UNIVERSITY OF SOUTHAMPTON

FACULTY OF ENGINEERING AND PHYSICAL SCIENCES

A Modular and Open Software and Hardware Architecture for Internet of Things Sensor Networks

by

Mihaela Apetroaie-Cristea

Supervisor: Prof. Simon J. Cox and Dr. Steven J.J. Ossont

June 25, 2020

UNIVERSITY OF SOUTHAMPTON

ABSTRACT

FACULTY OF ENGINEERING AND PHYSICAL SCIENCES

A Modular and Open Software and Hardware Architecture for Internet of Things Sensor Networks

by Mihaela Apetroaie-Cristea

We are experiencing a new industrial revolution, a cyber-physical systems revolution. The Internet of Things (IoT) is at the core of this development, and it is changing the way we perceive the world around us, impacting both business and technology.

Smart cars, smart thermostats, home automation hubs, smart lighting, and smart weather stations are common technologies met in most cities and households. The number of these devices is going to increase even more, and it is predicted to reach billions over the next few years. Although the IoT technology has reached maturity, we are still unprepared for the large number of devices predicted to reach the world in the near future. Managing high numbers of IoT devices requires stable infrastructures for deployment and management. It also requires creating sustainable infrastructures to minimise the impact on the environment.

In this thesis, we hypothesise that using flexible, open, modular, and reconfigurable hardware and software architectures at the base of smart city infrastructure can increase device longevity and minimise device management complexity at scale, while promoting sustainable development. The main contributions are: (1) identification of design requirements for building the next generation IoT device (2) reference architecture for flexible, modular IoT devices, (3) a novel, modular, open-source sensor board for building plug-and-play smart devices, allowing for complete removal/replacement of sensors and computing module, (4) a novel, modular, open-source software architecture, including: minimal Operating System (OS), over the air (OTA) updates, containerisation and remote device management, and (5) demonstration using a real-life application of environmental monitoring. The reference architecture presented in this thesis provides a robust, persistent, and reliable long-term solution for IoT deployements that addresses concerns regarding the negative impact of IoT on long term sustainable development.

Contents

List of Figures

List of Tables

Acknowledgements

This work is supervised by Prof. Simon J. Cox and Dr. Steven J.J. Ossont. Their guidance and support have been extremely valuable for the work presented in this thesis. I would like to thank my supervisors for being there for me every step of the journey, believing in me and getting out of their way to help me succeed.

Many thanks also to Dr. Mark Scott, Dr. Phillip Basford, Nana Okra Kwadwo Abankwa, Dr. Lasse Wollatz, Florentin Bullot, and Andrew J. Poulter for helpful feedback during our weekly group meetings and support over the last four years.

I would also like to thank my family and friends for the moral support and for believing in me.

I would like to dedicate this work to the memory of my grandfather, Dumitru Apetroaie, and Layla Apetroaie-Cristea.

Declaration of Authorship

I, Mihaela Apetroaie-Cristea , declare that the thesis entitled *A Modular and Open Software and Hardware Architecture for Internet of Things Sensor Networks* and the work presented in the thesis are both my own, and have been generated by me as the result of my own original research. I confirm that:

- this work was done wholly or mainly while in candidature for a research degree at this University;
- where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated;
- where I have consulted the published work of others, this is always clearly attributed;
- where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work;
- I have acknowledged all main sources of help;
- where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself;
- parts of this work have been published as: (please see Supporting publications section)

Signed:.....

Date:.....

Supporting Publications

Some of the work presented in this thesis has been also published in:

- **PLOS ONE**
  Apetroaie-Cristea, M., Ossont, S.J.J., Basford, P.J., Bulot, F.M., Scott, M., Cox, S.J.. An example architecture for managing distributed sensor networks at software and hardware levels. PLOS ONE Journal, 2020. *submitted, under review.*

- **MDPI Sensors Journal**
  Johnston, S.J., Basford, P.J., Bulot, F.M., Apetroaie-Cristea, M., Easton, N.H., Davenport, C., Foster, G.L., Loxham, M., Morris, A.K. and Cox, S.J. City Scale Particulate Matter Monitoring Using LoRaWAN Based Air Quality IoT Devices. MDPI Sensors. 19(1), 2019 (pp. 209 - 229).

- **Global Internet of Things Summit (GIoTS)**
  Johnston, S. J., Basford, P. J., Bulot F. M J., Apetroaie-Cristea M and Cox, S. J. IoT deployment for city scale air quality monitoring with Low Power Wide Area Networks, Global Internet of Things Summit (GIoTS). IEEE, 2018 (pp. 1-6).

- **IEEE World Forum on Internet of Things (WF-IoT)**
  Johnston, S.J., Basford, P.J., Bulot, M.J., Easton, N.H.C., Foster, G.L., Loxham, M., Apetroaie-Cristea, M, Morris, A.K.R, Cox, S.J. Testing Smart City environmental monitoring technology using small scale temporary cities. World Forum on Internet of Things (WF-IoT). IEEE, 2019 (pp. 578-583)

- **Nature Scientific Report**
  Bulot, F.M., Johnston, S.J., Basford, P.J., Easton, N.H., Apetroaie-Cristea, M., Foster, G.L., Morris, A.K., Cox, S.J. and Loxham, M. Long-term field comparison of multiple low-cost particulate matter sensors in an outdoor urban environment. Scientific reports 9. Nature, 7497 (2019), doi:10.1038/s41598-019-43716-3

- **ISC High Performance 2016**
  23rd of July 2016
  Poster + Presentation

- **Book Chapter**
  Johnston, S. J., Apetroaie-Cristea, M., Scott, M., and Cox, S. J. Applied Internet of Things. In Buyya, R. and Dastjerdi, A. V., editors, Internet of Things Principles and Paradigms, Chapter 15. Elsevier, 2016 (pp. 277-297)

- **Ubicomp 2016**
  Poster + UbiComp 2016 Adjunct Proceedings + ACM Digital Library
  Apetroaie-Cristea M, Johnston, S. J., Scott, M., and Cox, S. J. (2016). Low-cost indoor positioning system based on commodity hardware. ACM, 2016 (pp. 13-16).

- **World Forum on Internet of Things 2016**
  Johnston, S., Apetroaie-Cristea, M., Scott, M., Cox, S. (2016). Applicability of commodity, low cost, single board computers for Internet of Things devices. In World Forum on Internet of Things (WF-IoT). IEEE, 2016 (pp. 1-6).

Chapter 1

Introduction

The Internet of Things (IoT) introduces the concept of smart objects or things that are interconnected, uniquely addressable, able to cooperate with each other, and perform tasks that would benefit society. A dynamic area that captivates the attention of both industry and academia, some consider IoT the second technological revolution after the invention of the personal computer [35] and the fourth industrial revolution [128].

The term Internet of Things is relatively new [15], but its core areas and topics are met in literature from the early times of the computer. One of the main sub-fields of IoT - ubiquitous computing - is met in research literature and projects from the late 1980s. One example of such a project is Biosphere II [40], built over three years starting in 1987, an ecological life-support system with the scope of understanding the interaction between Earth's surface and the atmosphere. Computer systems were embedded in the experimental research area for administrative support, communication, and monitoring of ambient conditions. Weiser [154], who defined the term ubiquitous computing, was considering that the computers should be embedded everywhere throughout the physical environment, to help humans in a pervasive way and not remain the main focus of their attention. He also identified the main issues that had to be addressed for his idea to become a reality. These issues are the cost of computers, power consumption, the need for a wireless network to accommodate hundreds of high-speed devices, the need for a new network protocol, privacy issues. Nowadays, computers have decreasing power consumption and increasing computational power, smaller sizes, lower prices, the IPv6 (Internet Protocol version 6) can accommodate $5x10^{28}$ devices for every person in the world, while new security and privacy solutions are under continuous development.

All of these technological advances are leading to accelerated development of IoT technologies, which are becoming a ubiquitous part of our environment, like Weiser [154] predicted. Gartner [55] predicts that the number of Internet of Things (IoT) devices is going to increase to reach tens of billions in the near future. With this number of devices, IoT will have a strong influence on society and will change the way people perceive the

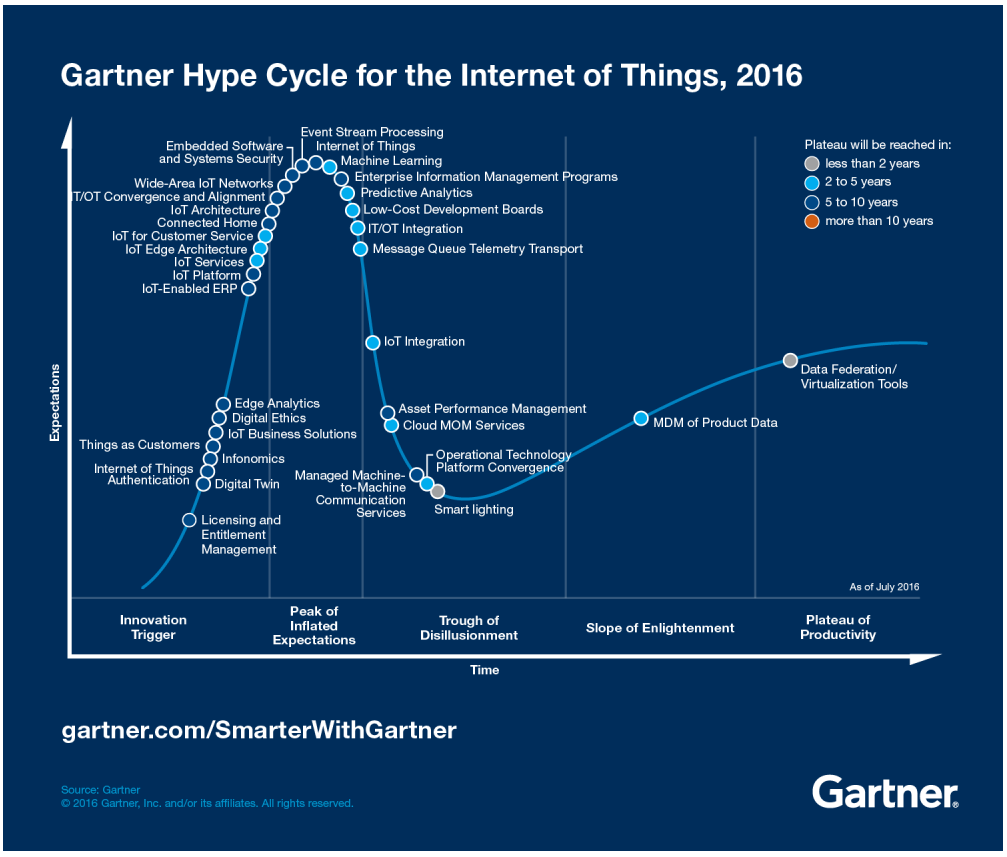


Figure 1.1: Hype Cycle for Internet of Things [148].

objects around them. Smart cities, smart buildings, smart homes, smart cars, smart manufacturing, smart transportation, and smart business management are just a few examples of technologies impacting our society.

According to Velosa [148], the IoT already reached maturity and it is situated in the productivity area on the hype cycle, Figure 1.1. Cities such as Amsterdam, London, San Francisco, Oslo, and Barcelona are already building smart infrastructures that benefit from these technological advances. Amsterdam has smart lighting, smart playgrounds, encourages circular businesses and energy-saving [105]. London is planning on building a smart infrastructure with an emphasis on transportation and resource-saving [56]. San Francisco is famous for the smart parking infrastructure [80]. Oslo benefits from smart parking and street lightning [32]. Barcelona has 500 km of fibre optic network, free public WiFi coverage, and smart lightning [20]. All of these cities are smart cities because they use technology to improve the living conditions, such as sensors and actuators, to collect data that is used to improve aspects of the daily routines, save resources and improve costs.

Hardware capabilities are improving, while the price and physical size are significantly decreasing. For example, the introduction of the Raspberry Pi Zero at the beginning of 2016 for $5 that is half the size of the first Raspberry Pi model of 2012 and has better

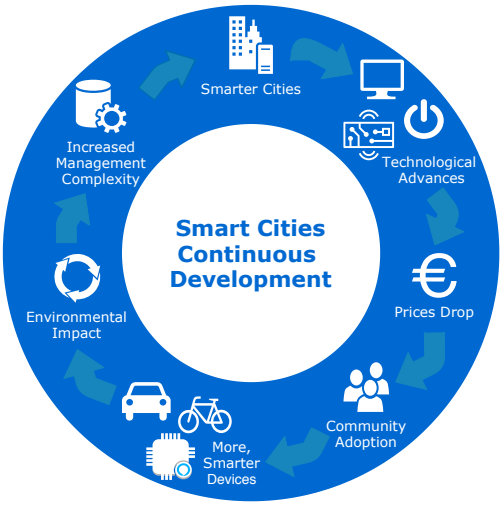


Figure 1.2: The continuous development of technology and its impact on smart cities

CPU clock and RAM. Raspberry Pi 4 Model B was introduced on the market this year (2019) and has four cores, more than double the CPU clock frequency and up to eight times more RAM than the Raspberry Pi 1 Model A, which was introduced on the market in 2012 for the same price of $35. This evolution is enabling significant technological progress; it leads to new research projects and development to be made in IoT.

With billions of IoT devices predicted to reach consumers over the next few years, the world as we perceive it will change drastically. An increased number of IoT devices can lead to increased costs, complexity, landfill, and clutter. Hence, finding ways of sustainably managing these devices at large scale is essential for the future of IoT. [56] revealed in the 2050 smart city plans for London that re-using existing infrastructures and developing infrastructures that serve multiple purposes is essential for smart cities as it reduces costs, pollution, and saves resources. The conjecture for this work is that using modular hardware and software, we are able to design an IoT device that is reconfigurable after deployment, and it can adapt to its environment. The next chapters explore the literature, the requirements for building such a device, and present a device architecture designed and built using these requirements, which is validated through an environmental monitoring use case.

1.1 Thesis structure

This thesis has six chapters. This is the first chapter, the introduction. It presents the structure of this thesis and the scope of this work.

Chapter 2, Internet of Things in our lives, presents the current status of the Internet of Things and general topics of interest in the area, as an extension to the work presented

in [69]. This chapter gives a better understanding of the current state of the technology, gaps in IoT device architectures and infrastructures, and the research questions and objectives addressed in this work.

Chapter 3 presents an indoor positioning system that has been developed based on low-cost commodity hardware as part of this research. The Chapter consists of a conference paper that was presented at the UbiComp2016 [13]; it also presents data that has been collected to assess the performance of the system further. The work presented in this chapter represents the first stage of this research, which helped with understanding the main issues in deploying and managing IoT devices at scale. The main scope of this chapter is to investigate the requirements for an open IoT device architecture that can adapt to technological developments and allows re-use of infrastructure.

Chapter 4 presents an environmental monitoring example that was designed and built for an outreach project in order to explore IoT architectures and answer the research questions in depth.

Chapter 5 presents a generic IoT device. The main component of the device is a generic plug and play board, developed using the HAT characteristics [90], the PiSEB [12]. This Chapter also presents the hardware implementation of a working environmental monitoring device example to validate the proposed solution.

Chapter 6 presents the generic, plug, and play software architecture that was developed for this work to fulfill the role of a working reference implementation and a skeleton for developing new applications. Our implementation is targeted to our device in particular, but it can be easily adapted to work in entirely different environments and devices. This Chapter also presents the software implementation of a working environmental monitoring device example to validate the proposed solution.

The last two chapters, Chapters 7 and Chapter 8, present the conclusions and future work for this research.

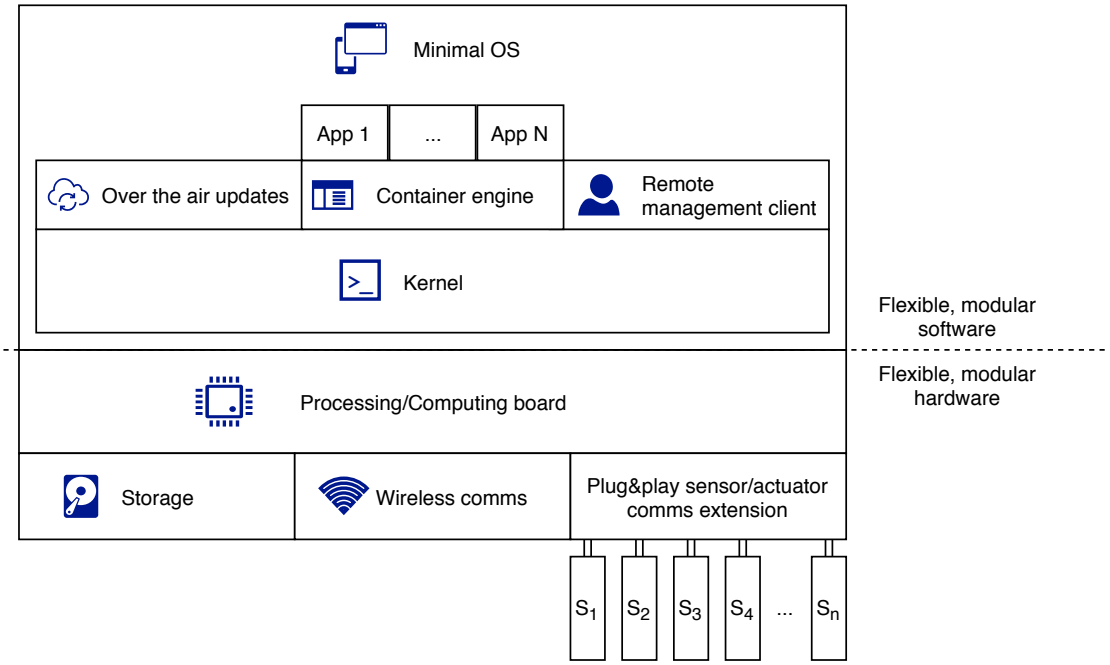


Figure 1.3: Proposed hardware and software modular architecture, where S represents a sensor

Chapter 2

The Internet of Things in our lives

The Internet of Things (IoT) is evolving at a high rate, from smart cities [96] to home automation [119], the idea behind IoT is the pervasive presence around us of various sensors and devices that send and receive data, are able to cooperate with each other and perform tasks that would lead to automation, increased efficiency and better management of processes.

IoT highly impacts our living, contributing to increased quality of life [54]. Cities around the world are adopting the IoT emerging technologies to build infrastructures that help saving resources, improve costs, and benefit the health [96, 105, 32]. All of these are possible due to great hardware and software advances [135], which allow for low-cost sensors and devices to be embedded into our everyday lives, taking us closer to achieving what Weiser [153] envisaged.

Sensor networks are at the core of IoT, street lighting, and air quality monitoring are examples that prove their significance to the area studied in this work [16].

With the predicted increase of IoT devices that are going to be deployed around the world in the near future, more research is required for achieving device scalability. Finding solutions to ease device interoperability, developing operating systems that facilitate operations on resource-restricted devices in out-of-reach constrained environments is essential at this stage. Over the air, secure updates are also an essential part of IoT, ensuring devices' reproducibility, ease of management, and deployment.

In this chapter, we discuss various aspects of state of the art in IoT and consider both the technologies involved and the challenges being tackled.

Figure 2.1: The United Nations Global Goals adopted in 2015 [54]

2.1 Towards ubiquitous computing

2.1.1 Sustainability

Every year, nearly 50 million tonnes of e-waste are produced globally, out of which approximately 40 million tones end up in landfill, and it is expected that by 2021 the annual e-waste volume will surpass 52 million tones [115]. A significant part of this e-waste is generated by IoT devices. IoT has a positive impact on sustainability, showing a strong correlation with 11 of the 17 Sustainable Development Goals (SDGs) [111]. IoT negatively impacts on responsible consumption and production, the 12^{th} SDG [57]. The UN has proposed tackling this issue by creating a circular economy, ensuring we get the most out of the deployed devices. For example, London is planning on re-using existing devices and infrastructure for providing new services [56] and to create an infrastructure that serves more purposes to minimise costs, save resources and reduce pollution for building new infrastructure. The main focus has been on developing new IoT services, and there has not been enough research done in the area regarding these devices after they become out of date or break.

Projects such as Freecycle [140] are an excellent initiative for minimising landfill. There is still a need to develop a very efficient method of re-using hardware that will accommodate the forecast of increase in IoT devices.

2.1.2 Internet of Things to address global goals

In September 2015, the United Nations (UN) adopted 17 Global Goals. The goals are illustrated in Figure 2.1. IoT is a technology that can be used to address these goals, and some of the work in the domain shows that the process has already started.

Internet of Things applications in the health domain, such as monitoring elderly people [45] or health monitoring devices that can be used from home to give a better insight of patients health at the comfort of their homes [121] are just a few systems that are very popular nowadays and are helping towards reaching the UN Global Goals.

IoT is contributing to advances in hardware and software. These lead to developing better education means, providing educators and students with better educational platforms [79] and more resources [42].

Low-cost sensors can be used in a wide range of areas, such as water, weather, and air quality monitoring [8, 68], to give us information that can help increase the quality of life. Smart monitoring devices, such as smart meters, smart sockets, and light sensors, can decrease the usage of water, energy, and similar valuable resources. Solar energy storage and sharing are also examples of applications that help decrease resource waste. Smart irrigation [52] and smart lighting are also reducing waste.

Smart cities infrastructures [56, 96] give birth to sustainable cities [105] with responsible goods consumption and circular business infrastructures, where recycling and resource sharing are reducing waste and littering.

Devices tracking our goods [137] and better software security solutions [117] are also contributing towards decreasing criminal activities.

2.1.3 Internet of Things in our cities

Cities across the world are embracing the IoT technologies. This section presents some of the most popular smart city initiatives.

Singapore

Singapore started the Smart Nations program in 2014 [107]. The idea behind the program is to create a distributed network of sensors across the city that will be used for different applications, which do not have to be known before the deployment of the sensors. Multiple cameras have been deployed to monitor littering, bus travelers' smartphones have been used to monitor street maintenance level, and motion sensors have been used to monitor the elderly [96]. Monitoring is also done in the areas of public transportation, air quality, and energy consumption. The data portal is open to developers and has a public API [66].

There is a plan for the future to collect the data from sensors centrally to store information on building architecture and weather conditions.

The main concern of this project is the protection of privacy and security. For example, under Singapore law, any decision to use the data from the sensors for law enforcement

does not need court approval. This can easily lead to privacy invasion. Creating ethical frameworks to solve these problems is still under development.

Amsterdam

In 2016 Amsterdam won Europe's Capital of Innovation award by the [50]. The plan [105] is to build data-driven city management. The projects that will be developed as part of Amsterdam Smart City are concentrated around:

- Infrastructure and technology - using IoT for crowd management and smart, dynamic pitch lighting on the Amsterdam ArenA. Green playground, where the whole family can enjoy fitness equipment that illuminates when someone exercises that also has illuminated benches in the evenings and augmented reality characters for entertaining.
- Energy, water, and waste - with projects such as storage and trade of solar surplus energy through home batteries, sharing energy between households to obtain energy-neutral neighbourhoods.
- Mobility - for example, bicycle kits that would make bikes connect with each other, collect data, and act as an anti-theft system.
- Circular city - projects such as encouraging local companies to use organic waste to develop new products to reduce sourcing, transportation, and waste management costs or building a circular data platform with multiple flows.
- Governance and Education - for example, the project City Protocol that aims to achieve city interoperability.
- Citizens and living - for example, citizens science projects, such as measuring weather conditions and air quality.

Although most of these projects are still under development, the Amsterdam smart city project is an example of a complex and highly data-driven city architecture that will become a reality in the near future.

Barcelona

Barcelona is one of the exemplary smart cities in Europe. The most outstanding technological advances in Barcelona are 500 km of fibre optic network, 670 WiFi hotspots at a maximum distance from each other of 100 m that provide free city-wide WiFi, 19,500 smart meters that monitor and optimize energy consumption, smart parking, digital bus stops with updates on bus location, USB charging stations, free WiFi and tools to help riders learn more about the city [20].

The Barcelona Lightning Masterplan that uses technology to enhance the efficiency and utility of city lampposts with more than 1100 lamp posts that have LED and brighten when pedestrians are in close proximity. They also are part of the WiFi network that provides free WiFi across the city and have sensors that collect data on air quality and monitor weather conditions for determining the amount of irrigation required for parks.

These create Barcelona's sensor network that is relayed through the Sentilo platform [122], which was created specifically for Barcelona, but it was made open-source in an effort to encourage other cities to join Barcelona's example of creating a smart city.

London

London is another example of a smart city. The open data platform London Datastore [41] has a public API and stores data on jobs and economy, transport, environment, community safety, housing, health, and more. London adopted transportation innovation technologies such as number plate recognition for congestion charge, WiFi on the Underground, and uses new technologies to reuse waste heat. There are some citizen science projects such as air quality monitoring [43] and weather monitoring logged to the [23]. London Air [151] is a project that aims to inform London citizens on the air quality within the city.

Breathe Heathrow [112] is another initiative of monitoring air quality and noise. Sensors are placed in people's gardens in an attempt to inform the population about the potential dangers they are exposed to around Heathrow airport.

London is a centre for technological advances, and multiple start-ups and companies are showcasing their technology.

London has committed to a development plan for 2050, which aims to improve London's infrastructure, with a focus on transportation, creating a circular economy, improved green infrastructure, energy and water, and digital connectivity [56].

San Francisco

San Francisco is renowned for the number of start-ups that are born in the city, which is why it is at the core of technological advances. From all these technologies, San Francisco is famous for its smart parking initiative [80].

Oslo

Oslo is another city that has plans for becoming a smart city. At the moment, they have parking monitoring and smart street lighting. They also have initiatives to assist sick and elderly patients and want to reduce greenhouse gas emissions by 50% [32].

2.2 Internet of Things hardware technologies

The IoT development is driven by significant increases in hardware capabilities and cost reductions. This section briefly summarises the most important hardware technologies that stay at the base of IoT and have been standing as technological pillars for the IoT development during the period of this research (2015 - 2019), with emphasis on the main applications and sensors that form the base of smart cities.

2.2.1 Single Board Computers

One of the most common IoT development boards is the Raspberry Pi - a series of credit-card size, single-board computers. The first Raspberry Pi was introduced on the market in 2012 at a target price of $35 and features the Broadcom BCM2835 system on a chip (SoC), a 700 MHz single-core ARM1176JZF-S Central Processing Unit (CPU) and Broadcom VideoCore IV @ 250 MHz Graphics Processing Unit (GPU). The latest version is Raspberry Pi 4 Model B that was released on the market in July 2019. It features the Broadcom BCM2711 SoC, a 1.5 GHz quad-core A72 64-bit CPU, a Broadcom VideoCore VI @ 500 MHz GPU and built-in WiFi and Bluetooth capabilities [135].

Raspberry Pi is not the only single-board computer that recently has emerged on the market. Companies such as Intel have released low-price single board computers, but there are also start-ups producing them, such as Pine64. A more detailed discussion about these boards, microcontrollers, and other releases in the area and their specifications can be found in the book chapter written during this research [69].

