT message contains 2 data fields: the rate at which the controller wants to either receive data readings from sensors, or receive PINGs from actuators, and the name of the controller.

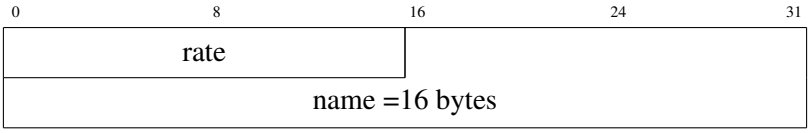

Figure 4.9: CONNECT message payload format

4.4.4 CONNECT ACK

Once a device receives a CONNECT or CONNECT ACK message, unlike when receiving a QUERY message, it must respond with the second or third part of the three-way handshake with either a positive or negative acknowledgement in order to ensure the connection has been set-up fully. When responding to a CONNECT or CONNECT ACK message, the device can either accept the connection or decline it, which could be based on device being unable to match the demands from the controller or due to a lack of available resources needed to support the connection.

The receiving state machine does the following on receipt of a CONNECT ACK message:

- if destined for an ephemeral channel which is in the CONNECT sent state,
  - check if the connection has been accepted, if so reply with another CONNECT ACK.
  - otherwise, close the newly opened channel.
- if destined for an ephemeral channel which is in the CONNECT ACK sent state,
  - check the accept field to confirm the connection has been fully accepted, if so move to the CONNECTED state
  - otherwise, close the newly opened channel.
- if destined for an ephemeral channel which is in the CONNECTED state, third handshake (CONNECT ACK) must have been lost, resend CONNECT ACK.
- if destined for an ephemeral channel which has not been allocated or does not exist, requesting channel must be confused.
- if destined for home channel, requesting channel is confused.

The CONNECT ACK message contains 2 data fields: accept, which declares whether or not the recipient accepts to create a channel, and the name of the device sending the message.

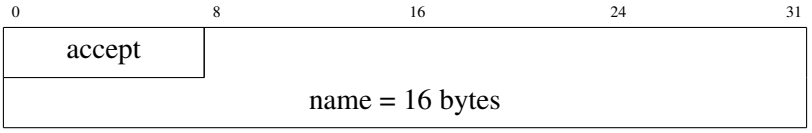

Figure 4.10: CONNECT ACK message payload format

4.4.5 RESPONSE

Once a channel has been set up and is in the connected state, the sensor connected to the channel can then send RESPONSE messages containing its sensor readings at the previously agreed upon rate.

The receiving state machine does the following on receipt of a RESPONSE message:

- if destined for an ephemeral channel which is associated with the sender channel and in the CONNECTED state
  - accept the data and pass to application on top
- if destined for an ephemeral channel which is associated with the sender channel, but is not in the CONNECTED state, then sender is confused.
- if destined for an ephemeral channel which is not associated with the sender channel, then sender is confused.
- if destined for home channel, then sender is confused.

The RESPONSE message contains 2 data fields: the data being sent, for example a temperature reading or a light reading, and the name of the sensor.

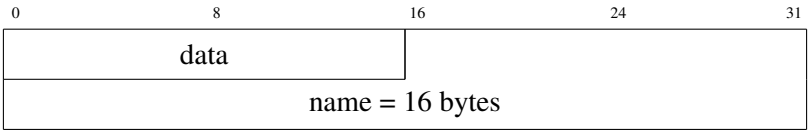

Figure 4.11: RESPONSE message payload format

4.4.6 COMMAND

Once a channel has been set up between a controller and actuator, the controller can then send COMMAND messages to the actuator indicating that it needs to perform its action. This is part of a reliable request-response sequence of messages and expects an acknowledgement in return to signify the command has been received, but not necessarily carried out.

The receiving state machine does the following on receipt of a COMMAND message:

- if destined for an ephemeral channel which is associated with the sender channel and in the CONNECTED state
  - executes command and replies to sender with a COMMAND ACK
- if destined for an ephemeral channel which is associated with the sender channel, but is not in the CONNECTED state, then sender is confused.
- if destined for an ephemeral channel which is not associated with the sender channel, then sender is confused.

- if destined for home channel, then sender is confused.

These is no payload specific data for the COMMAND message.

4.4.7 COMMAND ACK

The COMMAND ACK is the response to a COMMAND message, which must be sent upon receipt of the COMMAND message.

The receiving state machine does the following on receipt of a COMMAND ACK message:

- if destined for an ephemeral channel which is associated with the sender channel and in the COMMANDSENT state
  - cancels retry timer and changes to back to CONNECTED
- if destined for an ephemeral channel which is associated with the sender channel, but is not in the COMMANDSENT state, then sender is confused.
- if destined for an ephemeral channel which is not associated with the sender channel, then sender is confused.
- if destined for home channel, then sender is confused.

These is no payload specific data for the COMMAND ACK message.

4.4.8 PING

Once a connection has been made between two devices, a channel is created within each device to handle this connection. Due to the nature of devices, especially those on wireless networks or battery powered, devices can disappear off the network without gracefully closing the connection. If this occurs, then the connected devices must be able to detect if the connection is still alive, if not then clean up after it. To do this PING and PING ACKs are used to create a failure detector.

The receiving state machine does the following on receipt of a PING message:

- if destined for an ephemeral channel which is associated with the PINGing channel then respond with a PING ACK
- if destined for an ephemeral channel which is not associated with the PINGing channel, then drop message as PINGing channel is confused.
- if destined for the home channel, drop packet as PINGing channel is confused.

These is no payload specific data for the PING message.

4.4.9 PING ACK

The PING ACK is the response to a PING message for use in detecting failures in the network.

The receiving state machine does the following on receipt of a PING ACK message:

- if destined for an ephemeral channel which is associated with the responding channel then reset the PING timeout as well as the deletion timeout, to stop any further PINGs and ensure the channel is not deleted.
- if destined for an ephemeral channel which is not associated with the responding channel, drop the message as the responding channel is confused.
- if destined for the home channel, drop packet as responding channel is confused.

There is no payload specific data for the PING ACK message.

4.4.10 DISCONNECT

When a device has already formed a connection to another device and wishes to disconnect, perhaps due to either power concerns, the device is no longer needed or otherwise, a method for gracefully disconnecting the two devices and cleaning up the channels is necessary.

The receiving state machine does the following on receipt of a DISCONNECT message:

- always send the DISCONNECT ACK message back to the sender (otherwise ignoring it will likely cause another to be sent).
- if the destined channel is an ephemeral channel which is associated with the DISCONNECTing channel, then close the channel.
- otherwise drop message.

There is no payload specific data for the DISCONNECT message.

4.4.11 DISCONNECT ACK

The DISCONNECT ACK is the response to the DISCONNECT message used to disconnect two devices gracefully.

The receiving state machine does the following on receipt of a DISCONNECT ACK message:

- if destined for an ephemeral channel which is associated with the responding channel, then close the channel.
- if destined for an ephemeral channel which is not associated with the responding channel, drop the message as the responding channel is confused.
- if destined for the home channel, drop the message as the responding channel is confused.

There is no payload specific data for the DISCONNECT message.

4.5 Comparisons to other systems

This section will compare and contrast some of the key design choices of this new system with the design of other pre-existing protocols and systems.

