A platform for Internet-connected wireless sensor networks

Stuart Whitehead

9th April 2016

Abstract

Acknowledgements

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Project Motivation

1.2 Aims and Objectives

1.3 Report Structure

Chapter 2

Background

The term 'Internet of Things' captures the essence of the vision well; the vision of a world where everyday, physical objects are gateways to web-based services. These so-called 'smart objects' comprise of sensors to perceive their environment or context, as well as a notion of interconnectivity with other objects or services. Connected objects or services may react to collected data to trigger actions, and these actions may be digital or physical. The Internet of Things could be summarised as data collection, aggregation and reaction in the physical domain.

The boundaries between IoT and other trends help to define its place in computing. For instance the research area of Wireless Sensor Networks (WSN) carries similarities in hardware requirements and challenges. In particular, WSN also comprise of connected sensors and actuators (Mottola and Picco, 2011). WSNs differ from IoT because of the scope of connectivity—they are limited to specific applications and area coverage, such as a production line or a single building. In comparison, IoT applications have a global scope which encourages co-operation between devices, networks and services. Wireless Sensor Networks can, however, form one aspect of an IoT application.

The research area of 'Wearables' also overlaps with IoT. Wearables are smart devices designed to be worn or embedded within the body. They combine sensors with some form of connectivity, typically integrating with a smartphone (Wei, 2014). Consumer Wearables products are already on the shelves, such as Fitbit—a wrist-worn personal health tracker. The Fitbit wristband connects to a smartphone with Bluetooth Low Energy (BLE) and this smartphone then provides global connectivity through its wireless connection. Users can sign-in to a web-based dashboard which will collate and organise personal data. In this scenario, Wearables are collecting, aggregating and reacting to data in the physical domain. Wearables could therefore be described as a subset of IoT (as shown by Figure 2.1) because they are an application in a specific domain.

IoT can also take advantage of research conducted for the 'Big Data' trend. Over the past three decades, scalable and distributed data management have been a focus for the database research community (Agrawal et al., 2011). This research originally focused on traditional enterprise applications, but it has expanded to include the Cloud Computing paradigm across two classes of systems: update-heavy applications and ad hoc analytics & decision support. IoT is expected to generate large volumes of data and so by leveraging Big Data analytical processes, valuable information can be deduced. For example, Fitbit's web-based dashboard could use Big Data analytics to analyse personal health statistics and to find data trends.

IoT also goes some way to realise the Ubiquitous Computing concept. Ubiquitous Computing (also known as pervasive computing) envisages a world where users are surrounded by connected technology—technology in our homes, workplaces and recreational activities. In his seminal

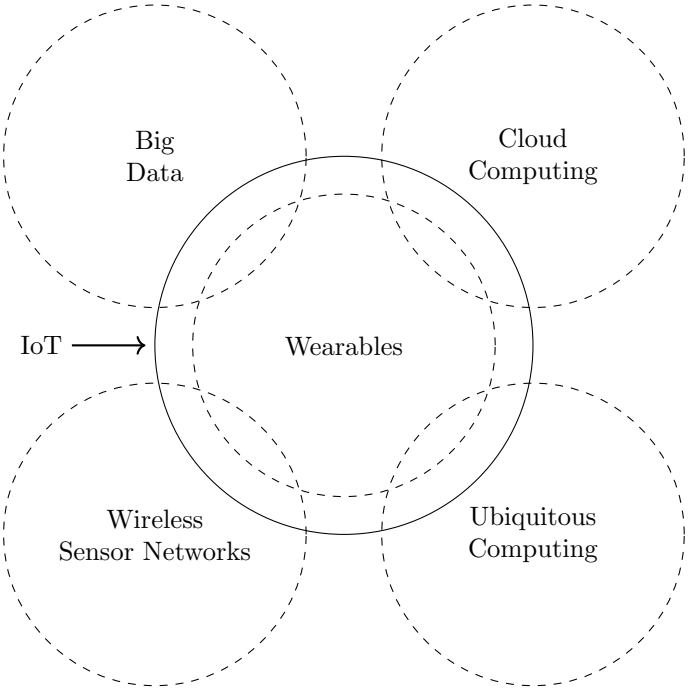

Figure 2.1: The boundary between IoT and other trends in computing

article, Weiser (1999) notes that "specialized elements of hardware and software, connected by wires, radio waves and infrared, will be so ubiquitous that no one will notice their presence". Lyytinen and Yoo (2002) take this further and suggest that we will not only be surrounded by technology, but it will be embedded in our social and physical interactions. Research areas such as this not only contribute to the IoT concept but also benefit from its research endeavours.

2.1 Opportunities

Why should we develop IoT solutions? Would our society actually benefit from them? These are the questions posed by stakeholders of all backgrounds. Pilot projects as well as industrial and academic research have shown that there are wide-reaching opportunities to be grasped. Until recently, these opportunities were achievable only in theory or laboratory scenarios, but due to performance improvements and cost reductions in the supporting technology, IoT is very much a reality.

Mattern and Floerkemeier (2010) describe a number of opportunities created by the evolving technical capabilities of IoT systems. Since much of the value of IoT is derived from the communication of data, the technologies supporting inter-device communication and cooperation are particularly important. Wireless Personal Area Network (WPAN) technologies such as Wi-Fi, Bluetooth, ZigBee and 6LoWPAN give objects the ability to network with Internet resources or each other. These WPAN technologies also provide device addressability mechanisms which allow objects to be uniquely identified from anywhere in the world, through extensions to protocols like MAC Addressing or IPv6. With this level of global interconnectivity, novel services and products become possible.

Another main characteristic of IoT is the ability to interact with the physical domain. Digital applications will interact with the physical world in two ways: through the monitoring and sensing of physical properties and through the actuation of the physical world in reaction to data. There are a massive variety of sensors available, even to hobby markets, and can measure any imaginable physical property. The reduced cost and improved support by hardware manufacturers will allow measurements to be collected at a scale and precision not previously possible.

Since IoT microcontrollers are becoming more powerful, there are opportunities for embedded information processing. Embedded information processing refers to end devices performing some form of data processing or storage before transmitting their data; for example, an end device might analyse its own data and only transmit if a specific threshold is met, rather than relying on Cloud Computing-based processing. This is an active area of research called edge computing (or fog computing) and allows for novel methods of data processing and is another method to optimise bandwidth and power consumption.

Along with these technical capabilities are big-picture societal and economic opportunities. The European Union is a strong example of an economy which can benefit from IoT uptake. The governmental body of the EU, The European Commission, recognises the potential opportunity: "One major next step in this development [of the Internet] is to progressively evolve from a network of interconnected computers to a network of interconnected objects, from books to cars, from electrical appliances to food, and thus create an 'Internet of things'." In 2009 the Commission published an action plan for embracing the Internet of Things.

The action plan (The European Commission, 2009) describes how IoT will improve citizen well-being both now and into the future. In the present, citizens will benefit from specific IoT initiatives—for instance, Internet-connected health monitoring systems could alleviate pressure on medical services for the ageing society. Other applications such as connected cars or smart waste management are expected to reduce the EU's carbon footprint and therefore protect resources for future generations.

The Commission's action plan also describes economic opportunities. IoT has the potential to provide many new income streams to businesses as well as finding other money through efficiency savings. First of all IoT opens up new markets which were previously not feasible nor even considered. These markets may include consumer products or novel service solutions in areas such as home security and monitoring. For organisations with complex supply chains or processes, IoT could help to maximise resources and stock control through large-scale automation and efficient information exchange.

2.2 Challenges

It is apparent that IoT is still an evolving area of research with numerous challenges to be addressed. In order for IoT to become a mainstream paradigm, the technologies and ecosystem surrounding them must converge and become more mature. Mattern and Floerkemeier (2010) state that IoT is not the result of a single novel technology but rather several complementary technical developments, so invariably, the challenges span many technical disciplines.

A good user experience must be offered by these technologies to encourage adoption. Mattern and Floerkemeier (2010) state that devices "need to establish connections spontaneously, and organize and configure themselves to suit their particular environment" which they coin 'Arrive and Operate'. The process of connecting a laptop to a home Wi-Fi router is representative of the current experience a user might have. They will turn on the Wireless Network Interface Card via a mouse and a Graphical User Interface and select their wireless network before inputing a password with a keyboard. This experience is far from ideal for constrained devices with a

simplistic user interface, or no interface at all. The challenge here is to develop technology which allows consumers to use smart objects with little or no configuration.

Device discovery and interoperability is required to build an open and extensible Internet of Things. Since the world of things is diverse, each type of device is likely to have different capabilities. Mineraud and Tarkoma (2015) state that the heterogeneity of existing platforms is considerable (numerous hardware, technologies, requirements and objectives) and as a result, IoT would benefit from a generic architecture which provides IoT developers with a common toolkit. Research efforts have gone some way to address these challenges, such as the IoT 'meta-hub' offered by Mineraud and Tarkoma, which provides a feed of available features for a given service.

IoT infrastructures must be scalable to handle the large number of connections and volume of data. By all estimates, IoT will have a much larger scope and footprint when compared to the existing Internet of computers. The European Commission puts this figure at 50–70 billion devices (on average, 10 per human). Mattern and Floerkemeier (2010) also note that things will be cooperating mainly within a local environment. This means that IoT devices, software and infrastructure must work equally efficiently in both small- and large-scale environments. This complexity is also reflected with the volume and variety of data being generated; a challenge currently being addressed with research around Big Data.

Fault tolerance, power supply and efficient wireless communication are challenges facing hardware manufacturers. There are a number of possible operating environments for IoT devices, such as office spaces, homes, public spaces and remote areas, so smart objects might depend on a self-sufficient power source. The industry is therefore challenged to produce long-lasting batteries. As well as improving power supply solutions, the industry should maximise power efficiency in other areas such as existing wireless communication technologies which consume too much overhead and power.

Technical challenges will continue to be addressed through academic and industrial research, however IoT will not achieve widespread uptake without social acceptance. This is the risk that the industry is carefully attempting to balance; the social acceptance of IoT products is imperative to their success but consumers and markets could be less accepting if industries try too much too soon. The European Commission's action plan describes practical challenges which need to be addressed.

Many aspects of digital technology is governed by public bodies and the Commission believes that IoT should not be any different. They note that technology will advance due to a normal cycle of innovation and that "simply leaving the development of IoT to the private section is not a sensible option in view of the deep societal changes that IoT will bring about." The main areas which may need to be governed are identification, information security and ethical & legal accountability. In terms of device identification, Mattern and Floerkemeier (2010) and (The European Commission, 2009) suggest that some form public registry (similar to those governed by the Institute of Electrical and Electronics Engineers) will be necessary. In terms of legal accountability, users may be reluctant to use IoT services unless there are safeguards in place. Without effective governance of these areas, IoT may suffer from stifled innovation; a mistrust with data and jeopardised system integrity.

Standardisation is a technical consideration but also impacts on the economic potential of IoT. The purpose of a standard is to give different entities and stakeholders a common language to design, build or communicate. The widespread adoption of a standard gives businesses of all sizes the opportunity to develop interoperable solutions. In being interoperable, standards-based solutions will be suitable for a wider market and in turn, this will encourage innovation and improve international competitiveness. If IoT is to maximise economic impact and to benefit from widespread adoption, there must be accepted practices or formal standards in place.

The greatest challenges to social acceptance are security and data privacy, particularly in

the EU where the protection of privacy and personal data are two fundamental rights. The challenge with this, in respect to IoT, is that new methods of collecting and using data will be invented. Does current legislation and safeguards protect the interest of EU citizens adequately? Who will own the data—the user whom it is about or the organisation that captured it? The Commission suggests that in response to this challenge, IoT components should be designed from their inception with a privacy- and security-by-design mindset.

2.3 Use Cases

Use-cases are a helpful mechanism for demonstrating the IoT concept. It is a difficult concept to explain because there are many cooperating components and ideas and for this reason, research papers exemplify their arguments with practical examples. Although IoT applications could be developed for virtually any industry there are domains where its application is more obvious. Table 2.1 presents six such domains for IoT, together with opportunities, challenges and related areas of research.

'Smart Cities' is a vision which convincingly demonstrates many benefits of IoT. The Department for Business Innovation & Skills (2013) describes the main issues facing local authorities which includes: piecemeal urban infrastructure, climate change targets, the changing nature of the high street and elderly social care. The economic downturn has also reduced budgets assigned to local authorities by as much as 30% between 2010 and 2013 making cost efficiencies another issue. By developing or retrofitting urban infrastructure with IoT connectivity, local authorities can monitor services in much finer detail. For instance, Bigbelly Solar is a smart waste and recycling system. The Internet-connected waste bins allow local authorities to monitor their status and to schedule collections when they are full. Bonomi et al. (2012) also describe a system of smart, connected traffic lights which can control the traffic flow through a whole city, reducing congestion and improving safety.

IoT in the utilities sector is one use-case which has started to become realised. The British Government is requiring energy companies to install smart meters for their customers and expect most to have a smart meter installed by 2020 (The Department of Energy & Climate Change, 2013). Smart meters monitor domestic energy usage on a per-site basis and can give real-time information about how much electricity is being used, expressed in pounds and pence. For individual households this has the benefit of enabling homeowners to save money and to reduce emissions. At a national level, smart meters are a step towards reducing energy consumption and meeting climate change goals. This use case is particularly interesting and well documented, so section 2.6 investigates the smart meter concept further. Additionally, smart thermostats like Nest also allow households to minimise their gas consumption and to reduce heating bills. This is achieved through usage optimisation based on occupancy patterns.

Logistics is one area which demonstrates the efficiency savings that an IoT application can make. The European Commission (2007) states that "Freight Transport Logistics focuses on the planning, organisation, management, control and execution of freight transport operations in the supply chain" and notes that it is one of the drivers of European competitiveness. To remain competitive, this industry must continue to improve and streamline infrastructure, fleet management and goods tracking. IoT could be used to share content-related data and location information for regulatory or commercial purposes. This is achieved through the use of inexpensive RFID tags (which store specific information about the item) and GPS sensors (which can identify precise locations). This would reduce time spent by workers manually recording goods and it will reduce errors at interchange points. The Commission has coined this the 'Internet for Cargo'.

Overlapping Research	Opportunities		Challenges		Example Use Cases
	Economic & Societal	Technical	Economic & Societal	Technical	
Cloud Computing	Citizen Well-Being	Communication & Cooperation	Governance	'Arrive and Operate'	Smart Cities
Big Data	Economic Prosperity	Addressability	Standardisation	Interoperability	Waste Management
Ubiquitous Computing	Ageing Society	Identification	Security	Discovery	Smart Traffic Lights
Wireless Sensor Networks	New Markets	Sensing	Data Privacy	Software Complexity	Utilities
Wearables	Business Optimisation	Actuation	Trust	Scalability	Smart Meters
Edge Computing		Localisation		Data Management	Logistics
		Embedded Information Processing		Fault Tolerance	
		User Interfaces		Power Supply	
				Wireless Communication	

Table 2.1: A summary of the current state of the Internet of Things

2.4 Existing Platforms

There are a number of IoT platforms both in service and under active development; in fact, Amazon announced their own platform, Amazon IoT, when writing this very comparison (Amazon Web Services, Inc., 2015). The aim of an IoT platform and the tools which it provides does differ from vendor to vendor, however common characteristics are beginning to emerge. We propose to compare platform offerings across five dimensions: hosting model, source availability model, connectivity, bidirectional communication support and trigger support.

2.4.1 Hosting

The hosting model offered by an organisation defines which stakeholder is responsible for maintaining the IoT application and its hardware infrastructure; either the platform provider or the customer. This is a notable differentiator between platforms and can be indicative of the platform vendor's business objectives. There are generally two variations in hosting models: Platform as as Service (PaaS) or self-hosting.

Platform as a Service

A Platform as a Service (PaaS) is a Cloud Computing concept. It is an abstraction of the complete application technology stack including both hardware and software. An organisation providing a PaaS would be responsible for the management and maintenance of the hardware such as servers, RAID backing storage and networking appliances like routers or switches. Depending on the software on offer, the vendor would also be responsible for managing and maintaining the server operating system as well as any supporting software packages and applications. Customers are typically provided access to this abstraction through a control panel or an Application Program Interface (API).