Low-cost single board computers and sensors have enabled high performance computing [42] and other projects [13, 3] that otherwise would be expensive and out of reach for most consumers.

2.2.2 Sensors

Sensor costs are significantly decreasing; for example, the price of a WiFi or Bluetooth dongle can be as low as $1. Gyroscopes, accelerometers, magnetometers, and GPS are widely available, low-cost, and embedded onto the majority of personal devices on the market. This enables new research projects, such as indoor positioning [77].

Extensive research is also done to achieve higher performance, for example, in the detection of gases [19].

The hardware developments have a significant impact on the Internet of Things enabling new research perspectives and advances.

Analysing the smart cities' initiatives and applications, it can be noticed that there are a common trend and a general agreement in the nature of the future smart cities applications.

Common sensors in a Smart City

One of the most common applications met in the future smart city is weather monitoring. Either citizen science projects or academia projects [68], weather stations seem to reach state of the art with highly accurate low-cost sensors and community platforms [23] that collect and analyse the data.

Smart transportation is another common theme among smart cities with live updates on bus locations, automatic sensing of passenger distribution in trains and WiFi hotspots in public transportation. Automatic recognition of plate numbers for congestion charges and counting cars on the motorway are some of the most common applications. For these applications, technologies such as WiFi, Bluetooth, GPS, cameras, and ultrasound are needed.

Smart parking is a widespread application. For example, the San Francisco parking system is based on a device that is drilled into the parking lot and contains a magnetometer, battery, processor, and radio communication for transmitting data [80].

Sustainability is another significant area that has applicability within the IoT area. Smart lighting, smart water meters, smart waste management are some of the applications under development in the area, and sensors such as motion, light, cameras, and long-range, low power radio transmitters are essential sensors needed for these applications.

Noise detection is also a popular application for smart cities. For example, monitoring the noise levels within the city and making open-source maps with noise pollution. ShotSpotter is a company that uses microphones deployed across cities to detect gunshots [34]. For these applications, sensors such as microphones are necessary.

Monitors for the structural integrity of buildings is another popular application [129]. Sensors such as accelerometers are necessary for this type of application.

Global warming [78] and increased pollution within the cities [97] are big concerns in our society. Air quality became one of the main concerns within the big cities due to health implications [30], which is why air quality monitoring devices are going to be embedded within all the future smart cities [105]. The most commonly monitored gases are Nitrogen Dioxide and Sulphur Dioxide. Particulate matter is also commonly monitored [130].

Gas Sensors

High-end air quality monitoring sensors are expensive [44]. With prices greater than £10,000, having a large number of such devices is infeasible, while having just a few of them cannot give a good description of the environment. This is why it is important to

have good and reliable low-cost air monitoring devices that would give sensitive results and would complement the few high-end devices that are already deployed in most cities.

The current generation of low-cost air quality monitoring devices (under $20) do not give reliable results, they do not give absolute readings, their accuracy is lower than the maximum healthy amount defined by Defra Directive 2008/50/E [29], and their range is not relevant to these values being sensitive to very high volumes of gases [83, 76], which is why they are not considered in this work.

According to Defra Directive 2008/50/EC [29], the threshold for Sulphur Dioxide is $500\mu g/m^3$, which means approximately 0.188 ppm, for three consecutive hours over $100km^2$ area and the threshold for Nitrogen Dioxide is $400\mu g/m^3$, which means approximately 0.21 ppm, for three consecutive hours over $100km^2$ area. According to [157] the Particulate Matter 2.5 (PM2.5) should not exceed $10\mu g/m^3$ as annual mean and $25\mu g/m^3$ as 24-hour mean. The PM10 should not exceed $20\mu g/m^3$ as annual mean and $50\mu g/m^3$ as 24-hour mean.

Hence, the air quality monitoring sensors should be sensitive enough to sense values within these thresholds. Table 2.1 shows some of the low-cost sensors with sensitivity and range that allow to detect air pollutants within the official healthy limits and could reliably be used as a reference for determining the air quality. The accuracy of these sensors is not high enough to generate data showing the exact pollution levels, but they can be used to indicate when the pollution levels change. Considering the current state of hardware and software evolution, it is expected that similarly accurate sensors will be available for under $20 in the near future.

2.3 Enabling technologies

2.3.1 Open Sensors Platforms

The IoT sensor data can be sent for visualization and sharing with the public to online platforms. This subsection exemplifies the most popular open-source platforms for data storage and visualization.

- Sentilo [122] is an open-source data management platform initially built for the smart sensors in Barcelona. Other cities across different countries have adopted the platform. For example, in Winchester, Hampshire, weather data and road temperature data is logged and can be visualised with Sentilo.
- The Things Network [142] is a community network that enables low-power devices to use long-range, low power radio frequency protocol, LoRaWAN, and Bluetooth for short-range.

Table 2.1: Gas sensors meeting the requirements for this project, containing costs and performance metrics; the cross-sensitivity is a measure of how much the values read by the sensor are influenced by other substances than the measured ones. Please note that the costs are valid at the time of this writing (2019) and might change in time

Distributor	Gas	Model	Resolution	Range	Half Life	Price ($)	Cross-sensitivity
Plantower	PM	PMS1003	1 ppb	0 - 500 μg/m3		25	high
Euro-gasman	NO2	4-NO2-20	0.1 ppm	0 -20 ppm	2 yrs	45	medium
Alphasense	NO2	A4-series	0.1 ppb	0 - 20 ppm	2 yrs	52	medium
Alphasense	SO2	A4-series	15 ppb	0 - 50 ppm	2 yrs	52	medium
Alphasense	NO2	B4-series	0.1 ppb	0 - 20 ppm	2 yrs	55	medium
Alphasense	SO2	B4-series	5 ppb	0 - 100 ppm	2 yrs	55	medium
Euro-gasman	SO2	4-SO2-20	0.1 ppm	0 -20 ppm	2 yrs	57	medium
Sensortech	SO2	EC4-20-SO2	0.1 ppm	0 - 20 ppm	1 yrs	88	medium
Sensortech	NO2	EC4-20-NO2	0.1 ppm	0 - 20 ppm	1 yrs	103	medium
Alphasense	PM	OPC-N2	0.38 to 17 ppb	0.84% @ 10^6 part./L		277	high
Libelium	NO2	NO2-A43F	0.1 ppm	0 - 20 ppm	2 yrs	290	medium
Libelium	SO2	SO2-A4	0.1 ppm	0 - 20 ppm	2 yrs	290	medium

- Open Sensors [114] is another community platform that allows users to log, visualize, and share data.
- WOW Met Office [23] is an open-source platform that allows citizens to log data from their own weather station devices.
- Freeboard [2] is an open-source platform for real-time data visualization, benefiting from a drag and drop interface.
- Plot.ly [116] is an open-source visualization library and online chart creation tool.

2.3.2 Scalability

The Internet of Things area is evolving very fast, and with the high increase of IoT devices that are predicted to reach the consumers in the next years, it is imperative to consider

further expansions of a system before deploying. When the number of devices is large, scalability is problematic at different levels, including data transfer and networking, data processing and management, and service provisioning [101].

Many IoT systems can be represented as client-server architectures, where clients are the devices deployed in the field, and the server is responsible for data collection and device management and monitoring. Sometimes the server is also responsible for passing messages between devices and sending actions to actuators.

There are two types of scalability: scalability on the server-side and scalability on the client-side.

Scalability on the client-side deals with hardware and software at the device level, such as power consumption, hardware durability, commissioning, networking, hardware maintenance, and reliability. These are presented in the next sections as they are a significant part of this work.

As the number of devices in a system grows, the server can easily become a bottleneck. There are many ways to scale a server, and there are software solutions and commercial cloud services that offer easy to use ways to build robust, scalable servers. In IoT, the biggest scalability concerns are around big data, namely having to support a large volume of data, sometimes coming at a high velocity. In systems with many types of sensors and other data sources, the data also comes in a wide variety of ways [37].

This subsection presents some of the solutions that can help achieve scalability on the server-side.

Designing a server (or more) to scale for supporting a large number of clients at the same time is usually addressed by a load balancer and more replicas of the server software. We are not going into the details of scaling servers, as this is not the main focus of this work. We will instead describe some common data-related issues that can arise while developing IoT systems.

One issue is picking the right data storage mechanism. There are plenty of choices, from generic relational databases to no-SQL databases specialised for different types of data to simply using files on the filesystem. There is no one size fits all solution, and we will describe a few options.

- Relational databases are digital databases that store data based on the relational model defined by Codd [39]. The data is organised in a set of tables, from which the data can be accessed or organised without the need for reorganising the database tables. The relational databases are written in SQL (Structured Query Language), which is the standard language for managing relational databases. The most commonly met relational databases are: SQLite [70], MySQL [59], PostgreSQL [104], Oracle [102].

- Document databases are databases designed for storing, retrieving and managing semi-structured data. The most popular document databases are: MongoDB [38], CouchDB [9], RavenDB [84].
- Time Series Database (TSDB) is a software system designed for handling time series. Frameworks such as Influxdata[108], OpenTSDB [134], Prometheus [118] are using TSDBs to manage large amounts of data without losing granularity and provide scalability. TSDB is ideal for large amounts of time-series data when flat databases are not a viable option due to limited storage and processing capacity.

Another issue is data processing at scale. A widely used software library and starting point for addressing this is Hadoop [147].

Apache Hadoop is an open-source software library for distributed computing. It is a highly-available service able to scale from single servers to thousands of machines, offering local computation, storage, and failure detection at the application layer. It includes:

- Hadoop Common with common utilities supporting other Hadoop modules
- Hadoop Distributed File System (HDFS) that provides access to application data
- Hadoop YARN, a framework for job scheduling and cluster resource management
- Hadoop MapReduce, a YARN-based system for parallel processing of large data sets.

The software libraries and packages described above can help significantly in developing reliable IoT solutions. However, setting up and maintaining all of the required databases, data processing pipelines, and device management solutions takes a significant amount of time and effort. Commercial solutions that offer easy to use and quick to set up systems for reliably managing big data are available from companies such as Amazon, Google, and Microsoft. These cloud-based solutions give developers the chance to build IoT applications without managing complicated server infrastructure. Below are some IoT-focused cloud platforms.

- Microsoft Azure IoT Suite [53] is a platform-as-a-service (PaaS) built on top of Microsoft Azure that offers device re-provisioning, ownership transfer, secure device management and device end-of-life management for IoT.
- Microsoft IoT Central [138] is an end to end software-as-a-service (SaaS) platform built on top of Microsoft Azure IoT suite . It is a fully managed SaaS addressed to IoT developers without cloud expertise that allows quick deployment and management of IoT devices.

- Amazon Web Services IoT [60] is a managed cloud platform with a software stack that enables connecting IoT devices to Amazon Web Services. The platform offers software tools that allow for ease of integration, secure communication, and management of devices with Amazon Web Services.

The cloud solutions provide us with server infrastructure, data storage solutions, and analytics frameworks. Web services providers such as Microsoft Azure [156], Amazon Web Services [67], Google Cloud Platform [75] offer a wide range of analytics, computing, database, mobile, networking, storage, and web solutions.

The actual server application software is the glue between the IoT devices and the data storage and analytics. NodeRED [81] helps writing the server application, and also with on-device application software. NodeRED is a web-based service that runs on the platform of interest (i.e., Raspberry Pi). It presents the user with a graphical interface to control data flow between modules. NodeRED comes with an extensive library of complementary modules for hardware communication, interaction with social media, and visualisation frameworks. It runs on Linux distributions, Mac OS X, and Windows.

2.3.3 Networking

Technologies such as LoRa [17] and Thread [95] provide networking stacks for IoT devices in different scenarios, for large areas (e.g., city-wide coverage) and small areas (e.g., in a house or a building), respectively.

- LoRaWAN is a Low Power Wide Area Network (LPWAN) specification intended for battery-operated wireless devices in regional, national or global networks. The standard provides interoperability, bi-directional communication, mobility, and localisation services. It operates at data rates between 0.3kbps to 50kbps, managed for each device employing an adaptive data rate (ADR) scheme to maximise the battery life of the end-devices and overall network capacity. Security is ensured at the network, application, and physical layers.
- Thread is an IP-based wireless networking protocol launched by Thread Group, that relies on low-power radio protocol called IPv6 over Low Power Wireless Personal Area Networks (6LowPAN). Thread sends small amounts of data. Hence it has low power usage. It is also based on mesh networks that can scale to hundreds of devices without a single point of failure and involves "banking-class" encryption. As it is a networking standard, it works with high-layer standards, such as AllJoyn and IoTivity.
- Sigfox [159] is a Low Power Wide Area Network (LPWAN) where the devices send data through the SigFox network to a SigFox base station (gateway). The base

station then detects, demodulates, and reports the messages to the SigFox cloud across three channels, at least every 10 minutes

- ZigBee [73] is a wireless-based open, global standard used for personal-area networks targeted at low-power applications with infrequent data transmission needs. It operates on the 802.15.4 standard and enables wireless mesh networks with low-cost and low-power solutions. It has data transmission rates of up to 250kbit/s, up to 100m transmission range and it supports up to $65,000$ nodes. Millions of ZigBee-enabled products exist on the market today, including in smart homes, connected lighting, and the utility industry.

- Bluetooth Low Energy [58] is a low-range, low-power wireless technology designed to be used for monitoring and controlling applications. It operates in the 2.4 GHz Industrial Scientific Medical (ISM) band and defines 40 Radio Frequency (RF) channels with 2 MHz channel spacing. It uses single-hop communication and can be used for connecting large amounts of IoT devices.

2.3.4 Operating Systems and Unikernels

Internet of Things devices are generally resource-constrained and built to meet specific functions, unlike personal computers. Hence they can benefit from more specialised OS distributions. Linux is the most popular base OS for IoT. Linux [132] is an open-source, Unix-like and mostly POSIX-compliant computer OS.

This section presents some of the OSs considered for the projects presented in this work.

- Raspbian [135] is a Linux-based OS developed for Raspberry Pi devices. The Debian-based OS has four versions: Wheezy, Jessie, Stretch, and Buster. The OS is specifically optimised for Raspberry Pi hardware and has more than 35,000 packages.

- DietPi [27] is a light OS for Single Board Computer (SBC), mainly built around Raspberry Pi. Its size can be as low as 400 MB, it is based on Debian Jessie, and it contains tools such as backup, cleaner, lightweight ssh server, automatic filesystem expansion.

- OpenWRT [51] is a Linux-based Operating System specifically designed to run on routers and Access Points (AP).

- TinyOS [82] is an open-source Operating System (OS) and platform designed for sensor networks. It is a flexible, application-specific OS that supports concurrent programs with very low memory requirements (many applications fit within 16KB of memory, and the core OS is 400 bytes) and efficient, low-power operation. It was designed to run on a generalized architecture where a single CPU is shared between

application and protocol processing. It makes use of event-based programming and enables system-wide optimisation by providing coupling between hardware and software.

- Alpine [48] is a small-size Linux distribution - a minimal installation requires 130 MB storage for embedded systems. It has its package manager, apk, and OpenRC init system, script-driven set-ups. The OS comes with read-only file systems only, and any changes made within the filesystem is discarded at reboot unless committed to the system. The downside of this OS is that its package manager has a limited number of packages, and porting packages from other distributions is time-consuming. Furthermore, some packages require rebooting during installation.

- NixOS [46] is a GNU/Linux distribution, that runs on top the Nix package manager. The package manager can provide incremental OTA updates, which have the same functionality as OSTree [149].

- Yocto project [125] was developed based on the OpenEmbedded project and allows creating and modifying Linux distribution. The deployment can be done directly on the hardware or using SSH. The Yocto images are build using Bitbake. Bitbake is the task executor and scheduler used by the OpenEmbedded build system to build images. Bitbake also allows building multiple targets at the same time, where each target uses a different configuration. The image build can be done via the terminal or using the Yocto project web interface, Toaster. The project is popular, and it supports builds for a wide range of machines and software packages.

- Windows 10 IoT Core [53] is a a version of Windows 10 optimised for embedded devices. It has support for Raspberry Pi, Arrow DragonBoard 410c, and Minnow-Board MAX. However, it lacks the community support, like other Linux-based OSs have, and it has not reached maturity at the time of writing (2019).

Containerisation and sandboxing are becoming popular choices for IoT because they ease application deployment and scalability. Some of the most popular Operating Systems built with an emphasis on these technologies are:

- Ubuntu Snappy [61] is a small size OS designed for IoT devices. The main feature of this OS is that applications can be sent over the air in snap packages. A snap is a zip file that contains the application to be deployed and its dependencies with a description of how the application should be run and how to communicate with other software.

  The applications are sandboxed or containerised. This isolates them from the underlying system adding security to the system.

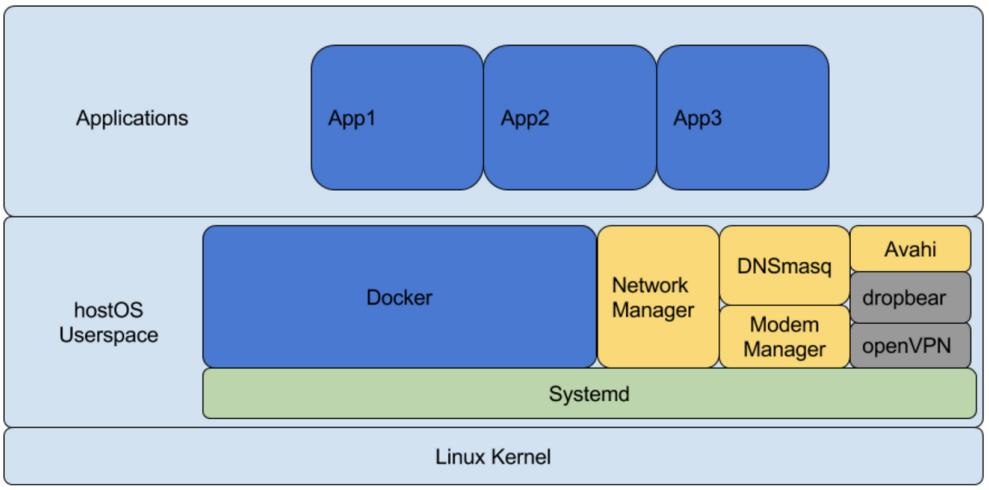


Figure 2.2: BalenaOS Architecture showing the main OS layer distribution [71]

The snaps are sent to the app store, where they become available for download on the IoT device. The apps are updated automatically in the background, and it is easy to roll back to a previous version in case of errors.

- Fedora Atomic [109] is a project between Fedora and Project Atomic that creates software frameworks with Fedora and Project Atomic for cloud servers.
- CoreOS [103] is an open-source lightweight Linux-based OS designed around containerisation and mainly for cloud based hosts and also supports a high number of physical machines. It has no package manager, assuming that all the applications run in containers. Container cluster management is done with Kubernetes. Dual-partition OS updates are also supported. It supports a wide range of x86-64, but it does not have support for ARM devices.
- BalenaOS [71] is an open-source operating system built around containerisation for IoT embedded devices. The core OS is built with Yocto and has the minimum set of components for stable operation. The applications are deployed in a Docker container for ease of portability and full image updates. The main OS builds can be made via Yocto Jethro, which is not supported anymore due to the release of two other more recent versions.

Unikernels [92] are specialised, single address space machine images build using operating system libraries. They run directly on hardware or hypervisor without the intervention of any OS. They have small sizes and are application-specific built, with minimal libraries and dependencies. They are ideal for constrained environments specific to Internet of Things devices. Some operating systems libraries that can be used for constructing unikernels are MirageOS, HaLVM, and Rumprun.

- MirageOS [92] is an operating system library for unikernels written in OCaml programming language and includes libraries to support networking, concurrency support and storage. It is fully event-driven, with no support for preemptive threading.
- HaLVM (Haskell Lightweight Virtual Machine) [155] is a library that enables writing programs that run directly on the Xen hypervisor. It is written in the Haskell programming language and can use non-Haskell libraries using the standard Haskell Cabal toolset. It allows for very lightweight, single-purpose Xen domains with minimal resource requirements to be run directly on the hardware and can be used for IoT devices.
- Rumprun [92] is built on a foundation of rump kernels, uses rump kernel drivers, has a libc and an application environment on top and provides a toolchain to write Rumprun unikernel applications. It can support existing application-level software without the need to port it, maintaining the unikernel small memory footprint advantage.

2.3.5 Over the air updates

Billions of IoT devices are going to be deployed across the world in the next years. These devices are going to be part of smart homes, smart cities, and will monitor the environment [62]. Managing all these IoT devices is a challenge, physically intervening to update and troubleshoot them will be nearly impossible. Researchers are working towards developing stable, reliable systems for remotely managing IoT devices. Over the air updates are a significant part of this work, and this subsection exemplifies some of the most reliable OTA strategies.

There are three main types of OTA strategies:

- Full image updates - the standard update for Linux embedded systems. There are two types of such updates:
  - **Single image approach** - assumes that the new image will boot, and has the possibility of recovering if the update can be performed from fallback factory bootloader.
  - **Dual image approach** - gives the possibility of rolling back to the previous working image in case of failure
- Incremental image updates minimises the size of the updates sending only changes composed of binary deltas for the modified files. It provides incremental atomic upgrades that can be easily deployed or rolled back and a complete history of deployments.

- Containers - built on top of an immutable core distribution, where the upgrades are deployed in containers. This cannot be used as a complete upgrade solution. Currently, there is a lack of tools providing this type of update.

The following software tools are built using these strategies:

- Mender [99] is an open-source remote updater for embedded devices. It is based on a dual image update framework. It has two client modes: *standalone*, where the updates are triggered locally and *managed*, where the client is a daemon and polls a server for updates. It uses the notion of commit when the updated has booted properly, and it toggles the inactive/active partitions in case of failure, which represents the fail-safe mechanism. It also contains an extra partition for data persistence, which remains unchanged during the update. The meta-mender layer can be used to build the client into a Linux OS distribution using the Yocto project.
- Swupdate [139] is a software update strategy for embedded systems. The updates can be made locally or over the air. It uses single image updates; hence it does not provide a rollback option in case of failure.
- swupd was initially part of the ClearLinux [18] project, and it is based on an incremental atomic upgrade mechanism. The updates are created and handled by a swupd-server and delivered as streams of bundles containing binary deltas to the swupd-client. The system does not require a boot after the update was made, and the meta-swupd layer allows building the mechanism into clients using the Yocto project.
- OSTree [150] mechanism is very similar to swupd, with the exception that it does need a reboot after the update has been installed. OSTree is a popular tool, used for projects such as Project Atomic, Fedora Atomic, Qt for Device Creation. The deployment process follows the steps below:
  - an updated image tarball is committed on the OSTree repository
  - an update/upgrade is triggered on the host side
  - a system restart is triggered on the host side for the new filesystem-set to take effect

  Cockpit [109] is a software tool that provides administration and visualization for hosts and container clusters and can be integrated with OSTree.
- Balena.io [21] offers an OTA update system for containers. The client-side of the system is open-source, but the server-side is proprietary. This would be a useful OTA for the project presented in this work, but we chose not to use it due to its proprietary nature.

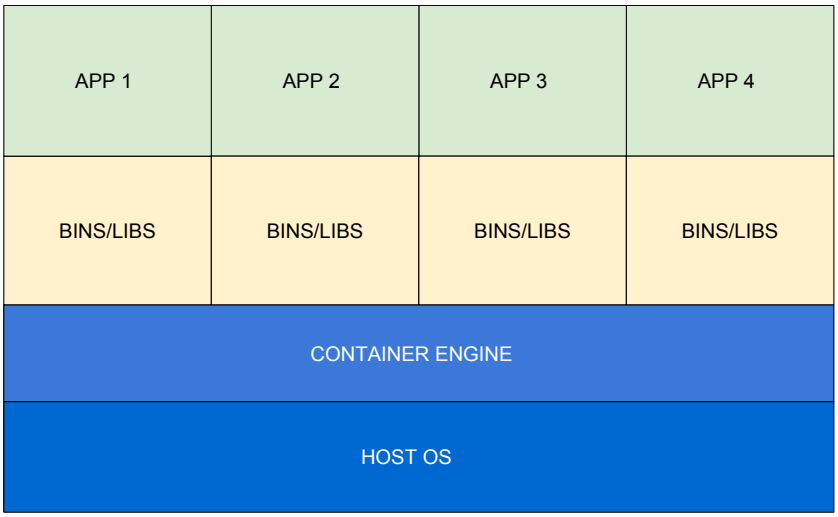


Figure 2.3: Figure showing the containerisation architecture and the relationship between container applications and the host OS

2.3.6 Containerisation

Containerisation is a technique of deploying software programs in an encapsulated environment that contains its own OS, the application and its dependencies, and has limited access to the underlying software system. The host OS and the container OS share the same kernel.

Containerisation can be easily confused with sandboxing. Sandboxing is a technique of isolating running software programs from the underlying software layer, providing a limited set of resources and read/write capabilities for the guest software. Sandboxing is usually used to run untrusted programs, i.e., in Windows. Containerisation limits the set of resources available to the guest software, as well as allowing the deployment of a new opperating system and dependencies in an encapsulated environment. As an example, web browsers run websites' javascript code in a sandboxed environment restricting access to resources and files on the host computer, but it does not deploy it in a container.

Containerisation and sandboxing are used for IoT applications because of the flexibility and increased security they offer around building, testing, running, and managing applications.

Containerisation is particularly useful for IoT devices as it increases reliability, speeds applications deployment, and migration, minimises the size of the OS and benefits applications and hardware upgrading. This section exemplifies some of the most popular containerisation software tools.

- Docker [100] containers share the same kernel with the host OS. They can run on Linux distributions and Microsoft Windows.