Distributed vs Centralised

One of the fundamental design decisions in creating the protocol, was to create a distributed system, which would relieve the system of a single point of failure, but in turn increase the complexity in discovering new devices. In contrast to this, the Java Messaging System (JMS) adopts a centralised approach, using a central name server to discover and locate devices on the network. Whilst JMS is not directly targeted towards "Internet of Things" networks, it does provide adequate abstractions that could be used in an "Internet of Things" network, such as the Publish - Subscribe model.

Because of the centralised name server, which is known a priori, it becomes very simple to join the network, by just querying the server for any devices or services that it is interested in. To contact the server it only requires a single unicast message, in contrast to the broadcast message sent to every device on the network for the proposed distributed system.

However, the centralised approach assumes that the network and devices used are reliable and are unlikely to fail, which is incompatible with the assumed traits of an "Internet of Things" network implemented on constrained devices. In the case where the central server fails, the new devices entering the network would no longer be able to connect, and if the central server was used to facilitate the communication via Publish - Subscribe, then the whole system would be rendered inoperable. In the constrained devices environment where resources are limited and wireless links are susceptible to interference, this becomes a significant problem.

Therefore, although the cost of device discovery does increase with the use of a distributed system, it offers far more fault tolerance than a centralised system.

Selective Unicast/Multicast vs Multicast

The xAP system chooses to use a multicast medium to interconnect all devices, where every device broadcasts its events/commands to every other device on the local network. This provides the benefit that devices don't need to explicitly set up connections between themselves and instead just listen for broadcasted packets on the network. Whilst this is truly distributed in nature, it does come at a considerable cost as the network scales to large quantities of devices. The core problem with this approach is that every device on the network is forced to receive and process every packet that is transmitted, regardless of whether it's relevant information or not. For constrained devices this is highly undesirable, as resources are severely constrained, especially battery life and processor speed. Using the radio on a constrained device is the most expensive operation, therefore by forcing all messages to be received, the power consumption on such a device becomes a significant problem.

Hence, the need to reduce the use of broadcast is necessary, but due to the distributed architecture its use is still necessary to discover devices. In the proposed protocol a compromise is made, in order to discover devices broadcast queries are used, but every other message in the protocol is unicast. This ensures that devices can still easily join and discover other devices on the network, but without the continual cost of each device receiving every devices data.

Another problem with the broadcast-only network approach is the issue of reliability; how can a device be sure that what it has sent, be it a command, sensor reading or otherwise, has been received, either by the device it wants to send it to, or anyone at all? A recipient could reply with an acknowledgement but that would further congest the broadcast-only network, especially in the case when multiple devices want to acknowledge the reciept.

Cloud vs Local

SmartThings, unlike the proposed design, chose a cloud-first approach in connecting the different devices on the network together. In comparison the the proposed design it does pose some benefits, such as instant and easy connectivity to other cloud services and making the devices accessible from anywhere with an Internet connection. But as previously discussed, this also poses several risks as well, these include security risks of the data both in transit and storage (who has control and access to the data or control devices, is it being intercepted?) and reliability risks (what happens if the cloud fails? Will all the doors unlock and the sprinklers not stop?). In regards to both risks, there have been various failures in the industry relating to both the security of data,[22], and the reliability of cloud services / online services [2, 10].

Adopting the alternative, the proposed system uses a local-first approach, which gives the user complete control and access to the data, enables the system to function without Internet connectivity and if desirted can be expanded by connecting to an Internet connection. This provides the safety and assurance of a local system without compromising the possibility of upgrading it to support the cloud based features which SmartThings benefits from.

Textual encoded packet vs Integer encoded packet

As one of the requirements, the new protocol must ensure an efficient use of resources, in which minimising the size of the packet is most important, because the protocol centres around sending, receiving and processing these packets. In order to achieve this, the packet header only contains a minimal set of fields, all of which are non-textual.

However, if the type of a device was described using a string representation, such as "Light Sensor", it would consume 12 bytes of data, not including any associated fields such as text length to allow for variable length identifiers. Instead, by using an integer encoding, whereby the types are predefined integers within the protocol, given the assumption that there are no more than 255 devices, the size of the field can be reduced to a single byte. Applying this to the rest of the header results in a total size of 12 bytes (not including the payload). This not only considerably reduces the cost of sending and receiving packets, but also reduces the complexity in parsing the data.

Compare this to xAP, which is based on the requirement of human readable messages, and exclusively uses a textually formatted packet header and payload. To illustrate xAP's textual format, an example header is shown in figure 4.12 In total, 87 bytes are used to represent a similar array of information to that contained within the

```
xap-header                          ← signals block is header
{
v=12                                ← defines protocol version
hop=1                               ← how many hops left
uid=FF123400                        ← a unique identifier for the source
class=xap-temp.notification         ← the type of payload the message contains
source=ACME.thermostat.lounge       ← text representation of source of the message
}
```

Figure 4.12: xAP packet header

designed system's header, which consumes 12 bytes. In contrast, xAP consumes almost 8 times more bytes. This has a large impact on the potential transmission cost and processing complexity if used on constrained devices, such as the TelosB mote.

Reliable vs Unreliable

The proposed protocol is designed to run on top of an unreliable link (e.g. UDP), enforcing reliability itself, but only when absolutely necessary. The reasons for this are related to power consumption, where reducing the number of packets both sent and received can have a significant impact on the overall battery life of constrained devices.

The reasons for not implementing the protocol on top of a reliable link, like TCP, is such that TCP requires an acknowledgement for each packet sent, therefore doubling the total packets sent and received. Not only this, but if packets are lost and resent, they will be received out of sequence; in the "Internet of Things" environment where most data sent is ephemeral, receiving packets late and out of sequence is pointless, as the interest in most cases is the present moment and not the past.

Therefore the choice to use an unreliable link, such as UDP, combined with building reliability on top for only absolutely necessary messages, such as connection handling, ensures that the overall sending and receiving of packets is reduced, reducing the overall power consumption, but without compromising the operation of the protocol.

CoAP

Comparing the proposed protocol to the aforementioned CoAP, currently in development in the IETF, both try to tackle the problem of designing a protocol suitable for the constrained devices. Taking similar approaches both protocols are based on a distributed architecture, whereby no central server has to be used to locate devices. Instead both allow the device to either know the target device a priori or discovery it via broadcast queries.

One of the significant differences between the two, is that CoAP is designed around generic, non-specific devices; whereas the proposed protocol classifies devices into the three main roles, sensors, actuators and controllers. Because of this, CoAP doesn't impose certain functions on devices, in the same way that the proposed protocol enforces that a sensor must transmit data at a set rate to its connected devices. However, CoAP does support the possibility of such functionality; devices can subscribe to one another, by doing so they will be informed by the subscribed device when an event has occurred, after which the subscriber can then decide to pull the event data from the subscribed device. This functionality works similar to a controller and sensor forming a connection and sending data, however because of the additional step of notifying the subscriber, additional packets must be sent and received by both devices.

Lastly, One of the core goals of CoAP is to ensure that constrained devices can easily transition into the World Wide Web, which is enabled by the RESTful style protocol, where commands (GET, PUT, etc) are used that are very much like those used in HTTP. This ensures minimal translation of packets needs to be performed. However, this is not the case in the proposed protocol, whereby instead an application would need to be built on top of the protocol to support WWW integration, due to the unrelated packet format.