Carriots is an example of a centralised PaaS and their website describes various benefits to a customer (Carriots, 2015). Primarily the customer does not need to allocate resources for infrastructure maintenance since this is managed by them, with the result being more time for the customer to focus on business needs. There is also less investment required on behalf of the customer—PaaS resources can be provisioned when and if they are needed, rather than

purchasing hardware upfront. Another benefit is the extra layer of support—system scalability, stability and fault tolerance are the responsibility of Carriots.

Self-hosted

An alternative to a PaaS is the self-hosted option. As the name implies, this option makes customers responsible for the maintenance and management the application infrastructure (although software updates may still be provided by the platform vendor). There are technical and legal arguments to support such a business decision.

The main technical aspect is that of control. With the self-hosted option, the system architecture can be optimised for specific use-cases, such as the global location of servers to reduce application latencies (the time taken for network transmissions between devices and servers). This also gives customers direct access to the core software and makes it easier to modify it for their own purposes (if the licence or software allows for that).

Self-hosting may also be necessary for legal compliance, especially in the realm of data protection. Depending on the country of operation, it may be necessary to retain data within country boundaries and this may not be enforceable with a PaaS. In October 2015 the Court of Justice of the European Union ruled that the Safe Harbour agreement between the EU and United States is invalid (Case C-362/14). This case found that the United State's data protection legislation was not adequate and means that any organisation sharing data under this agreement is now acting unlawfully. This problem could be avoided, or at least remedied more quickly, if data is managed from within the European Union. One example of a self-hosted platform is Nitrogen.io.

2.4.2 Source Availability Model

The Source Availability Model chosen by a platform developer defines the level of accessibility for their source code. This model comes in two forms—open or closed source—and can be further refined by a software license framework.

The source code of Open Source Software (OSS) is made publicly available. This could be served as a standard file download or, as is becoming common practice, on a code sharing service such as GitHub or Bitbucket. OSS will typically be provided under the conditions defined in a license, such as the GPL, MIT or Apache license frameworks. These vary on points like distribution, modification and notification of copyright (Github, Inc., 2015).

Closed Source Software is proprietary source code not made publicly available, although it may be necessary to distribute compiled binaries. This just means that the organisation does not wish to make source code available for modification or knowledge sharing.

In conjunction with the hosting model, the source availability is indicative of business objectives. Organisations wishing to commercialise their platform as a product offering will use a Closed Source model to protect their intellectual property. Conversely those organisations where knowledge sharing is an aim, or where the platform is a mechanism to sell other products or services (such as Sparkfun and their electronic hardware) may use an Open Source model.

2.4.3 Connectivity

The IoT concept *is* interconnectivity and communication, making device and application connectivity a very important topic. The platforms investigated support a range of connectivity protocols and concepts. These range from widely-accepted HTTP-based APIs to more niche protocols like CoAP and MQTT.

HTTP

The Hypertext Transfer Protocol (HTTP) underpins the internet. It benefits from widespread support and most notably, is used by web browsers to interface with web servers. RFC 2616 (Fielding et al., 1999) defines the specification for HTTP version 1.1 and notes:

> It is a generic, stateless, protocol which can be used for many tasks beyond its use for hypertext, such as name servers and distributed object management systems, through the extension of its request methods, error codes and headers.

As hinted at by this RFC 2616 quote, three important features of HTTP are: request methods, error codes and headers. To generalise the HTTP life cycle, a user agent will send a request to an origin server, the server will process the request and will return a response. The request will use an HTTP *verb* to define the action to take, such as `GET` or `POST`, and will include headers to specify other parameters. The response has an associated status code, such as `200 OK` or `404 Not Found`.

Most of the platforms investigated here support HTTP requests through a Representational State Transfer (REST) API. Fielding (2000) states that the REST architectural style emphasises scalability, independence of components, minimises latency and maximises security. These characteristics are achieved by applying constraints to the application, such as stateless communication, a uniform component interface and a layered approach to infrastructure technologies. REST methods are based on the HTTP verbs like `GET`, `POST`, `PUT` and `DELETE`. For IoT application developers, a REST API provides a scalable, uniform and robust interface between devices and the Internet.

Software engineers working on specifications for the Internet have foreseen the Internet of Things, even if it was taken in jest. RFC 2324 (Masinter, 1998) is an April Fool's joke and defines the Hyper Text Coffee Pot Control Protocol version 1.0 (HTCPCP/1.0). It specifies methods by which an Internet-connected coffee pot can be controlled, and even adds a new error code: "Any attempt to brew coffee with a teapot should result in the error code '`418 I'm a teapot`'. The resulting entity body MAY be short and stout."

WebSockets

The WebSocket Protocol is an independent TCP-based protocol that enables two-way communication. It uses an HTTP request as part of the handshaking process, which is treated as a connection upgrade request. RFC 6455 (Fette and Melnikov, 2011) states "The goal of this technology is to provide a mechanism for browser-based applications that need two-way communication with servers that does not rely on opening multiple HTTP connections (e.g., using `XMLHttpRequest` or `<iframe>`s and long polling)." This technology plays a vital role in enabling bidirectional communication between web browsers and servers, as will be covered in section 2.4.4.

WebSockets use standard HTTP ports for communication by default (port 80 and 443 for unencrypted and encrypted connections respectively). This has the added benefit of being compatible with common network and firewall configurations.

MQTT

MQ Telemetry Transport (MQTT) is a simple and lightweight messaging protocol. It is based on a publish/subscribe model and is specifically designed for constrained devices and low-bandwidth, high-latency or unreliable networks. In general, MQTT is used in conjunction with a message

broker. Clients subscribe to topics on this broker which will then forward any relevant messages. Clients are then free to do whatever they please with these messages (Banks and Gupta, 2014).

MQTT uses TCP/IP to transmit messages and operates on ports 1883 and 8883 for unsecured and secured uses respectively. Since these ports and not common, firewalls will typically block access. For this reason, MQTT can be tunnelled using WebSockets (and means this can be used in the browser, too).

CoAP

Constrained Application Protocol (CoAP) is another protocol designed with the Internet of Things in mind. Shelby et al. (2014) describe CoAP as "a specialised web transfer protocol for use with constrained nodes and constrained networks." Some headline features align with principles used with HTTP, such as utilisation of the REST model. RFC 7252 defines the proposed specification for CoAP. It specifically notes some main features:

- Web protocol fulfilling M2M requirements in constrained environments
- UDP [RFC0768] binding with optional reliability supporting unicast and multicast requests
- Asynchronous message exchanges
- Low header overhead and parsing complexity
- URI and Content-type support
- Simple proxy and caching capabilities

CoAP is likened to HTTP throughout the specification however there are some notable differences. HTTP uses the TCP/IP transport protocol whereas CoAP uses UDP, a protocol with significantly less overhead. UDP is not a request/response protocol so to provide a request/response interaction model, CoAP must deal with this through a logic layer.

UDP

One of the researched platforms also offers the User Datagram Protocol (UDP) as a connectivity option. RFC 768 (Postel, 1980) notes "this protocol provides a procedure for application programs to send messages to other programs with a minimum of protocol mechanism." To achieve 'a minimum of protocol mechanism', UDP does not establish a connection with the receiving device, so unlike TCP/IP it is connectionless. The protocol also specifies no mechanism for error correction so if a packet is lost in transit, it is lost forever.

Since data can be lost, this protocol should only be used in applications which can tolerate data loss. To give a real-world example, a battery-powered hygrometer sensor may be collecting data about the enviromental humidity. If UDP was used as the connectivity protocol, it would allow this sensor to wake up, transmit its reading and then fall back to sleep without being concerned about the packet's progress. If a packet was lost, it would not have business or safety impacts.

2.4.4 Bidirectional Communication

One main opportunity of IoT, as described in Section **??**, is to drive actuators in response to data. There are three aspects which define whether unidirectional or bidirectional communication is possible: the connectivity protocol, the software application and the network environment. To

illustrate the following points, a hypothetical scenario will be used, where one server and one actuator are connected together through the internet. In this case, unidirection communication means that only the actuator can initiate network requests whereas bidirectional communication allows either the server or actuator to initiate.

If the actuator is publicly addressable with a dedicated IP address, bidirectional communication is trivial to implement. Supposing that the HTTP protocol is being used for communication, both the actuator and server can initiate requests to each other by using the appropriate IP address. The server may send an HTTP request instructing the actuator to do something. In real-world scenarios, there are three factors which make bidirectional communication like this difficult: firewalls, Network Address Translation and IPv6 uptake.

Firewalls are common-place both in businesses and at home. They are a network security tool and can be either a dedicated hardware unit or software built into other devices, such as a domestic router. Their purpose is to screen both in and outbound connections which may have a malicious or compromising effect on the network integrity. The side effect which this has on IoT systems is that non-common protocols may be blocked, such as MQTT (since it uses ports 1883 and 8883). While devices behind a firewall may be publicly addressable, they may not be reachable, depending on the IoT application.

While obtaining a public IP address for each IoT device is a worthy aim, it is not achievable in many situations due to Network Address Translation. Network Address Translation (NAT) is a tool used to delay the depletion of IPv4 addresses and is very common with domestic internet connections. RFC 2663 (Srisuresh and Holdrege, 1999) states "NAT devices are used to connect an isolated address realm with private unregistered addresses to an external realm with globally unique registered addresses." In essence, this allows a number of privately addressed devices to communicate through one public address. In terms of IoT, this means than any devices connected behind NAT will not be publicly addressable—the server can not directly send a request to the actuator.

Internet Protocol version 6 (IPv6) will solve the issue of IP address depletion. It will offer over 340 undecillion possible addresses, compared to just under 4 billion available from IPv4, and will allow every connected device to have its own IP address for the foreseeable future (The European Commission, 2008). With regards to the Internet of Things, this is still not achievable. IPv6 is still not supported by many internet service providers (ISPs) and networks, including that of Robert Gordon University. For the Internet of Things, this means that universally supported IPv6 is still a while off.

There are some useable techniques to overcome these three issues and to enable (or mimic) bidirectional communication. The first of these solutions is HTTP polling. With standard polling, the actuator would periodically ask the server for updates. If this period was 1 second, the communication may feel like it was real-time and bidirectional. This is however wasteful because not every request will result in an update. A more efficient form of polling is called long polling. The actuator will send a request to the server, but if there are no updates yet, the server will not respond. When the server does have a message to send to the actuator, it will respond and then end the request. This is better, but implementations suffer from the added complexity of managing connections.

A final and more effective solution is to implement communication using WebSockets. As previously described, the WebSocket protocol allows for bidirectional, full duplex communication and it can solve every challenge already addressed. WebSockets use the same communication ports as HTTP (port 80 and 443) by default, meaning that firewalls will allow them. Networks that support web browsing will support WebSockets. Also, the WebSocket connection can be initiated by the client and still allow bidirectional communication. In this case, the actuator can open a WebSocket connection to the server and overcome any Network Address Translation in

Service	Hosting	Source	Connectivity	Bidirectional	Triggers
Amazon IoT	PaaS	Closed	MQTT, HTTP	Yes	Yes
Bug Labs Dweet	PaaS	Closed	HTTP	Yes (app)	No
Carriots	PaaS	Closed	HTTP, MQTT	No	Yes
Sparkfun	PaaS or self-hosted	Open	HTTP	No	No
Exosite	PaaS	Closed	CoAP, HTTP, UDP	No	Yes
Google Brillo	PaaS	Closed	Unknown	Unknown	Unknown
Digi	PaaS	Closed	HTTP	Yes	Yes
IFTTT	PaaS	Closed	HTTP	No	Yes
Newaer	PaaS	Closed	HTTP	No	Yes
Nimbits	Self-hosted	Open	HTTP	No	Yes
nitrogen.io	Self-hosted	Open	MQTT, HTTP	Yes	Yes
ThingSpeak	PaaS or self-hosted	Open	HTTP	No	Yes

Table 2.2: Characteristics of existing IoT platforms

place because it does not require a public IP address.

2.4.5 Application Triggers

An application-level feature common across investigated platforms is event triggers or rules. Where supported, platforms can trigger actions based on the data received from sensors and third-party APIs. Combining examples already mentioned, a hygrometer may send data to an IoT application. An application trigger could be configured to turn on a dehumidifier actuator if the humidity passes a certain threshold. If bidirectional communication is supported, this trigger would react in real-time. Triggers could also be configured to interact with other third-party software APIs.

2.5 Sensor Hardware

The Internet of Things is a multidisciplinary area of research and is just as dependant on advancements in electrical and electronics engineering as it is computer science. Since the hardware driving IoT sensor nodes is intertwined with the upper layers of software, it would be appropriate to investigate the current state of this area too. Rather than research the intricacies of how these pieces of hardware work, this paper will investigate existing end device hardware with regards to their implementation in IoT applications, particularly technical capability and energy efficiency.

With a similar method to Section 2.4, various makes and models of hardware boards were analysed to find comparable characteristics. The manufacturers chosen—Arduino, Raspberry Pi, Intel and Samsung—represent a cross-section of the field. These manufacturers aim their products at different markets, from the DIY 'maker' market of the Arduino to the professional, consumer electronics market of Samsung. This section will compare and contrast hardware models produced by these manufacturers across a range of key areas: processor architecture, memory, input/output options and wireless connectivity. Table 2.3 summarises each model and their specifications.

2.5.1 Processor Architecture

Just like any other computer system, the processor is the heart of the hardware board. It is responsible for running the operating system, orchestrating the various system components and for running the user application. The type and specification of the processor defines what can be

processed and how quickly, which is ultimately reflected in the responsiveness and capability of the IoT device. Processor throughput is not the only consideration when scoping an IoT device however—energy consumption is also an important factor. The specification of the processor can be categorised into smaller areas: processor type, instruction set, number of bits, clock speed and the number of cores.

There are two broad types of processor systems, namely microcontrollers and microprocessors. A microcontroller (also known as an embedded controller) is a complete computer in one chip. A single microcontroller chip will typically combine a central processing unit (CPU), small amounts of program memory and Random Access Memory (RAM) and input/output lines (Ibrahim, 2011). In comparison, a microprocessor requires other system components to function, such as external memory, input/output lines and a clock circuit. In this regard, microprocessor hardware cannot be upgraded because it is integrated whereas a microprocessor system can, but they are cheaper to manufacture.

Processor architecture can also vary between makes and models. (Ibrahim, 2011) notes that processors can be classified by the number of bits they process, with 8 bits being the most popular for microcontrollers. Processors with a greater number of bits are more powerful but are also more expensive. The instruction set used also effects the performance of the processor; Reduced Instruction Set Computer (RISC) instructions take only one CPU cycle to complete whereas Complex Instruction Set Computer (CISC) instructions can take multiple cycles.

Clock speed and the number of cores are two final factors of a processor's performance. The clock speed describes how many cycles the CPU executes in one second and is regulated by a crystal timer or internal digital clock oscillator (DCO) A greater clock frequency means more instructions which can be processed in a time period, and the more powerful the processor. Microcontrollers typically have clock speeds in the tens of Megahertz whereas microprocessors might run at hundreds of Megahertz or a number of Gigahertz. However due to physical limitations of hardware components, there is a practical limit in possible clock speed. In this case more cores are added to the processor allowing it to evaluate multiple instructions concurrently. With regard to power consumption, Mikhaylov and Tervonen (2010) state that clock speed is one factor affecting the lifetime of a battery powered device so the CPU should be carefully considered for its application.

2.5.2 Memory

In any computer system, different types of memory are used to store program instructions and the temporary data associated with them. While these types of memory can be split into more discrete categories, there are generally two types of memory: non-volatile and volatile. Non-volatile memory, such as Read-Only Memory (ROM) or Programmable Read-Only Memory (PROM), is used to store program instructions and fixed user data. Non-volatile memory is used to store this kind of data because it is not lost when power is disconnected. In comparison, volatile memory loses its contents when power is removed and so it is used for temporary or intermediate data.

Memory characteristics also contribute to the capabilities and performance of the hardware system. Primarily, the available memory size defines how much data can be stored. The Arduino boards researched have 32KB of program memory available compared to the Gigabytes available on some Samsung Artik boards. This constrains the number of program instructions which can be stored and means the software running on the Samsung Artiks could be much more complex. Memory word length also has implications. The word length is the number of bits which a single unit in memory can store, and is commonly found to be 8, 16, 32 or 64 bits. Larger word lengths can store larger pieces of instructions or data, such as larger numbers.