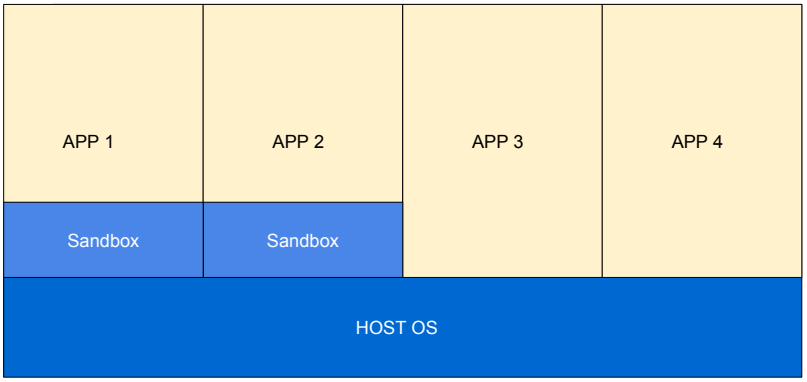


Figure 2.4: Figure showing the sandboxing software architecture and how sandboxed applications are isolated from the other applications

- Docker Engine is part of Docker, and it is the tool that helps building and running containers.
- Docker Hub is also part of Docker - a SaaS service for application sharing and managing.
- Docker Machine [144] is a tool that can be used for provisioning Mac or Windows with Docker or for installing and managing remote Docker hosts.
- Docker Swarm [144] helps manage Docker Engines at scale using an API. It helps to deploy clusters of Docker Engine hosts. The clusters act as a single computer, and the tasks are deployed per swarm only, and cannot be assigned individually per node in the swarm.
- Kubernetes [28] is a management tool for container clusters. Kubernetes offers means of networking, scheduling, load distribution, and data persistence for container-based architectures. It addresses containers using labels. The containers can be organised in pods, which can contain any number of containers, but generally, they contain one or two containers. It also has a replication controller and AK8S service that is persistent, provides discovery, load balances VIP layers, and identify pads by label sector. Volumes can be mounted as filesystems, and it also has a grouping mechanism, called namespacing.
- Singularity [91] containers can run an OS independent of the main OS. The container user privileges are shared between the container OS and the host, i.e., *root* access in the container is given to users that have root access on the host OS. This helps to preserve privacy and provide security.
- rkt [123] is an open-source command-line interface for running application containers on Linux with varying degrees of protection. The primary interface is a single executable that makes integration with existing containerisation tools easier. It

implements the App Container (appc) container format, and it can also execute containers in other formats, such as Docker containers.

2.4 Internet of Things applications

2.4.1 Air Quality monitoring

Array of Things

Array of Things (AoT) [36] is a project that uses a network of modular sensor boxes for urban sensing. At the time of writing (September 2019), there are 127 sensors deployed within the city of Chicago but the project is running in a few cities around the world. The sensors are measuring data on air quality, climate, traffic, and other urban features and are described as "fitness trackers" for the city. The data produced by the sensors is available as open data through the AoT platform.

The boxes are built using the Waggle platform [22], which has a modular architecture and supports by default sensors that measure: barometric pressure temperature, ambient light, acceleration, vibration, remote surface temperature, sound level, magnetometers. Besides these, the AoT boxes also include air quality sensors that measure Carbon Monoxide, Nitrogen Dioxide, Sulphur Dioxide, and Ozone. Further work will look into monitoring other urban factors of interest, such as flooding and standing water, precipitation, wind, and pollutants.

OpenSense

OpenSense[5] is a partnership between the National Air Pollution Monitoring Network in Switzerland and other parties in developing, deploying, and researching mobile air quality monitors. Up to date, they have developed a prototype of an air quality monitoring device that measures Ozone, Carbon Monoxide, Nitrogen Dioxide, and ultra-fine particles. The sensors have been deployed on ten trams in Zurich, and they collect data when the tram is stationary to mitigate the effects of tram movement on the data. A sensor has also been deployed at Switzerland National Air Pollution Monitoring Network. The sensor data can be accessed online over their Global Sensor Network (GSN) [4].

Stuttgart, Germany

The Open Knowledge Lab [141] in Stuttgart, Germany have developed a PM monitoring device for a citizen science project. The project uses the NodeMCU ESP8266 microcontroller, two pipes, DHT22 temperature, and humidity sensor and SDS011 fine dust sensor. The project has been crowdfunded, and anyone interested in mounting one of the sensors at their home can go to one of the public assembly workshops hosted by the Open Knowledge Lab and build their own sensor.

AirPi

AirPi [8] is a sensors kit built using Raspberry Pi. The kit contains sensors for monitoring weather conditions such as temperature, humidity, and UV levels. It also contains air quality monitoring sensors such as carbon monoxide, nitrogen dioxide, and smoke level. The sensor device records the data and logs it via the internet. The project is open-source, the kit can be bought for $ 90, but it can also be built from scratch as all the information needed is available online. The sensors used for air quality monitoring have the advantage of being low-cost but are not sensitive enough to provide reliable readings. Their sensitivity range is lower than the maximum safest amount of some of these gases in the air.

Home automation hubs

Very popular at the moment are home automation hubs. These devices are built for controlling different home IoT devices, such as smart thermostats, lighting, smart TV, and others. Google Home [110], Amazon Echo [119] and Wink Hub 2 [98] are just a few examples of these devices. Although the idea of being able to control all the home appliances with a single device has multiple benefits, such as price and extending the capabilities of home appliances, they are also greatly affected by interoperability limits being able to manage a defined number of devices. They also raise concerns regarding the privacy of the users and security.

2.4.2 Plug and Play platforms

Libelium [88] has developed multiple plug and play sensor boxes to be used for different applications like environmental monitoring and agriculture. These devices run on a custom-built microcontroller, Waspmote, which uses ATmega1281. It has a frequency of 8 MHz, 8KB SRAM, 4KB EEPROM, 128KB FLASH, 2GB SD card, seven analog inputs, eight digital input/output pins, 2 UARTs, 1 PWM, 1 I2C bus and 1 USB port.

The main disadvantage of the Libellium hardware is that, despite having plug and play features, each development platform has a specific range of sensors it can work with, which have to be bought from the Libellium company. Additionally, Waspmote is built-in, and it is the only processing board that can be used for their platforms. Although this decreases the complexity of the deployment, it also limits the use cases and makes it not suitable for research or ongoing development of smart cities. Furthermore, the cost of the Libellium products makes them unaccessible to small research teams, and they are targeted for entreprise use.

The Waspmote runs a dedicated program. It also has a system manager - Meshlium - which is a hardware Access Point that can also be used to run the system management server. The management server can be used for updating the applications (sketches)

OTA, networking, and monitoring the system. These characteristics are in line with the architecture proposed in this thesis. Although they are independent of each other and hence provide a level of modularity, they are unique to the Libellium ecosystem, and other tools are not available to use. Hence, their system limits the range of applications it can cover.

The Libellium devices have been used for mobile air pollution monitoring using buses [26] and for ambient and environmental conditions monitoring in the city centre in Ljubljana [86]. They have also been deployed in the Hampshire county for monitoring weather conditions and road temperature for optimising the amount of salt that is spread on the roads during the winter [87].

The BeagleBone Green is a low-cost, community-supported, open-source development platform, fully compatible with BeagleBone Black featuring AM335x 1GHz ARM Cortex-A8 with 512MB DDR3 RAM, 4GB 8-bit eMMC onboard flash storage. It also has two Groove connectors (SPI and I2C) and two 46 pin headers. This makes it a great development platform that can be used in the hobbyist and academic communities, but it cannot be easily used in production, as it would need further adjustments. It does, however, represent an excellent technology to be used with the hardware development HAT [12] presented in this work.

At the software level, the BeagleBone Green does not come with a pre-defined software stack, it is not dependent on any proprietary architecture, and it can run a wide range of OSs.

Zolertia [64] produces a series of commodity hardware platforms that allow fast development for IoT devices. The Zoul is a core module based on TI's CC2538 system on chip (SoC), it has an ARM Cortex-M3 with 512KB Flash and 32KB RAM, that can fit a wide range of prototyping boards. They also produce RE-Mote, a hardware development platform that includes, among others, two radios, external SD card storage, low-power operations, Real-Time Clock, and has available interfaces and connectors for plugging in different sensors. Another one of their products, Firefly, is a hardware development platform built on top of the Zoul module that has ultra-low power consumption and has most of the pins exposed for ease of development. It is compatible with breadboards and battery holders. Its main disadvantage is the limited processing power, given by the Zoul module, which in turn limits the range of applications it can cover.

BalenaOS [71] is an open-source OS built around containerisation for IoT embedded devices and part of the balena.io project. The core OS is built with Yocto and has the minimum set of components for stable operation. The applications are deployed in Docker containers for ease of portability, and full OTA image updates are also possible, using resinhup, a Docker-based tool developed by balena.io for full OS updates. However, although the OS is built using the open-source Yocto project, adding new capabilities or making modifications to it is highly difficult and time-consuming. It introduces low-level

specific modifications that are not compatible with other layers than the ones used in the Balena project. For example, in the project presented in this work, it was tried to integrate delta OTA updates and balenaOS, but failed due to the distribution incompatibilities between the layers introducing the balenaOS and OTA capabilities respectively. Furthermore, the project seems to be using old versions of the reference distribution, poky, of the Yocto project, always being a few branches behind the latest stable one. This introduces problems such as outdated software versions that cannot be used with present relevant developments.

The Project Atomic [109] is an initiative that supports re-designing the OS around containerisation. It comprises of a lightweight container OS, the Atomic Host, where the applications run in containers. The host has Kubernetes [65] installed by default, uses Docker [25] and is managed with rpm-ostree. The project also has different tools available such as AtomicApp for developing contained applications, Atomic Registry for registering the containers, and Cockpit to give visibility into the host and containers. It has been developed for server management and development and does not have support for embedded systems. Fedora Atomic [109] is a project between Fedora and Project Atomic that creates software frameworks with Fedora and Project Atomic for cloud servers.

Project Ara [158] was a Google project with the aim of developing a modular smartphone that could accomodate hardware and software updates, aiming to increase device lifespan and minimisation of e-waste. The proposed smartphone consisted of a frame and modules - such as cameras, speakers, WiFi - which could be updated over time. The project was based on Modu [106], a lightweight modular cell phone which allowed users to add keyboard, sporty chassis, camera, and MP3 player and which was bought by Google in 2011. At the bases of the Ara smartphone was a propietary frame, which could integrate Google-approved modules. In 2016 the project ended for unclear reasons. Modular consumer devices are becoming decreasingly common, with popular companies, such as Apple, providing high-performance un-upgradable devices.

In conclusion, although there are different development platforms for IoT devices, these impose limitations, such as limited processing power or a limited range of sensors they can cover. The available software even further increases these limitations. IoT devices are defined by both hardware and software.

2.5 Research Questions and Objectives

The literature review identifies the need for new IoT architectures where technology can be integrated after deployment in order to allow upgrading, maintenance, and re-use of existing infrastructures for new application deployment. Some research projects and companies have already made the initial progress in the area, but their solutions are

limited to hardware or software-only or built for specific use cases or can be used within vendor-specific ecosystems only. The main questions we are answering in this research are:

- Given the continuous evolution of IoT technologies, what are the requirements that need to be met by the IoT devices in order to make them adaptable to new applications and technologies?
    1. What hardware design requirements need to be met in order to build devices that are able to change their functionality after deployment and adapt to continuous technological development?
    2. What software design requirements need to be met in order to build devices that are able to change their functionality after deployment and adapt to continuous technological development?

To address these questions, we establish the following objectives for this thesis:

- To develop a generic IoT hardware platform that can be used with off-the-shelf sensors for smart city distributed sensor networks. Chapters 3 and 4 explore the requirements for the generic device.
- To implement this platform such that it can use a wide range of sensors, independent of manufacturer, interface, and other similar barriers often met with IoT devices. Chapter 5 presents a genric hardware development board that meets the design requirements, with an implementation of the device as an environmental monitoring use case.
- To be able to swap sensors during operation, allowing to update the device specifications and to add new capabilities and applications, as required. Chapter 5 presents the hardware implementation and validation of the generic device presented in this work.
- To explore potential sensors for a first use case and implement it. Chapters 3 and 4 explore potential sensors to be used for an use case, while 5 presents an environmental monitoring device, which serves as an use case of the solution presented in this work.
- To identify software requirements for this type of generic device and create a software architecture. These requirements are explored and detailed in Chapters 3 and 4.
- To explore existing software packages and tools that will enable or speed up the implementation of such an architecture. Chapter 6 detailes the software packages and tools considered for the implementation of the software architecture.

- To develop a reference implementation of the architecture that can serve as a working example and a starting point for new projects. A reference implementation is presented in Chapter 6.

In the following chapters, we are focused on answering the research questions by following these objectives.

Chapter 3

Indoor positioning system

The main role of this chapter is to identify the requirements for an IoT system deployed at different scales and how continuous technological evolution is influencing the IoT networks in order to answer this work's research question.

This chapter is organised into three sections. The first section presents an indoor positioning sensors network, and it is based on the *Indoor localisation system based on low-cost commodity hardware* paper that was presented at the UbiComp16 conference [13]. The second section presents the same technology but deployed at a larger scale and in two different environments, an office area and outdoors. The third section brings to light the issues that were faced when managing multiple IoT devices, how technological evolution is impacting the presented network, and identifies requirements for designing, deploying, and building the next generation IoT device that would solve some of the issues identified through this experiment.

3.1 Indoor localisation system based on low-cost commodity hardware

Pervasive indoor positioning is a key enabler for ubiquitous computing. Although a number of indoor positioning systems exist, we identify that there still is a barrier between these and the consumer. There is a need for a cost-effective, practical solution that can be easily designed, built, tested, and deployed.

We have built a low-cost indoor positioning system with off-the-shelf commodity hardware to provide an example of an accessible ubiquitous platform for developers, academia, private individuals, big and small companies, and institutions. The system is low-power, modular, performs auto-calibration, can track any WiFi-enabled device, and can be used for other applications alongside indoor positioning.

3.1.1 Introduction

A practical, cost-effective indoor positioning system (IPS) is one of the key enablers of ubiquitous computing. Although a large number of such systems [89] have emerged since Weiser defined ubiquitous computing [153], we believe that there still exists a barrier between these and the consumer. For example, the industry does not focus on developing ubiquitous solutions, and some commercial products [24, 152, 7] do not provide services for small spaces, aiming at large areas such as airports, universities, hospitals, and large retail companies. Furthermore, their products are often expensive, and the tracked asset requires specific localisation hardware such as smart tags. Table 3.1 details the costs and performance of some of the leading systems. High accuracy algorithms and systems have been developed in academia, but they are often not ubiquitous and require specialised knowledge [146, 94]. Fingerprinting-based indoor positioning is a popular choice, but collecting fingerprints requires user interaction, is impractical, and imposes a barrier on the user. There is a need for a low-cost, practical IPS that can be easily deployed. Weiser claims that in ubiquitous computing, we can only learn by experimenting. Hence, we need a simple system that allows for further experimentation with location-aware applications, but also with WiFi-based indoor positioning algorithms.

We present a low-cost WiFi-based IPS that uses Raspberry Pi devices for collecting and processing data. The system is ubiquitous, it collects the data without user input, and the remaining Central Processing Unit (CPU) capacity can be used for other applications, representing a valuable development platform for ubiquitous applications. It performs auto-calibration: for each node, all the other nodes of the system act as anchors, allowing for wall alleviation factor to be calculated. Each node can be battery powered and has a modular configuration, allowing for swapping and adding sensors. The system can track any WiFi-enabled device, and it works with Linux-based Operating Systems (OS).

Table 3.1: Cost and accuracy of indoor positioning systems that are currently on the market (2017) and have their pricing available. Please, note that the prices are subject to change, and the accuracy of the systems depends on the density of locator nodes per m^2.

Product	LN	Accuracy	Services	Installation	Tag	Licence	Support
Accuware	$ 30 - 40	2 - 4 m	$195 [1]	$ 60000	$ 59	$ 99 [2]	$ 75000 [3]
Indoors	$ 30 - 40 [4]	2 - 5 m	$ 499 - 999 [5]	ad-hoc		N/A	$ 95 - 990 [6]
infsoft	$ 170	2 - 5 m	N/A	$ 1000	N/A	5500	ad-hoc
Air-Go	$ 230	up to 2 m	$ 163 [7]	[8]	$ 51	N/A	ad-hoc

[1] monthly subscription to the cloud server - up to 50 tracked devices
[2] per device running Accuware software
[3] per year
[4] approximate price of an access point
[5] per month
[6] per month
[7] per year
[8] ad-hoc - approx. 15% of the total project cost

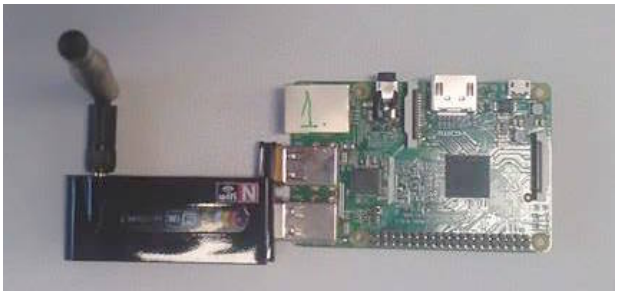


Figure 3.1: Locator node

	Model	**Price**
RPi 2	Model B	$ 35
WiFi	RALINK3070	$ 17
WiFi	EW 7811UN	$ 10
Power Supply	T6143DV	$ 7
SD card	SanDisk Ultra 64 GB	$ 12.5

Table 3.2: Details of the components of the locator nodes used for this study including prices and models. The prices are valid at the time of writing (2017) and they might change in time.

3.1.2 Design of the Indoor Localisation System

The system has two components: locator nodes that collect data from WiFi-enabled devices and a server that runs on one of the nodes. Figure 3.1 shows one of the locator nodes and Table 3.2 details the hardware components of each locator node. The two WiFi cards are used for monitoring the tracked devices and for connecting to the local network respectively.

The tests presented in this work have been performed using Raspbian OS. OpenWrt, which is an OS specifically designed to deploy on Access Points and routers [51], has also been successfully tested.

The Free Space Path Loss theory [120] and the trilateration algorithm [93] have been used to compute locations of the tracked devices. The final location estimate is assumed to be the centre of the smallest circle that can be formed using the best k solutions yielded by the trilateration algorithm.

The best k solutions are filtered based on the Euclidean distance between them and the locator node with the highest signal strength. k, and the number of locator nodes were chosen to be two based on iterations made on multiple data sets.

Each locator node collects data (the Media Access Control address, received signal strength) on channel 11 using the Python library `scapy` and sends it to a `HTTP` server, that computes the coordinates of the active WiFi devices and displays them on a web page in real-time. The data from the locator nodes is sent continuously to the server and saved in log files, but not used for coordinate computations if it has been in the system for more than one minute.

Besides tracked devices, the locator nodes are also monitoring each other. Knowing the real location and the signal strength variation at each locator node allows for the wall alleviation factor to be calculated, where wall alleviation factor represents the variation in signal strength due to LOS obstacles in the environment. This enables the system to adapt to changes in the environment.

3.1.3 Preliminary Testing

3.1.3.1 Methods

The tests have been conducted in a busy office area of 30x13 m^2. The office, shown in Figure 3.2, contains computers, tall metal drawers, thick pillars, four Access Points, printers, microwaves, Bluetooth enabled devices, and other furniture and equipment typical in an office environment. Figure 3.3 shows a 2D top view representation of the testing area and the positions of test points.

The measurements have been taken using a Fluke 414D laser distance meter that has an accuracy of $\pm$ 2 mm, and the tracked object was a Kazam Thunder 45L smartphone.

The tracked device was held stationary at each of the test points. The results represent the average of all the coordinate estimates obtained at each of the test points.

The error has been calculated as the Euclidian distance between the coordinates yielded by the positioning system and the physical location of the tracked device.

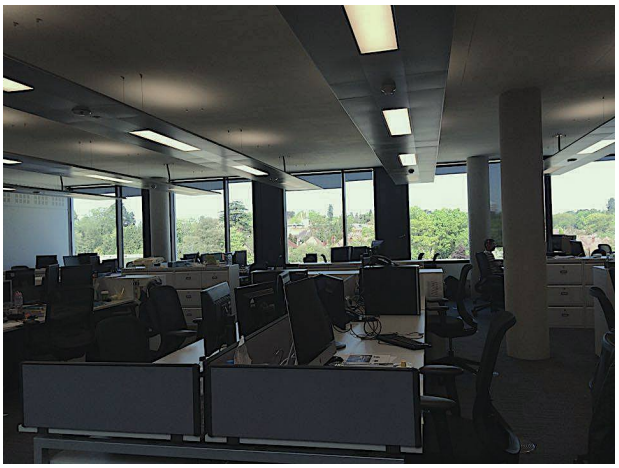


Figure 3.2: Image of a section of the office area illustrating the testing environment

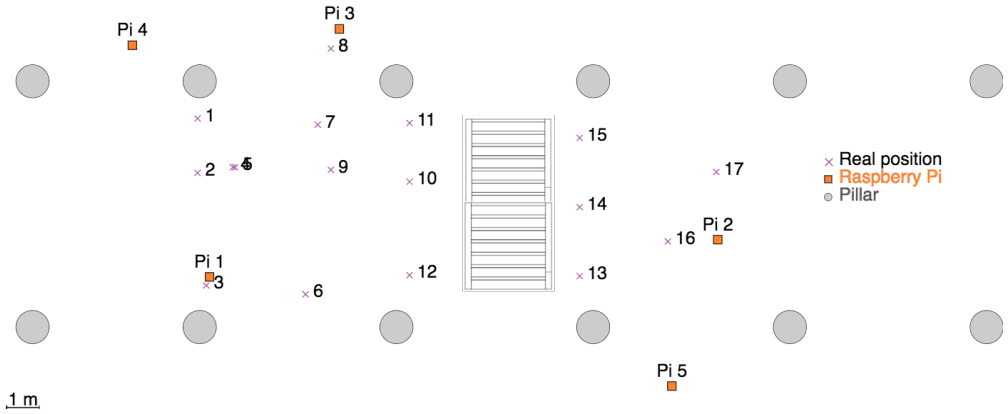


Figure 3.3: The testing area layout showing the position of the locator nodes (Pi) and the locations where data is collected. The test points are numbered in an ascending order from left to right hand side along the width of the test environment.

3.1.3.2 Results

Tests were taken using five locator nodes, and the results are illustrated in Figure 3.4. It is noticed that the accuracy decreases from the left to the right-hand side of the testing area. The results are influenced by the positioning of the locator nodes relative to the measurement points and to each other, and by interference and Line-of-sight (LOS) obstructions. To prove this theory, we re-calculated the tracked device coordinates in the test section, where the nodes are evenly spread, and less LOS obstructions and interference are present. We chose to use three locator nodes only (one, three, and four) to re-calculate the coordinates at the first nine test points. The results show a decreased locating accuracy, as illustrated in Figure 3.5, supporting our theory. Furthermore, comparing these results with the ones that were obtained previously at the same points,

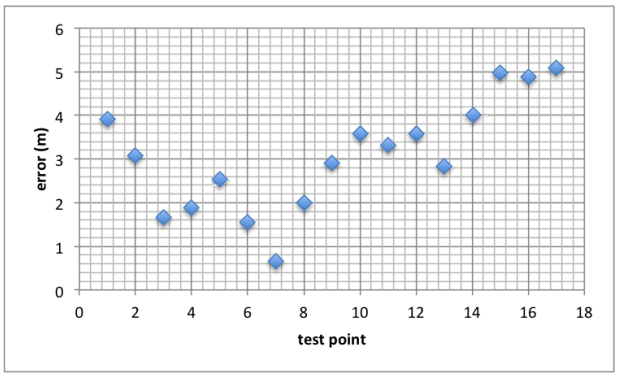


Figure 3.4: Graph illustrating the error variation at 17 measurement points. The accuracy of the system is ±3.09 m with a standard deviation of 1.24 m

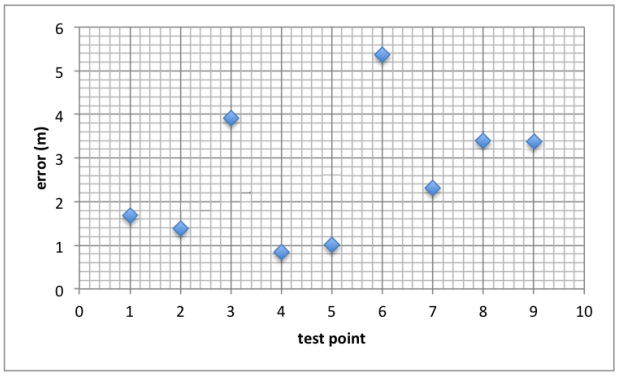


Figure 3.5: The error variation at the first nine measurement points when three locator nodes have been considered: Pis 1, 3, 4. The accuracy of the system is ±2.59 m with a standard deviation of 1.44 m.

it is noticed that better accuracy is obtained when more locator nodes are used. It demonstrates the impact of the number of locator nodes on the accuracy of the system.

In conclusion, the results presented in this section suggest that better accuracies could be obtained by increasing the number of locator nodes and minimising the line of sight obstructions within the testing area. In order to confirm these findings, more experiments are designed, where the number of locator nodes is increased. The experiments are conducted in two different environments, to clarify the impact of LOS obstructions on the accuracy of the system.

3.2 Further assessment of the system performance

The work presented in the previous section describes a low-cost IPS system that can be used as a development platform for location-aware applications. Preliminary tests demonstrated 2 - 3 m average accuracies.

This section presents how the device performs in two different environments under more in-depth tests.

3.2.1 Methods

To further assess the performance of the system, we developed an experiment that used six locator nodes to track ten devices simultaneously. The locator nodes and tracked devices were attached on top of 1.5 m poles. The locator nodes were positioned such that they represented triangles vertices for better node distribution across the assessed area. The tracked devices were ten Raspberry Pi 3 model B. Each of the tracked devices was stationary during the measurements, and we took measurements for five minutes at each of the test points. The ten tracked devices were moved across the testing area in eight configurations, which yielded approximately 80 data points.

The server was located on a Raspberry Pi, which was also an Access Point to which the locator nodes were connected to transmit data without an Internet connection. The tracked devices were connected to another Access Point, the EE 4GEE WiFi. The Raspberry Pi devices used for the locator nodes were also replaced with the Generation 3 model B ones, which reduced the cost of the system by $10. All the other methods remained unchanged from the previous tests described here.

The system was tested in two different environments: in the office area described previously and outdoors (Southampton Common) in clear line of sight conditions. The test area and positions of the locator nodes and the tracked devices relative to each other were kept the same in both tests, and the only change is that the second environment does not have any office furniture and equipment, and there are fewer radio devices producing interference. The system was tested in the second environment to assess the influence of LOS obstructions and interference on the overall performance of the system.

3.2.2 Discussion and analysis of the results

Preliminary data processing gives an accuracy of 6.58 m for the tests conducted in the office area and 9.5 m for the tests conducted in the park. The error variation with respect to both x and y is illustrated in Figure 3.6 for the office tests and Figure 3.7 for the park tests. These results are different from what we have obtained before - the error is greater.