Chapter 5

Implementation

Following on from the design, this chapter discusses the process of taking the protocol from design to implementation on constrained devices. The chapter discusses the challenges faced throughout the implementation process, and the development on the event based operating system, Contiki, for the TelosB mote.

5.1 Implementation break-down

In order to manage the implementation of the protocol, the development was broken down into functional components; these components were then implemented individually and tested, ensuring that the system functioned correctly before developing the next component.

Because of the functional similarities between the sensor and actuator, the initial development focused upon the communication model betweet the sensor and controller, with the sensor later modified to support the actuator functionality.

The implementation was broken down into the following:

1. Test basic broadcast and unicast communication between two devices.
2. Define packet structure, payload structures and device types.
3. Implement broadcast device discovery and unicast response mechanism.
4. Implement single channel device connections between devices after discovery.
5. Implement sending sensor data and pings between connected devices.
6. Implement multiple channel support for devices.
7. Implement channel closing.
8. Implement API bindings for application support.
9. Implement mock applications on top of API.

5.2 Initial challenges

This section will discuss the challenges faced which initial choice of available hardware and software platforms.

5.2.1 Arduino

As described in the background chapter, 2.3.1, the Arduino is unanimously the most popular open source constrained device available for consumers; thus developing an implementation of the protocol for it would allow it to be used with all the Arduino compatible devices currently available.

However, after much research into the platform, it was discovered that the Arduino did not natively support hardware interrupts from the available Ethernet network board.[3] Instead any program expecting data from the network is forced to poll the Ethernet driver to check if any packets have arrived, resulting in a busy-wait. This busy-wait method of acquiring data from the network is extremely inefficient and directly opposite to what the protocol requirements and design aims to achieve, thus the Arduino was not chosen as the target development platform.

5.2.2 TelosB and operating system choice

After deciding not to develop for the Arduino platform, the alternative platform available, due to the resources available in the School of Computing Science, was the TelosB mote. As described in section 2.3.3, the TelosB mote is a widely used device within academia for research into wireless sensor networks and constrained operating systems. Because of this research, there are several operating systems to choose from when developing for the device, including TinyOS and Contiki.

Between TinyOS and Contiki there are some considerable differences, both in the programming model, event based vs hybrid (threaded & event driven), and the structuring of the operating system, discussed previously in section 2.3.3. Because of Contiki's programming model and language similarity to native C, it was chosen over TinyOS, in the hope that it would prove to be considerably easier to develop for.

5.3 Telos B Mote and Contiki implementation

After choosing to develop for the TelosB mote with Contiki, the next step involved getting to grips with how to best implement the system on the platform. This section discusses the process of implementing the system on the chosen platform, the various issues encountered, the overall structure dictated by the platform choice and the resulting system created.

5.3.1 Contiki & Protothreads, a hybrid multi-threaded and event driven system

For implementing the system on Contiki, the hybrid programming model Contiki uses must first be understood. When developing an application in Contiki, the application can use one or more protothreads to run. Contiki's protothreads are lightweight, stackless threads which allow for linear code execution within an event-driven system, and also provide functionality to wait on events, either posted from other protothreads or from external events, such as a button press. This enables a developer to mix multiple threads of execution with an event-driven system to create a reactive system; this not only takes advantage of the constrained devices but also helps reduce the power consumption by sleeping when all threads are waiting on external events.

5.3.2 Overall system architecture

Before implementing the system, the overall architecture had to be decided upon, shown in figure 5.1. In particular the method by which separate components, implemented as protothreads, interacted with each other within the system. For interaction two methods could be used:

- Callback functions, by passing pointers of functions from the application to the API, when an event arrives such as a sensor reading, the appropriate function in the application can be called, passing it the related data.
- Posting events, by defining new protocol specific events which the application program can wait on, and when an event occurs, such as a sensor reading arriving, the event is then rippled up to the application passing with it the related data.

For the majority of the system, the event based approach was chosen because fits in better with the event driven style of the rest of Contiki and also means the application can wait on events to occur and then wake up directly. However, to simplify retrieving sensor readings and calling actions from the application, function pointers to functions which return the wanted data or carry out an action will be used, this enables the system to call the functions directly without having to wait for an event to be posted to the application and then waiting for one to return.

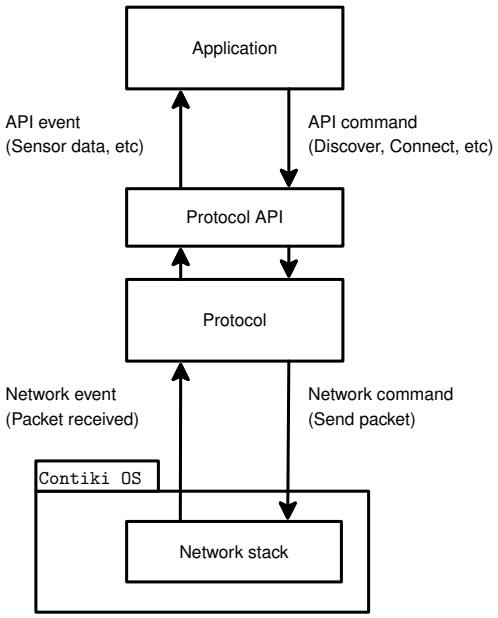

Figure 5.1: Overall system architecture

5.3.3 Choosing the transport layer: uIP vs Rime

Within the Contiki OS, several communication stacks are implemented, including a miniature IP stack called uIP (IPv4 & IPv6) and the native Contiki Rime stack. Both stacks claimed to support the mechanisms required for the protocol, both broadcast and unicast, so either could be used. In an effort to enable easy integration into other IPv4 networks the uIPv4 stack was chosen. This meant that packets passed around the network could be routed to a gateway and sent to a neighbouring IPv4 network and possibly then on to the Internet without the need for translation.

5.3.4 Problems Encountered

Whilst learning about and developing the system in Contiki, a variety of issues and problems occurred, each of which affected the time spent and complexity of the project in some way.

Documentation

Whilst Contiki is a well developed and established operating system for constrained devices, such as the TelosB mote, the documentation for the OS and its libraries are mostly outdated, incomplete and provide no easy place to get started. Although most of the C syntax, structure and libraries remain consistent and intact, it's still fairly difficult to learn quite how the Contiki protothreads actually work and integrate with the system. Because of this a considerable amount of time was spent searching for up to date documentation.

Constrained resources

Using the Contiki OS, which provides a significant amount of the standard C libraries that C programmers are familiar with, can provide some initial problems. An example of this is the inclusion of malloc, which in a normal x86 pc environment is very useful to enable run time allocation of data structures. But when used in a constrained environment where RAM is extremely limited, it becomes useless due to the limited heap space and the fragmentation of memory.

Because of this, it is necessary to use static allocation to ensure memory is allocated correctly, but with this another problem arises. Because statically allocated memory is allocated at compile time, the size of the memory required needs to be known at compile time as well. When allocating fixed data structures this is not a problem, as the size of the data is already defined and known at compile time. Whereas, when allocating memory for the packets in the proposed protocol, it becomes difficult due to the varying payload lengths. In order to resolve this, the size to allocate for a packet can be defined as, size of the packet header + size of the largest payload. Doing this ensures that memory is always available for any packet, but at the same time for packets which don't have any payload the overallocation can waste a significant portion of the limited memory. In the implementation, this method of allocation is chosen, and each channel in the system is allocated one packet to send and receive messages.