2.5.3 Input/Output

The Input/Output (I/O) lines of a hardware board are the interface between sensors or actuators and the processor (or beyond). Without this physical connectivity, developers would not be able to capture information about the physical world, stopping the IoT concept in its tracks. This aspect of hardware boards is therefore important to the flexibility and future potential of IoT applications. To ensure compatibility between hardware platforms and external hardware like sensors, various industry standards have been developed.

Before any I/O standards are discussed though, it is first important to establish the importance of the operating voltage for an individual electronics circuit. The hardware boards analysed operate at various logic voltages; for instance most Arduino boards run at 5V whereas the native voltage of the Intel Edison is 1.8V. The standard logic voltage also effects I/O operation. Digital sensors may have their own, incompatible standard logic voltage—a microcontroller running at 5V would damage a sensor running at 3.3V. If directly interfacing between processors and sensors, a logic level shifter might have to be implemented so careful planning is required. Logic voltage is another aspect which affects the lifetime of a battery-powered node (Mikhaylov and Tervonen, 2010).

General Purpose Input Output (GPIO) pins are a blank canvas for engineers and developers. They are digital I/O pins with no default use and as their name suggests, can be configured as input or output. Since they are digital, GPIO pins can only read or write with 2 discrete values, logic high (1) or logic low (0), and these values usually respect the standard system voltage. Depending on the device, a number of GPIO pins might support Pulse Width Modulation (PWM). PWM is a technique for emulating analogue levels by pulsing the digital value at differing frequencies. Using a light bulb as an example, PWM could emulate the effect of dimming the light, like what could be achieved with an analogue potentiometer.

While GPIO pins allow for basic I/O operations, there is a need for communicating with more complex data. There are various available standards for interfacing between low-level electronics components. The datasheets for the selected model note some common standards, namely: Serial Peripheral Interface (SPI), Universal Asynchronous Receiver/Transmitter (UART), Inter-Integrated Circuit (I^2C) and Integrated Interchip Sound (I^2S). There are a variety of uses for these standards so this paper will go no further than noting that they exist.

2.5.4 Connectivity

Interconnectivity is yet one more important aspect to hardware designed for the Internet of Things. Since IoT devices are becoming increasing mobile and wire-free, connectivity is being discussed almost exclusively in regards to wireless communication protocols and standards. As was discussed in Section **??**, some IoT-specific wireless standards have already been established. Many of these technologies are supported by the various hardware platforms investigated, validating their purpose. The standards supported by these hardware platforms are IEEE 802.11 (Wi-Fi), Bluetooth and IEEE 802.15.4 (Low-Rate Wireless Personal Area Network).

Wi-Fi is a technology which has become synonymous with wireless connectivity and the Internet. The IEEE 802.11 standard states that the purpose of Wi-Fi is "to provide wireless connectivity for fixed, portable, and moving stations within a local area" where a local area could refer to a home, classroom or office space. Wi-Fi benefits from wide utilisation in all areas and industries so wireless coverage is likely to be in place already. Aust et al. (2012) state that the radio frequencies used by Wi-Fi (2.4GHz and 5GHz bands) are becoming crowded and are increasingly resulting in radio interference. This is being addressed in a new standard, IEEE 802.11.ah, which makes use of the 900MHz frequency band. The two main potential advantages of this standard are greater range and less power consumption, which is beneficial to IoT devices.

Even with this standard, however, end devices are impeded by the large overhead of upper layer protocols used in conjunction with Wi-Fi.

The Bluetooth standard has been repurposed as an effective wireless communication protocol for IoT devices. Prior to the introduction of Bluetooth 4.0 in 2010, it was primarily used in multimedia applications such as streaming stereo audio (Chang, 2014). With the introduction of Bluetooth 4.0 came new features, namely Bluetooth Low Energy (BLE), which consumes such a small amount of power that a sensor can run from a coin battery for months or years. For an average usage scenario, BLE could be expected to achieve a practical maximum range of 77 meters (Labs, 2015) which makes it ideal for use-cases such as body sensors and intelligent transport systems.

One final wireless connectivity standard supported by the investigated hardware is IEEE 802.15.4 Low-Rate Wireless Personal Area Networks (WPAN). The standard states that it " provides for ultra low complexity, ultra low cost, ultra low power consumption, and low data rate wireless connectivity among inexpensive devices" and covers the physical layer (PHY) and Medium Access Control (MAC) sublayer specifications. This IEEE standard has manifested into various competing technologies, in particular ZigBee and 6LoWPAN. ZigBee is a set of layers built on top of 802.15.4 and adds three important features: routing, ad-hoc network creation and self-healing mesh (Faludi, 2010). In comparison, 6LoWPAN is an adaptive layer defined in RFC 4944 and allows IPv6 packets to be transmitted using the short addresses in 802.15.4 (Hui and Culler, 2008). The average range for ZigBee and 6LoWPAN is 291 and 191 meters respectively (Labs, 2015).

2.6 Case Study: UK Smart Meters

2.6.1 Overview

The smart meter project by the British government serves as an ideal case study to exemplify an Internet of Things application. In Section 2.3, this paper described some stand-out use cases, which were collated from various sources. Upon researching the smart meter concept further, this massive governmental project—currently being developed for the United Kingdom utilities sector—was discovered. As more research on this project was carried out, it became clear that it represents a significant commentary in regard to the IoT concept.

In comparison to other potential case studies, the smart meter project is important for many reasons. First of all, the transparent and objective nature of public projects does prevent an operational bias. It would be possible to evaluate case studies provided by commercial organisations; for instance, Digi provide case studies across a number of industries. This however introduces the risk of bias—it is in the business's interest to make a sale. In order to satisfy stakeholders, it is also necessary for the public project to unambiguously document discussions, actions and expectations. Furthermore, the scale, purpose and complexity of this project truly puts the Internet of Things concept to the test.

Before analysing the technical details of this project, it is first important to understand the purpose of smart meters. The project was announced by the Department of Energy and Climate Change (Decc) in 2009. It ambitiously aims to replace the electricity and gas meters for all residents of the United Kingdom with so-called 'Smart Meters' by 2020, giving a projected time frame of 10 years (The Telegraph, 2009). For consumers, these systems comprise of two main components: the electricity/gas meter and the in-home display. The display will wirelessly connect to the meter and will provide a near real-time indication for the resources being consumed, expressed in pounds and pence (The Department of Energy and Climate Change, 2009). Utility

Board		Processor			Memory		Input/Output			Voltage	Connectivity
Make	**Model**	**Bits**	**Cores**	**Speed**	**App**	**RAM**	**Analogue**	**Digital**	**PWM**		
Arduino	Uno	8	1	16MHz	32KB	2KB	6	14	6	5V	-
	Yún	8	1	16MHz	32KB	2.5KB	12	20	7	5V	Ethernet Wi-Fi
	Pro Mini	8	1	8MHz/16MHz	32KB	2KB	6	14	6	3.3V/5V	-
Raspberry Pi	2	32	4	900MHz	SD Card	1GB	0	40	0	5V	Ethernet
	A+	32	1	700MHz	SD Card	256MB	0	40	0	5V	-
	Zero	32	1	1GHz	SD Card	512MB	0	40	0	5V	-
Intel	Edison	64	2	500MHz	4GB	1GB	6	20	4	1.8V/3.3V	Wi-Fi a/b/g/n Bluetooth 4.0
Samsung	Artik 1	32	1	240MHz	4MB	1MB	2	0	0	Unknown	Bluetooth 4.0
	Artik 5	32	2	1GHz	4GB	512MB	2	47	2	1.8V/2.4V	Wi-Fi a/b/g/n Bluetooth 4.0 ZigBee/802.15.4
	Artik 10	32	4+4	1.3GHz/1GHz	16GB	2GB	6	51	2	1.8V/2.4V	Wi-Fi a/b/g/n Bluetooth 4.0 ZigBee/802.15.4

Table 2.3: Summary of hardware boards

companies will also be provided with remote access to the meter, primarily allowing them take readings.

With an estimated cost of £10.9 billion (The Department of Energy and Climate Change, 2014a), the project must deliver benefits to consumers, suppliers and the government. One of the main drivers for meters are the potential financial savings. For the consumer, they are more aware of their own energy usage which allows them to reduce their spending. Into the future, this could become automated with Smart Appliances which wait for cheap energy. For the supplier, large savings will be made through the automation of meter readings and billing. Operational maintenance will also become streamlined as smart meters can immediately identify faults. For the government, efficiency savings will reduce the national environmental impact and will help the United Kingdom meet climate change targets (Thomas and Jenkins, 2012). The total projected savings from the combination of these points is £17.1 billion, resulting in a net benefit to the British economy of at least £6 billion (The Department of Energy and Climate Change, 2014a).

2.6.2 Governance

Effective governance and organisation is imperative for the success of the smart meter roll-out in the United Kingdom. Governance was discussed, in board terms, as part of Section **??** and found three main areas which may require governing: device addressability, information security and ethical & legal accountability. Similar issues face the British government. The Department for Energy and Climate Change have been responsible for managing the smart meter programme since April 2012 and delegates governing roles to various actors in the project. Some of the main actors include the Office of Gas and Electricity Markets (Ofgem), the purpose-built Data and Communications Company (DCC) as well as various subcontractors and licensees. These bodies govern the smart meter project roll-out across many areas, including two which are of special interest to this paper: consumer protection and technical implementation.

Widespread social acceptance is one of the main challenges facing IoT products and services; consumer protection offered by Decc and its subcontractors goes some way to earn this public trust. First and foremost, the energy data generated by smart meters is useful to consumers, suppliers and the government. In response to the privacy issues which this creates, the Decc has developed the Data Access and Privacy Framework (DAPF). The Department of Energy and Climate Change (2015) notes that this governing framework serves three main purposes:

- Protect consumers' interests, including by addressing concerns that consumers may have about privacy;
- Enable proportionate access to data by authorised parties to ensure that benefits can be delivered; and
- Promote competition and innovation in the developing energy services market.

The DAPF is expected to be finalised by 2018. The Decc also addresses specific consumer concerns with governance and legislation. For instance, there are restrictions in regards to the supplier-led installation as detailed by the Smart Meter Installation Code of Practice, including: prohibition of sales during installation visits and the mandatory provision of energy advice.

Governance also extends to the technical standards and specifications used for the smart meter project. One important challenge facing this project is hardware and software interoperability between the large number of parties. The Smart Metering Equipment Technical Specifications (SMETS) document was drawn up in 2012 and represents the technical standards to be followed

by suppliers and systems developers for the in-home products. The responsibility of Wide Area Network communication has been delegated to the Data and Communications Company (DCC). This organisation specifies its own standards for connecting to the WAN, which go as far as XML schema for inter-system communication.

2.6.3 Implementation

From the perspective of this paper, the technical implementation is the most interesting aspect of the smart meter project. There is a huge level of complexity and interconnectivity covered by SMETS and DCC specifications. The completed system will be a distributed and connected platform for smart meters in the United Kingdom and can be split into three constituent components: energy consumers, wide-area connectivity and service users. The implementation is summarised in Figure x.x.

Before discussing the implementation of specific technologies it is first worthwhile to understand the high-level system architecture. The energy consumers for the smart meter project are homes and small businesses. As was previously noted, there are three product types which each premises will have: smart meters (gas and electric), in-house displays and communication hubs. The communication hubs have two responsibilities, namely to provide premises-wide wireless connectivity—dubbed the Smart Meter Home Area Network (SM HAN)— and to connect each SM HAN to the nationwide Wide Area Network (WAN) provided by the Data and Communications Company. Service users are the utility companies like British Gas or Scottish and Southern Energy (SSE) as well as authorised third-parties such as online energy comparison websites. Authorised service users will gain bidirectional connectivity with their customer's smart meters through the DCC WAN.

The SMETS and Communications Hub Technical Specification (CHTS) documents complement one another to unambiguously define the functional requirements for smart meters and the home area networks. Since the communication hub will provide wireless connectivity across a single premises, its functional requirements are indicative of how this network will operate. The Department of Energy and Climate Change (2014b) state that the HAN interface of the communications hub will use ZigBee over the 2.4GHz frequency band. The CHTS also notes the minimum capacity of the SM HAN: four Electricity Smart Metering Equipment (ESME), one Gas Smart metering Equipment (GSME), one Gas Proxy Function (GPF), seven Type 1 devices and three Type 2 devices. Type 1 devices are those which can issue any command to ESME or GSME meters whereas Type 2 devices can only issue read requests (The Energy & Utilities Alliance, 2014).

Individual metering devices have their own functional requirements which will inform their hardware implementation. The high-level expectations of a ESME or GSME device are: a clock, a data store, meter containing one measure element, a HAN interface, a load switch, a Random Number Generator and a user interface (The Department of Energy and Climate Change, 2014c). It is also important to note that ESME or GSME devices must be capable of storing 13 months of readings taken at 30-minute intervals, even after power loss. Thomas and Jenkins (2012) compared the UK requirements against an existing smart meter, the Elster REX2. This meter has has an 8-bit Texas Instrument SoC with a clock speed of 32MHz, 4KB of RAM, 256KB of application memory along with 21 GPIO lines and up to 8 analogue inputs. These specifications would place the required hardware in the realm of the Arduinos researched in 2.5 however the authors argue that the UK devices would need to be more powerful and with greater memory due to the data storage and potential computational requirements.

The final piece to the smart meter project is the Wide Area Network managed by the Data and Communications Company. The DCC is responsible for secure communication, access control

and scheduled data retrieval for this project. Since the installation environments will differ from premises to premises, various connectivity methods are being employed, such as cellular, broadband and 802.15.4 mesh radio. This is, in essence, the platform supporting the whole project. The DCC will maintain a web-based service where service users can issue HTTP requests with XML document payloads to achieve tasks. The possible functions are defined in the DCC User Interface Specifications (DUIS).

2.7 Conclusions

Chapter 3

Specification

To obtain a complete and practical understanding of the IoT paradigm, the software implementation accompanying this paper will comprise of two distinct components: a generic IoT application framework and a demonstration application. Mineraud and Tarkoma (2015) state that developers of IoT applications would benefit from a generic, standardised architecture, and that is the aim of the framework component. The second component has two main purposes. First of all, it will implemented the framework component with the aim of validating design decisions. The second purpose is to demonstrate the opportunities present in IoT. This chapter will define the requirements for both of these components.

To help reference these two software components throughout the remainder of this paper, they have been given product names. Haar is a cold sea fog which forms on the east coast of Scotland and is a fitting metaphor for IoT being all around us. It is also a nod towards the Fog Computing paradigm, and towards Robert Gordon University's location in Aberdeen. The generic framework will be called the Haar Engine and the demonstration application will be named Haar.

3.1 Functional Requirements

3.1.1 Framework

The target users of the Haar Engine are designers and developers of IoT services and applications. The general aim of a framework is to encapsulate design decisions into a logical and elegant toolkit. The Haar Engine aims to perform the same task by encapsulating the knowledge which has been collated in Chapter 2. The following subsection specifies its main requirements.

User and Device Management

The following requirements have been derived from the analysis of existing platforms and the requirements of use cases.

1. The framework shall model users
   (a) A user shall be identified by a unique username
   (b) A user shall have an associated profile including: full name, email address
   (c) Users shall be assigned a privilege level: standard or administrator

2. The framework shall authenticate genuine users only
3. The framework shall model connected IoT devices
   (a) A device shall be identified by a unique identification number
   (b) A device shall be designated as either an input (sensor) or output (actuator)
4. The framework shall authenticate genuine devices only
5. The framework shall authorise and facilitate the management of devices
   (a) A device will have one owner
   (b) Access to device data and configuration options shall be restricted to the device owner by default
   (c) The device owner shall be able to make device data public
   (d) A device owner shall be able to grant access permission to specified users

Device Connectivity

A main aspect of the IoT concept is the interconnection of devices. The Haar Engine is responsible for managing the connectivity between devices and itself. The issues surrounding bidirectional communication was described in Section 2.4.4, and these should be addressed by this framework.

1. The framework shall establish a connection with end devices
   (a) Devices shall be addressable and identifiable whilst the connection is open
   (b) The connection shall remain open under normal operating conditions
   (c) Under error conditions, the connection shall be re-established
2. The connection shall be bidirectional
   (a) Input devices shall be able to transmit data packets to the framework
   (b) The framework shall be able to transmit data packets to output devices
   (c) The connection shall be tolerant of network obstacles such as firewalls and Network Address Translation

Data Storage

IoT applications will collect a large volume of data. The data collected by these applications will vary in structure and type, so the Haar Engine must be tolerant of this. The developers potentially using this framework will have specific domain knowledge and so should be empowered to tailor the system for their needs.