The best solutions are filtered based on the difference between the Euclidean distance between them and the locator nodes and the distance calculated using Free Space Path Loss formula. In order to check if this approach is correct, we re-calculated the coordinates of the tracked devices filtering the best solutions using the distance between the real coordinates and the coordinates yielded by the trilateration algorithm. The average error obtained in this case is 2.86 m for the tests conducted in the office area, displayed

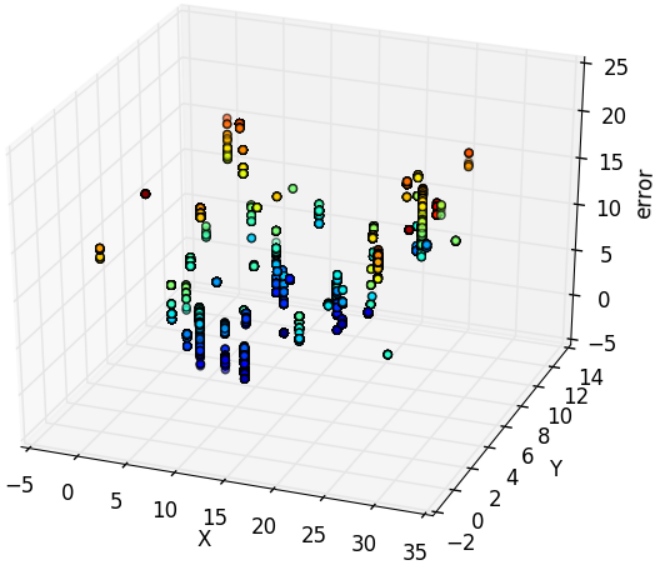


Figure 3.6: Variation of the position estimation errors at the measurement points for the indoors experiment.

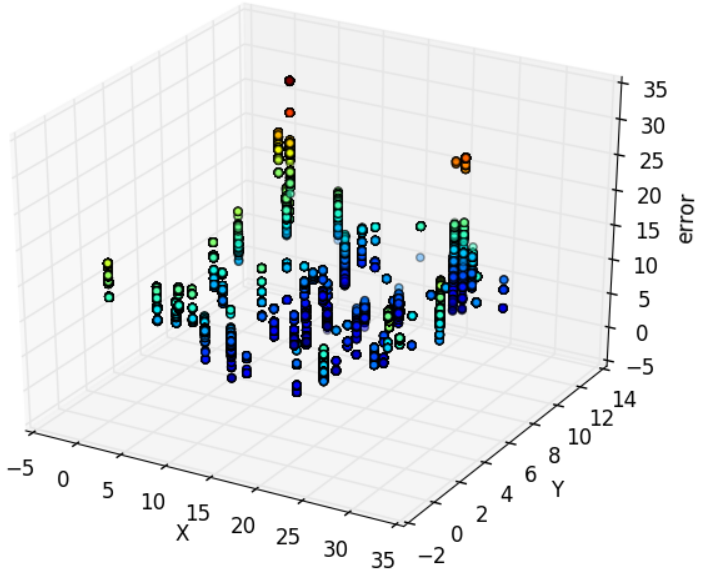


Figure 3.7: Variation of the position estimation errors at the measurement points for the outdoors experiment.

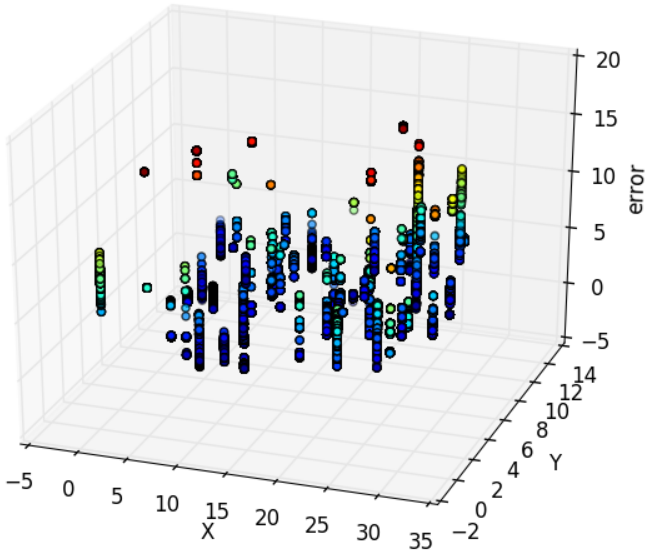


Figure 3.8: Variation of the position estimation errors at the measurement points if the method of choosing the best solution would be ideal. This is the data for the indoor measurements.

in figure 3.8, and 3 m for the tests conducted in the park, shown in figure 3.9, This means that improving the way we choose the final coordinates of the tracked devices could improve the accuracy of the system.

Figures 3.6 and 3.7 also show that for each point, we have found more than one result. This is explained by the fact that we took multiple measurements at each of the points.

Being able to find a way to automatically choose the best result at each of the points in real-time would reduce the system error. One way of doing so is by defining a scoring method for the solutions and applying an upper bound that, if exceeded, the solution is discarded. This is the approach we chose when processing this data.

The scoring mechanism creates a score for each of the solutions yielded by the trilateration algorithm per reading. The score is the squared difference between the distance used in the trilateration algorithm and the distance between the solution considered and the locator nodes multiplied by the signal strength. This way, the strongest signal will yield a small score. The solution with the smallest score is considered the best.

Another approach was to minimise the signal noise by comparing the signals between two locator nodes, i.e., the signal between locator nodes 1 and 2, as seen at locator node 1 compared with what is seen at locator node 2. Doing this for all the combinations of locator nodes, we are able to minimise the overall system error by approximately 1 m.

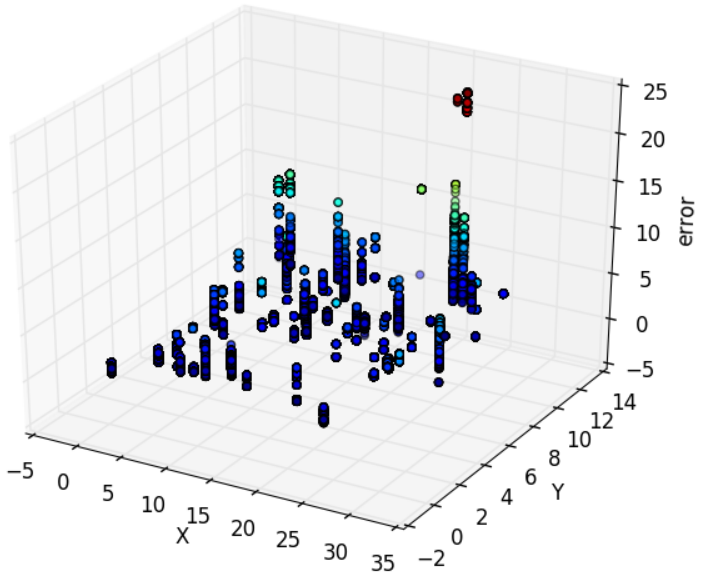


Figure 3.9: Variation of the position estimation errors at the measurement points if the method of choosing the best solution would be ideal. This is the data from the outdoor measurements.

Finding the optimum coordinates for the tracked devices by minimising the mean square error, the score described above didn't improve the results for the system. The values obtained using this method were 6.87 m for the office measurements and for 9.66 m for the park measurements.

Python (using scikit-learn) machine learning (ML) algorithms, neural networks, and linear regression have been used in the effort of finding a scoring algorithm that would yield the best solution given by the trilateration algorithm per reading. However, the algorithms failed to find any correlation between the best solutions and signal strengths. Using ML for replacing the trilateration algorithm was not successful either. The algorithms have been trained with data varying from 20% to 60% of the data. Data augmentation has also been used for training purposes in the effort of eliminating the need for calibration prior to the deployment, which is the aim of this work. Using the models created in this way reduced the solutions pool to only two: the origin and the middle of the test area, for any number of data points. This helped us understand that the noise level is high, and finding a correlation between the signal strength and device location is difficult. Further developing the algorithms could lead to better results, but it was decided that ML might not be the best approach to solving the given problem.

The received signal strengths at each locator node are subject to unknown levels of noise. If we calculate the estimated distance d_i between locator node i and the device we're

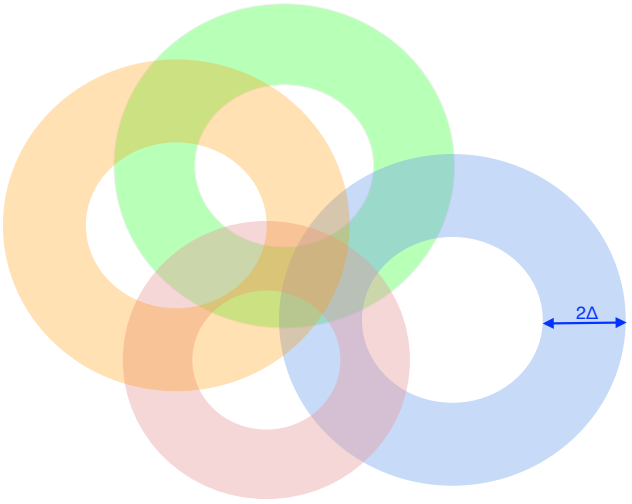


Figure 3.10: This image illustrates how the solutions interval can be considered as the intersection between the rings created as a result of combining the signal strength and the parameter Δ; where Δ represents the signal noise.

trying to locate using the Free Space Path Loss formula and draw a circle of radius d for each locator node, we will notice the circles do not necessarily have a common intersection point. We can overcome the noise by adding a parameter Δ, as described by [133]. Then for each locator node, instead of using a circle with radius d_i we use the area between two circles of radius $d_i + \Delta$ and $d_i - \Delta$, as shown in figure 3.10. We call this area the ring r_i. The middle point of intersection between all ring areas r_i for $i = 1, 6$ is the estimation of the location of the tracked device. For quick experimentation, this method was implemented using the OpenCV library by simply drawing each ring black with 0.3 opacity and thresholding the image for the darkest colour to obtain the intersection area. The centre of that area is the prediction point. Rendering the image size is such that 1 centimetre = 1 pixel. Using this approach, we get accuracies of 8.23 m for the indoors experiment and 9.05 m for the outdoors one.

Using different methods does not increase the accuracy of the system from 6.87 m for the

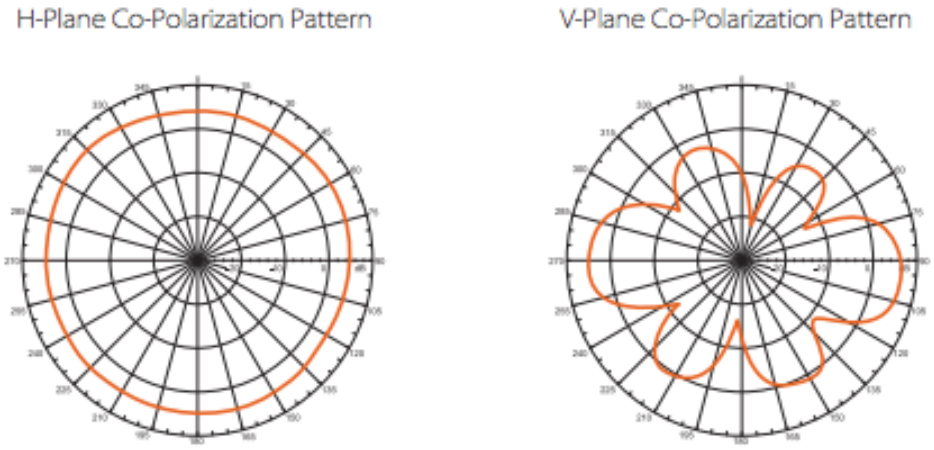


Figure 3.11: Radiation pattern of the omnidirectional, high-gain antenna used for this project [143]

indoor measurements and 9.05 m for the outdoor measurement. The system performed the worst in the park experiment. This means that the interference is introducing significant noise in the data readings. By adding more WiFi devices that send data using the same channel, more interference is created (co-channel interference), which is why the preliminary testing yielded better results. Devices spacing and density also have a significant impact on the data.

The antenna used for the testing is 5dBi gain omnidirectional antenna, and its radiation pattern is described in figure 3.11. As both the tracked devices and the locator nodes were placed on 1.5 m poles, we are only interested in how the antenna behaves in the horizontal plane. Looking at the respective radiation pattern, we can see that the radio signal is radiated equally in all directions.

Furthermore, changing the Access Point from Eduroam (provided by our University IT department) to a mobile one, as compared with the preliminary experiment presented in the first section, might also have caused the difference in the results. Different networks have different forms of interference mitigation.

Even if outdoors, there were fewer obstructions, the ground, the poles, the devices themselves, and the people walking around in the area are forms of obstruction and introduced noise in the data. It is also possible that some of the locations of the tracked devices were not measured correctly, some of the devices were mislabelled, and others fell during the measurements.

Attenuation, refraction, reflexion, and diffraction are altering the signal received at the locator nodes, introducing a high error that is very difficult to mathematically model. Signals are minimised, increased, and attenuated when they come in contact with each other and the environment. General patterns are very hard to find with such a system

that is highly unpredictable, considering all the processes that the radio waves are undergoing. Systems that perform calibration before deployment can give good accuracies because they can create a model of the environment. Being able to change the hardware can also increase the accuracy of the system. However, trying to get a precise indoor positioning system without any of these is highly complicated and has not been achieved up to date.

There are ways of increasing the accuracy of the system, for example, knowing if the tracked device is stationary or not could help us filter the data at each point more accurately. Accurate timing could also lead to improved accuracy. Using other algorithms for data filtering and using different localization techniques than trilateration might also lead to increase accuracy.

3.3 Conclusions and future work

The system helped us understand how WiFi-based positioning using off the shelf hardware can be used. This study shows 6.87 m accuracy for the indoor measurements and 9.05 m accuracy for the outdoor measurement. Using more hardware than WiFi dongles to track devices in a less radio-dense environment could yield better accuracies. Depending on the application and the environment, the system could be used to determine the number of devices in the room and even their general distribution, although obtaining precise coordinates is challenging.

Looking deeper into radio propagation and its characteristics could also lead to a better understanding of the system and how to improve it. However, this is not the purpose of this work, and although the accuracy of the system could be improved by employing different technologies, we have not identified a clear path towards significantly improving the system within the constraints of only WiFi and without fingerprinting the target environment.

During the research undergone in this chapter, multiple systems using more sensors or other technologies have emerged, which rendered better accuracies. After undergoing an in-depth literature review, it has been noticed that, given the accelerated rate at which the technology is evolving in this area, most of the devices built for smart applications have a relatively short life, where most of them are disposed of in favour of better versions. Hence, deploying smart devices at a large scale can be wasteful, while the complexity of building on top of existing systems increases exponentially.

The experiment presented in this chapter can be improved by using an entirely different set of sensors [14]. Every year, new research studies in indoor positioning introduce systems with improved accuracies, which use different sensors and mathematical modelling techniques. Hence, a durable indoor positioning system deployed at a large scale

requires incorporating these new technologies after deployment in order to efficiently integrate with new advances in the IoT area for the generation of better services and applications. This aspect, customisation, needs to be met by new IoT architectures. In this work, a customisable architecture allows updating to new technologies after deployment, for example, sensors, actuators, and applications, with no vendor-specific restrictions.

Adding new sensors and actuators to the existing devices deployed in this work could also lead to better accuracies [49]. Adding new sensors to the deployed devices could also enable new applications to be deployed without the need for building new infrastructure. The ability to expand devices after deployment is also a requirement that needs to be implemented for new IoT architectures to increase life span and decrease the complexity of deploying new smart applications. In this work, extensible refers to the property of architectures to integrate new technologies after deployment, such that new applications can be added to already existing infrastructures to enable the creation of a circular economy for sustainable city development.

Replacing sensors and adding new ones needs to be enabled not only from hardware but also from the software level. For example, in the experiment presented in this chapter, data processing has been done using different methods. If the data processing would have been done on the edge, all of these applications had to be deployed on the devices during operation. New technological advances in hardware drive new software developments. Hence the next generation IoT devices architecture needs to meet the design requirements at both hardware and software levels to enable increased life span and minimise device management complexities.

In order to increase the life span of IoT devices, new methods of maintaining them that save time and costs need to be developed. Maintainable refers to the property of an IoT system architecture to enable device upgrades and repairs via modularity in order to increase life span. The next generation IoT device also needs to be accesible, and not rely on vendor-specific ecosystems or require extensive training in order to be able to use it.

The remaining of this thesis is exploring potential designs for meeting the design requirements identified in this chapter: accesibility, customisability, extensibility, and maintainability.

Requirement	**Definition**
Accesibility	An accessible architecture can be used and maintained by the general user and does not require a high level of training. It should also not be restricted by vendor specific ecosystems or cost.
Customisability	A customisable architecture allows updating to new technologies after deployment, for example, sensors, actuators, and applications, with no vendor-specific restrictions. This allows increasing the device life-span and reduces e-waste.
Extensibility	The property of IoT architectures that allows using already deployed infrastructures for new applications, allowing the creation of a circular economy for sustainable city development, while reducing the complexity of deploying new technologies and reducing the e-waste.
Maintainability	The property of an IoT system architecture to enable device upgrades and repairs via modularity in order to increase life span and reduce e-waste.

Table 3.3: Modular device architecture requirements and definitions

Chapter 4

Generic IoT device for an environmental monitoring use case

This chapter explores the potential of meeting the design requirements identified in the previous chapter -availability, customisability, extensibility and maintainability - using a generic IoT device which includes a wide range of sensors to support new applications after deployment. Air quality is an important global concern, and it has been used as a use case for this chapter.

Southampton is the 5th most polluted city in the UK. London has been regularly exceeding the yearly European pollution limit in the first calendaristic months of each year [124]. In order to address these issues, there a need for high granularity data across cities. The city council is providing a few high-quality base stations for monitoring the air quality, i.e., 3 in Southampton, but these are not enough in order to find the main causes of the pollution and find mitigation methods. Hence, in this work, we focused on environmental monitoring devices as use cases for our novel technology.

4.1 Development

Chapter 2 contains the analysis of some of the most significant smart cities and their applications, along with some of the sensors required for achieving these applications. It was concluded that weather (humidity, temperature, pressure, UV levels, light) and air quality (Nitrogen Dioxide, Sulphur Dioxide, Particulate Matter) monitoring sensors along with magnetometer, accelerometer, gyroscope, ultrasound, microwave, camera, microphone, GPS, WiFi, Bluetooth, and LoRa are some of the most common sensors used for IoT applications. Some of these sensors can be connected such that they can be updated with newer models, using connectors such as the Grove ones [131]. Table 4.1 shows the sensors chosen for the first prototype, along with their costs.

Table 4.1: The sensors chosen for the first version of the generic IoT device.

Sensed quality	Distributor	Model	Precision	Price (USD, 2017)
Distance	seeed studio	De-Lidar TF02	1 - 10 cm	149.00
Temperature	Adafruit	MCP9808	$\pm 0.25^o$ C	5.15
Luminosity	Adafruit	TSL2591	188 uLux	7.2
Pressure	Adafruit	BMP180	0.03hPa / 0.25m	10.3
Ultra Violet	Adafruit	Si1145		10.3
Humidity	Grove	AM2302	0.1 %RH	15.5
Sound	Thomann	t.bone GC 100	-40 dB ± 3 dB	17.0
Images	Raspberry Pi Foundation	Camera Module	up to 240 frame/s	30.0
Acceleration and Orientation	SparkFun	MPU-6050	2048 - 16384 LSB/g 16.4 - 131 LSB/^{o}/sec	41.3
SO2	Alphasense	B4	15 ppb	55.0
NO2	Alphasense	B4	0.1 ppb	55.0
Particulate Matter	Alphasense	OPC-N2		277.0
Location	Adafruit	Ultimate GPS	-165 dBm	39.95

The majority of these sensors have been chosen because they have been proved to work with Raspberry Pi, the chosen SBC for the first prototype, and because they are easy to use and have available documentation and libraries. Using Raspberry Pi 3 was a decision made based on availability, price, and the required features for this project, such as processing power and cost. This is not the final Single Board Computer (SBC) choice.

The Alphasense gas sensors have been chosen because they are a popular choice in literature, their readings have good accuracies, and their cost is within the range of this project. They could be used to complement the existing air pollution monitors provided by city councils [33].

4.2 Theoretical use case - Southampton Smart City

This section presents a theoretical use case in Southampton in order to illustrate the benefits of a generic device. This use case has not been implemented and it is only used for presenting the benefits of generic devices deployed at a large scale in smart cities.

For the purpose of this example, it is assumed Southampton has 1000 generic IoT devices that contain the sensors described in table 4.1. The devices are spread throughout the

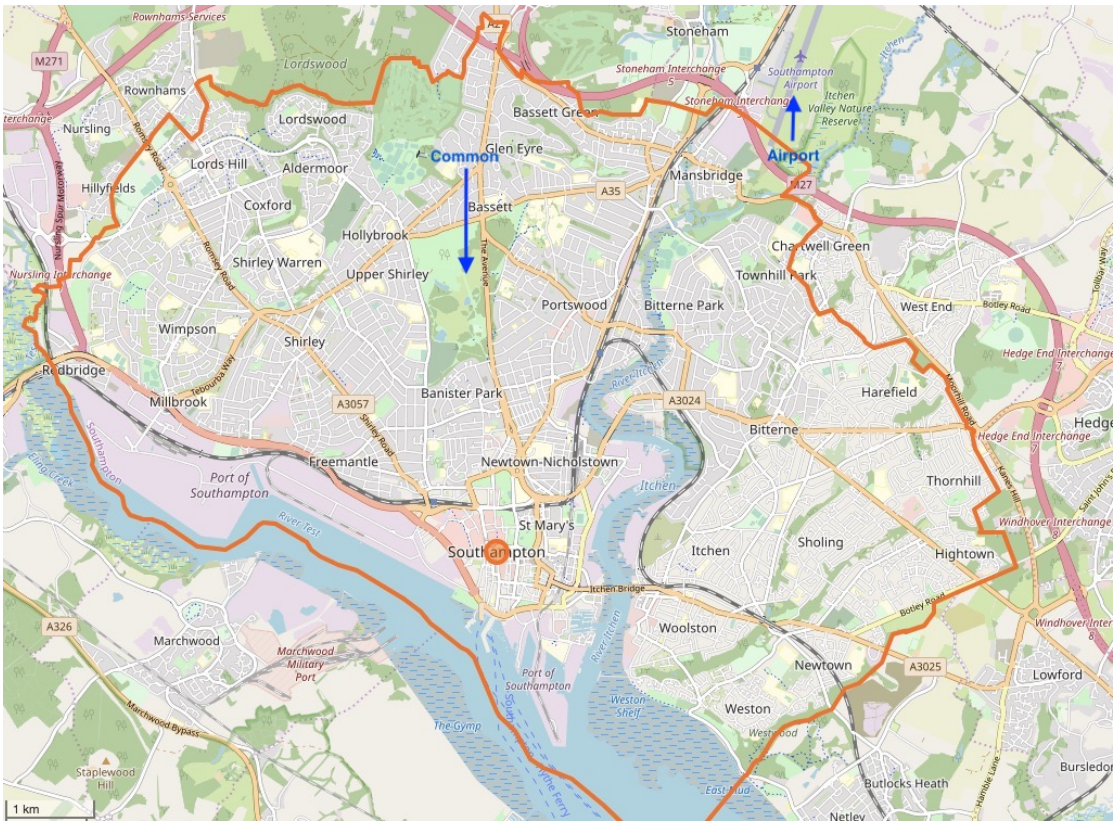


Figure 4.1: Map of Southampton taken from OpenStreetMap [113].

city along main roads, near schools, airport, and in the Common, please refer to figure 4.1 for a map of Southampton.

The sensors along roads perform air quality monitoring, speed monitoring, and plate number recognition during the day. During the night, they control street lightning and perform air quality monitoring.

The sensors near schools perform air quality and security monitoring during school activity and weather monitoring otherwise.

On the Common, the sensors are used for weather and air quality monitoring during the daytime. During the night, they perform street lightning control, detect WiFi devices, and make use of the cameras for public security.

The sensors near the airport are used for noise and air pollution monitoring at all times.

This example shows how the same device is used for different purposes within the city, which are part of the same infrastructure. It also shows that the devices can be deployed around the city without the need for a defined purpose before deployment.

The device can also make use of the sensors it contains to change its purpose on demand, performing temporary functions. The following paragraphs illustrate two examples of situations when the sensor can be re-purposed.

There is a 5K marathon on Southampton Common July every year, and some of the generic devices are re-purposed for taking pictures at the event and for monitoring the UV levels. The devices will be used for these sole purposes during the event, and they will return to their initial activity after the event ends.

The City Council wants to optimise the salt spreading on the roads during the winter period. Some of the sensors along the main roads are re-purposed to monitor the weather conditions. They re-gain their initial functions during the other seasons.

4.3 Air quality monitoring

Air pollution is becoming a big concern across the whole world. China is fighting against air pollution every day [63]. Complex systems have been designed for households to help the population keep safe, such as multiple air monitoring devices per home and air filtration.

Cities in Europe are also concerned about the level of air pollution and are developing projects for monitoring air quality and a better understanding of air pollution. As noted in the literature review, Chapter 2, cities such as Amsterdam, Zurich, Stuttgart, London, and Singapore are already working towards building air quality monitoring devices to be spread through the cities and give a better understanding of pollution distribution.

Air quality monitoring is one of the key concerns for future smart cities. We consider that having a generic IoT device that is able to perform air quality monitoring is essential, which is why the first sensors that have been incorporated and tested are air quality sensors.

The major air pollution gases are Nitrogen Dioxide, Sulphur Dioxide, and Particulate Matter. High end, expensive, air quality monitoring devices that collect data about these gases are already present in most cities, for example, Southampton has three air quality monitoring devices. Due to their high prices, too few of these devices are present, and they are not enough to get an understanding of the air quality in the whole city, which is a common problem in the UK and other countries in the world.

We identify the need for having low-cost mobile IoT air quality monitoring devices that would collect data at multiple locations across cities. In this research, we have developed a mobile, battery-powered air quality monitoring device that was demonstrated at the Engineering and Science day at the University of Southampton to collect data on an uni-link bus, and we now present some further details of it.

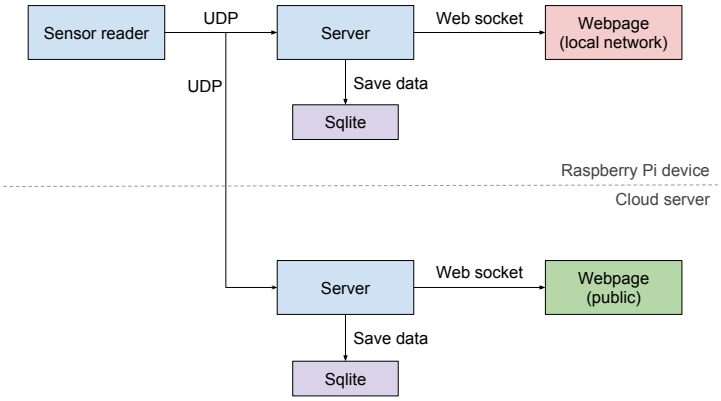


Figure 4.2: Device architecture for the preliminary test, showing how the data was collected, stored and displayed.

4.4 Preliminary testing

This subsection describes the preliminary tests of the air quality monitoring device at the University of Southampton Science and Engineering Day. The subsection describes implementation details, the data collected, how the demonstration impacted the public, and the lessons learned from this activity.