As an alternative, which could save space at the cost of complexity, two pools of packets could be allocated; one pool stores packets with only the header, which would be used for sending messages with no command-specific payload, and the other pool stores packets with the maximum payload size, which would be used for all other command types. The complexity increases due to the need to ensure packets are allocated correctly between channels, ensuring that enough are available to service any incoming or outgoing request. In the case of resending packets, due to timeouts or repeat requests, if packets were returned to the pool after each use, the data for the retransmission would need to be regenerated; whereas in the implemented case, packets are only ever used by the channel to which they were assigned, so if the last packet needed to be resent, the data would still be contained within the stored packet, without the need of regeneration.

IPv4 Broadcast support

After having chosen to use the uIPv4 UDP stack for communication, it was later discovered that the broadcast mechanism was just a stub. The uIPv4 stack is implemented on top of the native Rime stack, so all IP packets are routed around using the Rime primitives for messages. Through much investigation it was found that when a broadcast packet is sent, it is given the normal "all ones" broadcast address, 255.255.255.255, but when the

uIP layer is passed the packet to send, it detects the destination is outside of the local network and tries to find a gateway to send it over. Instead, the address should be recognised as a broadcast destination and then broadcast over the network.

After many attempts at contacting the Contiki developers community, no answer or solution was found; so instead a fix had to be created. As mentioned previously the uIPv4 stack is run over the Rime stack, and as discovered, using only a unicast Rime connection. In order to allow for broadcast packets to be sent, a new broadcast Rime connection needed to be created and correctly set up without breaking the existing code base. Once the broadcast code was in place, the system was tested to ensure it worked correctly and as expected for both unicast and broadcast packets.

5.3.5 Final implementation and API

After deciding upon the architecture of the system, the communication stack to use and overcoming the various problems which arose throughout development, the implementation of the protocol on the TelosB motes with Contiki was completed.

As shown in figure 5.2, the implementation of the protocol uses only one protothread to receive, process and send any messages. The protothread is only active when an event occurs, either from the UDP stack, indicating a message has arrived, or from one of the timers in the system. In the implementation there are two types of timers, the clean up timer and the channel timer. The clean up timer is created once for the whole system and manages packet timeouts for reliable messages, occurring every 20ms. The channel timer, is instantiated once for each channel in the channel table; depending on the device, it either signals when the device should send out a sensor reading on a particular channel, which is defined when the channel is set up, or it defines when the actuator should send a ping to the connected controller to signify it is still alive. In the case of the controller it is used to signify by when it should receive a packet from a channel, defined as 3 times the predefined rate.

For communication between the protocol and the application, as discussed previously, events are used to pass information back to the application and commands are used to initiate actions, such as discovering devices, connecting to devices or closing connections.

Whilst the implementation adheres to the protocol design and specification proposed in chapter 4, it does have some limitations in its current form. These include:

- Channel table limited to 10 channels due to memory constraints.
- Two devices can create multiple channels between themselves.

5.3.6 Testing and Simulation

Throughout the implementation it was absolutely necessary to test the code as often as possible to ensure that it functioned correctly and as expected. Tests were performed both on real devices and within the provided simulator, Cooja, for the platform developed on, Contiki on TelosB motes. The Cooja simulator, shown in figure 5.3, provided the ability to test scenarios which wouldn't be possible with the real devices, including testing with 10's of devices and with varying conditions such as radio strength and interference.

To test the implementation, unit tests were performed on each component of the system as it was developed, this was done to ensure that that each component functioned correctly before being integrated into the rest of the system. After which, the system was tested to see if the new component functioned correctly and did not break

the existing code base. Using the Cooja simulator made it extremely easy to perform these tests and see how any changes effected the operation, without the need to reprogram each mote individually.