1. The framework shall store data received by devices
2. The framework shall provide an interface for using different database types
   (a) The framework shall provide a default 'out of the box' database connector
   (b) Application developers shall be able to create their own database connectors

Data Access API

The Haar Engine must be able to provide access to stored data. Primarily this access will be for use in its own application, however third-party access would enable interoperability between external services and applications.

1. The framework shall provide structured access to stored data
   (a) Access shall be granted to authenticated and authorised users only
2. Application developers shall be able to modify data access
   (a) Application developers shall be able to add custom data searches
   (b) Application developers shall be able to add custom data filters

Real-Time Event API

The ability to react to data in real-time enables novel use cases. As with the Data Access API, the primary purpose is to satisfy the needs of a specific application. However, third-party access to this API would allow for further real-time integrations.

1. The framework shall provide a real-time event API
   (a) Client applications shall be able to listen for device events
   (b) The real-time API shall leverage bidirectional connectivity with devices
2. The API shall provide real-time access to data received from devices
3. The real-time API shall enable developers to create application rules and triggers

3.1.2 Demonstration Application

Haar is the second software component for this project. The aim of Haar is to validate the Haar Engine in practice, as well as to demonstrate the opportunities present in the IoT concept. This will be achieved through the implementation of an arbitrary web-based application which sufficiently covers key features.

Devices

The IoT concept enables the interconnection of devices so this is a logical requirement to begin with. Haar should comprise of at least two separate local device networks to demonstrate global interactivity. These local networks should themselves comprise of a number of devices with enough variety to validate Haar Engine use cases.

1. The application shall comprise of two local area device networks
2. Devices of these networks shall be capable of establishing a connection between themselves and the web application
3. The device networks shall comprise of two sensor devices
   (a) The device network shall contain a sensor device which measures a single data point, such as temperature

   (b) The device network shall contain a sensor device which measures more complex data, such as a vector of wind direction and speed

4. The device networks shall comprise of a single actuator device
   (a) The actuator device shall be flexible enough to represent a variety of data types

Dashboard

The web-based dashboard will be the main interface between a user and their devices.

1. The dashboard shall be accessible on the World Wide Web
2. The dashboard shall authenticate users with a username and password
3. The dashboard shall list profile details
4. The user shall be able to manage their devices
   (a) The dashboard shall list devices owned by the user
   (b) The user shall be able to add a new device
   (c) The dashboard shall show whether devices are connected or not
   (d) The user shall be able to transfer ownership of the device
   (e) The user shall be able to grant access permissions to other users
5. The dashboard shall enable users to manage application rules
   (a) The dashboard shall list existing rules
   (b) The user shall be able to create new rules
   (c) An actuator device shall be able to react to sensed data
   (d) User devices shall be able to integrate with third-party services [See third-Party Integration overleaf]
6. The dashboard shall display data for a given device [see Data Visualisation below]
7. The dashboard shall be able to trigger actuators with virtual controls

Data Visualisation

There is an opportunity to develop a rich user interface for the dashboard. Since so much of IoT relies on data, it would be appropriate to develop example data visualisation tools for it.

1. A section of the dashboard shall be dedicated to data visualisation
2. A generic visualisation tool shall plot temporal data as a graph
   (a) The date range shall be configurable
   (b) Multiple data sources shall be comparable on one graph interface
   (c) The graph shall leverage the real-time event API to update when new data is received
3. Specialised visualisation tools shall reflect specific device types. For example, a virtual thermometer would visualise a temperature
   (a) Specialised visualisation tools shall leverage the real-time event API to update when new data is received

Third-Party Integration

Interoperability between IoT services is described as a challenge in Section 2.2. Haar should therefore demonstrate an ability to integrate with third-party services.

1. The application shall provide pre-defined third-party integrations
2. The integrations shall be listed as third-party rules
3. One integration shall demonstrate reaction to a third-party event
4. One integration shall trigger a third-party service in response to new data

3.2 Non-functional Requirements

In addition to the features required for Haar and the Haar Engine, there are a number of additional considerations. Since both of these software components are destined for use in the same system, this section will detail their non-function requirements collectively.

3.2.1 Constraints

Network Environment

The potential operating environments in which IoT devices could be implemented vary from domestic household appliances to heavy industrial applications. These networks will all have different configurations, capabilities and customisability. In order to reduce deployment issues, the software components must operate in a common network scenario. A typical home network will be used as a baseline, where only common firewall ports will be allowed.

Public IP Address

IPv6 is the successor to IPv4 and will allow every connected device to have a unique, globally addressable IP address. While this is the direction the industry is heading in, many businesses and Internet Service Providers are not prepared for its adoption. Network Address Translation (NAT) is still commonly deployed and prevents devices on these networks from being publicly addressable. It should therefore be assumed that end devices will not have a public IP address.

Initial Setup

In a domestic situation, consumers would be expected to connect products to their internet connection by themselves. This is an important user experience to get right. While this project acknowledges the importance of this, it will not be considered a requirement for Haar so more effort can be afforded to developing a robust application.

Sensor Nodes

To demonstrate the various framework use cases, the application should connect a variety of sensor types. To ensure this project remains achievable, sensor nodes will be limited to those whose data can be represented by a single data interchange format like XML or JSON. Binary data such as images or audio will not be considered for this project.

3.2.2 Performance Requirements

Since this project is a research endeavour, it neither requires nor promises any performance benchmarks. In saying that, the implemented product should meet general performance expectations, especially in regards to any web-based user interfaces.

3.2.3 Security Requirements

This particular software product will not store sensitive personal details, nor will it need to satisfy criteria for professional accreditation. However security is an important topic in IoT and so real-world security measures should be implemented. In particular, any device or application connectivity should be implemented over encrypted connections to prevent potential network sniffing. Also any live demonstration versions of this software should be hosted on up-to-date hosting services to prevent any bug exploitations.

3.2.4 Software Quality Attributes

Extensibility

The application domain of an IoT project could be in virtually any industry. It is important that this software product is extensible, meaning that the core functionality can be extended to cover specific use-cases.

Portability

The server infrastructure supporting a generic platform such as this could vary between organisations and use-cases. For instance it may be beneficial to use a Platform as a Service such as Heroku or conversely a range of dedicated servers may be in use. This project should be platform agnostic.

Scalability

It is difficult to predict the exact number of sensor nodes and the data which they will produce. The software components should be scalable, either horizontally and/or vertically.

Testability

It is important that core system functions execute as designed. This project should therefore be testable through assertions where possible and appropriate.

Robustness

With many devices relying on a system for data and connectivity, system stability is an important focus. The framework should strive to be as fault tolerant as possible, however this is the sum of many parts. Reliable hardware infrastructure and high quality tested code all play a part.

Reusability

Of course a main feature of a generic framework is the ability to be reused. Two different organisations with different use-cases should both be able to use the framework without issue.

Chapter 4

Design

The purpose of this chapter is to document architectural design decisions for Haar and the underlying Haar Engine. It is an important stage in the development of these system components because it combines the requirements specified in Chapter 3 and the knowledge gained in Chapter 2. The process has been highly iterative and has led to the upcoming system architecture.

One fact which has become clear throughout the design process is that the design and implementation are not mutually exclusive. Some ideas presented in Chapter 2 are dependant on specific technologies, such as the ability to facilitate bidirectional communication (Section 2.4.4). For this project, the available tools and technologies have therefore informed the design, and this has been integrated to the design process.

The design process for each functional requirement has consisted of three iterative stages, as shown in Figure 4.1. First of all, the initial research is consulted to check if any knowledge could inform the design. The requirement and existing knowledge were then used as a basis as a small feasibility investigation for various technologies. Based on these findings, the design was updated. This process was repeated for individual requirements, as well as between requirements to ensure the complete system will operate adequately.

4.1 Haar Engine

The first component to get design attention is the Haar Engine. Since this will support the main system functions required for IoT applications, its design is heavily informed by the available technologies.

One major design decision is the programming environment used to implement the Haar Engine. It is important because it informs how the code can be structured. First of all, the chosen programming language and ecosystem must be able to support the requirements outlined in the previous chapter. Beyond that, the language and framework must provide suitable constructs so application developers can easily extend and modify for their own needs. Various languages and architectures were investigated (such as PHP, Ruby, Python and Go), however one stood out as being most suitable for this project: Node.js.

Node.js is a server-side implementation of JavaScript. More specifically it implements Google's V8 JavaScript engine and it is ideal for this project due to its event-driven nature. In general, it is based on the event loop concurrency model - a single-threaded process which loops continuously over a message queue (Mozilla Developer Network, 2016). Haar Engine is expected to be focussed on input and output (IO) operations. IO operations using more traditional techologies such as PHP, Python and Ruby are synchronous in nature and block futher execution of the process. In

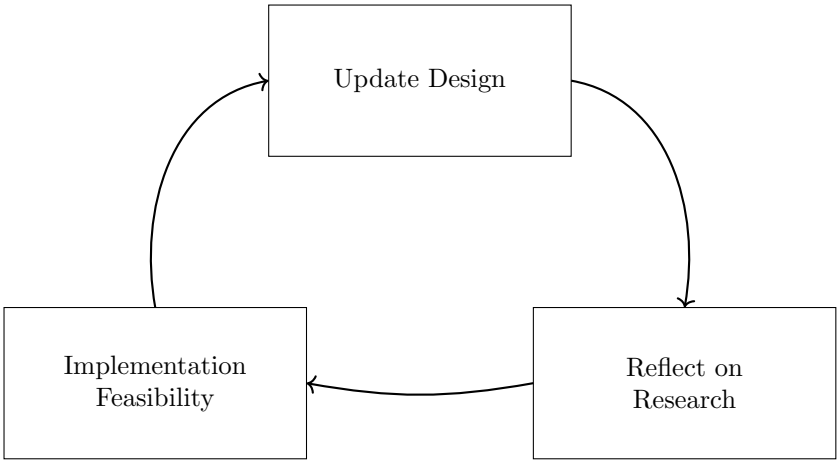

Figure 4.1: Iterative steps when designing for functional requirements

comparison, JavaScript (and therefore Node.js) can continue processing other instructions while waiting for an IO operation to complete.

There are various web application frameworks developed on top of Node.js, with the *de facto* standard being Express. The Express framework is based on the concept of middleware, where middleware functions anonymously pass objects to the next one until a web request has been completed. This concept could therefore be used to an advantage for system aspects like authentication.

Bearing these language features in mind, a final design for the framework has been defined. A logical view of the main system components can be seen in Figure 4.2. The design decisions for each of these framework components have been described in the following subsections.

4.1.1 Data Modelling

The functional requirements in Chapter 3 make reference to a number of modellable objects. In particular these are users, devices and rules, and they all have a notion of ownership and authorisation within the framework. During initial research, it was found that an existing platform (Nitrogen.io) abstracts similar objects into a collection called principals. Throughout the design process, users, devices and rules have have proven to be first-class citizens of the framework, so the Haar Engine will adopt the same terminology.

The three principal objects set foundations for the complete data model as well as other framework components, so it is important that they are modelled correctly. This subsection will first describe the traits of each principle before expanding their relationships into an entity-relationship model. Figure 4.3 illustrates the relationship between the principals.

Users

The user principal is fundamentally responsible for authorisation within the Haar Engine. This will model an end-user account such as an individual person or a company, who will access their account using a unique username and a password. The user is responsible for managing their own devices and rules. The Haar Engine will identify user principals with a unique identifier. This would allow users to change their username while maintaining relationships with other principals.

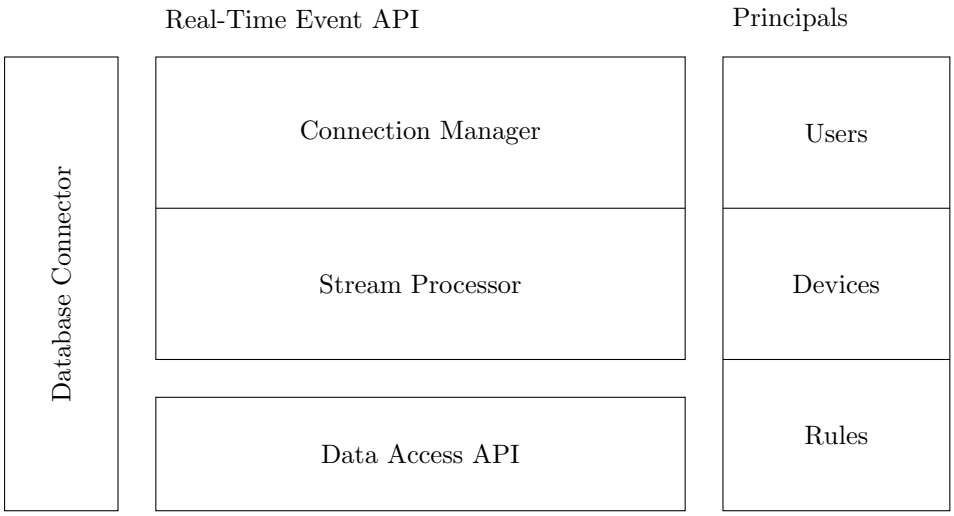

Figure 4.2: A logical view of system architecture for the Haar Engine

User principals will be the gatekeeper to devices associated with their account. First of all, they will be able to attach new devices to their account. Users will also have the ability to share access to sensor devices with specific users, or make that data publicly available. Finally, user principals will have the ability to transfer ownership of the device to another Haar Engine user account and relinquish administration control over it.

User principals will also be able to create, modify and delete rules. When a new rule is created, the user will be designated as the owner of it. Unlike devices, users cannot share rules with another user, nor can they transfer ownership. It is foreseeable that a rule is configured to watch sensor data which is owner by another user. Since access to a third-party sensor cannot be granted, the rule would not have appropriate access permissions to run.

Devices

The device principal will model a single IoT device. The device can be one of two types (either a sensor or an actuator). Devices will be owned by one user, however further users may be granted access to access sensor data. Devices will be identifiable with a unique identifier in the Haar Engine. This identifier will be used for the purpose of authentication and authorisation against user principals.

The device model will also describe the type of data which it supports. For sensor devices, the principal will note the type of measurement (a single value or a vector of values) and the unit(s) of measurement. In the case of actuators, the principal will describe possible output types (single value or a vector of values), as well as value limits. These data type descriptions will be used when defining rules.

Rules

Rule principals define triggers within the Haar Engine. They are created and owned by a user principal. A rule will consist of a watch target, trigger criteria (such as threshold values), an action target and an action. The watch target is a sensor to monitor and when changes are detected, its output values will be compared against trigger criteria. If the criteria is met, the

action target (actuator) will be updated in accordance with the action to take. The rules may run infinitely or they may be configured with a start and end date.