4.4.1 Implementation

An air quality monitoring device has been designed and built for a demonstration at the Southampton Science and Engineering Day. It contained Nitrogen Dioxide, Sulphur Dioxide, Particulate Matter, Pressure, and Temperature sensors. The client streamed live data to a cloud server via UDP, which displayed the real-time data stream on a web page that contained three charts (Air Quality, Temperature, and Pressure) and logged all the data collected in an SQLite database. The server was replicated locally on the Raspberry Pi client, for redundancy in case of Internet failure. Figure 4.2 shows the architecture of the device.

The client was written in Python using popular libraries, such as the Adafruit library for the MCP3008 ADC adapter chip to read the data from the sensors.

The server was written using JavaScript and node.js, and the graphs have been generated using the JavaScript Chart.js library.

UDP was chosen as a data transmission protocol for its simplicity and to support the data being collected and displayed live, which means that having the data sent continuously was more important than not losing any of the data. We are aware that this system, although reliable enough for its intended purpose at the time, is not secure enough and would require enhancement if deployed more widely.

The client is a Raspberry Pi 3 Model B. For the air quality monitoring, we chose Alphasense sensors: the A4-series for the Nitrogen Dioxide and Sulphur Dioxide with the 2-way sensor circuit. The 2-way circuit is an in-house integration circuit made by Alphasense for the two sensors to have minimal noise and for better integration with other circuits. The Plantower PMS1003 Particulate Matter [72, 127] monitoring device was also used. These sensors have been used for the report of quality vs. price they are providing. According to literature, these sensors give good sensitive values that can be used as a good and reliable reference for air quality.

4.4.2 Data

From the data collected during the experiment, it can be noticed that the gas readings, figure 4.3, and the Particulate Matter ones, figure 4.4, are most of the time within the healthy threshold. The only times when the limits are exceeded were when the device was placed close to the exhaust pipe of a bus, first peak, and close to the exhaust of a 1996 Diesel car. These results are expected, and it can be concluded that the sensors' readings are sensitive within the healthy threshold, they can detect when the air has healthy, low amounts of the monitored polluting gases and when values exceed these values. These sensors are not meant to provide absolute values for air quality, but they can be used as references for when the air quality is changing.

Another observation made during the experiment is that the sensors readings are greatly affected by pressure and temperature variations, and by comparing the four graphs, figures 4.4, 4.3, 4.5 and 4.6, it can be noticed that peaks in the gas and PM readings can be correlated with peaks in the temperature and pressure readings.

From figures 4.3 and 4.4, it can be noticed that NO2, SO2 and PM2.5 sensor readings are following the same trend with the same peaks and stable periods, although there is no real reason why the three should be so closely correlated during the experiment. All the sensors measure different gases, and although the peaks are caused by highly polluting events, an increase in particulate matter is not necessarily related to an increase in SO2 and NO2. This leads to the conclusion that the reference voltage for the sensors was varying during the experiment, causing some noise in the data.

Furthermore, the sensors were placed in a box, and figure 4.5 shows how the temperature rises due to the Raspberry Pi heating up during operations. The sensors are temperature sensitive, and a future design should take into consideration this finding.

4.4.3 Security and Privacy

The air quality monitoring device has been mainly developed for exploring the research requirements of the generic IoT device presented in this work and has been used for an

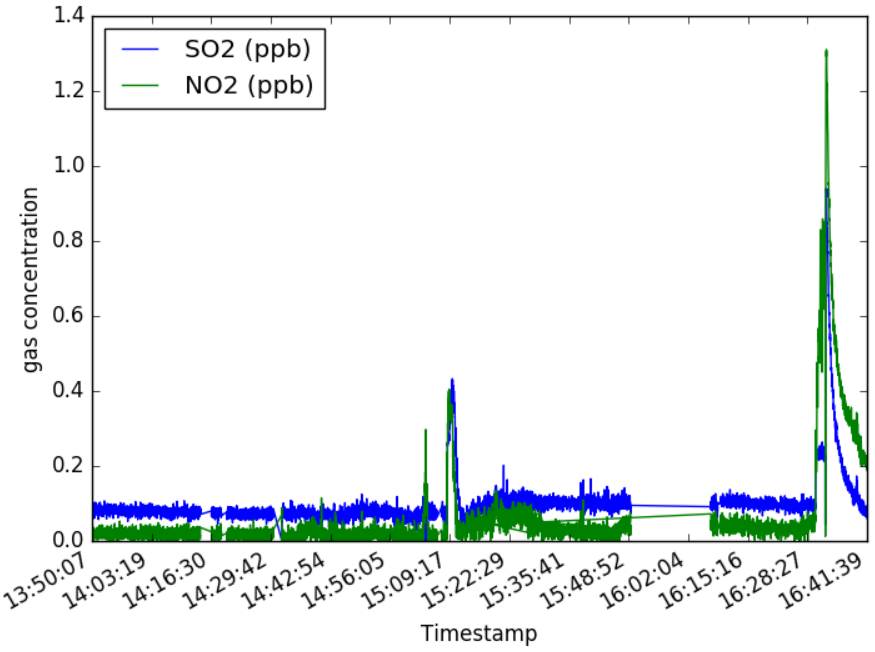


Figure 4.3: Graph showing the variation of Sulphur Dioxide and Nitrogen Dioxide over time during the preliminary testing - the peaks are recorded when the device was placed at the pipe exhaust of a bus and a car respectively.

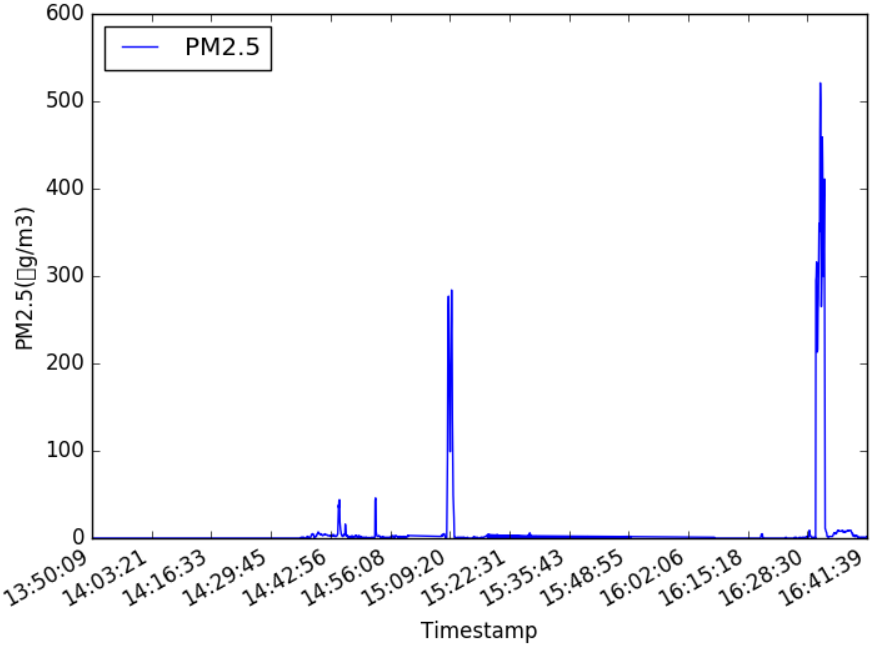


Figure 4.4: Graph showing the Particulate Matter 2.5 variation over time during the preliminary testing - the peaks are recorded when the device was placed at the pipe exhaust of a bus and a car respectively.

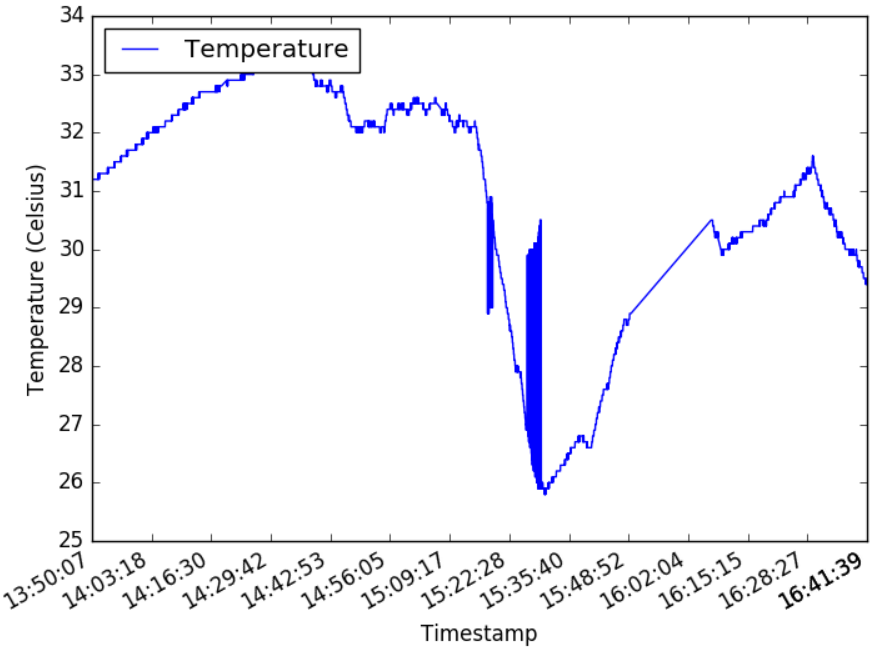


Figure 4.5: Graph showing the temperature variation over time during the preliminary testing

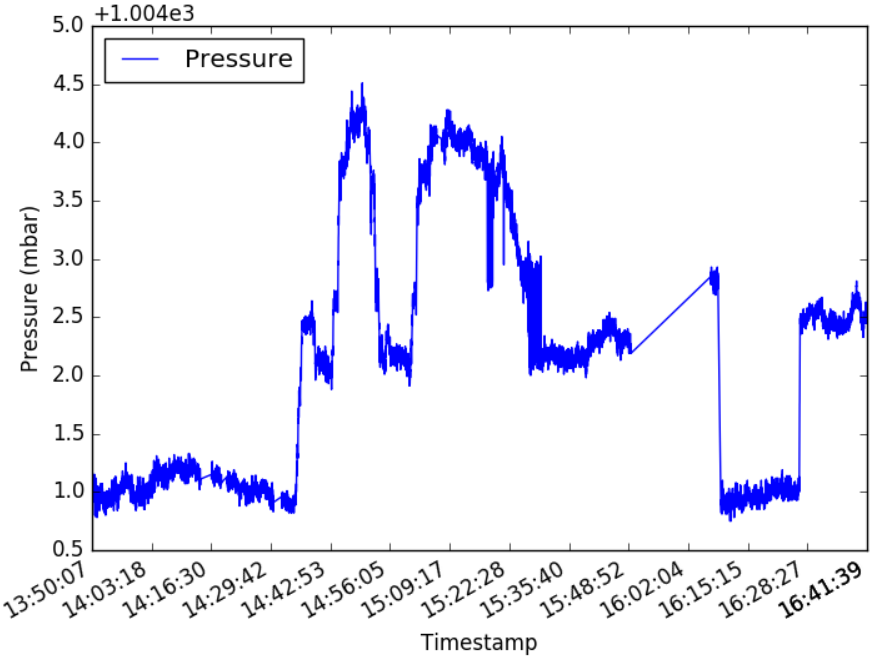


Figure 4.6: Graph showing the pressure variation over time during the preliminary testing.

outreach project at University of Southampton. The outreach project was short lived and did not handle any sensitive data. As such, security and privacy concerns were only considered in the context of future versions and larger scale deployments.

The data collected during this experiment is not of a sensitive nature and it has been collected during an outreach project which aimed to show the public the importance of science in our lives. Hence, loss of data and accuracy were not signifcantly important for the experiment. The two main security measures taken for this experiment were keeping the system behind the University network without any sensitive public endpoint and following good practices for server (and Raspberry Pi) configuration: changing default passwords, adding SSH keys for our user accounts, and only listening to the network addresses and ports we needed. A public URL for the online webpage was made available in case the demonstration device had to use mobile data to show the dashboard as opposed to eduroam. For the short-lived demonstration this proved to be sufficient. For a more permanent (or repetitive) deployment more measures need to be taken, such as encryption of the data in transit (to prevent both evesdropping but also inserting malicious readings into the system).

Data reliability can be improved for a larger deployment by using a reliable message broker that could handle queuing messages in the case that the network connection of the sensor node goes offline. This has only been handled in the demonstration by running two server instances, one on the sensor node (the Raspberry Pi) and one on a different server (see Figure 4.2). However, there was no synchronization between the two databases to handle missed or duplicate data points.

4.4.4 Outreach

The project was demonstrated at Engineering and Science day at the University of Southampton on the 18th of March 2017. We spoke to approximately 20 families (approx. 50 people) that participated at the event.

Families were composed of parents and their children in middle school or younger. Most of the parents seemed to understand air pollution, and the children, although too young to fully understand and relate to the implications of our project, seemed excited to interact with us, take part in our demonstration and try to understand what we are doing. Everyone seemed to be impressed with the whole event and enjoyed the activities and projects at the University.

Visitors found value in the Air Quality monitoring project, most of them mentioning how worried they were with the recent news on Air Pollution in the UK and how the project could help gain a better understanding of overall air quality within the city. They expressed interest in the visualisation software used for displaying data in real-time, the sensors, and the architecture of the monitoring device. The majority of people knew

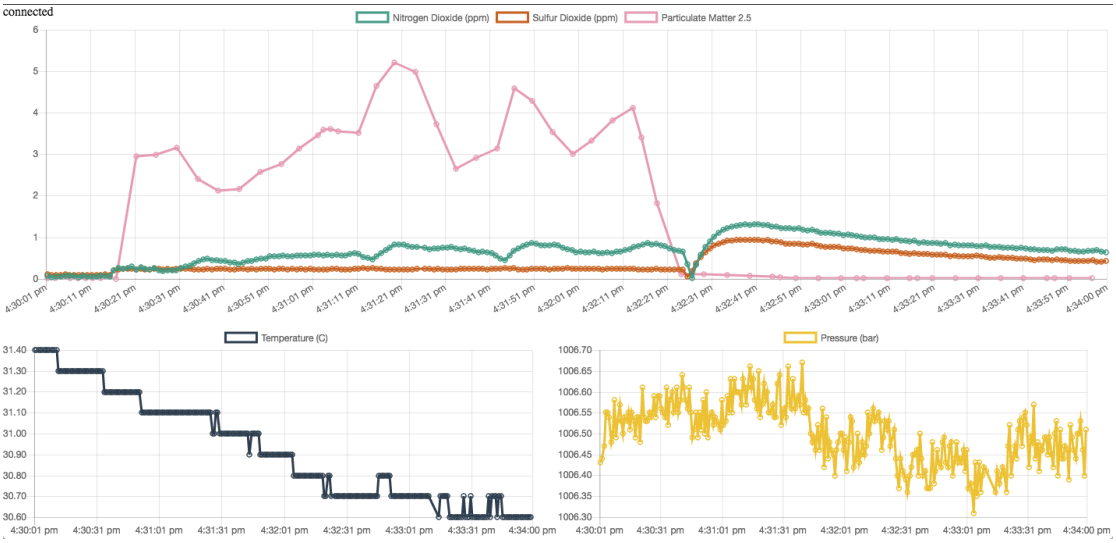


Figure 4.7: Example of the charts used in the preliminary testing; the data shows how the particulate matter readings rise when amount of dust particles in the air increases.

what a Raspberry Pi device is. Part of the visitors were able to identify current issues of Internet of Things devices such as power provisioning in restricted environments.

Children seemed to be impressed with changes in the data. For example, we asked them to find a way of generating dust particles, such as shaking their clothes or their chairs and follow the graph changes for air particle monitoring. Figure 4.7 exemplifies the data behaviour in one of these situations. We also asked the bus drivers to turn on the engine while we put the air quality monitoring device close to the exhaust (nobody was standing behind the bus at any moment while the engine was running) to show visitors how the Sulphur Dioxide, Nitrogen Dioxide, and Particulate Matter are increasing and the effect of vehicles on air pollution.

One of the most interesting conversations we had was with a teacher who found the project very interesting and thought that it could benefit schools as a citizen science project. She also thought that having kits for students to build their own air quality monitoring devices would be an excellent educational program.

Overall the whole activity was a success. Everyone seemed to enjoy themselves, and it was a good way for us to gain feedback on our work while trying to explain the impact the technology can have on society.

4.5 Lessons learned

In conclusion, the outreach bus demonstration was useful, showing that the air quality sensors readings are sensitive enough to be used for complementing the high-end air

pollution monitoring devices, that are not enough for a good understanding of the air quality. The sensor readings are not accurate enough to provide absolute values for air quality, but they can be used to detect changes in the environment. It was confirmed that the temperature and pressure variations have an effect on the quality of the sensors' readings. This experiment also showed that a more stable circuitry needs to be implemented.

The system presented in this work meets the extensible requirement, as new applications can be deployed using the existing sensors. Adding new applications allows extending the scenarios the device can cover. However, a device with a limited number of sensors, permanently soldered on the board, is not enough, as new sensors with significantly better qualities can be developed in the future, rendering such a device of limited use. Hence, updating the system in such circumstances would require updating the whole infrastructure, which implies that the customisation requirements are not met. The device needs re-designing due to the voltage irregularities illustrated in this chapter. In order to change the circuitry, the whole device needs to be changed, or a time-consuming process of de-soldering the components would need to be employed. Scaling this process is expensive and time-consuming. Hence the maintainability requirement is not met also.

All the work done so far has demonstrated the need for a modular system, where new technologies can be integrated during operation instead of building a device with a large number of sensors, which might not necessarily be needed or could become outdated. The literature review presented in Chapter 2 presents a method for achieving these requirements. Table 4.2 presents detailed methods for achieving the design requirements introduced in Chapter 3.

Table 4.2: Methods for achieving the design requirements - customisation, expansion, maintenance and accessibiliy- hw represents hardware and sw represents software

Level	**Method**	**Explanation**
Hw	Flexible, reliable hardware architecture	IoT is a fast paced area, which is why building devices that enable easy hardware upgrades would further its evolution.
Sw	Flexible, resilient OS	Most of the IoT OSs are designed based on the one-size-fits-all mentality or they are proprietary and cannot be customised easily. Neither of these options is optimal, and we need an OS that can be easily customised for each individual application it is meant to run and it should also be easy to modify as needed during operation.
Sw	Secure, robust OTA updates.	OTA updates are critical for a long-lived system as they provide essential bug fixes, security patches but also functionality upgrades (adding new features) and compatibility with newer software. Enables management of multiple versions of the OS simultaneously to aid managing an entire fleet of devices.
Sw	Separation between the OS and applications	Flexibility in application development and portability; eases deployment and management of applications.
Sw	Remote management of devices.	Enables monitoring the devices, installing updates, changing configuration parameters and installing, removing, starting or stoping containerised applications.

Chapter 5

Hardware architecture

In the previous chapters, we identified the design requirements and technologies required for building the next generation IoT device, Table 4.2. This chapter presents an implementation of this device and the required technologies at the hardware level. The next chapter explores modularity at the software level.

5.1 Introduction

Hardware is developing rapidly, and its availability increases. For example, the Raspberry Pi computer series, which was released on the market in early 2012, has impacted the IoT development significantly - giving birth to important IoT projects in the hobbyist world [74], academia [42, 68, 13] and industry [47].

Indoor positioning is another example of technology that has evolved rapidly, from using one sensor like WiFi or Bluetooth to using multiple ones, i.e., adding gyroscope and accelerometer. Networking technologies like LoRaWAN are offering large scale networking and positioning. This evolution has led to better, improved, and advanced IoT technologies and services and made older ones obsolete. Hence, in this work, we are presenting a device that can adapt to these fast technological advances in order to save time, money and to lead to the fast development of new IoT devices and services.

5.2 Motivation

Developments in hardware lead to new IoT products, which appear on the market at an increased rate - each of them with better performance than previous ones. Over the last five years (2014 - 2019), 11 models of Raspberry Pis have been released as part of five Raspberry Pi families: 1, 2, Zero, 3, and 4. Each release brings improvements such

as better processing power, new capabilities such as included WiFi and Bluetooth and, in the case of the Zero family, smaller size and lower cost. This hardware evolution is beneficial for IoT development, but it also has a negative impact on sustainability and increases the complexity of managing IoT devices. Every year, nearly 50 million tonnes of e-waste are produced globally, out of which approximately 40 million tones end up in landfill, and it is expected that by 2021 the annual e-waste volume will surpass 52 million tones [115]. A significant part of this e-waste is generated by IoT devices. IoT has a positive impact on sustainability, showing a strong correlation with 11 of the 17 Sustainable Development Goals (SDGs) [111]. IoT negatively impacts on responsible consumption and production, the 12^{th} SDG, [57]. The UN has proposed tackling this issue by creating a circular economy. The development plan for 2050 Smart City London emphasizes being able to re-use existing infrastructure instead of disassembling it, and installing a new one and using infrastructure for multiple purposes are two of the main goals of the project [56].

Based on the work done in the first steps of this project, we have identified the requirements and methods necessary for building IoT devices with an increased life span and reduced management complexity. The reference architecture presented in this work can be used at the base of a circular economy and reduce the negative impact IoT has in SDG 12.

5.3 Proposed Solution

Re-purposing and updating hardware is a big challenge that has not been explored enough in the literature up to this point. Although there are companies that are trying to build plug and play devices that essentially allow users to add, change and update hardware, they are still limiting the user to a small range of proprietary sensors that can perform specific tasks [88, 8].

The main component designed and built in this work is a plug and play development board, PiSEB, illustrated in Figure 5.2, that has a modular design, as depicted in Figure 5.1. It allows access to multiple interfaces, such as SPI, I2C, UART, via plug and play sockets. The sensors can be attached using JST connectors to any of these sockets. It also has connectors for a microcontroller and/or a Single Board Computer (SBC) - it currently has been used with Raspberry Pi and PyCom/Fypi, but it can be be used with other single board computers and microcontrollers with the same pinout or using an adaptor. This is the second iteration of the board and schematics for both versions can be found in Appendix A and Appendix B respectively. Additionally to the interfaces shown in Figure 5.2, the board has the following features:

- Wide range of voltage input 12V - 24V.

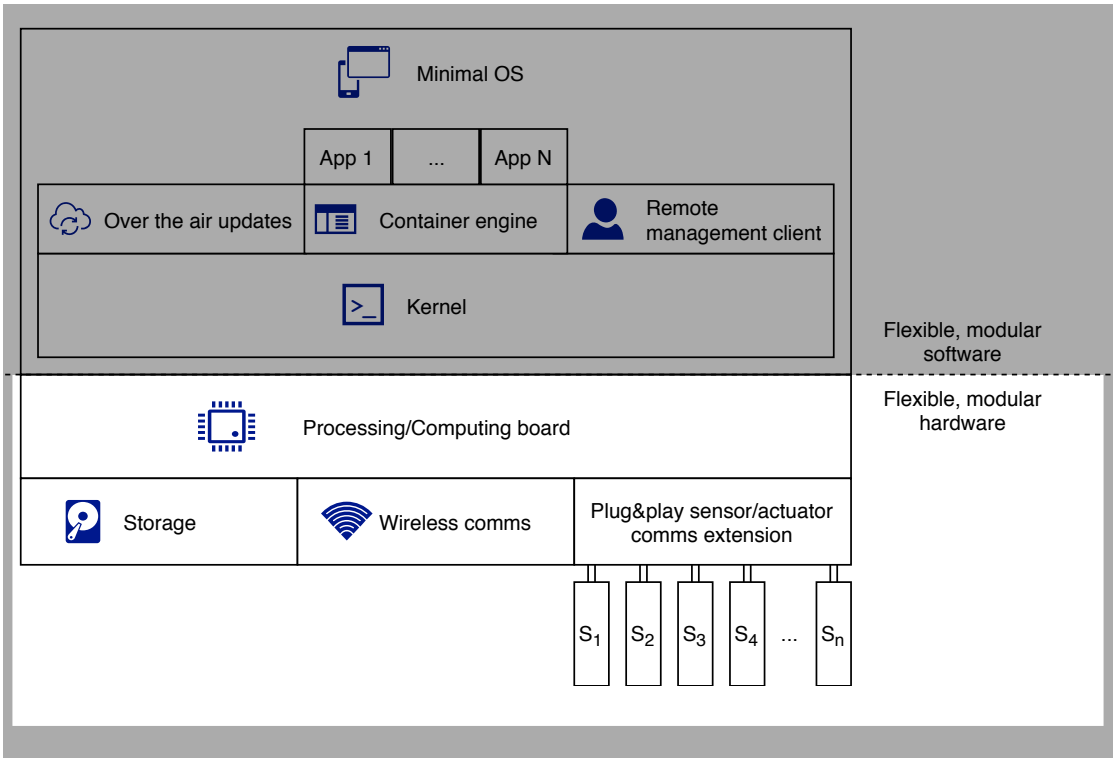


Figure 5.1: Reference architecture, highlighting focus of this chapter implementation, the architecture at hardware level.

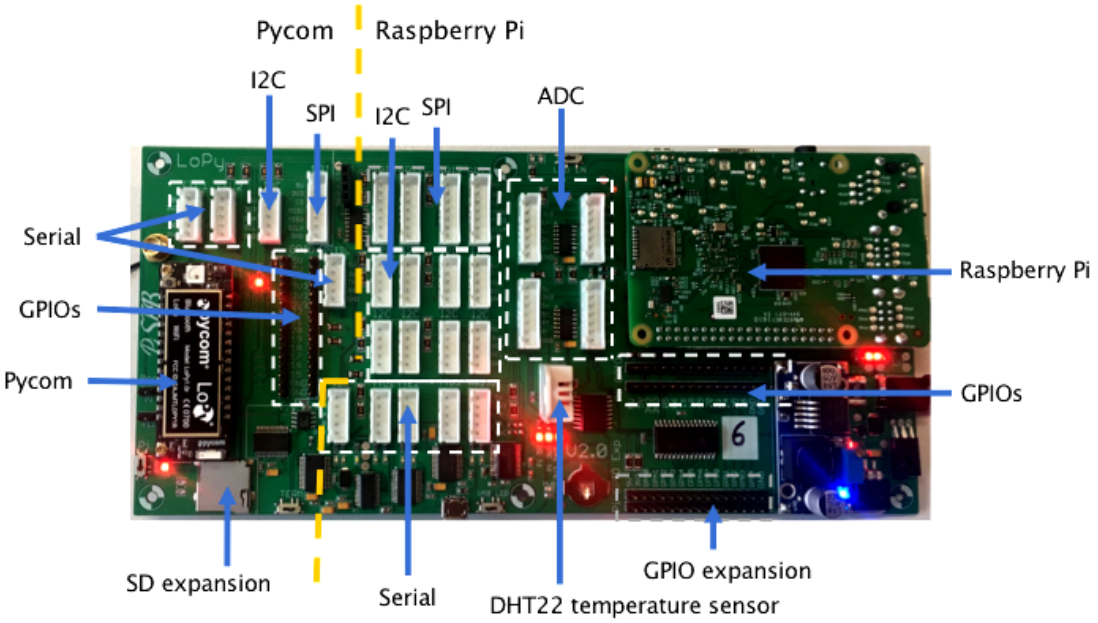


Figure 5.2: PiSEB development board [12] designed as part of this research, where the continous line delimitates the microcontroller (pycom) and SBC (Raspberry Pi) interfaces

- Reverse polarity protection.
- Remote control for the Raspberry Pi (via the microcontroller).
- EEPROM for each Raspberry Pi, adhering to the Hardware on Top (HAT) specifications [90].
- Debug/status Light Emitting Diodes (LEDs) that can be disabled to save power.
- micro-SD card extension for the Pycom microcontroller.
- On-board real-time clock.
- On-board DHT22 temperature and humidity sensor.