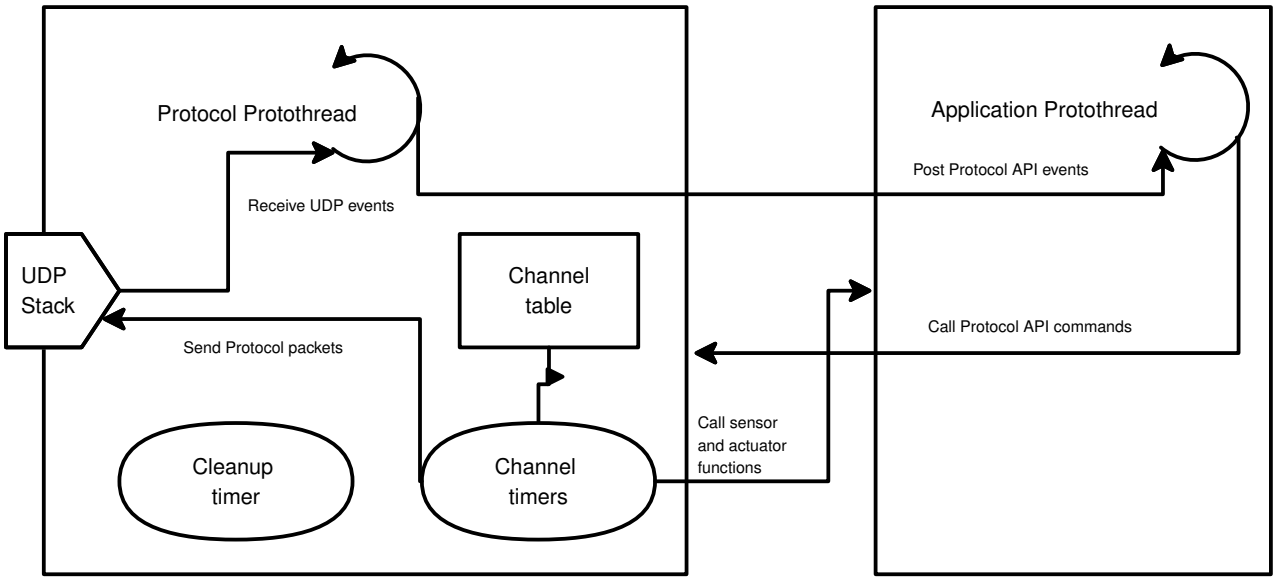

Figure 5.2: Communication and threading architecture

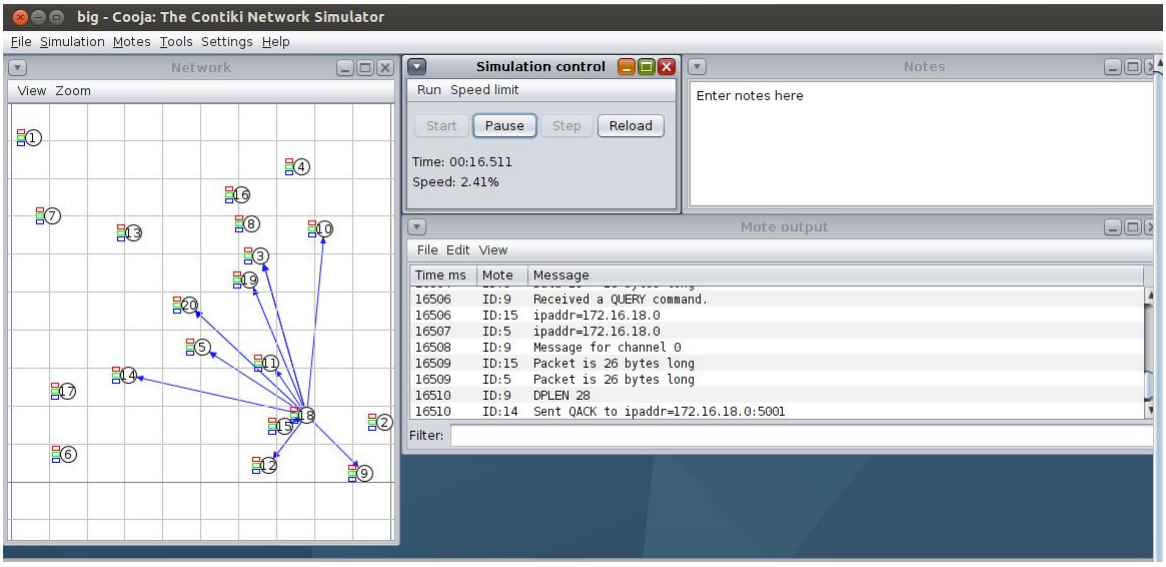

Figure 5.3: Cooja Simulator

Chapter 6

Evaluation

This chapter evaluates and discusses how successful the design and implementation was at meeting the proposed requirements for the project. To do this, the chapter will discuss how the protocol was tested to satisfy the initial use cases, the implementation size, followed by a comparison of the performance characteristics of the protocol and its implementation to other existing protocols / systems discussed in chapter 2.

6.1 Testing

To ensure the protocol worked throughout development, both the simulator and TelosB motes were used to test some of the scenarios described in the use cases contained in chapter 3.

6.1.1 Sensor logging

In order to test the multiple channel support of devices, the scenario of sensor logging was set up; where one or more controllers search the network for available sensors and request data from them at regular intervals. To demonstrate this, the scenario was set up in the Cooja simulator, with one controller device, three light sensor devices and three actuators; the controller was set to locate and connect to three sensing devices which could sense light.

In figure 6.1, the controller is labelled as mote 1, and can be seen to receive messages from sensor motes 3, 5, 7. Motes 8, 9, 10 are actuator motes, listening for any broadcast queries which match their type. In the Mote Output box, each mote prints out its temperature locally, as designated by the Mote column, and sends it to the controller, where it also prints it.

6.1.2 Light/Presence detector

In order to ensure multiple types of devices could be connected, so that a closed loop system could be created, a scenario involving a controller, light sensor and light switch was created. The controller attempts to locate and connect to a light sensor and light switch actuator on the network, using the readings from the light sensor to decide when to turn the light switch on and off. The purpose of such a scenario is to demonstrate the possibility of the “Internet of Things” sensing either, the presence of a user by obscuring a sensor, or the change in lighting conditions, which would require a light to be turned on to assist the user. To test and demonstrate this, the

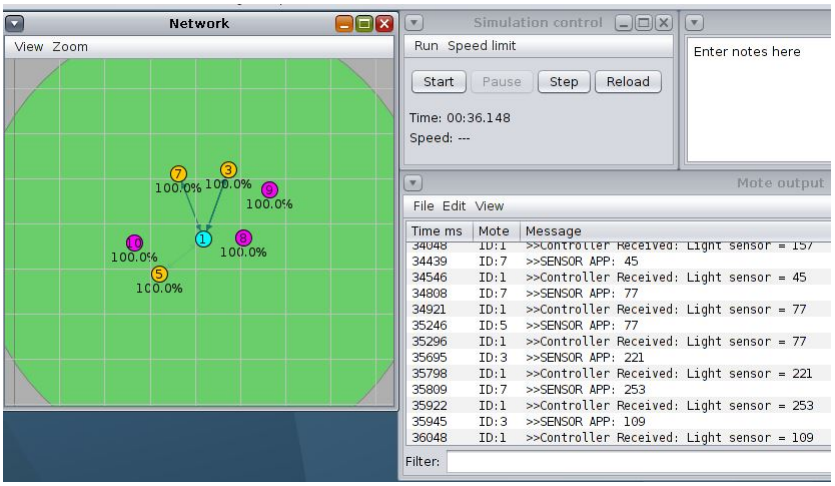

Figure 6.1: Sensor logging scenario

scenario was set up using both real motes and the simulator, with one controller device, one light sensor device and one light switch actuator device.

In figure 6.2, the controller is labelled as mote 2, and can be seen to receive messages from sensor mote 1 and send messages to actuator mote 3. In the Mote Output box, the various print outs can be seen, displaying sensor readings received, decisions based on the sensor readings and the commands sent and received. In the network view box, it's possible to see the LEDs on mote 3 alight, due to the recent command to turn on the lights.

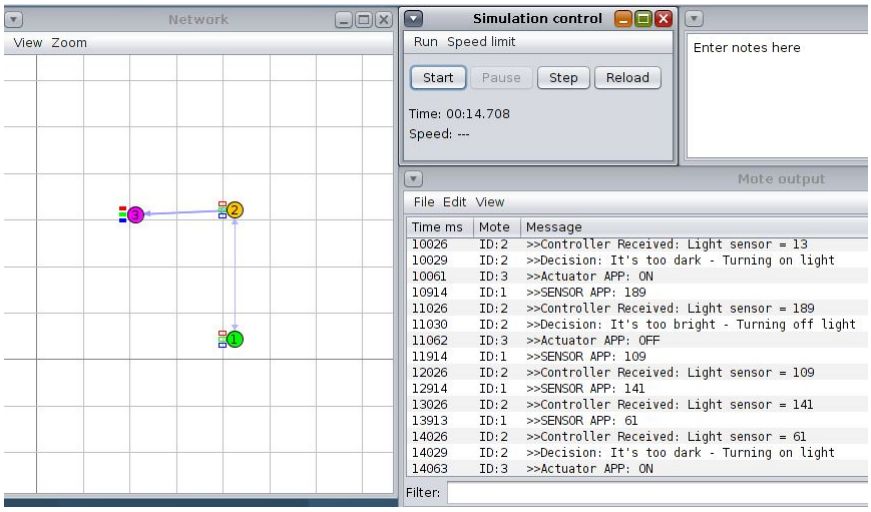

Figure 6.2: Light/Presence detector

6.2 Implementation size

Creating a lightweight protocol is one of the primary requirements, thus ensuring that the protocol could be implemented on even the most constrained devices, such as the TelosB or Arduino. Because of this, each aspect in the protocol had to be made as low cost as possible, whilst still maintaining flexibility and upgradeability. With this in mind, the TelosB implementation performs well in regards to consuming device resources when compared to a variety of other applications, as shown in figure 6.3.

To demonstrate the amount of ROM and RAM occupied, a simple hello world program was created, which

upon receiving an event, output “Hello world” to a connected serial port. This provided the baseline from which to compare the implementation against; also included as another baseline is the same hello world program with uIP included, which shows how much is added to the binary when uIP is included, without any additional code being written.

When comparing the Hello world app with the Hello world app with uIP included, a significant difference can be seen in the size of ROM consumed as well the RAM, leaving around 18,000 bytes ROM and around 3,300 bytes RAM left for an application and other libraries.