Entity Relationship Model

1. Entity Relationship Diagram using Bachman Notation.

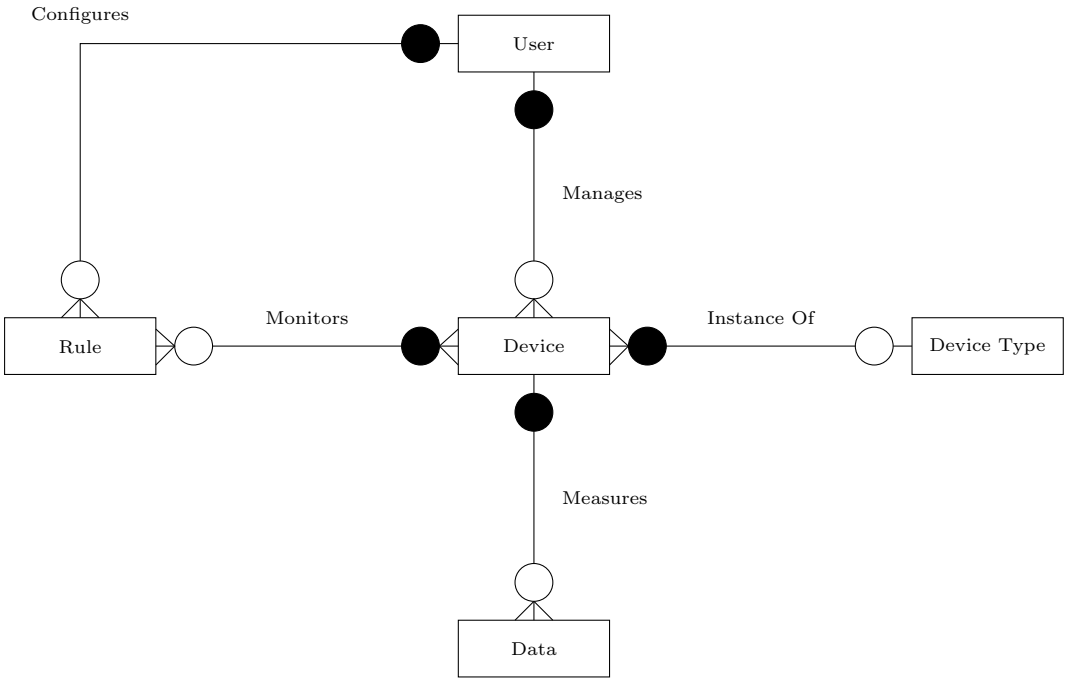

Figure 4.3: Entity Relationship Diagram for the Haar Engine

2. Entity Sets

- User(id, username, password, role)
- Device(id)
- Device Type(id, type, structure)
- Data(id, values)
- Rule(id, watch target, criteria, action target, action)

3. Relationships

- Manages — User manages Device [1:m][m:o]
- Configures — User configures Rule [1:m][m:o]
- Instance Of — Device instance of Device Type [m:1][o:m]
- Measures — Device measures Data [1:m][m:o]

- Monitors — Rule monitors Device [m:1][o:m]

4. Constraints and Assumptions

- Device Type structure is an array of objects. Each object describes a possible measurement.
- The Device Type of a device will never change

4.1.2 Database Connector

Efficient data storage is an important performance consideration. The requirements in Chapter 3 also state that application developers should have the ability to change the database type to suit their own application. The design solution arrived at for this requirement is a database connector API.

The database connector API will provide a set of foundation methods for operating on principal models and data generated by devices. The framework will use a consistent programming interface and this will allow application developers to change the connector for one of their own design. In a more strictly-typed programming language, object-oriented constructs such as abstract classes or interfaces might be used to define class signatures. With JavaScript, programming techniques like function composition might be more appropriate.

Whilst the Haar Engine will provide the capability of changing the database connector, an 'out of the box' connector will be provided. A NoSQL database fits the purpose of the framework better than an SQL-based database because the data stored could be of varying formats and structures. There are a variety of NoSQL databases available, which will be discussed in the implementation chapter.

4.1.3 Data Access API

There are a number of mature standards used for structured data access on the web, so this requirement of the project is about choosing the most appropriate. Two widely-used access standards are Simple Object Access Protocol (SOAP) and Representational State Transfer (REST). These standards also use data interchange formats such as eXtensible Mark-up Language (XML) or JavaScript Object Notation (JSON).

A REST API was found to be a better fit than SOAP for the Haar Engine. The REST access operators integrate much easier with potential implementation technologies like Node.js and Express and it is a simpler standard to use. The JSON data interchange format is based on JavaScript object constructs so it naturally integrates with a JavaScript-based application.

Authentication will have to be enforced by every component of the Haar Engine, including the Data Access API. One characteristic of REST APIs is that they are stateless, meaning no session data is stored server-side and that every request will have to be inspected for appropriate access permissions. Two authentication techniques which will be used are a username and password combination, as well as token-based authentication. Clients will first authenticate with their username and password and if accepted, the REST API will return a unique access token for use in subsequent requests.

4.1.4 Real-Time Event API

One main characteristic of IoT is the ability to react on sensor data in real-time. This makes a robust and effective real-time API a priority for any given application. Giving structured access to this feature is challenging due to the points described in Section 2.4.4. For this reason, the problem has been divided into two constituent components: opening and maintaining a true

bidirectional communication channel, and efficiently handling messages between the two nodes of the channel.

One technology which goes some distance in helping with this is MQTT. As described in Section 2.4.3, MQTT is a publish-subscribe protocol which facilitates the transmission of messages. MQTT is a very robust protocol however given project constraints and additional the additional complexity which it might create, it has been deemed unfeasible for this project. Instead, the JavaScript project Socket.io can open and maintain WebSocket connections, with fallbacks if the WebSocket protocol is unavailable. The message-passing concepts of MQTT could be applied to Socket.io.

Connection Manager

The connection manager is responsible for navigating potential network issues to maintain a bidirectional communication channel. Choosing to use Socket.io and the WebSocket protocol should alleviate any of the issues regarding firewalls and NAT. Socket.io provides an API for managing WebSocket connections. Clients can connect to namespaced URIs which encapsulate different categories of client—the Haar Engine will use two default namespaces: sensors and actuators.

It is also the responsibility of the connection manager to authenticate clients, whether they are users or devices. Socket.io also solves this problem with the use of its own middleware capability. This middleware could make use of authentication mechanisms already in place for the Data Access API.

Stream Processor

Once a connection has been established, it is the responsibility of the stream processor to act on messages. The Socket.io API allows clients to join one or more rooms within a given namespace and this capability will be leveraged by the stream processor. For example, multiple sensors will each have their own room under the sensor namespace. The stream processor can listen for any sensor events and store the data. Alternatively, stream processing rules (based on the rule principal object) could listen for messages in a single sensor room and act accordingly.

4.2 Haar

The main purpose of the second component, Haar, is to validate the implementation of the Haar Engine. Most of the heavy lifting will be done by the Haar Engine, however this application also requires design attention, especially in regards to the end devices. There are three main components to be investigated: the back-end application, front-end application (dashboard) and the wireless sensor network itself. Figure x.x. illustrates the relationship between these components.

4.2.1 Back-End Application

The back-end application is simply an instance of the Haar Engine. It is expected that the data models will be extended for one or two application-specific fields (such as a profile picture and biography for users) however it will generally be left as designed. The back-end application will be self-contained and separate from the front-end application.

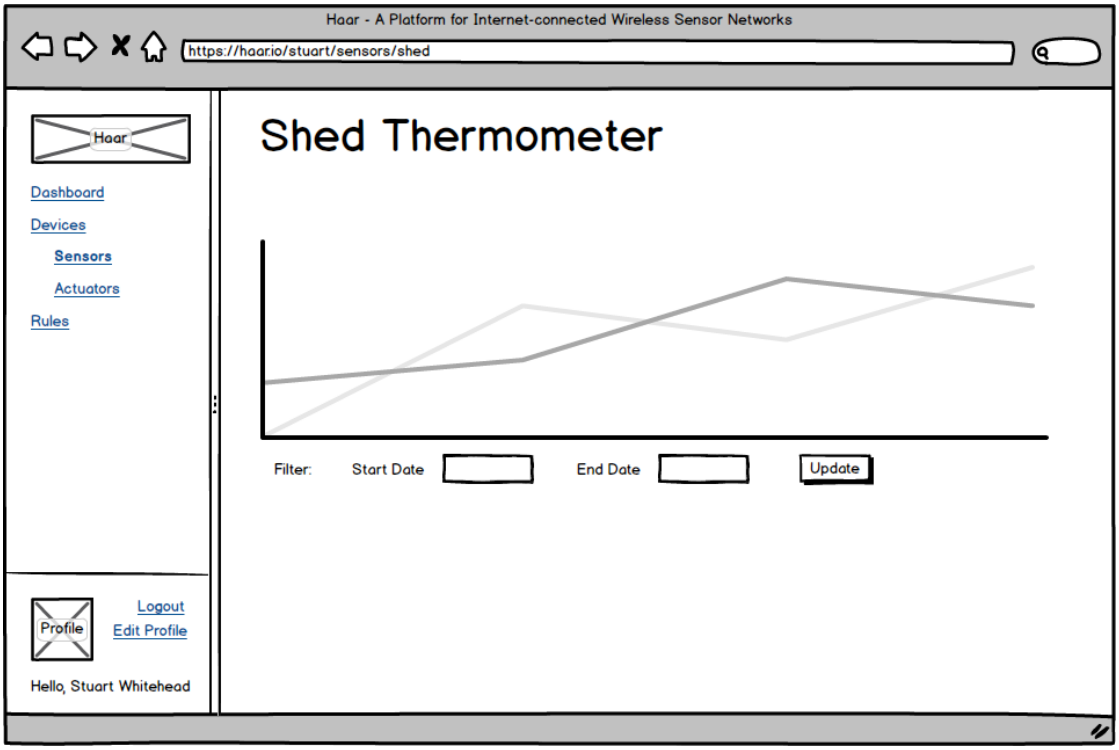

Figure 4.4: UI layout for desktop-sized web browsers

4.2.2 Front-End Application

The front-end application is, in essence, simply a user interface for the back-end APIs. It will make use of the Data Access API and Real-Time Event API to manage users, devices, rules and the authentication for all of them. The main challenge presented by the dashboard is handling very dynamic, real-time data in a structured and efficient way for a variety web browsers. The implementation details of this will be discussed in detail as part of the next chapter.

What can be discussed at this stage is the design, layout and behaviour of the user interface. Figure 4.4 illustrates the general layout which will be used. On desktop-sized web browsers, the window will be split into two panes—one used for navigation, and one for displaying the current page content. On mobile-sized web browsers, only one pane will be displayed at a time. By default, this will be the content pane and when the navigation button is clicked, the navigation pane will be shown.

The behaviour of the user interface is a challenging aspect. It is expected that aspects of the user interface will react in real-time to new data, so it will also have to open a connection to the Real-Time Event API. This will be possible by using the Socket.io library client-side. The challenges begin when the user starts to navigate to different pages of the dashboard. Traditionally this would load a completely new page, however real-time connections would have to be renegotiated and this is inefficient. Instead, dashboard pages will be loaded asynchronously. Care must then be taken to achieving this in a structured way with in its implementation.

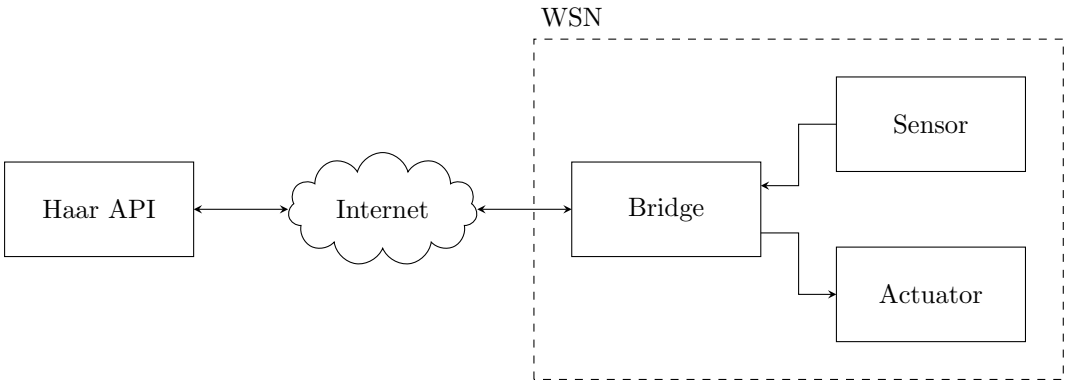

Figure 4.5: System architecture for Haar devices

4.2.3 Wireless Sensor Networks

The third and final component for the Haar application is the Wireless Sensor Network. The WSN comprises of the physical sensors and actuators which create an interface between the physical and digital worlds. To fully demonstrate the capability of the Haar Engine, there is a requirement for a variety of sensor types.

Before designing individual devices it is first important to plan the data flow between them. It has already been decided that the Haar Engine will support bidirectional communication through the use of the WebSocket protocol, meaning that end devices must also support it. However, to maintain a persistent WebSocket connection any given end device must be awake 100% of the time. IoT devices should be as power efficient as possible and maintaining a WebSocket connection would prevent them from being able to sleep. Similarly, the WebSocket protocol requires extra software overhead to manage connections; simpler, more efficient protocols already exist for IoT device communication.

As described in Section 2.5.4, ZigBee is a lightweight wireless communication protocol ideal for use in IoT solutions. ZigBee, however, is not compatible with the TCP/IP protocol used for Internet communications. To compromise on this issue, a bridge solution will be used. A bridge device will maintain the WebSocket connection with the Real-Time Event API; when a WebSocket packet is received, the bridge will translate it into a ZigBee packet and vice-versa. This system architecture is shown in Figure 4.5.

End Devices

To successfully demonstrate the IoT framework, a number of devices are required. Ideally, these devices should be a true representation of an IoT device and not just mocked using software emulation. For the purpose of this project, a true IoT device is considered as: constrained resources, low power and inexpensive.

Out of the hardware platforms investigated in Section 2.5, the Arduino development platform is the most suited. The Arduino hardware boards and software tools are open source giving complete transparancy to their implementation. The Arduino IDE also comes packaged with all the software required to develop and install the c/c++-based programs. There are a variety of hardware board and accessories and they are all inexpensive and widely available.

Most Arduino boards have limited connectivity and very rarely have a wireless option built in. As a result, an additional Radio Frequency (RF) hardware component is required. After

researching available hardware, the XBee radio chip is the only feasible solution which supports the ZigBee protocol. The XBee chip is a very flexibile device and actually contains its own microcontroller. This characteristic will allow Haar devices to delegate communication duties to the XBee device whilst only needing to manage the intepretation of sensor data itself. Another benefit of using XBee is that configuration and debugging software is available for computers.

As described in Section 3.1.2, devices with enough diversity are required to fully demonstrate the IoT concept. Thought was first given to the output device; this was required to be flexible enough to reflect a variety of sensor data. An existing IoT product called the Good Night Lamp product provided inspiration for this device. The Good Night Lamp is a cellular-connected bedside lamp in a master-slave configuration. When the slave lamp is turned off, the master lamp will also turn off, no matter where in the world they may be. The developers behind this product suggest that this creates an emotive connection with a loved one, such as that between a child and their travelling parent.

To build on the concept of the Good Night Lamp, the output device for Haar will be an RGB colour lamp. Firstly, an RGB actuator like this has three data channels - red, green and blue. This characteristic can be used to demonstrate a vector of sensor data being transformed and used to trigger an output. Additionally, colours can add additional meaning to a context. For example, if used to reflect a temperature, a blue colour suggests coldness whereas a red colour suggests warmth. This subjectiveness adds a nice human aspect to the device.

Three different input devices will also be built. In order to demonstrate the temperature example just discussed, a sensor which measures the ambient temperature will be developed. The temperature sensor only measures one datapoint, so further devices which measure a vector of data will also be required. Since the output devices is an RGB lamp, it would be nice to demonstrate an RGB colour sensor, too. Each channel of the RGB sensor could be mapped to the respective channel on the output so it simply relays the sensor colour. Finally, a 3-axis gyroscope sensor will be built. Since the gyroscope has 3 axis (x, y and z), they can each be mapped to a channel of the output device. Depending on its sensitivity, a gyroscope could produce a high volume of data, so it will be useful when testing the limitations of the system.

The next stage in planning the devices is to design their schematics. Section 2.5.3 discussed how the operating voltage is an important consideration and that it affects battery life and compatibility with sensor chips. Most Arduino boards operate at 5V however a number of models operate at 3.3v, such as the Arduino Pro Mini as noted in Table 2.3. The XBee radio chip also operates at 3.3v, so this would be a suitable system voltage to use.

In terms of the sensors components, there are two varieties to choose from: analogue or digital. Analogue components are very cheap, widely available and very robust but often require extra calibration steps to glean accurate results. In comparison, digital components are more expensive but often come packaged with helper libraries, use well-supported digital protocols and take accurate readings ‘out of the box.’ Since this project is up against tough time constraints, the extra support provided by digital components makes them more suitable for this project.

The following diagrams illustrate prototype circuits for all device types. There are a number of common characteristics between them which should be explained first. The XBee Explorer is an expander board for XBee radio chips and exposes a prototype-friendly pinout. The basic pins (power, ground, serial in and serial out) are connected to the Arduino board, along with two more pins: CTS and DTR. These are represented by the yellow and orange wires respectively. The CTS pin can be used to determine when the XBee chip is ready to receive data, and the DTR pin can be used to send the XBee chip to sleep. Additionally, the grey and brown wires are used to connect digital sensor chips to the Arduino.