The board has been designed in Eagle, the schematics are open-source and can be accessed on Github [12] in order to allow uptake in the IoT community.

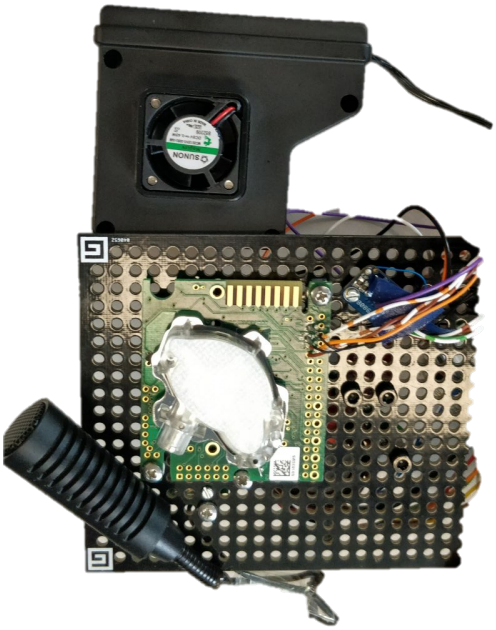
For this project, we have built and tested twelve of these boards with the purpose of developing environmental monitoring devices to be used across Southampton for air quality and weather monitoring. This is not the only use for this board, and it can be used for any application across a smart city, as there is no restriction on the input sensors. It can also serve as a testbed for IoT technologies, generate collaborations between researchers, and can be built by anyone in the world given its open-source availability.

5.4 Environmental monitoring example

The environmental monitoring device, shown in Figures 5.3a and 5.3b, has been built and tested in order to prove the applicability of the PiSEB board for IoT devices. The sensors used for this example are detailed in Table 5.1. An environmental monitoring use case has been chosen for this prototype due to the increasing concerns in today's society regarding air quality and weather.

The type of sensors used for this example has been chosen based on the literature review of smart city applications. The sensors are generic, widely available and can be used for a wide range of IoT applications, allowing face validation of different characteristics of the system architecture presented in this work.

The majority of these sensor models have been selected because they have been proven to work with Raspberry Pi, the chosen SBC for the prototype, and because they are easy to use and have available documentation and libraries. For communications, the Pycom FiPy has been chosen as it provides LoRa, WiFi, Bluetooth, Sigfox, and cellular LTE-CAT M1/NB1, which costs 62 USD at the time of development.



(a) Selection of sensors used in the generic environmental monitoring box, showing a particulate matter monitoring sensor (left), CO2 monitoring sensor (center) and a microphone (right) on the structure used inside the Stevenson cage.

(b) Atmospheric sensing housing for environmental monitoring device built as a use case for the reference architecture.

Figure 5.3: The hardware used for the environmental sensor prototype.

The Alphasense gas and particulate matter sensors have been chosen because they are popular in literature [31], provide good quality at a reasonable price, their readings have good accuracy. They can be used to complement the existing air pollution monitors provided by city councils.

The Raspberry Pi 3 Model B+ (Raspberry Pi 4 Model B was not released at the time of this work) has been chosen because it has the required specifications for this project. This is not necessarily the final SBC choice. Considering the rate of hardware advances during the last few years, building a device that can be upgraded once new hardware becomes available is essential, which is why we are designing the device presented in this work such that both hardware and software allow for upgrading.

5.5 Face validation - results and conclusions

The environmental monitoring device has been deployed on the roof of Building 176, Boldrewood Campus, the University of Southampton, in order to validate the design.

Table 5.1: The sensors chosen for the environmental monitoring IoT device, where the last three sensors were added after deployment; prices presented are the ones at the time of the writing.

Measurement	**Sensor**	**Price (USD, 2019)**
Distance	Seeed Studio De-Lidar TF02	149
Luminosity	Adafruit TSL2591	7
Pressure	Adafruit BMP180	10
Ultra Violet	Adafruit Si1145	10
Sound	Thomann t.bone GC 100	17
Images	Raspberry Pi Foundation Camera Module	30
NO2	Alphasense B4	55
Particulate Matter	Alphasense OPC-N2	277
Location	Adafruit Ultimate GPS	40
Temperature	Adafruit MCP9808	5
Humidity	Grove AM2302	16
Acceleration and Orientation	SparkFun MPU-6050	41

We used PoE for power and connectivity. The only use of the FiPy was to control the Raspberry Pi, e.g., rebooting when needed, but it is planned to be used in the future for LoRaWAN connectivity. The sensors were used out of the box for the purpose of this application, with the exception of attaching JST connectors, where needed.

The environmental monitoring device consists of two components: one stevenson cage with a dome on top for the air quality and weather monitoring sensors and a waterproof enclosure with the PiSEB board and computing units, as depicted in Figure 5.4.

The plug and play and modularity of the board allowed for extended testing of the sensors, and it eased upgrading the hardware, for example, upgrading from Pycom Lopy to PyCom Fipy. It also allowed for ease of building multiple versions of the device, where some of the sensors presented in Table 5.1 were omitted - these versions are not presented in this work as they have not been deployed yet.

The main advantage of the PiSEB board drawn by this experiment is the ability to field test sensors that otherwise would have been used for large scale deployments without the possibility of modification. For example, the Alphasense OPC sensors, although presenting important advantages and working reliably in laboratory settings, have been found unreliable during field testing due to incomplete readings and high power usage [31], hence they could be easily changed for more reliable sensors.

Another advantage presented by using the modular, plug and play PiSEB board was the ability to use a high number of sensors concurrently and explore power usage and computing capabilities, while experimenting with data collection rates and different sensors and interfaces, for example, the Alphasense can be used via USB or SPI, and using the

Figure 5.4: Reference architecture, highlighting focus of this chapter implementation, the architecture at hardware level.

PiSEB board has allowed us to understand that using the sensor with the SPI interface presents a more stable, reliable connection.

Three weeks after deployment, it was concluded that the on-board DHT22 humidity and temperature sensor was prone to failure due to environmental conditions and did not offer long-term reliability. The air quality sensor readings are influenced by humidity and temperature; hence the DHT22 sensor readings are essential for the presented application. As a result of these findings, new humidity and temperature sensors have been added to the device, as detailed in Table 5.1. The new sensors were added via the plug-and-play sockets available on the PiSEB board, in the Stevenson cage in order to obtain more relevant results for this application. The example illustrates the maintainability, and customisability of the architecture presented in this work. The device was modified after deployment to replace the faulty sensor with better-suited ones, similar or entirely different from the replaced sensor. Traditionally, the entire device is replaced for another one, which means that all the ten sensors, extension board, and computing modules become part of the e-waste, despite most of them being fully functioning. Other modular architecture systems are not flexible enough to allow changing a sensor for another model, usually having a limited range of sensors they can support. The modular architecture presented in this work saved 13 electronic modules, 100% of the functioning modules, and 92.3% of the total device electronic modules, leading to increased the lifespan of the device. For this exercise, an electronic module is defined as one sensor, one computing board, or the PiSEB extension board, and each of them can be replaced separately in case of failure.

Table 5.2: Summary of face validation tests and results. For this exercise, an electronic module is defined as one sensor, one computing board, or the PiSEB extension board, and each of them can be replaced separately in case of failure.

Design Requirements	**Test**	**Result**
Customisability	Temperature and humidity sensors upgrade to Adafruit MCP9808 and Grove AM2302 respectively	In-field testing and building of new technology and applications
Extensibility	An orientation and acceleration sensor was added to the device after deployment	66% less electronic modules needed to deploy new application
Maintainability	In-field replacement of DHT22 faulty sensor	Save 100% functional electronics and 92.3% of the total electronic modules
Availability	Board resources available open-source. Board plug and play design does not require expert knowlege to utilise. Any sensors can be used, and there are no vendor-specific requirements.	World-wide availability

An additional acceleration and orientation sensor has been added to the device using the plug-and-play board designed and built in this work, PiSEB. Testing and debugging without disrupting the existing applications or affect the robustness of the device. Adding a new sensor to the existing device allowed using existing infrastructure for building and deploying new applications. The implications of this extension are: saving time and costs, reduced deployment complexity, reduced management complexity. Traditionally, enabling new applications for smart cities requires new devices and infrastructures to be deployed - in the presented case, at least one computing device, one sensor board, and one sensor. The presented architecture reduces the number of deployed electronic modules by 66%.

In conclusion, the hardware design presented in this chapter, designed and built following the reference architecture proposed in this work, meets the design requirements (customisability, extensibility maintainability and availability) identified in this research. These findings are summarised in Table 5.2.

5.6 Known issues and limitations

The PiSEB has a wide range of interfaces in order to cover a wide range of sensors, but this can also impose limitations. The board can be too bulky for some applications, which can be solved by accessing the open-source files of the board and re-building it with a lower number of interfaces. On the other hand, other applications could require more interfaces and can be solved similarly using the open-source files.

The PiSEB is a generic board that tries to cover a wide range of sensors by providing plug and play accessibility to different interfaces. However, some sensors might need different interfaces, and as a result, they cannot be attached to the board.

The plug and play sockets are JST, but not all the sensors come with JST connectors built-in. Hence these might have to be attached before using the board.

The board allows for connecting a wide range of sensors, but electronic knowledge is required in order to use the board at its full capabilities, as having a large number of sensors within a device could cause errors due to wiring, interference, power usage, or computing power requirements.

The PiSEB can be used with any microcontroller and SBC. However, the pinout used for attaching these are the ones used by the Pycom microcontrollers and the Raspberry Pi, hence attaching something with a different pinout might require an adapter or building another board using the open-source files.

The PiSEB has been developed and built for minimising the complexity of adding new technologies to existing distributed sensors networks, but it still requires minimal physical intervention in order to update any of its hardware components.

The number of sensors that can be used within an application is also limited by the capacity of the computing unit.

Chapter 6

Software architecture

This thesis proposes using modular hardware and software architectures for reducing the complexity of managing distributed sensor networks in the era of constant technological advances, where building on top of current architectures needs to be made a priority in order to enable better management of these devices, but also to enable further technological enhancement of existing systems.

In the previous chapter, we discuss an open-source modular hardware architecture, that allows updating the computing unit and the sensors, and is not limited to any proprietary technology, answering research question 1. To complete this type of device design, in this chapter, we are developing a modular software architecture that can be used independently or concomitantly with the hardware architecture presented in the previous chapter of this thesis, answering the research question 2.

6.1 Introduction

The IoT can be the solution to many challenges in the current socio-economic environment. For example automatic water meter reading by driving a car on the street [136], mobile air quality sensors to get a better understanding of the air pollution in the cities and its origins [85], citizen science projects such as weather stations and air quality monitoring devices are just a few projects that have a high impact on society and are models of distributed sensor networks in restricted environments.

Deploying and managing such devices is a challenge, and problems related to the power supply, software updates, and recovery in case of failure can arise in restricted environments when we cannot easily access the device. This chapter is looking into finding ways of deploying and managing distributed networks of IoT devices. It presents a software architecture that is designed to be used for large distributed networks of IoT devices operating under restricted environments. The chapter explores existing tools that can

be used to design more reliable IoT devices and proposes an architecture that aims at reducing the complexity of managing smart devices and related emerging technologies.

Our architecture has three main modules: Over the Air (OTA) updates, Operating System, and containerisation. The solution proposed here is not revolutionary, but rather evolutionary. Others have introduced similar concepts, but they are either designed for cloud servers [109] or do not support popular embedded systems architectures [103]. Furthermore, our overall hardware and software architecture presented in, Chapter 5 and this Chapter, aim to encompass the whole modular architecture required.

The most similar architecture is that of [71]. Their solution has been tested at the initial stage of this project and has been proven to be less flexible and too complex for the purpose of this research, as described in chapter 2. For example, BalenaOS, although very similar to what we are building, cannot be easily integrated with other packages or modules other than the ones developed by the same company, specifically for their proposed architecture. The main similarities between BalenaOS and the OS built-in this work are modularity, minimal Yocto built OS, over the air updates, and containerisation. However, BalenaOS - although built with Yocto - is generally a few versions behind the latest stable Yocto development, and customising the factory-built is not straightforward, which restricts most of the users to use it as provided, along with their proposed solutions for either of the modules. This, although not contradictory to the modularity they are claiming, does not allow free choice for the technologies used in each of the OS modules. The over the air updates provided by BalenaOS are Docker-based (in the sense that the updater itself runs in a container) and use a dual image system. In this work, we have tried to integrate other OTA technologies with the BalenaOS (OSTree-based incremental updates), but this was proven to be too complex and time-consuming, requiring low-level modifications of Yocto layers to solve compatibility issues. The nature of the Balena built requires its own customised containerisation engine, which conflicts with any other type of engine a user might choose instead. All of these difficulties have proven that, although based on similar ideas presented in this thesis, the Balena technologies are not as flexible as required for our work, and the development based on their layer was aborted. A key difference between our solution and balenaOS is that our solution encourages modifying the OS if required and the incremental updates system enables complex fleet management with devices being able to run different versions of the OS on demand (and, for instance, changing between them at boot or having different OS branches for different sets of devices). Our containerisation module, albeit recommended for ease of deploying and managing applications, is entirely optional.

The architecture proposed in this chapter is based on open-source stable IoT tools that are already available and have been proven to be reliable for embedded devices. The main purpose of this project is to prove the need for modular hardware and software architectures for reducing the complexity of managing distributed sensor networks and

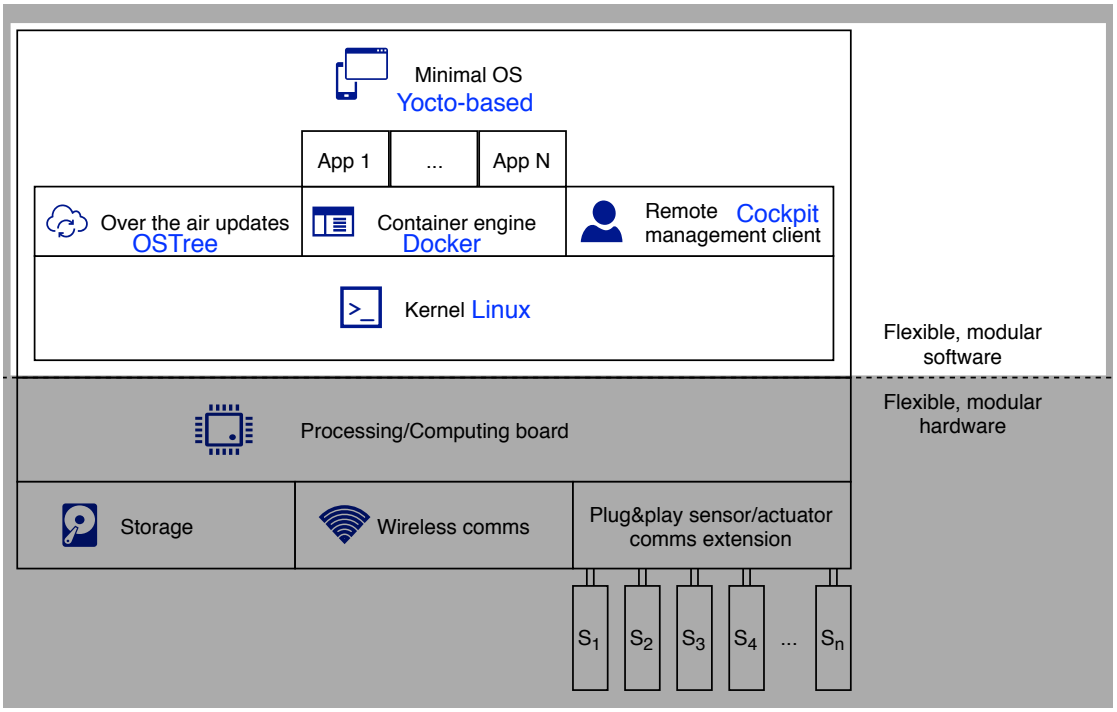


Figure 6.1: Reference modular architecture, showing the main technologies used for the software implementation. Please note that these technologies are examples, as the main advantage of the architecture presented in this work is not being locked to any specific technology.

ease their development, and the specific tools used in our applied example are just suggestions, while other tools might be more suitable for other projects.

6.2 Requirements

Table 4.2 identifies the technologies required for implementing a system architecture that meets the design requirements identified in this work and follows the reference architecture. This chapter focuses on the software level, as shown in Figure 6.1, presenting a method of implementing the reference architecture with applications in environmental monitoring. The specific technologies used for this implementation are just examples, and better or more suitable ones can be added at a later stage, as the main goal of the presented architecture is not locking a potential user to any specific technologies.

6.3 Operating Systems

The current OS distributions for SBC, such as Raspberry Pi, are not built for large scale deployments having increased size due to user-friendly features and lacking the required specifications for large deployments, such as reliability in case of power failure. Most

SBCs have constrained resources, which is why a large operating system with built-in libraries and applications, such as Raspbian, cannot be efficiently used for production. These Operating Systems (OSes) are made for out of the box operations, for exploring the capabilities of the boards and for ease of deployment for educational projects or the hobbyist community, which is why they contain libraries and binaries that are most of the time not necessary for the applications they are running when in production, increasing their size and complexity.

The Yocto project [125] gives flexibility for building customised Linux distributions. The Yocto project was developed based on the OpenEmbedded project and allows creating and modifying Linux distributions. The deployment can be done directly on the hardware or using SSH. The Yocto images are built using Bitbake. Bitbake is the task executor and scheduler used by the OpenEmbedded build system to build images. Bitbake also allows building multiple targets at the same time, where each target uses a different configuration. The image build can be done via the terminal or using the Yocto project web interface, Toaster. The project is popular, and it supports builds for a wide range of machines and software packages. However, using the Yocto project also has some disadvantages. Every change requires a new built, for example, adding a new package. Furthermore, not all Linux packages have Yocto layers or recipes, and some of them are unmaintained or have very specific requirements that lead to incompatibilities with a high number of other packages. For example, the *meta-virtualization* layer for building Docker on a Raspberry Pi is still under development and can cause problems during the building process or after. Furthermore, the bitbake compiler, as well as the more user-friendly compiler built for the Yocto Project Toaster, require a high learning curve. The error output logs do not offer enough information for efficient debugging, making the development process time-consuming and complex.

The biggest software component is a minimal OS built using the Openembeded Yocto project, the Rocko branch (latest stable at the time of the last upgrade of our stack; we regularly upgrade our OS development stack to the latest stable versions of packages and poky branches) of poky. Yocto gives advantages such as flexibility around building an application-specific OS. It allows adding capabilities such as read-only root filesystem and OTA updates, avoiding clutter, and keeping a minimal size OS. It enables building the same OS for different boards. At the moment, the OS built for in this work has been built and tested for Raspberry Pi 2, Raspberry Pi 3, Raspberry Pi 3 Model B+, and the Up-Board.

The OS can be used as provided or can be further customised to fit the requirements of the project it is being used for. The core capabilities of the OS are OTA updates, containerisation via Docker, and device management with Cockpit.

6.4 Remote device management

The Cockpit is a modular server management tool that can be easily extended via plugins. It provides a server-to-server (device-to-device) communication model based on SSH. It officially supports plugins for managing Docker containers, storage administration, network configuration, inspecting logs, and provides easy in-browser terminal access. For this work, a plugin has been developed and implemented in order to enable interaction with OSTree. Any of the devices registered in the Cockpit "network" can be configured to (passively) listen to an HTTP port. The Cockpit shows a web page that allows administrators to see all the devices set up with Cockpit.

The Cockpit client has been built into the OS using the meta-remote-management Yocto layer [10]. The meta-remote-management is an open-source layer that has been developed and implemented as part of this project. For this validation stage, the server has been tested and run on an internal server, in the first stage, and then it was moved to a cloud server.

6.5 Over the air updates

In Chapter 2, three types of OTA systems have been introduced: full image updates, incremental updates and container-based updates.

Full-image updates systems work by dividing the storage space into more partitions, usually two: A and B. One is the primary partition, which the systems run from, and the other one is used to download and apply an update when needed. If the update fails, the system falls back to the old partition. The downside of this type of system is the large bandwidth required for each update - typically downloading a full OS image with each update. Another disadvantage is the inefficient use of SD card space; 50% of the space is only used when applying a new update and cannot be used for applications or data. For our system, we initially tested the Mender full image updates but decided that we wanted a solution that uses minimal bandwidth and maximises the space available for data, such that it can be used in more restrictive environments also. While full image updates need high bandwidth for deployment, which can be challenging to achieve in restricted environments, they offer the advantage of entirely changing the Operating System, while the dual partition system can also be used in case of complete OS failure. The size of the update can restrict the pool of radio technologies that can be used for transmitting the updates. For example, LoRa has low bandwidth, but it has a high transmission range, between 20 km in LOS scenarios and 2 km in urban areas, which makes it a desirable technology for IoT sensor networks deployed in restricted environments [145].

Container updates refer to the system used to update container image, and they are ideal for application deployment and updates but not for the entire OS. We use Docker with a Docker registry to provide a way to deploy applications to a fleet of devices.

For the final version of our software stack, we use a system of incremental updates called OSTree for the OS, which provides a way to send OS updates to clients efficiently (only send deltas, not entire images) and be able to manage entirely different operating systems (or OS versions) on the server-side and the client. The clients (devices) have a read-only root filesystem and a (git-like) repository of OS versions that can be used to boot. The update system manages which OS version the device boots into, and this mechanism can be used for recovery in case of failure. When applying an update, new files and metadata are downloaded to the repository, but old ones are not altered, thus if an update gets corrupted, previous versions should remain usable. The main advantage of incremental updates is a smaller size, requiring smaller bandwidth. Rollback to the previous version in case of failure is also possible. However, with this type of update, we can only make changes to the current operating system, and there is no option for recovering in case of complete OS failure.

6.5.1 OSTree

The core capability of the system is OTA updates using OSTree. This allows the OS to be updated remotely even after the initial deployment, and also supports working on multiple versions (branches) simultaneously.

OSTree provides "git for operating system binaries" [149], designed to parallel install multiple independent bootable Unix-like operating systems atomically. It is comprised of a userspace content-addressed filesystem and an administrative layer that atomically installs the OS. It records and deploys complete bootable filesystem trees, but has no built-in knowledge of how a given filesystem tree was generated, the origin of individual files, dependencies, or descriptions of individual components.

OSTree-based updates have two components: a client and a server. Read-only OS trees are replicated on the client by atomically downloading contents from the server. System crashes or power problems during the update operation do not lead to system corruption; the system reboots into the old version or the new updated one.

At the client level, OSTree deployments rely on a top-level OSTree directory. Each device has an OSTree repository and a set of deployments. The deployments rely on hardlinks into the repository. Hence each update would only increase the OS size proportionally with the new files only, plus some constant overhead. Applying delta updates on Android apps can reduce bandwidth with up to 50% [126]. The OSTree kernel parameter sets which deployment the device boots. The device will boot into a chroot equivalent that points to one of the deployments. The OSTree-based filesystem is described in Table 6.1.

Table 6.1: OSTree-based filesystem

Path	Description
/usr	Read-only bind-mount content to prevent corrupting the repository. The main advantages of such systems are reliability and predictability.
/etc	Read-write directory. A deployment default configuration, which is in /usr/etc, is copied into the /etc directory inside the deployment. An update is performed by a three-way merge between /usr/etc, /etc and the /usr/etc of the new version of the OS. This way, new defaults do not override current configuration if it was changed, but new configuration files are copied normally.
/var	It is preserved across updates, shared between deployments with the same OSNAME. Typically /opt, /src, /mnt, /usr/local are symlinks to sub-folders inside /var/
/home	Indirectly symlinked to /sysroot/home (via /var/home), thus it is shared between all deployments.
/sysroot	Points to the physical sysroot.

OSTree also provides a C shared library that allows computing the contents of /usr directly on the client, assembles a new filesystem tree, and records it to the local OSTree repository.

The server-side can be a simple static HTTP(S) server that makes the bare/archive repository available. The OSTree client is then able to pull in the required metadata and files for updates. Static binary deltas can be pre-computed between specified versions to reduce bandwidth.

More advanced server software can be used. For instance, access to the repository can be restricted using an authentication mechanism such that only authorised devices can download the OS. Furthermore, access can be under fine-grained access control for different branches of the OS.

In the real-life application presented in this work, the OSTree client was built in the OS using the open-source Yocto layer meta-updater [6]. The OTA server was built, deployed, and tested on both an internal server and a cloud server. For simplicity, HTTP has been used at this stage, although for the large scale deployment, a more secure protocol will be implemented (i.e., HTTPS and authentication). A Cockpit plugin has been developed and implemented for monitoring and deploying the updates.

6.6 Containerisation

Whilst containerisation is not new, Linux Containers (LXC) has been used by companies such as Google for application deployment for almost a decade , containerisation became much more popular and accessible over the last few years, since containerisation tools such as Docker have been released.

In the architecture proposed in this thesis, containerisation is used for providing an application layer to the devices, as presented in Figure 6.1. All applications run in a container, and they are isolated from each other and the host OS, with appropriate access to the required hardware peripherals (sensors, cameras, etc.). This method of deploying applications allows for easy remote management, deployment, and updates but also enables the development of applications in simulated environments, which is ideal for rapid prototyping and automated testing.

The main advantages of containerisation are:

- Reliability - deploying applications as part of isolated systems makes the system less prone to failures, i.e., if an application fails, it will not affect the other applications on the localhost. It also eliminates software conflict problems, driver compatibility, and library conflicts.
- Security - having the applications isolated from each other minimises the risks of having the whole system compromised due to security issues at the application level.
- Rapid Deployment - once the application is built within a container, it can easily be deployed on any hardware running Linux distributions. This also makes upgrading the hardware very easy.
- Ease of updating - the applications can be easily updated or upgraded, simplifying the process of re-purposing embedded devices systems.
- Minimising the size of the host OS - having applications and their dependencies deployed in containers minimises the size of the main OS, which makes deploying OS updates over the air much easier.

Adding a containerisation layer does have a cost: the need to have a container runtime available in the host OS but also that each containerized application is larger in size (container image + application size).

Container updates can be made within the container only, and, using the correct tools, anything can be changed from binaries and libraries to the whole application. We have decided to use containerisation for application deployment and management, which is used on top of the main operating system and its independent updating mechanism.