The sensor implementation of the IoT protocol with uIP and a minimal sensor application (used to connect a sensor to the API and print out a message), consumes only an additional 2428 bytes ROM and 994 bytes RAM with 10 channels and 554 bytes RAM with 5 channels. Similarly the actuator implementation performs on par, only consuming a minimal amount more in both cases. Unlike the two other implementations, the controller implementation consumes a much higher, 3848 bytes of ROM but with a minimal increase in RAM, due to the necessity to discover, communicate and connect to both of the other device roles.

During runtime, the amount stored in RAM due to the protocol should not change, because of the static allocation for all data structures made at compile time.

Setup	ROM (bytes)	RAM (bytes)	Channels	Notes
Hello world	19,812	5,538	n/a	...
Hello world	29,494	6,764	n/a	with uIP stack.
Sensor app	31,922	7,708	10	with uIP stack, IoT Protocol and light sensor
Actuator app	32,032	7,750	10	with uIP stack, IoT Protocol and LEDs
Controller app	33,318	7,808	10	with uIP stack, IoT Protocol
Sensor app	31,922	7,318	5	with uIP stack, IoT Protocol and light sensor
Actuator app	32,032	7,320	5	with uIP stack, IoT Protocol and LEDs
Controller app	33,318	7,378	5	with uIP stack, IoT Protocol

Figure 6.3: Table showing implementation sizes against stock Contiki

6.3 Comparison to other systems

To better demonstrate the lightweight and scalable nature of the protocol, as well as the fault tolerance which it can provide, two comparisons are made, one on the number of packets received within the network and another on the case of resilience to failures in the network.

6.3.1 xAP - Receiving packets

As described previously in chapter 2, xAP is implemented on top of a multicast network, to enable a distributed, highly fault tolerant network. But as a consequence, the cost of all devices broadcasting packets has a great impact on devices in the network, especially as the number of devices increases.

To contrast this against the proposed protocol, the graphs below show the number of messages received by different types of devices on the network on both the proposed protocol and xAP.

xAP does not differentiate between different devices explicitly, but to better understand the activity of each device both graphs display the messages received by controllers, actuators and sensors. Another key difference to

take into consideration is that xAP does not implement reliability on any of the messages, however does suggest the use of “heartbeats” to signify a device is still alive on the network, therefore “heartbeats” have been included as pings [27].

To observe how both protocols performed as the network scaled, two scenarios are demonstrated:

- Scenario one: 1 controller, 1 actuator and 1 sensor
- Scenario two: 1 controller, 2 actuators and 2 sensors

Whilst neither scenario demonstrates a large number of devices, both clearly show how the traffic changes as the network scales.

To simulate a simple “Internet of Things”, the following messages are sent within the network:

- Sensors send sensor readings to the controller every one second.
- Controllers send commands to actuators every 4 seconds.
- In response to the commands, actuators send acknowledgements to the controller every 4 seconds.
- Sensors send a PING message to the controller as part of a liveness detector every 10 seconds, actuators don’t need to send PING messages when they are receiving messages from the controller and returning acknowledgements, unless they do not receive a message within an expected period of 10 seconds, like the sensor.

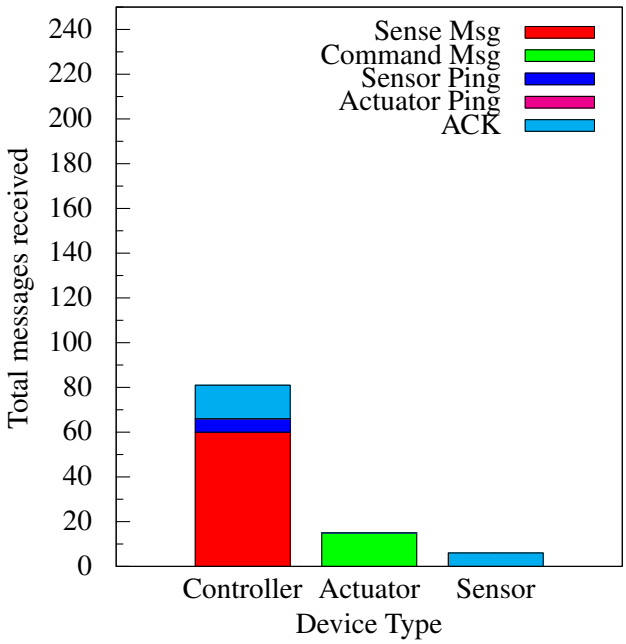

Figure 6.4: Messages received in 1 minute using IoT protocol

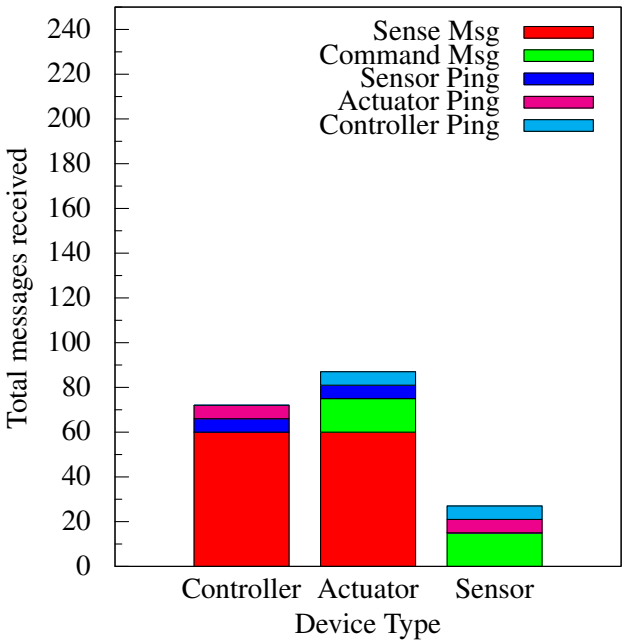

Figure 6.5: Messages received in 1 minute using xAP protocol

As shown in figure 6.4, the controller receives the most messages, due to it’s request for data from the sensor and the acknowledgements from the actuator. In the case of the actuator and sensor, relatively little messages are received but reliability is kept for commands and liveness checked for both.

Comparing this to xAP in figure 6.5, the controller receives less messages than the equivalent in the proposed protocol, but in the cases of the actuator and sensor, far more unnecessary messages are received and then thrown away by the devices. This creates a huge waste of time spent receiving and processing the messages for no reason; in cases where the devices are battery powered this has an even greater impact, reducing the battery life of the device.

The second set of graphs, figures 6.6 and 6.7, show how this initial observation evolves as the network scales up to contain double the number of sensors and actuators. As expected, in the proposed protocol the number of messages received by the controller doubles in proportion to the number of devices connected to it, however the number of messages received by the individual actuators and sensors remains the same as in the previous scenario. This is not the case in xAP however, instead the situation for the actuators and sensors continues to worsen as the number of devices increases.