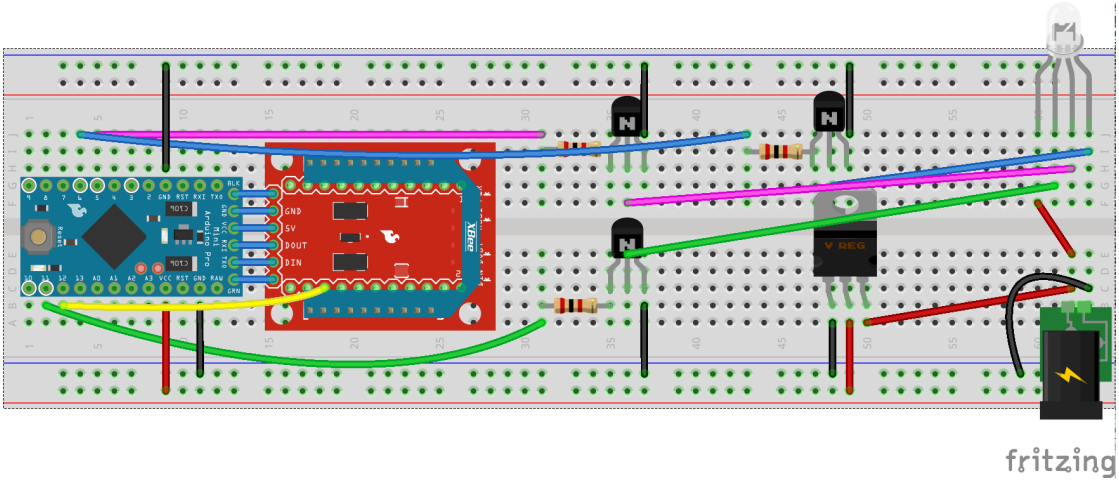

Figure 4.6: Breadboard layout for RGB LED actuator

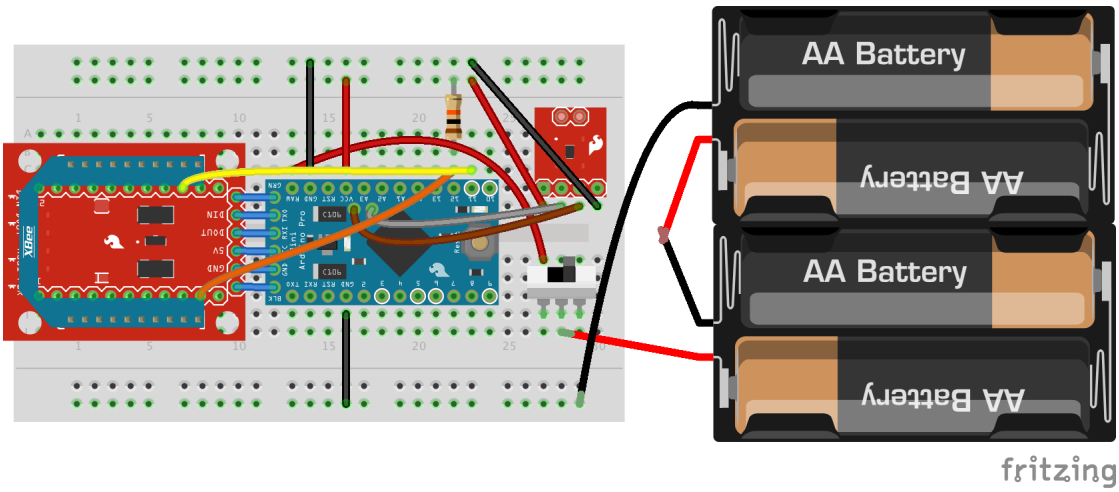

Figure 4.7: Breadboard layout for temperature sensor

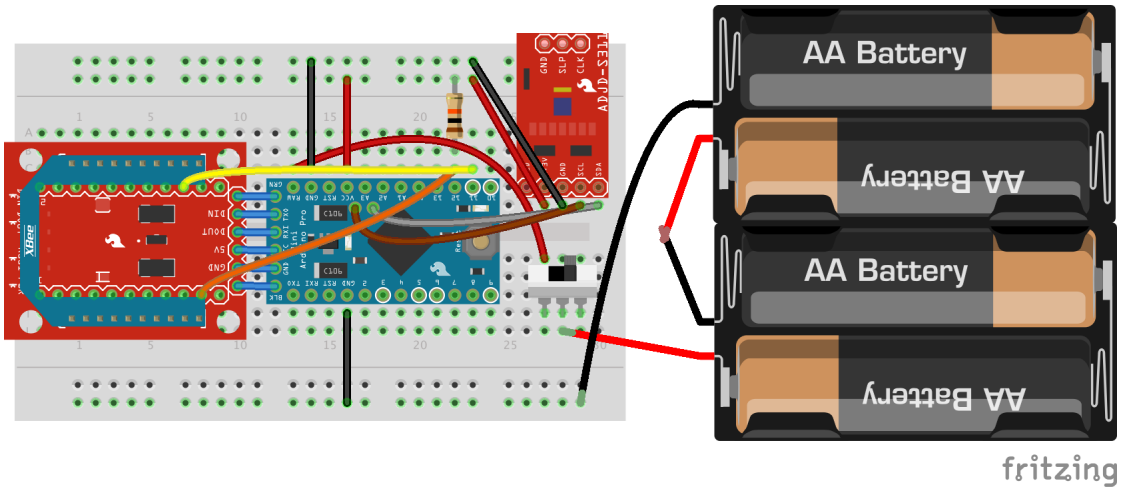

Figure 4.8: Breadboard layout for colour sensor

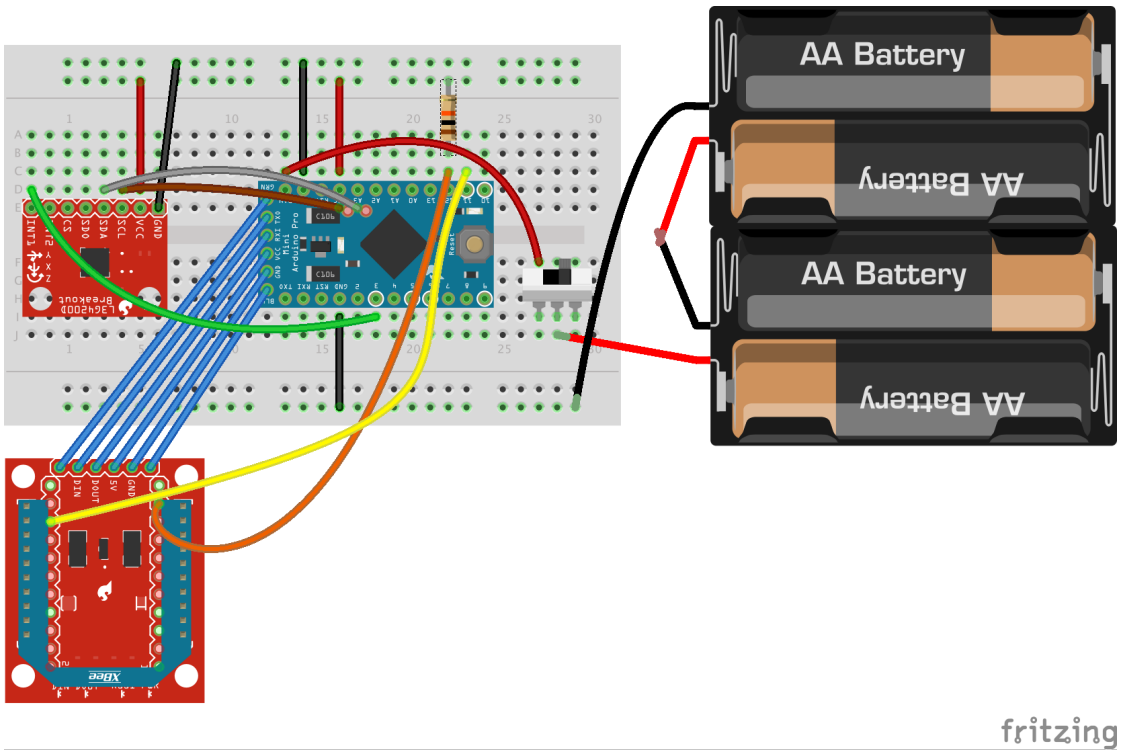

Figure 4.9: Breadboard layout for gyroscope sensor

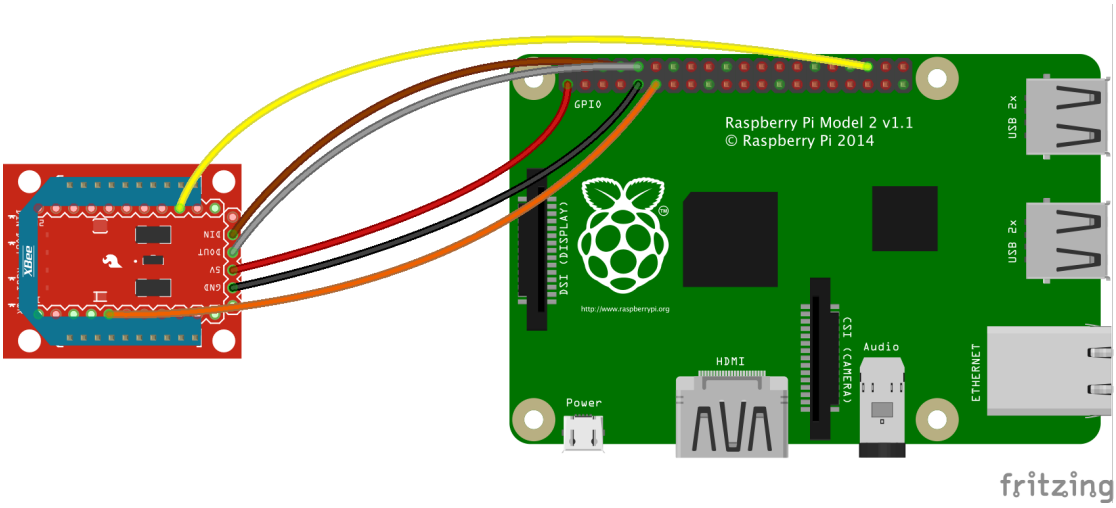

Figure 4.10: Breadboard layout for bridge device

Bridge

Since the purpose of the bridge device is to translate packets between TCP- and ZigBee-based protocols, it requires a different design approach to the end devices. This device must be able to support multiple modes of connectivity. Its software stack must be able to open and maintain a WebSocket connection as well as to parse and transmit ZigBee packets. The additional overhead of supporting multiple communication protocols means that this device will require more system resources than a simple microcontroller. At the same time, this prototype device should remain inexpensive and developer-friendly.

Out of the hardware platforms summarised in Table 2.3, the Raspberry Pi model 2 (RPi2) bests fits this criteria. The RPi2 has a native ethernet port which would allow it to interface with the Internet. Additionally, it provides a number of GPIO pins which could be used to interface with its own XBee radio chip. A combination of these two features means it can satisfy the connectivity requirements. The RPi2 is also a fully-fledged computer with a 32-bit, quad-core CPU running at 900MHz and 1GB of RAM. These specifications are more than sufficient to service resource demands.

The Raspberry Pi community manage a Linux distribution for use on the platform. Since this will be used as the operating system, any program which runs on Linux could be used to implement the bridge software. For the same reasons Node.js is being used to develop the Haar Engine (non-blocking I/O operations, real-time libraries) it will be used to implement the Haar Bridge. The Socket.io client library can be run within server or web browser environments, meaning that it can be used for handling the WebSocket connection on the RPi2. The RPi2 will interface with the XBee chip using the serial GPIO pins.

The following diagram illustrates the bridge device prototype. Since this device is expected to handle many more IO operations than an end device, both of the hardware flow control pins (CTS and RTS) from the XBee have been connected to the RPi2. These are illustrated by the yellow and orange wires.

Chapter 5

Implementation

5.1 Development Process

A software implementation can be set up for success or failure before any code has been written, especially whilst working within a team. This is down to the development process and policies in place. Of course, one process may work for one team or project and not another.

A well implemented process can provide many benefits. First and foremost, everyone in a development team clearly understands their roles and responsibilities. The process can also enforce a hierarchy of decision making, or encourage peer collaboration. Beyond these people-based benefits, a development process can aid in the development of a robust, maintainable codebase. A developer who can understand a codebase and make confident updates is an efficient one.

Although the output of this project is the work of one person, arguments supporting a process-first implementation still apply. The following section will detail the main tools used in supporting this project.

5.1.1 Version Control System

The use of a Version Control System (VCS) is highly recommended in any project. As the name suggests, a VCS manages iterative updates of project files. To generalise its functionality, a VCS maintains a timeline of file states. Different states of the same file can be compared to investigate which changes were made and by whom, and if needed a file can be reverted to a previous state. This means that the complete history of a codebase is recorded. Without using a VCS, code updates would simply overwrite the existing file and old states would be lost forever.

There are a number of Version Control Systems available to developers, such as Mercurial and Subversion. Git is another VCS and was created by Linus Torvalds, creator and maintainer of the Linux kernel. Git is arguably the most popular VCS amongst the open source community, thanks to its use in the development of the Linux kernel. Its command-line interface is highly portable and many community-oriented services support it extensively, such as Github and Bitbucket.

Git along with the web-based service GitHub have been chosen for use in this project thanks to their universal support. But simply selecting a tool is not enough when implementing a development process - guidelines on how to use it should also be decided. There are a number of well-defined development workflows using Git. This project has chosen to implement the 'Git Flow' workflow. Git Flow utilises various features of Git and GitHub, including branching, merging and pull requests.

Describing the Git and the Git Flow workflow could fill an article on its own, so this description will be very brief. In Git, a branch is an instance of the complete codebase. Multiple branches with different versions of the same codebase can be developed in parallel. The merge command copies the changes of one branch into another. A Pull Request is a request to merge one branch into another, which can be inspected and commented on before the merge is completed.

The Git Flow workflow adds structure to these features. The master branch is the production codebase. The develop branch is the in-progress codebase. Feature branches (such as feature/-menu or feature/users-controller) are used to develop a specific feature of the project. Once the feature has been built, a pull request is raised to merge it into the develop branch. Once all features have been merged and tested, a pull request is raised to merge the develop branch into master which is then deployed. This workflow has been employed for this project.

Pre-commit Hook

The Git VCS also supports lifecycle hooks. These hooks allow developers to run bespoke scripts to enforce custom policies, such as limiting access to specific branches. One helpful trick has also been employed for this project through the use of the pre-commit hook.

No matter how descriptive or strict development policies are, developers are only human. Humans can make mistakes and they can be lazy, meaning that poor quality code can be committed to the codebase. The pre-commit hook can be used to enfoce a specific coding standard. It works by testing the committed code against a set of rules. If the tests pass, the code is committed to the codebase but if they fail, a warning message is displayed. By using this technique, the code for this project is guaranteed to be of a certain standard.

5.1.2 Docker

Developers face a number of challenges when developing and deploying modern web-based applications. Primarily, the environment in which code is executed can differ wildly. For any given project, one developer might use a Windows machine whilst their colleague could use an Apple Mac. The staging environment could use one version of Linux whereas the production enviroment could use another. All of these environments could potentially run differing versions of the same software meaning that bugs in the project might not be traceable or reproducable. Development teams have attempted to normalise these enviroments through the use of package managers, however these still leave room for misconfiguration.

A relatively new tool called Docker has been taking the development world by storm since its release in 2013. Docker builds on a technology called Linux Containers. Traditional virtual machines run a complete operating system (kernel, program binaries and any other files) on top of a host machine. In comparison, Docker (and Linux Containers) allow images to use the same kernel of the host operating system. This means that each container is very lightweight but can still benefit from the complete isolation of program processes.

Docker containers are instances of a Docker image. Docker images are compiled according to a Dockerfile which comprise of commands such as COPY (to copy source files to the image) and RUN (to run Unix commands within the Linux Container). This is where the advantages of Docker become clear - explicit, reproducable environments can be built as if working directly on a server. Since Docker is a portable tool, every developer working on a project can run exactly the same environment as each other. When the application is ready to be deployed, the same Docker image can be executed on a production server.