6.6.1 Applications in containers

The OS is configured to install Docker by default. Docker was chosen due to its ease of use, availability, and the wide range of tools built around it. All the applications are running inside a Docker container and are deployed via a Docker image. This simplifies the development and deployment of applications and separates the core OS and dependencies from application-level code.

In this context, applications are programs that can interact with peripheral devices (such as sensors) attached to the host. For a device with a camera and a motion sensor, an example application is a program that reads values from the sensor and sends a request to a server when motion is detected; additionally, it records a short video and sends it to the server as well. Another application could analyse the video on the edge device in order to enhance privacy; for example, people counting [36].

This comes at the cost of having to support a container engine. Each application is larger in size due to being packed in a container image. For applications that share dependencies, this will result in having the same dependencies stored multiple times, once in each container. We consider that, besides the advantages listed above, avoiding dependency conflicts and offering the freedom of choice for programming languages and libraries, the ability to create and deploy applications that would work reliably on different host OS environments, and the ability to develop applications in very different environments than the deployed target, outweigh the disadvantages.

Each application container is stored in a Docker registry, located on the main server - in our case, the local and cloud servers, respectively. Each application can be tested before deployed in the field while updating an application adds only the new layers that are not present on the device. The applications can be deployed on any of the connected devices via the Cockpit manager, which allows deployment and monitoring of the application containers. In the implementation presented in this work, multiple applications that read and process data from the sensors have been tested. The data was then sent to the server and stored in a simple database. During this validation stage, other available technologies have been tested, such as Microsoft Edge Compute - which is based on containerisation. It has been decided that at this stage of development, this type of platform is not useful due to lack of robustness, i.e., an intermittent connection between server and client.

6.7 Server-side requirements and considerations

The system requires a few components to be offered on the server-side. These are a remote OSTree repository, a Docker registry to pull applications from, a server running Cockpit (not mandatory), and any components required by applications.

The basic server-side solution used during this validation stage for the OSTree remote repository is a static HTTP file server making the archive OSTree repository available at a URL. This allows devices to pull updates when they become available in the remote repository. This can be protected using an authentication scheme. The system has also been implemented and tested on cloud services such as Microsoft Azure and Google Cloud Platform. A standard Apache HTTP server or Nginx server can be used.

Deploying applications via Docker containers requires a Docker registry. Docker registry is an open-source project that can be easily deployed on a server. Alternatively, Docker Hub offers free hosting for Docker images that are made public and paid options for private storage. All major cloud services providers also offer a hosted solution. For this validation stage, different solutions have been successfully implemented and tested: local server, Google Cloud Platform Docker registry, and Microsoft Azure Docker registry. Although all of them have been successfully tested, it has been found that the most convenient implementation was the one on the local server. Cloud solutions had the benefit of providing a ready-built solution; however, in our testing, it seemed to add unnecessary complexity. We acknowledge that cloud-based solutions might be favorable to self-hosted in large deployments due to ease of scalability; there is, however, a cost in development time to set up the clients to work with the chosen cloud provider.

Running Cockpit on a Linux server and adding all devices in Cockpit from the interface was found the most convenient method at this validation stage. The server SSH public keys were copied to the authorised keys file on all the devices to avoid typing passwords. This made Cockpit on the devices is more easily accessible, and the devices did not need to support the Cockpit web server. A potential problem with this is key invalidation in case the server's private key gets compromised.

The applications running in Docker containers have their own server-side components, and there are many choices for how to implement data collection and processing pipelines. All major cloud providers have scalable solutions focused on IoT use cases - they provide advanced features like device provisioning, key rotation, and telemetry stream analysis. For small deployments, simple HTTP applications backed by a database would suffice. For this validation stage, both options have been successfully tested.

6.8 Power

The consumer SBCs highly available on the market are known for having power issues such as power consumption, power provisioning and are prone to power failures. At the moment, there is a lot of effort in trying to produce devices that are more power-efficient and reliable and to build batteries that last longer. Plus, using solar and wind power for IoT devices is a popular choice, especially when the devices are deployed in restricted environments. Power over Ethernet (PoE) is a popular choice for providing network

connectivity and power to IoT devices, but there is a limit to the amount of power the devices can consume when connected this way.

Both hardware and software are prone to power failure. For example, the Raspberry Pi can fail to reboot after a power failure when it runs the default Raspbian in the case that some of the configuration files get corrupted during power off.

In order to avoid this, we created a read-only root file system. This has been achieved using Overlay FS [51]. Overlay FS has been built into the Linux kernel since v3.18 and provides a mechanism to create a read-only root file system and then overlay a read-write file system on top. This can be built using Yocto.

Alpine OS was also considered for this work due to its small image size and read-only root file system [48]. However, it was not used due to the complexity of the process of adding new packages, like using a specific set of commands for making them permanent before the reboot, steps that could be easily overlooked during the development phase, as well as the small range of packages available to install in the Alpine package manager.

6.9 Implementation details - other technologies

We deployed a Mender server and built a client using the Yocto project that has been tested with Raspberry Pi 2 and BeagleBone Black. The complexity of building and deploying the system on the two devices is similar. The main difference between Raspberry Pi and BeagleBone Black is that Raspberry Pi supports a larger number of packages, while the BeagleBone built was limited to minimal capabilities due to incompatibilities between packages and number of features implemented. OS distribution for Pine64 can also be built using Yocto, but this is also limited.

Using Mender was a decision made based on availability, ease of deployment, and available documentation. Building the Mender client can be done via the Yocto project. The Yocto project only runs on Linux distributions and requires high storage memory for building images due to the number of files and directories created during the build. Building Mender into the OS distribution will increase the image size by 1GB. This is significantly high compared with the space the other packages occupied for the first client trial: 200MB.

Sending updates via Mender requires a new image to be built using Yocto. The image is compressed using a Mender tool called *mender-artifact* into an *artifact* that contains the new root filesystem and information about the update such as version and compatible devices. The artifact can be sent using the Mender server. The update is sent to the client, and once it has been installed onto the free partition, the device is rebooting using the new root filesystem. If the update is successful, the client communicates this to the server. Otherwise, if the client is not communicating with the server or the update is

unsuccessful, the client will rollback to the previous version at the next reboot. For the first test, we have successfully tested the Mender project by building the client and sending an update via the webserver.

However, due to the large size of the updates, it was decided that Mender was not suited for the type of application we are interested for this work. Hence, although it has been tested successfully, it was decided to be discarded in favor of incremental updates, OSTree.

For building, deploying, and managing applications, it was decided to use Docker as it is a very popular containerisation tool, very well documented with available tutorials, and can easily be integrated with other containerisation tools. Singularity was one of the other options we looked at, and we developed a package for the meta-virtualization layer in order to enable installing it using Yocto. Replacing Docker with Singularity is an option that will be further assessed in the future. We also deployed Docker Swarm to test if it can be used for managing clusters of devices. However, it was decided that not being able to control which device is performing the assigned tasks in swarm is not ideal for managing IoT devices, although it is an excellent tool for other applications, such as High Performance Computing.

The layer meta-cloud-services allows adding integration with popular cloud services, such as Azure, Amazon, and Google. Its capabilities have been tested during this research in the effort of using Azure edge, Azure Servers, and the analytics tools for our implementation. This is still work in progress at the moment.

6.10 Face validation - results and discussion

In order to validate the architecture presented in this work, the system has been built and tested as an environmental monitoring device presented in the previous chapter, Chapter 5. This section presents the results and conclusions from the software testing stage, summarised in Table 6.2.

6.10.1 Implementation details

The OS is a set of configuration files, patches, and recipes using the Yocto Project. We used the Rocko branch of poky and made a recipe called meta-remote-management, which contains minimum packages require for running the environmental monitoring device described in this work.

The OTA capability has been built using the *meta-updater* [6] layer. The meta-updater layer requires a board-specific layer on top, with the purpose of providing a bootloader

that is able to use OSTree's boot directory. It creates a fitImage which allows for packing the device tree and kernel together, allowing for both to be updated over the air, which would not be possible to achieve without additional update scripts if the normal configuration was being used, as the devices tree is typically on the /boot partition and overlays are configured in config.txt. This created some difficulties for the RaspberryPi kernel, which is typically configured with device tree overlays and adding special *dtparam* commands in config.txt. To edit the device tree parameters, we had to look up the source of the overlays and manually apply them in the original device tree source file. This is achieved with a patch appended to the linux-raspberrypi recipe.

The *meta-virtualization* [1] layer provides the containerisation technology, in this example Docker. Docker is not the only option provided by this layer, it also enables installing singularity and LXC, for example.

We also built our own layer, *meta-remote-management* [10], to enable the use of the Cockpit project, as well as providing application-specific support such as configuration files, patches, and the standard configuration offered in a distribution.

To make our layer compatible with more devices we included patches and configuration for the RaspberryPi in a separate layer, *meta-remote-management-raspberrypi* [11].

6.10.2 Experiments

Replacing the DHT22 sensor, as illustrated in the previous chapter, required relevant software modifications to be made using the Cockpit manager, OTA updates, and containers. The existing Docker application was modified and tested on a Raspberry Pi in the lab to ensure it was functioning as required before deployment. The relevant configuration files have also been built in the OS using the Yocto Project, and tested on a Raspberry Pi. The application was then updated on the Docker registry, and the new OS was also deployed on the server in the OSTree repository. The software was commissioned to the environmental monitoring device using Cockpit. The application image and the OS update sent only the difference between the versions running on the device and the newly modified ones. The architecture presented in this work allowed simple testing before deployment; the modular design has mechanisms that ensure the robustness of the system in case of failure. An application error does not affect other applications or the main OS. An OS update failure leads to automatic fallback to the previous OS working version. The read-only format of the sensitive directories also minimises vulnerability. The system logs and health are monitored remotely via the Cockpit manager. The example illustrates the maintainability, customisability, and extensibility of the architecture presented in this work.

At the software level, updating the OS and the running applications is traditionally done either physically, by changing the SD card, via full OS updates, or manually using the

package manager, which are limited by the available bandwidth and time-consuming. For example, on a generic embedded systems OS, such as Raspbian, each package update requires a full download of the package and required dependencies (if not up-to-date). Our experiments showed 50% smaller downloads than these traditional approaches, for example, updating the strace package to the latest version requires 932.16 kB download when using a traditional approach, but only 470.0 kB when using the implementation presented in this paper. This result agrees with other experiments in the literature [126]. Reduced update download leads to reduced costs due to bandwidth usage, saves time, and expands the possible deployment environments to more bandwidth constricted sites. A built-in computing board further minimises use cases and applications these devices can run. Devices that do not have delimitation between the host OS and applications are more prone to failure due to errors. The reference architecture presented in this paper is more efficient as application errors are contained within the application container and do not affect the rest of the system, while the OS has a read-only format for sensitive directories and it can be rolled back to previous versions. The system presented in this work allows testing and debugging before deployment and consistent images across multiple devices, increasing the robustness of the deployments.

Adding an additional sensor, such as the acceleration and orientation sensor addition presented in the previous chapter - Chapter 5 - required a new application to be built, tested, and deployed over the air using the software stack developed in this work to test the extensibility of the system. The application was deployed in a container, which allowed testing and debugging without disrupting the existing applications or affect the robustness of the device. Adding a new sensor to the existing device allowed using existing infrastructure for building and deploying new applications. The implications of this extension are: saving time and costs, reduced deployment complexity, reduced management complexity.

Increased hardware capabilities allow software stack developments that include new capabilities and better performance. As the software evolves, adds new features, fixes, and security patches, older versions become obsolete. Newer hardware capabilities and performance increases enable more possibilities in software, which is reflected in newer software releases. As a result, old devices/hardware becomes out-dated, unable to run new software applications or to run them sub-optimally. This leads to the decommissioning of these devices.

An environmental monitoring device built five years ago on a non-modular architecture with a Raspberry Pi 1+ might still have fully functioning components (in reality components might break). However, the lack of software support, limited processing capabilities, and other hardware limitations would lead to device end-of-life. The reference architecture presented in this paper allows efficient updating of the device components in order to increase its life span. For example, as new Raspberry Pi models become available, they can easily replace the older versions via the plug-and-play sensor board

Table 6.2: Summary of software face validation tests and results.

Design Requirements	**Test**	**Result**
Customisability	Over the air update to add new software package after device deployment	50% smaller downloads and reliable rollback capabilities
Extensibility	Contained software application deployed over the air to enable the use of new sensors	Robust testing and deployment of new technology
Maintainability	Cockpit device manager	Fault detection and resource maintenance
Availability	The software platform is built using open-source applications and tools. All the software developed as part of this research is available open-source.	World-wide availability

without the need for changing other electronic components. The software stack can be updated accordingly over the air to enable the hardware update, and new applications can be deployed remotely. The design of the system allows making these updates for less bandwidth, time, and cost. Using the Cockpit manager, multiple devices dispersed within the city can be updated remotely, to meet new requirements. These increase the life span of the devices and decrease management complexities. It also allows using existing infrastructure for the deployment of new applications, creating new opportunities for quick prototyping and rapid iterations of smart city applications.

In conclusion, the end-to-end system presented in this work proves that using the reference architecture developed in this work meets the design requirements - customisation, extension, and maintenance - allowing for developing devices with increased life span and reduced management complexity. The main specifications of the software stack developed as part of this research are:

- The system is open-source and does not depend on any proprietary technologies.
- The modules can be used independently from each other.
- The technology used for each module is non-restrictive and can be swapped for more suited one if needed.
- The applications and the OS have remotely and automatically updating capabilities.

- The applications can be run independently from each other.
- The applications are built such that they can be run independently of the devices and main OS.
- The system needs to be resilient. i.e., handle faults like power failure, file corruption, unreliable network, etc.
- Applications can be developed and tested without necessarily using the target devices.
- The whole fleet of devices can be easily managed and monitored remotely.
- Minimal OS that can be highly customised to adapt its capabilities or port it to new hardware.

6.10.3 Security and privacy

The security and privacy of this system was not the focus of this research project. This section aims to provide a list of suggestions for future work and clearly describe the security measures, limitations and issues of the project.

Software updates. The ability of the system to provide OTA updates is powerful and useful, but if not properly secure it can be dangerous. The current stack for software updates is a proof of concept, and it requires more security features for a large deployment.

The server which stores and serves the software updates patches to the clients must have strict controls of who can publish a new update. This can be achived simply by limiting access to the server (which it has been implicitly done), but in a larger organization more controls might be required, especially if this server is to be horizontally scaled.

The updates themselves must be cryptographically signed such that the client nodes can verify that the update has not been modified (maintain integrity) and that it comes from a trusted source. This has preliminary support and will be developed in the next stage.

The updates server must be using HTTPS (or other encrypted protocol) to serve the updates and, if required, provide appropraite access control such that only trusted clients can download software update patches. The latter can be implemented using machine-to-machine authentication, for instance using JWT tokens.

Node-to-server authentication. The network of devices that our system provides must not allow other devies to join in and pretend to be part of the fleet. This can be done by implementing a method of machine-to-machine authentication (ie. using JWT tokens as mentioned above, or SSH keys) but also it is required to have a way of provisioning authentication information the first time to each node. The initial provisioning

can happen at first boot by generating a set of SSH keys, printing out the public key fingerpring and asking the user to manually "trust" this fingerprint on the server. In the exemplar use case the MAC address of the clients are white listed for initial provisioning, instead of the public key fingerprint to allow to easily add and remove devices without user input and a screen.

Passwords and SSH access for large deployments. In the presented implementation, the password login for nodes is disabled by default and authorized SSH keys are pre-copied to each device at image build time. The keys can be updated via a system upgrade. This method provided the security features required without compromising simplicity. Manually keeping SSH keys up to date for a large deployment in a multi-user scenario is error prone. Another approach (after node-to-server authentication is setup) is to have the server maintain a list of authorized keys (one for each system user, and the server's public key such that server's Cockpit instance can see all nodes) that the clients periodically fetch. In this way, there is no need to use passwords for node access and no need to manually copy SSH keys to a large fleet. This has the key advantage of allowing each system user to manage their own SSH keys (similar to GitHub).

VPN The presented deployment used VPN to ensure no outside access was granted to the network. All nodes were deployed under the VPN, as well as the server and all users who required access to the system. This has been used as an extra security measure.

Containerized applications. Applications can be deployed to the nodes in docker containers, which is a good solution to decouple the environment of the system from the application environment and allows for much greater flexibility in application development. Containers also offer a strict environment for running applications. The current deployment, however, uses priviledged containers to facilitate access to peripherals from application level. This voids to some extent the restricted envrionment and allows application to access much of the host system. It is not a security issue in our case, where the applications were our own development. On the other hand, a much stricter environment is needed in a context where applications are developed by third parties (ie. a city may employ other companies for application development for their smart city infrastructre). There is a need for an extra software layer, one that provides controlled access to peripherals for third party applications. This is described at length in the next section, where the issue of concurrent access to the same peripheral by many applications is also discussed.

6.10.4 Known issues and limitations

In this subsection, we discuss the known limitations of our proposed system and implementation at the software level.

Yocto Project is a set of tools that allows creating Linux OS images for a variety of target devices. It enabled us to create our own OS that is configurable, extensible, modular,accessible and can be built for many devices. However, the suite of tools is somewhat complex and hard to navigate, having a steep learning curve. This results in users that intend to adapt our OS being forced to learn how to use many of the tools in the Yocto Project. Operations like building OS images, updates, and adding certain packages can be quickly learned by following instructions, but not all packages can be built using Yocto easily as they might not have already made recipes. Creating new Yocto recipes is similar to making new packages for different Linux distributions, but the process is rather involved, especially if not already familiar with the Yocto project.

On the note of usability of the system, containerisation and developing applications to run inside Docker containers could be another challenge faced by researchers not already familiar with the technology. However, Docker has extensive documentation and tutorials that should make this process straightforward, and it can prove to be a worthwhile investment given the growing popularity of containerisation.

Automated testing is routinely used in software development, but currently, our system has no mechanism for efficiently testing the OS (and other functionality like the update system) other than manually performing checks on target devices. A step towards mitigating this is to add support for the QEMU emulator, which will allow for developing a toolkit for automated testing and rapid experimenting with the whole system, including the update system and containerised applications. An emulator built can also speed up development work on the OS itself and other modules but adds the complexity of supporting an extra target. Also, there is no guarantee that modules that work on the emulator will run as expected on other devices, which is the reason this was not included from the beginning. This limitation does not apply to the containerised applications, which can be efficiently designed to be testable without using target devices.

In our current setup, there is a lack of separation between containerised applications and hardware peripherals. Applications running in containers are given full access to the hardware peripherals (sensors, actuators), and this is the current way of using these devices. Two problems arise. First, two or more applications may access the same sensor or actuator at the same time potentially creating unknown outcomes which would be very difficult to detect or debug (for example, two applications trying to rotate the same motor without knowledge of each other - at least one of them will get unexpected results). Second, there is no mechanism in place to guarantee strict access control to peripherals. This is best explained with an example: for a device that has a camera and a motion sensor, users might desire to grant access to an application to access the motion sensor but not the video stream. This can raise potential security and privacy issues: a vulnerability in any application can create a back door to the whole device.

Our current monitoring system, Cockpit, currently requires manually adding all the monitored devices via the user interface using their IP. All the devices must be accessible from the server that runs the user interface. Another limitation is that one needs to either type the username and password of these devices (same as used for SSH) or copy the public key of the server (web server machine) to the *authorized_keys* file of all target devices. Although not having a significant impact on small deployments, the process can be prolonged when using many devices. This can be solved by implementing the SSH key management method described in the previous section and making sure that all the IPs of clients are added to the server's Cockpit instance.

Currently, we use a simple HTTP server to make the OSTree repository available for target devices to access for system updates. This is easy to set up and sufficient for development purposes; however, it lacks a few features. There is no fine-grained access control, although this can be easily solved, for example, by password-protecting the server or running it behind a firewall. There is no separation between development branches of the OS and those that are meant to be deployed as updates, and there is no easy and documented method for *pushing* an update to the update server. The latter two issues can be solved by creating a new web service to replace the standard HTTP server for updates; it will have its own OSTree repository that it serves via HTTP for updates, but it will also support pushing new versions or branches of the OS to it, similar to how a git bare repository works.

It is currently difficult to make specific low-level configurations for some target devices like the Raspberry Pi. The Raspberry Pi device can be configured during the boot process by using Linux kernel overlays and special commands in the *config.txt* file from the boot partition (*dtparam* and *dtoverlay*). Adding or removing these dynamically configures the Linux kernel to load (or omit) specific overlays and settings. Our build does not use the overlay system and instead compiles the device tree and kernel image into a single *fitImage*. Configuring the Raspberry Pi device (and others that use similar systems) must be done by patching the Yocto recipe for the kernel (linux-raspberrypi for the Raspberry Pi). There is no simple workaround to address this issue, and we accept this as a known limitation.

We have focused our work on the end device software and hardware. The server-side counterparts to our system are basic and formed by many parts that are not integrated with each other. The cockpit web server simply needs to run on a Linux machine. The update server only needs to expose the OSTree repository, and a Docker registry is required for making (private) Docker applications available to download on the end devices. This simplistic approach has its benefits, namely the fact that any of the subsystems can be used independently of each other and that it is easy to set up using standard tools. However, a more integrated solution can enable more enhanced device management features and data collection as well as offer the possibility to add more layers of security and access control for administrators. Additionally, it can be released as an easy to set

up all-in-one solution that can save time for new users and make sure all modules are functional out of the box.

Chapter 7

Conclusions

The work presented in this thesis has been generated in order to find the requirements for building IoT devices that are adaptable to new applications and technologies. Chapters 3 and 4 focus on finding these requirements and answering the research questions 1 and 2. The next chapters introduce new technology developed based on the design requirements - customisability, extesibility, maintainability and availability - testing its adherence to the requirements as well as its feasibility for in-field deployments.

This work gives an overview of the research done for designing, building, and testing an IoT device that is customisable, extensible, maintainable and available. The main research contributions of this work are:

- Reference architecture, introduced in the first chapter of this thesis, Chapter 1. The main advantage of this architecture is the end-to-end characteristic, which introduces both hardware and software modularity. This approach to designing and building IoT devices is not explored enough in literature and provides more flexibility around building durable IoT devices that can withstand the high rate of technological advances in the IoT area and can be sustainably maintained.
- Design requirements, identified in Chapters 3 and 4 for designing and building IoT devices with an increased life span and minimal added management complexity for adding new technologies and extending deployed infrastructures: accessibility, customisability, extensibility, and maintainability.
- Method - presented in Chapter 5 and Chapter 6 - which introduce open, modular sensor board and software platform respectively. These have been used for implementing the reference architecture as an environmental monitoring use case. The implementation is an example, and each of the technologies and modules used are not final, where each of them can be modified and adapted after deployment when improved, better-suited, or more cost-effective solutions become available.

Increasing the lifespan of IoT devices requires modular architectures that allow updating after deployment. The next-generation IoT device allows swapping, adding and removing sensors (or other peripherals)

During this research, we have developed and built a hardware extension board, PiSEB, that can be used with off the shelf sensors, actuators, computing boards, and microcontrollers. This board has been implemented and tested using a wide range of sensors required in a real-life scenario, environmental monitoring, proving its adherence to the design requirements. Face validation tests include adding and replacing sensors in order to expand the device capabilities for sensing environmental factors, as well as maintaining the device. The sensors used for the example presented in this work were chosen based on availability, while manufacturer and communication protocols did not impose any limitations on the sensors selection process.

In order to provide a complete implementation of the reference architecture and enable the type of high-level flexibility required for the next generation IoT device, a software stack adhering to the design requirements has also been developed. It consists of three core modules: minimal OS built using Yocto, incremental OTA updates powered by OSTree, and system monitoring using Cockpit. The modules are interchangeable and not depending on each other, and the OS is highly customizable with Yocto. This has been successfully tested with Raspberry Pi 2, Raspberry Pi 3, Raspberry Pi 3 Model B+, the Up-Board, and Odroid C+. The software stack has been implemented and tested as part of the real-life environmental monitoring example. The tests have proven that the software stack enhances the device architecture by enabling hardware and software updates for sustainably extending the life of IoT devices. The resulting device has the necessary capabilities for allowing further expansion and serves as a working example and a starting point for new projects.

One of the most significant contributions of this work is the end-to-end modular, open system reference architecture, which, unlike other related work, includes both hardware and software levels. Modularity at both the hardware and software levels increases granularity and offers a complete solution for optimally meeting the design requirements to increasing device lifespan while reducing management complexities. Libellium and Balena are using similar architectures at hardware and software levels, respectively, but their work is not extensible, customisable or end-to-end (both hardware and software).

A significant advantage of the system presented in this work is the open-source aspect, as well as not locking a potential user to specific technologies. Due to this, the implementation and integration of the architecture presented here to specific applications are less complex and time-consuming. This also allows collaborations between different parties and eases the re-use of deployed infrastructures.

Another main advantage is modularity. Each module is of the system is independent, which, unlike other architectures, allows the users to chose which modules are necessary and what technology is a better fit for their use-case. This also enables further development after deployment, such as adding or updating components, allowing new technologies to be built upon existing infrastructures, as well as maintaining optimal operations and continuous adherence to the state-of-the-art. This results in an increased device life span and a significant drop in deployed devices.

The system architecture presented in this work is generic, which is very different from other ones in the literature, which limits the user to a specific use-case. This enables collaborations between different parties, but also device re-purposability during operation.

The architecture presented here details both software and hardware requirements for developing robust, next-generation IoT systems, which solves scalability problems, reduces devices management complexities and e-waste as well as enabling the next generation IoT innovations.

At the software level, the main disadvantage is given by the complexity of running the Yocto Project. The sensor board designed and built for this research, PiSEB, is generic but has a limited number of sockets and supports a limited number of interfaces. The hardware board presented here is an exemplar built for proof of concept. The main components of this work are open-source to allow uptake in the IoT community.

In this thesis we have researched and developed an open modular architecture for the hardware and software required for Internet of Things sensor systems. This opens the door for new robust, resilient, and cost effective ways to monitor and study the world in which we live.

Chapter 8

Future work

The next steps are to continue iterating on the IoT generic device, build, test, and deploy it at scale. The focus will be on covering some of the known limitations, testing the system with a deployment of 100+ sensors around the city of Southampton. As part of this deployment, extra tools will be developed to help manage the whole fleet, including server-side software for device management and data collection, but also software development tools for creating OS updates faster and aiding the development and testing of containerised applications.

The immediate next steps are:

- Developing a deployed device meta-data storage on the server-side. Each device has a unique identifier, but allowing filtering, searching, and deploying updates and applications to groups of devices filtered by meta-data stored about each device would ease the management of large fleets. Such metadata can include the geolocation of the deployed node, list of sensors available, date of last hardware maintenance or check, os versions and other application-specific data points (for example, whether the device is in a typically crowded location or a quiet residential area, or whether there are other restrictions in place due to local laws and regulations and deployment spot). This can be implemented as a plugin to Cockpit.
- Stress testing the devices in simulated constrained environments is also a step we are planning to take in the future. This includes controlled random power and network outages in critical moments of the system operation: during reboots, system updates, application deployments, etc. Finding weak spots in our software and hardware solutions will enable us to find mitigating solutions.
- Using Long Range Wide Area Network (LoRaWAN) for sending and receiving data because it has low power and wide range, which makes it a very attractive technology for IoT devices, especially those deployed in remote areas where other

networks might be out of reach or too expensive to setup. Deploying updates, applications, and real-time device management will not be available for devices to connect via LoRaWAN, but developing a solution that allows managing these devices with intermittent high bandwidth networks is feasible.