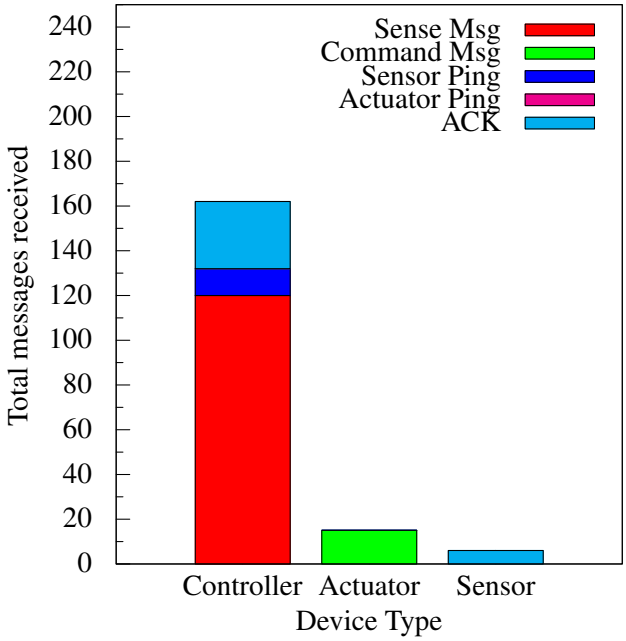

Figure 6.6: Messages received in 1 minute using IoTprotocol x 2

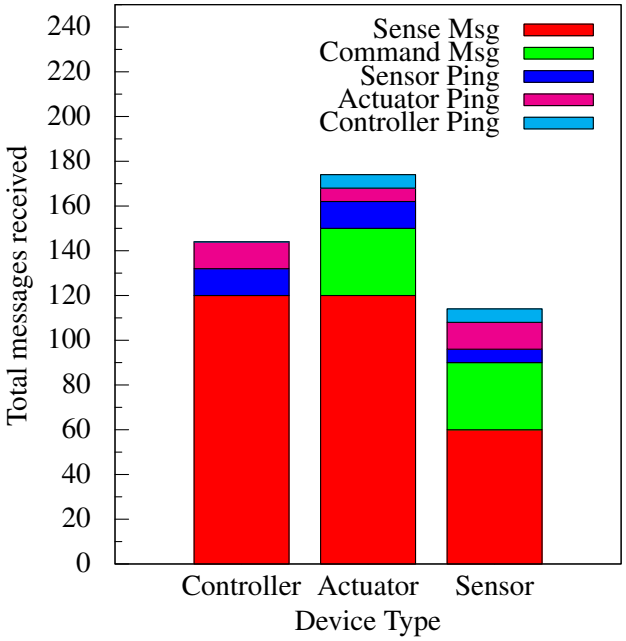

Figure 6.7: Messages received in 1 minute using xAP protocol x 2

In a network with a limited number of devices, devices can cope with the unnecessary messages received, but as soon as the network size starts to increase, devices will very quickly be swamped by unwanted data.

In regards to the proposed protocol, devices received no unwanted messages, saving time and power for the devices, however it can be observed that the controller still received lots of messages as a result of the high frequency of messages from the sensors. Whilst in this implementation the controller is also implemented on the TelosB motes, a better solution would be to create a controller from perhaps a low-power ARM or x86 pc, such as the Raspberry Pi, which would be much better suited to receiving that amount of traffic. Another alternative, is to distribute the control around the network by creating multiple controllers, which each control certain parts of the network, in turn also increasing the fault tolerance of the network.

6.3.2 JMS & xAP - Failure

As previously discussed, JMS provides adequate abstractions for implementing a centralised "Internet of Things" network, running on devices with no resource constraints. In contrast to this both xAP and the proposed protocol are designed around a distributed architecture in order to reduce the effects of a failure within the network.

In the situation where a single sensor or actuator in the network were to fail, xAP would be least affected due to its connectionless set up; devices would not require any cleaning up of state upon realising the failure, because no state is kept for connections. However, in the case of JMS and the proposed protocol, a device would need to ensure that when a device failure is detected, that it is in fact no longer available and isn't due to a transient network failure; once ensured, the device detecting the failure can then remove any state relating to the failed device, reducing the resources consumed.

In the situation where a controller in the network fails, with the proposed protocol, connected devices would try to ping the failed device to ensure it's not a transient failure and then remove the connection state related to that controller. Because the controller facilitates the co-operation between the sensors and actuators within the network, the relationship between these two devices is now lost. In the case of xAP, because it doesn't have the concept of a controller within the network, every device can contain some processing within it, further distributing the control around the network; but due to the connectionless architecture, devices don't understand when a recipient of their data fails, and in the case of the only recipient dying, the device can't know that there is no device out there to receive its data, and so will continue to broadcast the data unnecessarily. Whereas, in the case of JMS, when the central server fails, which facilitates the publishing and subscribing, the entire network of devices becomes inoperable because of their sole reliance on this server, with no way to communicate directly to one another.

Whilst the proposed protocol does have the controller role, which if fails, can have a significant impact on the network, the possibility of distributing the role of the controller among several devices is also available. By doing so, controllers could be set up within different areas across the network, or even act as redundancy, detecting when the original controller fails and then taking over the connections with minimal network down time. When compared to xAP, whilst xAP provides robustness through absolute distribution of logic, it does so at the cost of excessive broadcasting; whereas, the compromise observed within the proposed protocol, ensures that the criticality of failures can be reduced without any significant costs to the network.

Chapter 7

Conclusion

This chapter concludes the report, reviewing the initial objectives, assessing the successes and contributions resulting from it. The chapter ends discussing some future work which could follow on from the project.

7.1 Summary of Project

The original purpose of this project was to try and solve the problem of creating a lightweight protocol for the "Internet of Things", by designing and implementing a new protocol which could run on the most constrained devices, scale up to the more powerful devices we use everyday and scale out to 10's / 100's of devices within an environment such as the home or office.

The design of the protocol was heavily influenced by existing systems currently available, and seeked to adapt their concepts to better fit the "Internet of Things", such as xAP's distributed broadcast architecture and TCP's reliability concepts. By doing so, a protocol which could run efficiently on the most low-power devices, saving power and reducing the strain on the CPU, was created.

Evaluating this against other implementations, the protocol proves its ability to scale across different network sizes, and withstand failures in a manageable fashion when they occur, with the possibility of distributing control across the network and even creating redundancy to further reduce the impact. The overall size of the protocol implementation shows that it has a minimal impact on the resources of a device, even whilst offering 5 to 10 connections per device, which on a small to medium network would prove sufficient for both the actuator and sensor roles. In the case of the controller, porting it across to a more powerful platform suits the role far better, given the quantity of connections and data it will usually handle and request.

However, given the development constraints, including both the limited time and availability of different platforms, further work developing the protocol would be necessary to improve, test, port and validate its applicability in a real heterogeneous "Internet of Things" environment.

Overall, the project has been very successful in designing and implementing a protocol which meets the initial requirements, surpassing existing implementations in its applicability to constrained devices within the "Internet of Things" paradigm.

7.2 Future Work

This section describes some of the possible future work for developing this protocol further, from improving its feature set to trying to maximise the use of available resources.