Docker also has powerful composition tools. Anything beyond a simple script or application is typically the sum of many smaller, more specific applications. A simple example of this would

be a dynamic web application backed by a database. In Docker terms, this application would be composed of two images - one image for the dynamic web application, and another for the database. Docker can configure automatic links between instances of these images and this is aided with the helped command-line interface 'docker compose'.

Docker Compose makes use of another configuration file, 'docker-compose.yml'. In this file it is possible to specify which Docker images make up a complete application (which can be custom builds using a Dockerfile, or pre-built images from the global Docker repository). Using this file Docker Compose will intelligently start Docker images in order of their dependencies. Using the dynamic web application and database example, the database would be started first and then the application. Docker will also set up lots of other clever tools, such as a long list of environment variables referring to specific links. These environment variables can then be used within applications to configure things like database connections.

Once applications have been wrapped up into Docker images and compositions, it is then necessary to deploy them to servers. This can of course be achieved manually using the docker CLI, however the Docker ecosystem also includes Docker Cloud, a first-party container deployment service. Docker Cloud uses four more important terms to describe an application:

Node a server which can host Docker containers

Node Cluster a grouping of multiple servers which can host Docker containers

Service one or more running container instances of a Docker image

Stack an application composed of multiple services

Docker Cloud is very simple but hugely powerful. Scaling is an important aspect of any web application and is handled with grace. Docker Cloud will intelligently distribute container instances across all available nodes based on their loads. It will then lay a private network between all services of a stack (irrespective of its host node) so all containers can transparently communicate. This makes load balancing truly simple.

Docker has been chosen as a critical piece of infrastructure for this project. Flexibility, robustness and maintainability are three important aspects when developing any application and Docker aids these areas very effectively.

5.1.3 Continuous Integration

The development process has so far defined how the codebase will be developed and how it will be hosted, but there is still an important step missing. How does newly developed code get deployed in a way which ensures no downtime and without bugs? This is where a technique called Continuous Integration (CI) is used.

Continuous Integration is the glue between the processes at each end of the development cycle. The main ethos behind it is to deploy little and often. This allows developers to catch issues early and to pin-point its cause quickly. ThoughtWorks, Inc. (2015) describe CI with a number of key points, including: automated, self-testing builds and automated deployments. As it turns out, the Git Flow workflow, Docker and Continuous Integration all work incredibly well together.

As has already been discussed, Docker images provide explicit, reproducable environments and this is great for testing purposes. Since all Docker environments will be built equal, it means tests run on a local machine will behave the same on a production server. And because Docker Compose can specify which images needs to be used, any service with access to this information can build the correct application dependencies.

Travis CI is one such continuous integration service which supports the Docker ecosystem. It has been employed in this project to enforce a CI workflow and it follows these steps:

1. Code is developed using the Git Flow workflow
2. All commits to branches and pull requests trigger tests on Travis CI
3. Each test builds the Docker environment to ensure consistency
4. If tests pass, the Docker image is published to the Docker repository
5. The Docker repository will deploy new images through Docker Cloud

This workflow contributes to a solid development process which can be scaled to a large codebase and team of developers.

5.2 Haar Engine

Chapter 3 (Specification) states that this project will create two main software components: Haar Engine and Haar. This section will discuss the implementation of the former - the generic IoT application framework.

To recap, the programming environment chosen for Haar Engine is server-side JavaScript in the form of Node.js. JavaScript itself is an implementation of the ECMAScript standard. ECMAScript has seen a number of updates and is currently on a version dubbed ECMAScript 2015 (Babel, 2016). This latest version adds a long list of new syntax and tools to the langauge. Client-side JavaScript has historically suffered from poor universal compatibility due to differences in browser implementations, however since Node.js is a server-side tool and under a developer's control, this project was implemented using native ES2015 syntax.

Another important point of discussion is the database support offered by this framework. Originally, section 4.1.2 (Database Connector) specified that the framework should offer a Database Conector concept where a developer can implement a database of their own choosing. Whilst investigating possible database types, it was decided that this options was simply too broad giving the time constraints for this project. Rather, Haar Engine will support one, highly extensible database solution. The NoSQL databse MongoDB was selected for this purpose.

MongoDB was selected for a number of reasons. First of all, the flexibility of a NoSQL versus a relational SQL database allows the schema to be flexible and to change with the data. This makes it simpler for a developer to add new data structures. In comparison, to modify an SQL database the developer would have to first update the database schema using a Data Definition Language (DDL). MongoDB was selected from the avaiable NoSQL database solutions because its data structure is based on JavaScript Object Notation (JSON). This makes it easy to interface with the Node.js application code. There are also a number of helper packages available for Node.js.

Once the fundamental technologies were chosen, the next step was to go ahead and build the framework. Since the data models and Data Access API are used to configure the fundamental aspects of the system, they were developed first. The structure of Haar Engine is based on the Model View Controller (MVC) architecture. The models were defined using the Mongoose package, a tool which helps to give structure to the otherwise freeform MongoDB database. Controllers were then built using the previously discussed Express package. They are responsible for authentication, data handling and response building. Controllers respond with JSON objects which are considered the views of this system.

Chapter 4 (Design) also discussed that the RESTful Data Acess API will be responsible for managing authentication and authorisation. JSON Web Tokens (JWT) are a widely-used protocol for stateless applicaton security and were employed for the Haar Engine. Once a user successfully authenticates using their username and password, the Data Access API will generate, sign and return a JWT. A JWT includes so-called 'claims' - a small JSON payload with information about the entity. The JWT must be attached to all subsequent requests to the Data Access API to be granted access. The Data Access API can then verify the JWT and access its claims for use within the application.

Once the fundamental software components were in place, it was then time to build the *raison d'etre* of this project - real-time, bidirectional communication. The specific implementation of this feature had to be revised from the initial plan in Chapter 4 (Design). In that chapter, the JavaScript package Socket.io was identified as a suitable tool. However during development of the Real-Time API, limitations in Socket.io meant that not all features of the publish-subscribe concept could be effectively implemented.

An alternative JavaScript package called Primus was used as the basis for the real-time feature instead. Using Primus, Haar Engine successfully allows clients to publish device data and to subscribe to data events. In a similar way to Socket.io, the middleware feature of Primus allows custom functions to be executed in order to authenticate connections. This allowed Haar Engine to reuse existing the existing JWT authentication mechanism.

As should be done with any modestly-sized software application, a suite of unit tests was developed for Haar Engine. The unit tests developed embraced the stengths of Node.js and Docker. Since the Docker environment ensures that all dependencies are started, it means that the unit tests can use a live database rather than mocked data. When testing the models, the database was cleared and seeded with data before each test to ensure testing consistency. When testing the APIs, HTTP requests are made to the application in order to replicate real-world scenarios.

One final point worthy of discussion is the evaluation and execution of device rules. As a reminder, rules operate on sensor data and trigger an actuator. To provide as much flexibility as possible, each rule is a fully-fledged JavaScript script with access to variables and system functions. This, however, presents its own challenges. Accepting user input - especially code- is dangerous and could compromise the integrity of the host application, and this scenerio is no different.

To protect from the application from malicious or buggy user-generated rules, they are executed in a sandbox environment with access to only their required variables. This prevents user-submitted scripts from tampering or otherwise affecting normal execution of the framework. This is achieved through the use of the Node.js VM module. An additional benefit to using this sandbox technique is the ability to catch syntax errors. When storing a new rule, it can be evaluated with example data and any resulting syntax errors can be returned to the user as validation.

5.3 Haar: API

The second software component produced for this project is Haar, a demonstration implementation of Haar Engine. The main purpose of this demonstration application is to validate the design decisions and to show, by example, that the theory behind this IoT framework works. The implementation of Haar has been split further into four sub-components: API, Dashboard, Bridge and Nodes.

Haar API is the actual implementation of Haar Engine. Since the aim of Haar Engine is to

provide a starting point for an IoT application, the sole purpose of Haar API is to bootstrap the framework and show that it can be included as a project dependency. Quite simply, Haar API includes Haar Engine as a dependency from Github.

5.4 Haar: Dashboard

Haar Dashboard is a web-based user interface built on top of the suite of Haar Engine APIs. It utilises both the Data Access API to configure system settings and the Real-Time API to subscribe to live data events. Whilst not necessary for the operation of the underlying IoT framework, the Haar Dashboard UI makes configuring devices simple and illustrates how a viable consumer solution might work.

Section 4.2.2 (Front-End Application) briefly discussed the challenges facing this user interface. In particular, certain components of the user interface must react to real-time events. To address this challenge, the UI has been developed as a so-called 'single-page JavaScript web app'. JavaScript web apps come with their own challenges. Web pages using this technique are typically not built on the application server but are rather 100% client-side code. Due to this, single-page web apps can experience a long initial delay as the client-side script is executed. Another issues is that search engine optimisation and accessibility are affected because no HTML body is returned by the server - there is nothing for search engine crawlers or accessibility tools to parse.

A clever JavaScript framework called React solves these issues. React is a project developed by software engineers at Facebook. It is based on the concept of a tree composable components, where a single component encapsulates all of the logic which it requires to execute. What is most clever about React is its use of a virtual Document Object Model (DOM). React will first build up the tree of components using this virtual DOM before 'diffing' it against the actual DOM. If anything has changed, React will only update the affected components. Using this technique, React can be used on a web application server. It can build up a webpage and send it as the body of an HTML response. Once the client-side code has loaded, React will then bootstrap the application based on the existing body. This concept is known as Isomorphic or Universal JavaScript.

As was discussed for the implementation of Haar Engine, the latest features of JavaScript are defined in the ECMAScript 2015 standard. Some web browsers can be slow to implement new features whilst other, older browsers may not implement them at all. This is a headache for developers; new features and syntax might solve problems in a more effective way, but they may not be supported by every target browser. This is where so-called front-end tooling is used. A variety of tools exist to convert new, uncompatible syntax into older, widely supported syntax. In particular a package called Babel has been used in this project to compile ES2015 syntax into ECMAScript 5 compatible code, meaning it can be run in all modern browsers. Additionally, a tool called Webpack has been used to package server-side JavaScript modules for use within the browser.

The features which React provide solve half of the front-end engineering problem. React defines how data is manipulated and displayed, but it does not specify how the data itself is managed. For that reason, Facebook have also invented a development pattern called Flux. Flux provides a strict method by which data is added or updated then introduced to the React component tree. The fundamental rule of Flux is that the data flow is unidirectional; that is, data cannot be mutated directly but rather through a pre-defined process. Figure 5.1 illustrates this process. Redux has become the *de facto* standard of this pattern and will be used in this project.

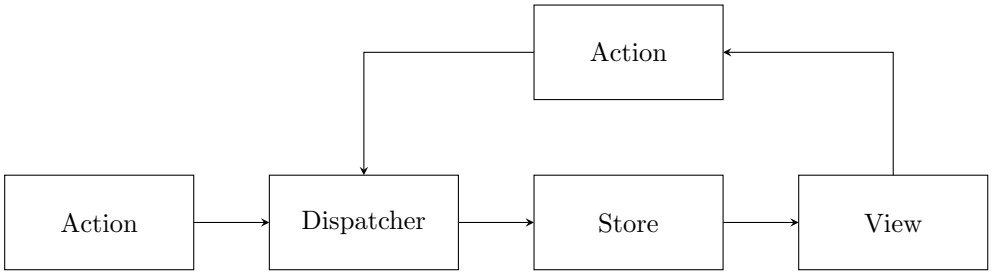

Figure 5.1: Data flow in a Flux React application

The Flux development pattern is also very effective at managing real-time data events. The Primus package used in Haar Engine also has a client-side library builder and this has been used in Haar Dashboard to manage the WebSocket connection. Haar Dashboard will subscribe to data streams managed by Haar API and when data is received, a Redux action will be dispatched and handled appropriately.

5.5 Haar: Nodes

The end devices built to demonstrate the IoT concept have been dubbed Haar Nodes. The first step in their implementation was to physically build the hardware circuits. The Arduino Pro Mini development boards are very minimal and do not have pin headers—if required, these must be soldered on manually. Since these boards were to be used in breadboards, the headers were soldered on. The same went for the sensor components; these arrived without pin headers so these were soldered too.

The XBee Explorer also had to be wired up to the Arduino. As it turns out, the FTDI (data communication interface) of the Arduino has the same footprint as the FTDI interface of the XBee Explorer board. This meant that these two components could be connected through the use of additional headers rather than wires.

Once the hardware was in place it was then time to develop the device software using the Arduino IDE. There are three main components to this software: sleep mechanism, sensor interfacing and XBee communication. All of the sensor devices are battery powered so power conservation was an important consideration. Arduino's deepest sleep mode was employed to save as much power as possible, but this presented its own challenges. When in this deep sleep, nothing except an interrupt will wake the Arduino.

Two different interrupt methods were used to wake the device. The temperature and RGB colour sensors were developed as cyclic devices—that is, they will take a reading then sleep for a pre-determined length of time. The Watchdog Timer feature of the Arduino boads is used to raise the interrupt in order to wake these devices. In comparison, the gyroscope had to use a different approach. This is because it works best when reacting to movement rather than sampling data at a predetermined interval. The gyroscope component has the ability to raise its own hardware interrupt when movement is detected, so this mechanism was used to wake the host Arduino device.

When the Arduino wakes up, it then has to take a sensor reading. This aspect was simplified thanks to the I2C protocol libraries which were available for each sensor component. When each sensor device is awoken, it will retrieve the datapoint value by using the appropriate library. The final stage in taking a sensor reading is to send it to the connected XBee device and this was

achieved through the use of the xbee-arduino library.

The Arduino board communicates with the XBee chip using a serial communication channel and there are a number of implementation details to discuss. First of all, the XBee can operate in two modes: transparent or API. With transparent mode, the XBee chip simply emulates a wired serial link whereas the API mode allows more advanced features to be used. API mode was chosen for this project due to the extra features it provides.

Since the XBee chip contains its own microcontroller, this project attempted to implement sleep mode for it too. The XBee can be configured to use pin hibernation; when the DTR pin is logic high the XBee will sleep and when the pin is logic low, it will be awoken. The operation of this was sporadically unsuccessful and the root cause of the problem could not be identified. As a result, this feature was left out with the acknowledgement that more battery power will be consumed.

The ultimate purpose of the Haar Nodes is to interface with the Haar demonstration application. In order to do that, the sensed data must be transmitted to Haar API via the WebSocket connection. A common data-interchange format is required so that both the devices and the API are compatible. Arguably, the choice of format should belong in the design chapter, however it is dependent on the implementation of the Haar Engine APIs and models. Four different formats were considered: comma-separated value (CSV), JSON DB Schema, Compressed JSON DB Schema and Compressed JSON Map.

Each of the formats have been described and explained below and there were a number of aims to satisfy. Firstly, size of data transmitted has an effect on power usage. More data means that the XBee chip has to transmit for longer periods, consuming battery power. The maximum packet size of an XBee transmission is 72 bytes (Digi International Inc, 2016). In order to limit transmission to one packet, data size should aim to be less that this limit. Additionally, the format chosen should be robust and make it simple to distinguish the datapoints collected.

Comma-separated value

The first data format explored was a simple comma-separated string. This created the smallest payload size (11 bytes for the given example).

```
123,123,123
```

This format is not flexible, however. It requires the datapoints to maintain strict order, and requires all devices to know what this order is. The CSV string does not describe its contents at all - a developer would have no indication what the data refers to.

JSON DB Schema
The second format explored replicates the format of the Data Mongoose model used in the Haar Engine. Haar Engine explicitly defines 'name' and 'value' properties for use in validation. The size of the given example is 76 bytes, more than the 72 byte limit.

This is the most descriptive format and requires no extra transformation before being sent to the Haar API. However, it is unsuitable because of its payload size.

```
[
  {
    "name": "x",
    "value": 123
  },
  {
    "name": "y",
    "value": 123
  },
  {
    "name": "z",
    "value": 123
  }
]
```

Compressed JSON DB Schema
Whilst investigating the previous format, it became apparent that it was unnecessarily expanding on the syntax of a JSON object. By their very nature, they are a key-value store. This means that the format can be condensed into the shown example (36 bytes).

This format is both machine and human readable. Additionally, it does not require much additional trasformation in order to be submitted to the Haar API.

```
[
  {
    "x": 123
  },
  {
    "y": 123
  },
  {
    "z": 123
  }
]
```

Compressed JSON Object
Looking at the previous example, it is clear that there is a redundant structure in place. This format compresses the datapoints into a single JSON object, rather than an array with multiple nested objects. This yields an example payload size of 25 bytes.

No information is lost between this format and the full JSON DB Schema option, meaning that it can be expanded to be successfully submitted to the Haar API and pass validation rules. This is the most effective format and has been chosen for use in the Haar Nodes.

```
{
  "x": 123,
  "y": 123,
  "z": 123
}
```

Figures 5.2 and 5.3 show the finished prototype hardware for the temperature sensor and RGB LED actuator. The RGB colour and gyroscope sensors are similar to the temperature sensor.

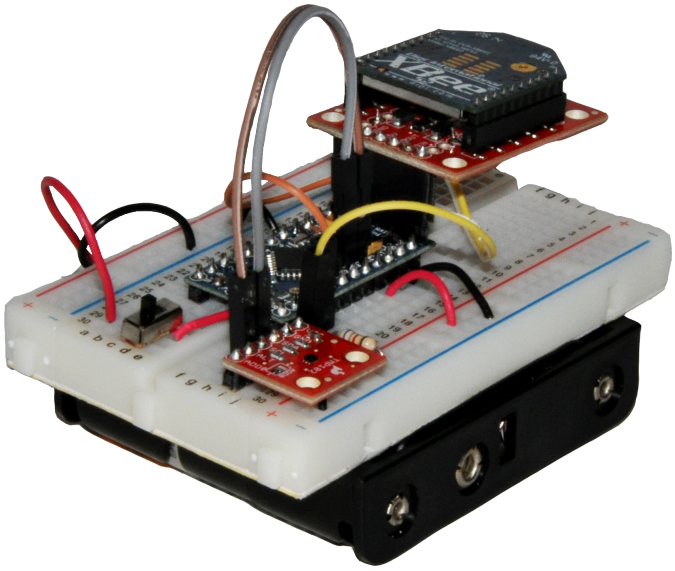

Figure 5.2: Prototype temperature device

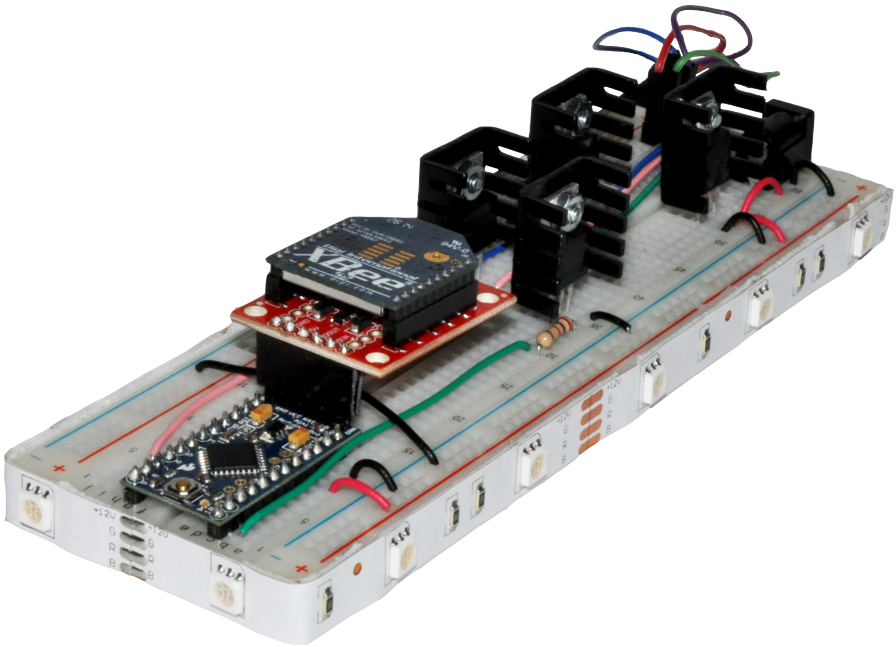

Figure 5.3: Prototype RGB LED output device

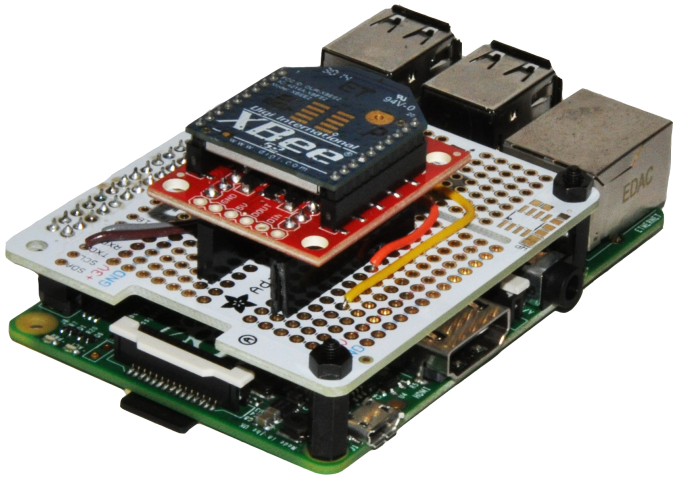

Figure 5.4: Prototype bridge device

5.6 Haar: Bridge

The final piece for the complete implementation is the bridge device. Like the end devices, the hardware for this had to first be built. When sourcing the Raspberry Pi devices for this component an additional component, the RPi HAT prototype board was found. Rather than connecting the RPi to the XBee chip with simple wires, this prototype board was used to build a more professional prototype. The RPi HAT required additional soldering, namely the headers to connect it to the RPi, all circuit wires and the XBee Explorer pins. It should also be noted that the Raspberry Pi model B+ (RPiB+) was procured rather than the model 2 and this was down to its cheaper cost. The RPiB+ is less powerful with a single-core 700MHz processor and 512MB RAM, but is still expected to be sufficiently powerful. The completed hardware can be seen in Figure 5.4.

As planned, the RPi-maintained distribution of Linux was installed on the RPiB+. A small Node.js application was developed. It made use of the same JavaScript packages which have been implemented in the Haar Engine and Dashboard. One additional role which the application played was to expand on the Compressed JSON Object data format described in the previous section so it could be submitted to Haar API.

Chapter 6

Results and Evaluation

Chapter 7

Conclusions

Chapter 8

Future Work

Bibliography

Divyakant Agrawal, Sudipto Das, and Amr El Abbadi. Big data and cloud computing: Current state and future opportunities. In *Proceedings of the 14th International Conference on Extending Database Technology*, EDBT/ICDT '11, pages 530–533, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0528-0. doi: 10.1145/1951365.1951432. URL http://doi.acm.org/10.1145/1951365.1951432.

Amazon Web Services, Inc. Announcing aws iot, now in beta. Seattle, WA: Amazon Web Services [Online], Oct 2015. URL https://aws.amazon.com/about-aws/whats-new/2015/10/announcing-aws-iot-now-in-beta/. [Accessed 10 October 2015].

S. Aust, R.V. Prasad, and I.G.M.M. Niemegeers. Ieee 802.11ah: Advantages in standards and further challenges for sub 1 ghz wi-fi. In *Communications (ICC), 2012 IEEE International Conference on*, pages 6885–6889, June 2012. doi: 10.1109/ICC.2012.6364903.

Babel. A detailed overview of ecmascript 2015 features. Babel [Online], Apr 2016. URL https://babeljs.io/docs/learn-es2015/. [Accessed 6 April 2016].

Andrew Banks and Rahul Gupta. Mqtt version 3.1.1. OASIS Standard, Oct 2014. URL http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.pdf.

Flavio Bonomi, Rodolfo Milito, Jiang Zhu, and Sateesh Addepalli. Fog computing and its role in the internet of things. In *Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing*, MCC '12, pages 13–16, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1519-7. doi: 10.1145/2342509.2342513. URL http://doi.acm.org/10.1145/2342509.2342513.

Carriots. What is carriots. Madrid: Carriots [Online], November 2015. URL https://www.carriots.com/what-is-carriots. [Accessed 12 November 2015].

Case C-362/14. Maximillian Schrems v Data Protection Commissioner. OJ C351, Oct 2015. URL http://curia.europa.eu/juris/document/document.jsf?text=&docid=172254&pageIndex=0&doclang=EN&mode=req&dir=&occ=first&part=1&cid=199525.

Kuor-Hsin Chang. Bluetooth: a viable solution for iot? [industry perspectives]. *Wireless Communications, IEEE*, 21(6):6–7, December 2014. ISSN 1536-1284. doi: 10.1109/MWC.2014.7000963.

Digi International Inc. Sending data through an 802.15.4 network latency timing. Digi International [Online], Apr 2016. URL http://knowledge.digi.com/articles/Knowledge_Base_Article/Sending-data-through-an-802-15-4-network-latency-timing. [Accessed 9 April 2016].

Robert Faludi. *Building Wireless Sensor Networks: With ZigBee, XBee, Arduino, and Processing*. O'Reilly Media, Inc., 1st edition, 2010. ISBN 0596807732, 9780596807733.

I. Fette and A. Melnikov. The WebSocket Protocol. RFC 6455 (Proposed Standard), December 2011. URL `http://www.ietf.org/rfc/rfc6455.txt`.

R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616 (Draft Standard), June 1999. URL `http://www.ietf.org/rfc/rfc2616.txt`. Obsoleted by RFCs 7230, 7231, 7232, 7233, 7234, 7235, updated by RFCs 2817, 5785, 6266, 6585.

Roy Thomas Fielding. Architectural styles and the design of network-based software architectures. Irvine, CA: University of California, 2000. URL `https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm`.

Github, Inc. Choose a license. San Francisco, CA: Github, Oct 2015. URL `http://choosealicense.com/`. [Accessed 14 October 2015].

Jonathan W. Hui and David E. Culler. Ip is dead, long live ip for wireless sensor networks. In *Proceedings of the 6th ACM Conference on Embedded Network Sensor Systems*, SenSys '08, pages 15–28, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-990-6. doi: 10.1145/1460412.1460415. URL `http://doi.acm.org/10.1145/1460412.1460415`.

Dogan Ibrahim. Advanced pic microcontroller projects in c. Oxford: Newnes, 2011. URL `http://www.myilibrary.com?ID=127324`. [Accessed 1 December 2015].

Link Labs. M2m & iot technologies explained. Annapolis, MD: Link Labs, 2015.

Kalle Lyytinen and Youngjin Yoo. Introduction. *Commun. ACM*, 45(12):62–65, December 2002. ISSN 0001-0782. doi: 10.1145/585597.585616. URL `http://doi.acm.org/10.1145/585597.585616`.

L. Masinter. Hyper Text Coffee Pot Control Protocol (HTCPCP/1.0). RFC 2324 (Informational), April 1998. URL `http://www.ietf.org/rfc/rfc2324.txt`. Updated by RFC 7168.

Friedemann Mattern and Christian Floerkemeier. From active data management to event-based systems and more. chapter From the Internet of Computers to the Internet of Things, pages 242–259. Springer-Verlag, Berlin, Heidelberg, 2010. ISBN 3-642-17225-3, 978-3-642-17225-0.

K. Mikhaylov and J. Tervonen. Optimization of microcontroller hardware parameters for wireless sensor network node power consumption and lifetime improvement. In *Ultra Modern Telecommunications and Control Systems and Workshops (ICUMT), 2010 International Congress on*, pages 1150–1156, Oct 2010. doi: 10.1109/ICUMT.2010.5676525.

Julien Mineraud and Sasu Tarkoma. Toward interoperability for the internet of things with meta-hubs. Technical Report arXiv:1511.08063v1, University of Helsinki, Helsinki, Finland, nov 2015. URL `http://arxiv.org/pdf/1511.08063v1.pdf`.

Luca Mottola and Gian Pietro Picco. Programming wireless sensor networks: Fundamental concepts and state of the art. *ACM Comput. Surv.*, 43(3):19:1–19:51, April 2011. ISSN 0360-0300. doi: 10.1145/1922649.1922656. URL `http://doi.acm.org/10.1145/1922649.1922656`.

Mozilla Developer Network. Concurrency model and event loop. Mozilla Developer Network [Online], Apr 2016. URL `https://developer.mozilla.org/en/docs/Web/JavaScript/EventLoop`. [Accessed 6 April 2016].

J. Postel. User Datagram Protocol. RFC 768 (INTERNET STANDARD), August 1980. URL http://www.ietf.org/rfc/rfc768.txt.

Z. Shelby, K. Hartke, and C. Bormann. The Constrained Application Protocol (CoAP). RFC 7252 (Proposed Standard), June 2014. URL http://www.ietf.org/rfc/rfc7252.txt.

P. Srisuresh and M. Holdrege. IP Network Address Translator (NAT) Terminology and Considerations. RFC 2663 (Informational), August 1999. URL http://www.ietf.org/rfc/rfc2663.txt.

The Department for Business Innovation & Skills. Smart cities background paper. *London: Department for Business Innovation & Skills Publications*, Oct 2013. URL https://www.gov.uk/government/uploads/system/uploads/attachment_data/file/246019/bis-13-1209-smart-cities-background-paper-digital.pdf.

The Department of Energy & Climate Change. Smart meters: a guide. *London: Department of Energy & Climate Change*, Oct 2013. URL https://www.gov.uk/guidance/smart-meters-how-they-work. [Accessed 29 November 2015].

The Department of Energy and Climate Change. Smart meters: a guide. London: Department of Energy and Climate Change [Online], Oct 2009. URL https://www.gov.uk/guidance/smart-meters-how-they-work.

The Department of Energy and Climate Change. Third annual report on the roll-out of smart meters. London: Department of Energy and Climate Change, Dec 2014a.

The Department of Energy and Climate Change. Communications hub technical specifications. London: Department of Energy and Climate Change, Nov 2014b.

The Department of Energy and Climate Change. Smart metering equipment technical specifications. London: Department of Energy and Climate Change, Nov 2014c.

The Department of Energy and Climate Change. Consultation on the timing of the review of the data access and privacy framework. London: Department of Energy and Climate Change [Online], Mar 2015. URL https://www.gov.uk/government/consultations/consultation-on-the-timing-of-the-review-of-the-data-access-and-privacy-framework.

The Energy & Utilities Alliance. Smart metering device assurance scheme operator services. Kenilworth: Energy & Utilities Alliance [Online], Sep 2014. URL http://www.eua.org.uk/sites/default/files/SMDA%20Scheme%20Operator%20RFP%20v1.2%20(Final).pdf.

The European Commission. Freight transport logistics action plan. *Luxembourg: Publications Office of the European Union, COM/2007/0607*, Oct 2007. URL http://www.ipex.eu/IPEXL-WEB/dossier/dossier.do?code=COM&year=2007&number=0607.

The European Commission. Action plan for the deployment of internet protocol version 6 (ipv6) in europe. *Luxembourg: Publications Office of the European Union, COM/2008/0313*, May 2008. URL http://www.ipex.eu/IPEXL-WEB/dossier/document/COM20080313FIN.do.

The European Commission. Internet of things — an action plan for europe. *Luxembourg: Publications Office of the European Union, COM/2009/0278*, June 2009. URL http://eur-lex.europa.eu/legal-content/EN/TXT/?uri=celex:52009DC0278.

The Telegraph. Energy suppliers to install smart meters in all households by 2020. Dec 2009. URL `http://www.telegraph.co.uk/news/earth/energy/6708822/Energy-suppliers-to-install-smart-meters-in-all-households-by-2020.html`. [Accessed 5 December 2015].

Lee Thomas and Prof. Nick Jenkins. Smart metering for the uk. HubNet [Online], Jun 2012. URL `http://www.hubnet.org.uk/position_papers`.

ThoughtWorks, Inc. Eliminate blind spots so you can build and deliver software more rapidly. Chicago, IL: ThoughtWorks [Online], Apr 2015. URL `https://www.thoughtworks.com/continuous-integration`. [Accessed 5 April 2016].

J. Wei. How wearables intersect with the cloud and the internet of things : Considerations for the developers of wearables. *Consumer Electronics Magazine, IEEE*, 3(3):53–56, July 2014. ISSN 2162-2248. doi: 10.1109/MCE.2014.2317895.

Mark Weiser. The computer for the 21st century. *SIGMOBILE Mob. Comput. Commun. Rev.*, 3(3):3–11, July 1999. ISSN 1559-1662. doi: 10.1145/329124.329126. URL `http://doi.acm.org/10.1145/329124.329126`.