- Applications
  - In order to fully understand the applicability of the protocol and to ensure it provides the correct abstractions, applications need to be developed for the protocol. Within the project timescale, only a subset of the original use cases could be implemented and tested, with more time the remaining use cases should be implemented at the very least to test the protocol's uses.
- Expanding the protocol
  - In the current protocol, sensors and actuators were only observed to be very primitive, single function devices, when in reality the possibility of a device being able to sense or perform more than one action exists. Therefore the need to be able to somehow store, search and interpret a devices functions is necessary. Perhaps a further step after discovering a device could be used to investigate the different functionality available to it.
- Further power saving techniques
  - In the current protocol, by reducing the number of unnecessary packets sent and received, the battery life of the constrained devices is significantly improved. To further this approach, compacting several high frequency sensor readings or actuator commands into one packet, which is then sent a lower frequency, could reduce the power consumption further, whilst still maintaining the same granularity as before when timing is not an issue.
- Security
  - Before being able to truly connect to the Internet, the protocol needs to ensure adequate security measures are put in place; encryption and authentication need to be considered and implemented, to ensure that eavesdropping on the data is not possible, as well as the assurance devices are who they claim to be and can't be spoofed on the network. The Datagram Transport Layer Security (DTLS) protocol could be used for ensuring packets can't be eavesdropped or forged; currently there is support for DTLS on Contiki, [9] .
- Porting to different platforms
  - Just as with applications, the protocol needs to be ported across a variety of platforms to really test its ability to perform and meet the requirements. By porting to other platforms, the true heterogeneous nature of the "Internet of Things" can be realised and utilised to the full extent of its capabilities; allowing not only constrained devices to interact, but also smart phones, tablets, home pcs and even Internet services, providing a much more dense and rich "Internet of Things".

Appendices

Bibliography

[1] 2013: The year of the Internet of Things. http://www.technologyreview.com/view/509546/2013-the-year-of-the-internet-of-things/. Accessed: 21/03/2013.

[2] Amazon Cloud outage. http://aws.amazon.com/message/680587/. Accessed: 21/03/2013.

[3] Arduino Network Interrupt support. http://arduino.cc/en/Main/ArduinoEthernetShield. Accessed: 21/03/2013.

[4] Arduino numbers in 2011. http://www.adafruit.com/blog/2011/05/15/how-many-arduinos-are-in-the-wild-about-300000/. Accessed: 21/03/2013.

[5] Botanicalls. http://www.botanicalls.com/about/. Accessed: 21/03/2013.

[6] CoAP library for C. http://sourceforge.net/projects/libcoap/. Accessed: 21/03/2013.

[7] CoAP library for Java. http://code.google.com/p/jcoap/. Accessed: 21/03/2013.

[8] CoAP library for TinyOS. http://docs.tinyos.net/tinywiki/index.php/CoAP. Accessed: 21/03/2013.

[9] Contiki DTLS support. http://tinydtls.sourceforge.net/. Accessed: 21/03/2013.

[10] Google Mail outage. http://news.cnet.com/8301-1023_3-57415281-93/gmail-users-experience-outage/. Accessed: 21/03/2013.

[11] Internet of Things first coined. http://www.rfidjournal.com/article/view/4986. Accessed: 21/03/2013.

[12] LG Smart Appliances. http://www.lg.com/us/press-release/lg-smart-appliances-for-2012-deliver-connectivity-efficiency-through-smart-thin Accessed: 21/03/2013.

[13] Manufacturer's smart home. http://edition.cnn.com/2011/TECH/innovation/01/07/internet.connected.appliances/index.html. Accessed: 21/03/2013.

[14] Qualcomm announces IoT platform. http://www.qualcomm.com/media/releases/2013/01/07/qualcomm-and-att-support-internet-everything-development-platform. Accessed: 21/03/2013.

[15] Raspberry Pi Lego Super Computer. http://raspberrypicloud.wordpress.com/. Accessed: 21/03/2013.

[16] Raspberry Pi Media Centre. http://www.raspbmc.com/about/. Accessed: 21/03/2013.

[17] Raspberry Pi sold 500,000 in 8 months. http://www.wired.com/opinion/2013/09/raspberry-pi-insider-exclusive-sellout-to-sell-out/. Accessed: 21/03/2013.

[18] Roomba, autonomous vacuum robot. http://www.irobot.com. Accessed: 21/03/2013.

[19] Size of the Internet. http://www.guardian.co.uk/business/2009/may/18/digital-content-expansion. Accessed: 21/03/2013.

[20] Smart Things IoT platform. http://smartthings.com/. Accessed: 21/03/2013.

[21] Social funding platform, Kickstarter. http://www.kickstarter.com/help/faq/kickstarter%20basics#Kick. Accessed: 21/03/2013.

[22] Sony Playstation Network Breach. http://blog.us.playstation.com/2011/04/26/update-on-playstation-network-and-qriocity/. Accessed: 21/03/2013.

[23] TelosB Sky Motes. http://www.willow.co.uk/TelosB_Datasheet.pdf. Accessed: 21/03/2013.

[24] Tweeting Plant. http://blog.makezine.com/2008/02/25/how-to-make-plants-talk-t/. Accessed: 21/03/2013.

[25] Twine "Internet of Things" Thing. http://supermechanical.com/. Accessed: 21/03/2013.

[26] xAP: eXtensible Application Platform. http://www.xap.com/. Accessed: 21/03/2013.

[27] xAP: Heartbeats. http://www.xapautomation.org/index.php?title=Protocol_definition#Device_Monitoring_-_Heartbeats. Accessed: 21/03/2013.

[28] xAP: Protocol specification. http://www.xapautomation.org/index.php?title=Protocol_definition. Accessed: 21/03/2013.

[29] David Braginsky and Deborah Estrin. Rumor routing algorthim for sensor networks. In *Proceedings of the 1st ACM international workshop on Wireless sensor networks and applications*, WSNA '02, pages 22–31, New York, NY, USA, 2002. ACM.

[30] Castellani. CoAP to HTTP mapping. https://datatracker.ietf.org/doc/draft-castellani-core-http-mapping/. Accessed: 21/03/2013.

[31] Adam Dunkels. Rime — a lightweight layered communication stack for sensor networks. In *Proceedings of the European Conference on Wireless Sensor Networks (EWSN), Poster/Demo session*, Delft, The Netherlands, January 2007.

[32] Adam Dunkels, Bjorn Gronvall, and Thiemo Voigt. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *Proceedings of the 29th Annual IEEE International Conference on Local Computer Networks*, LCN '04, pages 455–462, Washington, DC, USA, 2004. IEEE Computer Society.

[33] Jeong-hwan Hwang and Hyun Yoe. Design and implementation of wireless sensor network based livestock activity monitoring system. In *Proceedings of the Third international conference on Future Generation Information Technology*, FGIT'11, pages 161–168, Berlin, Heidelberg, 2011. Springer-Verlag.

[34] Chalermek Intanagonwiwat, Ramesh Govindan, and Deborah Estrin. Directed diffusion: a scalable and robust communication paradigm for sensor networks. In *Proceedings of the 6th annual international conference on Mobile computing and networking*, MobiCom '00, pages 56–67, New York, NY, USA, 2000. ACM.

[35] Sukun Kim, Shamim Pakzad, David Culler, James Demmel, Gregory Fenves, Steven Glaser, and Martin Turon. Health monitoring of civil infrastructures using wireless sensor networks. In *Proceedings of the 6th international conference on Information processing in sensor networks*, IPSN '07, pages 254–263, New York, NY, USA, 2007. ACM.

[36] Yongsheng Liu, Yu Gu, Guolong Chen, Yusheng Ji, and Jie Li. A novel accurate forest fire detection system using wireless sensor networks. In *Proceedings of the 2011 Seventh International Conference on Mobile Ad-hoc and Sensor Networks*, MSN '11, pages 52–59, Washington, DC, USA, 2011. IEEE Computer Society.

[37] Z. Shelby. IETF, Constrained RESTful Environments - Resource Discovery, 2012. RFC6690, 1.2.1.