- Testing the devices with many sensor configurations to find whether there are incompatibilities or if the devices anyhow become unstable due to too many peripherals connected (e.g., CPU load too high, power issues, bandwidth issues via the peripheral interfaces, network bandwidth issues or data storage limitations), and how to mitigate for these situations.
- Using the system in a real research project which will allow us to write comprehensive documentation for new users, tutor other researchers into how to use our systems, and learn which areas need to be improved in terms of accessibility and ease of use.

In this architecture, data processing was considered to be addressed at application level. However there are data-related challenges that can solved at architecture level via auxiliary services. For instance

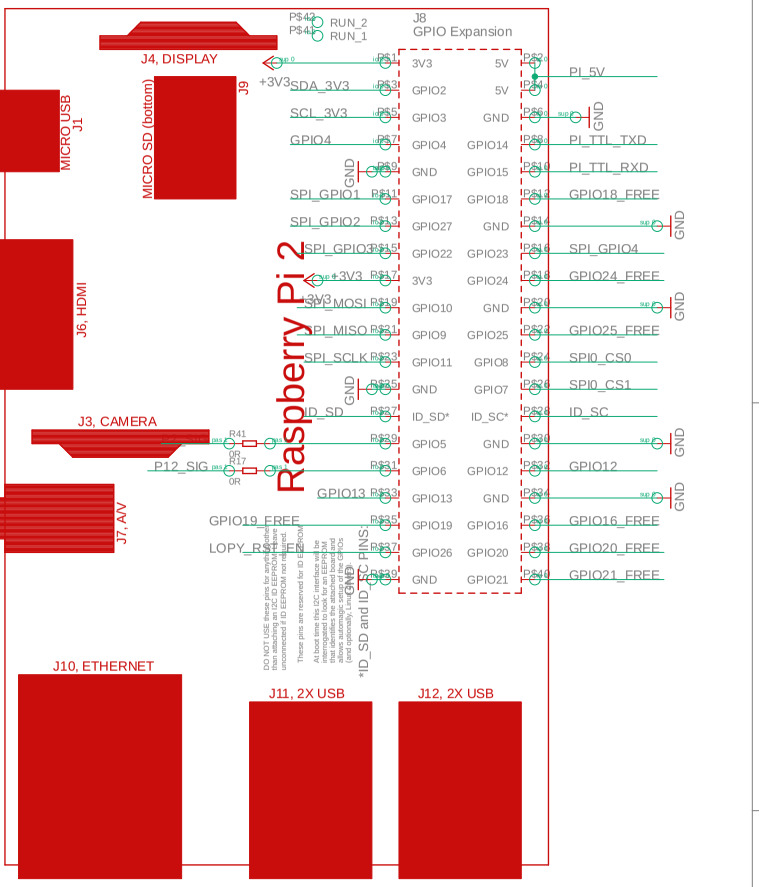
- Persistent local data storage to account for possible network connection loss. This can be achieved by creating a data partition with strict write accessibility, where only a specific application or user can write data. This way, each application has a dedicated writeable location.
- A broker has to be used in order to ensure that there is no data loss due to poor system malfunction or loss of network connectivity. This is a common problem for most applications and can be provided as a service on the client nodes, as opposed to allowing each application to handle this separately.
- Data has to be sent and stored securely, which can be achieved by using encrypted data transmission protocols and encrypting data at rest, but also by ensuring applications don't have access to each other's data.
- Part of this work explored existing services that offer data storage and processing. Their compatibility with the software stack presented in this work has been tested, but more work is required to ensure good functioning at large scale. As a result of these tests, Microsoft Azure, Amazon Web Services and Google Cloud Platform will be considered for this step.
- A generic solution for streaming data from sensors to servers can be provided to aid and speed up development of applications but also ensure best practices for security and privacy are followed.

This future work will be done as part of the EPSRC Doctoral Prize program that I will start after finishing my Ph.D. candidature.

Appendix A

PiSEB v1.0 Eagle Schematics

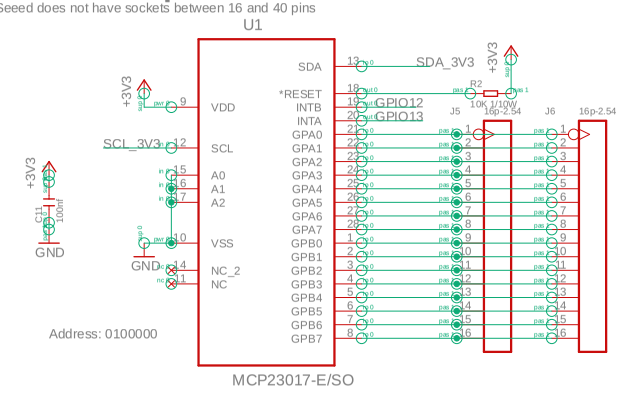
This appendix includes the Eagle Schematics of the first version of the PiSEB board.



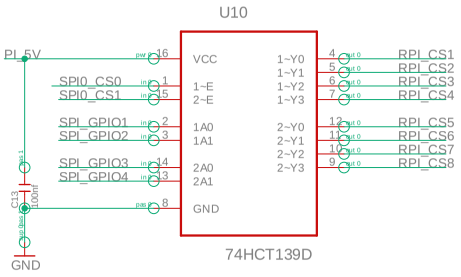
Port Expander

Added new header with free rpi gpio and 5V, 3V3 and GND

Seeed does not have sockets between 16 and 40 pins

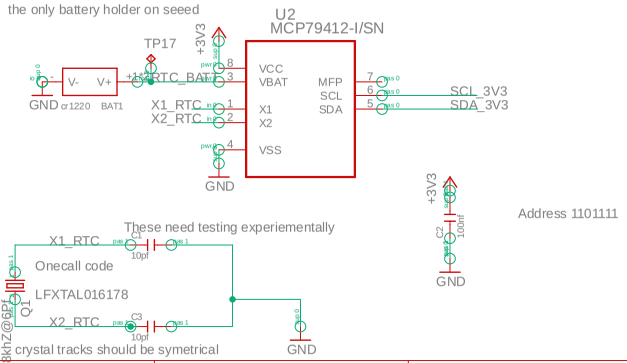


SPI Mux

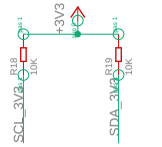


RTC

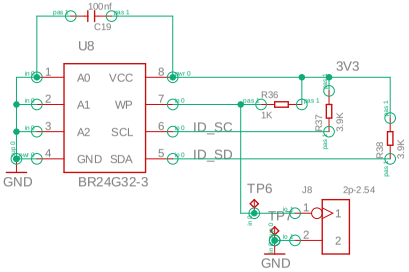
Do we want this permenantly powered given the Pi can be switched? What's the battery life like?



I2C Pullups

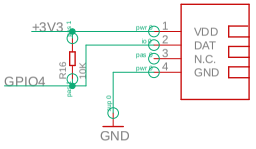


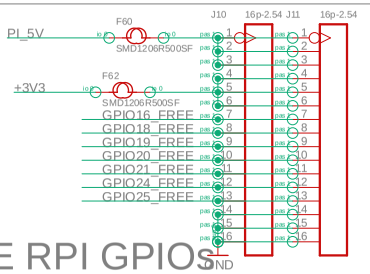
RPI has 1.8K pull up resistors for i2c



ID EEPROM

DHT22

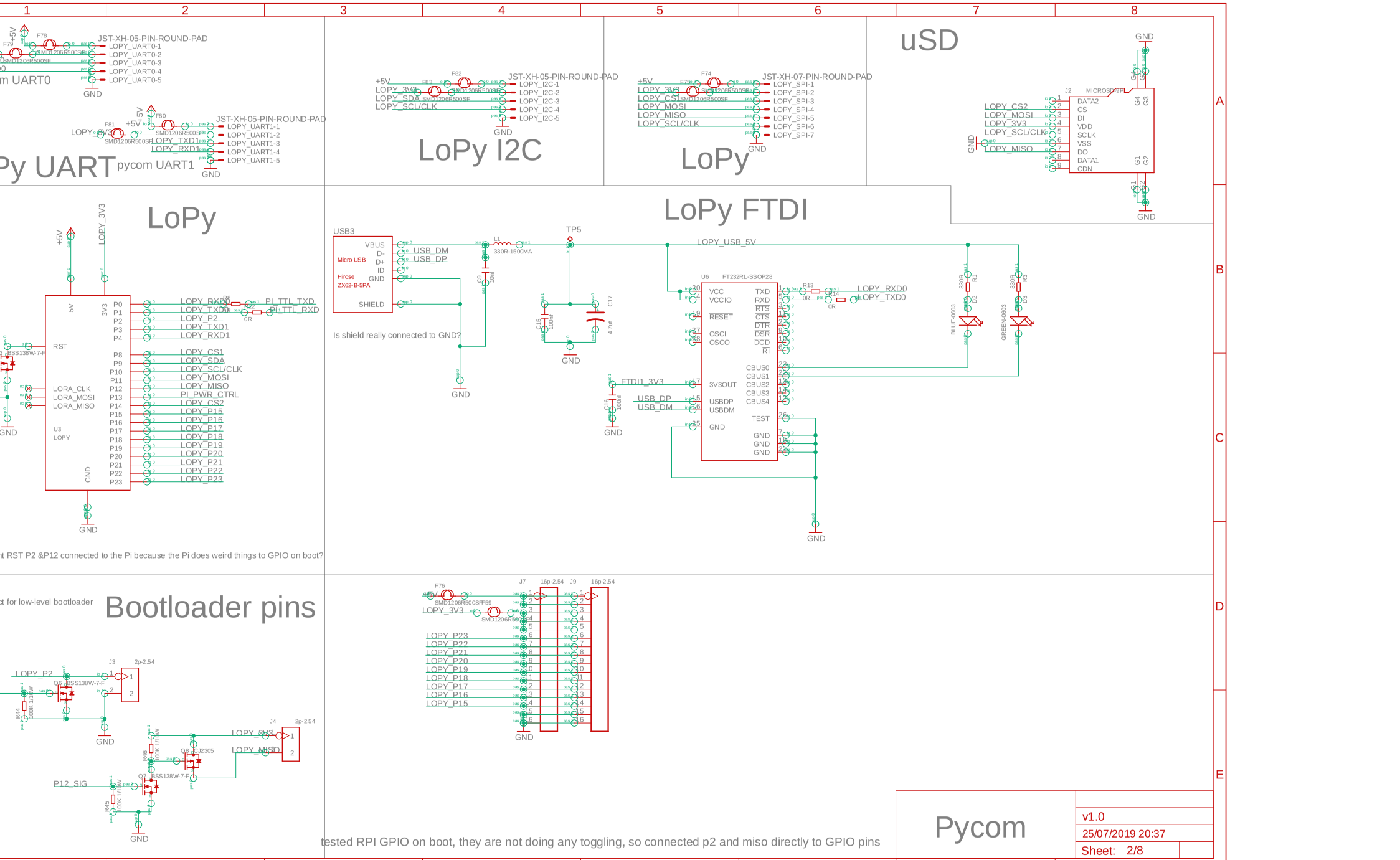




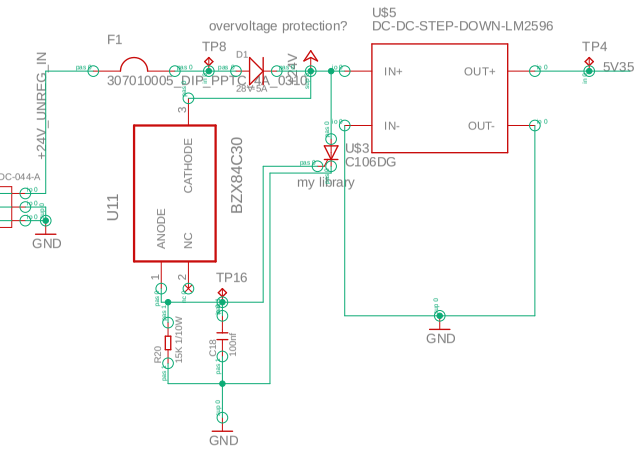
FREE RPI GPIOS

Pi & Digital

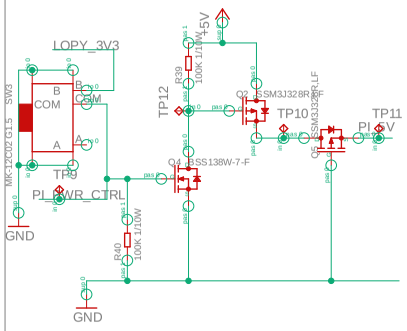
v1.0	
25/07/2019 20:37	
Sheet: 1/8	


JST-XH-05-PIN-ROUND-PAD
LOPY_UART0-1
LOPY_UART0-2
LOPY_UART0-3
LOPY_UART0-4
LOPY_UART0-5
m UART0
GND
JST-XH-05-PIN-ROUND-PAD
LOPY_UART1-1
LOPY_UART1-2
LOPY_UART1-3
LOPY_UART1-4
LOPY_UART1-5
LOPY_TXD1
LOPY_RXD1
Py UART
pycom UART1
JST-XH-05-PIN-ROUND-PAD
LOPY_I2C-1
LOPY_I2C-2
LOPY_I2C-3
LOPY_I2C-4
LOPY_I2C-5
LOPY_SDA
LOPY_SCL/CLK
LoPy I2C
JST-XH-07-PIN-ROUND-PAD
LOPY_SPI-1
LOPY_SPI-2
LOPY_SPI-3
LOPY_SPI-4
LOPY_SPI-5
LOPY_SPI-6
LOPY_SPI-7
LOPY_CS1
LOPY_MOSI
LOPY_MISO
LOPY_SCL/CLK
LoPy
uSD
MICROSD_SPI
DATA2
CS
DI
VDD
SCLK
VSS
DO
DATA1
CDN
LOPY_CS2
LOPY_MOSI
LOPY_3V3
LOPY_SCL/CLK
LOPY_MISO
LoPy
LOPY_3V3
RST
LORA_CLK
LORA_MOSI
LORA_MISO
LOPY_RXD0
LOPY_TXD0
PI_TTL_TXD
PI_TTL_RXD
LOPY_P2
LOPY_TXD1
LOPY_RXD1
LOPY_CS1
LOPY_SDA
LOPY_SCL/CLK
LOPY_MOSI
LOPY_MISO
PI_PWR_CTRL
LOPY_CS2
LOPY_P15
LOPY_P16
LOPY_P17
LOPY_P18
LOPY_P19
LOPY_P20
LOPY_P21
LOPY_P22
LOPY_P23
nt RST P2 &P12 connected to the Pi because the Pi does weird things to GPIO on boot?
LoPy FTDI
USB3
Micro USB
Hirose
ZX62-B-5PA
VBUS
D-
D+
ID
GND
SHIELD
USB_DM
USB_DP
Is shield really connected to GND?
330R-1500MA
TP5
LOPY_USB_5V
FT232RL-SSOP28
VCC
VCCIO
RESET
OSCI
OSCO
3V3OUT
USBDP
USBDM
TXD
RXD
RTS
CTS
DTR
DSR
DCD
RI
CBUS0
CBUS1
CBUS2
CBUS3
CBUS4
TEST
FTDI1_3V3
BLUE-0603
GREEN-0603
ct for low-level bootloader
Bootloader pins
J3
J4
2p-2.54
16p-2.54
LOPY_P2
P12_SIG
LOPY_3V3
LOPY_MISO
tested RPI GPIO on boot, they are not doing any toggling, so connected p2 and miso directly to GPIO pins
Pycom
v1.0
25/07/2019 20:37
Sheet: 2/8

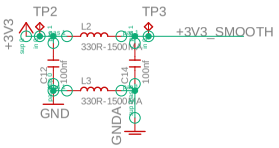
DC&DC



Pi Power Control

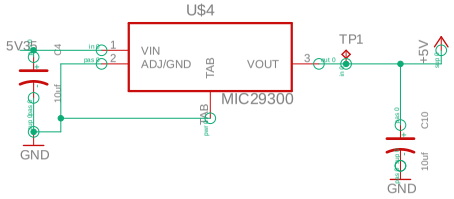


Pi filter for 3V3



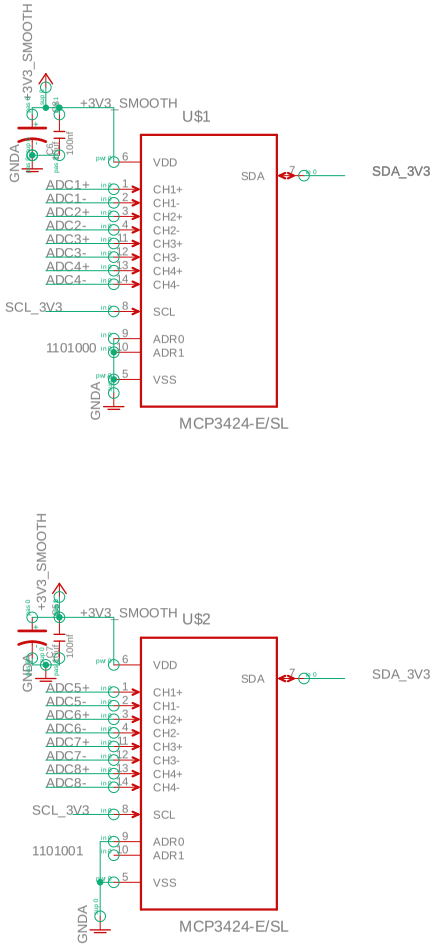
LDO

TP4 & TP1 should be close together

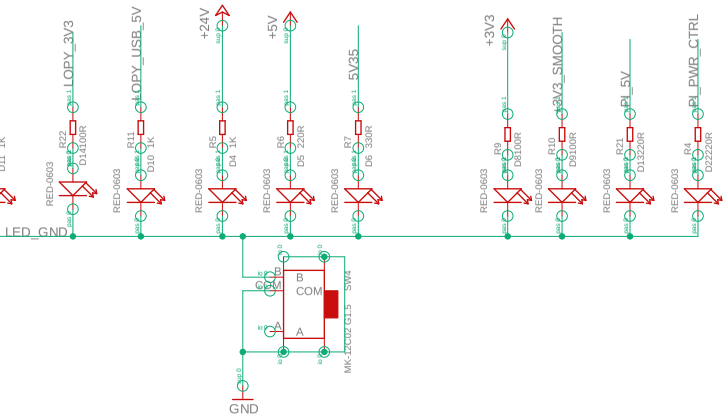


Needs good ground plane for termal issues

PSU	v1.0
	25/07/2019 20:37
	Sheet: 3/8



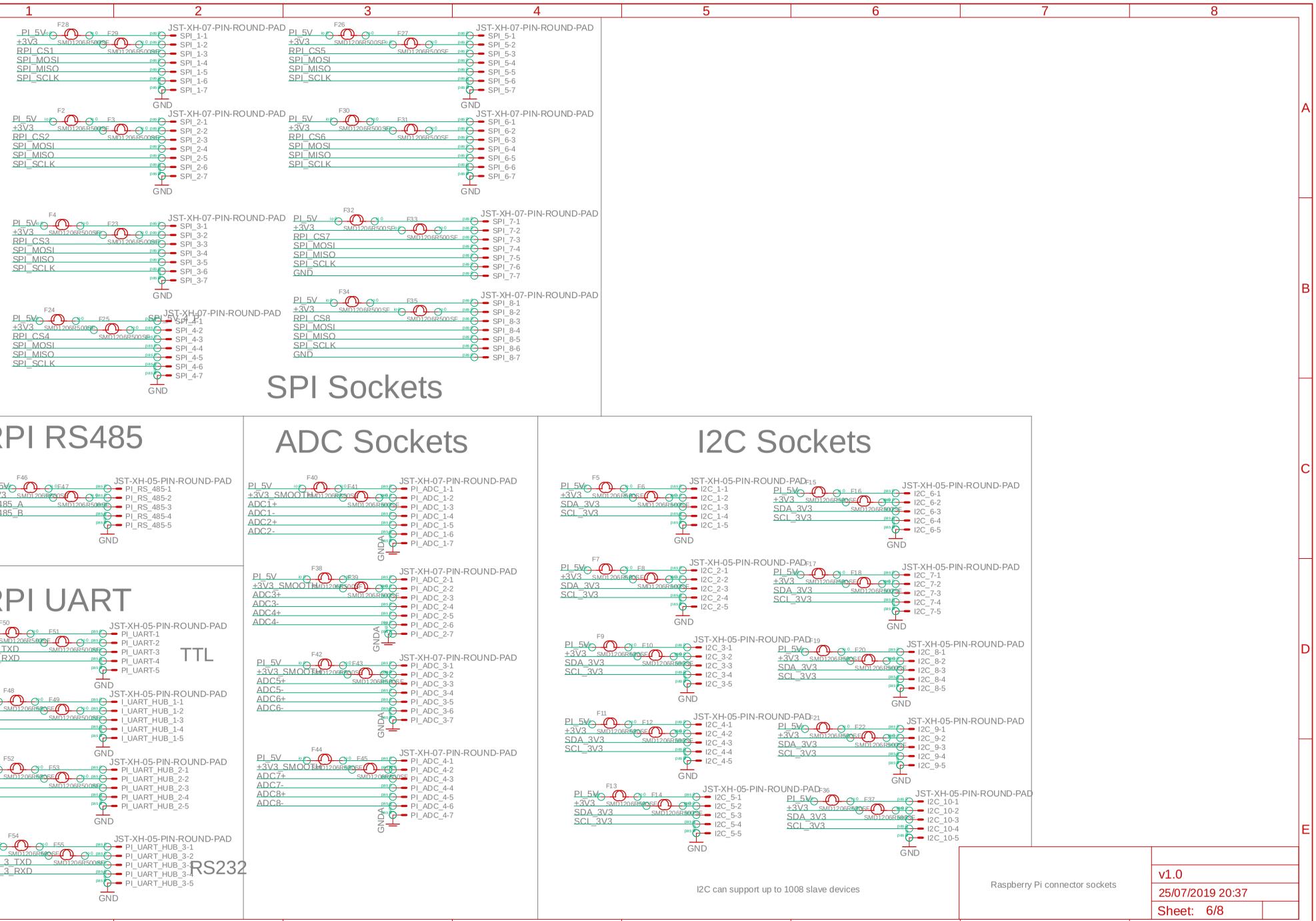
ADC	v1.0
	25/07/2019 20:37
	Sheet: 4/8

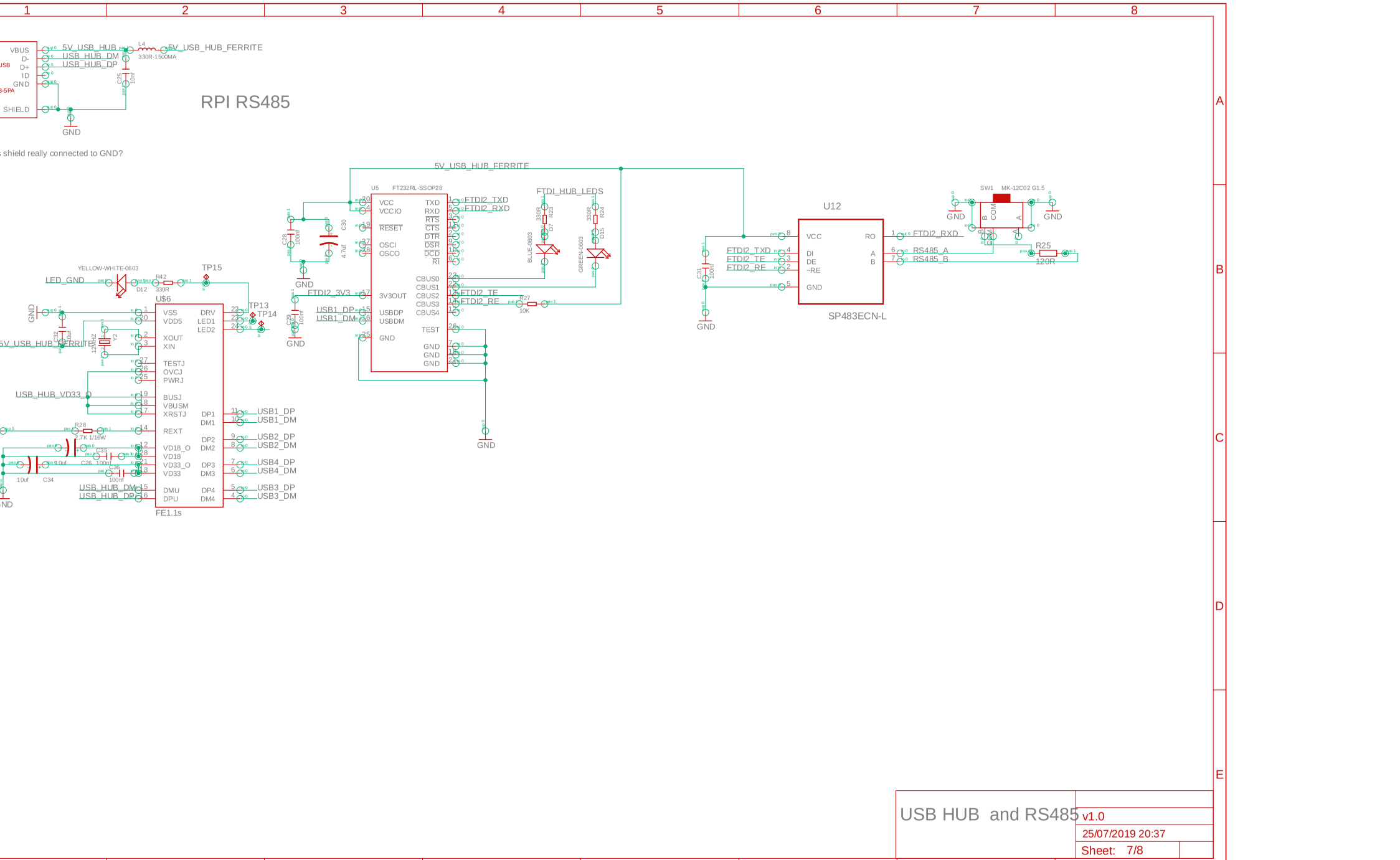


H1 MOUNT-HOLE3.0
H2 MOUNT-HOLE3.0
H3 MOUNT-HOLE3.0
H4 MOUNT-HOLE3.0
H5 MOUNT-HOLE3.0
H6 MOUNT-HOLE3.0

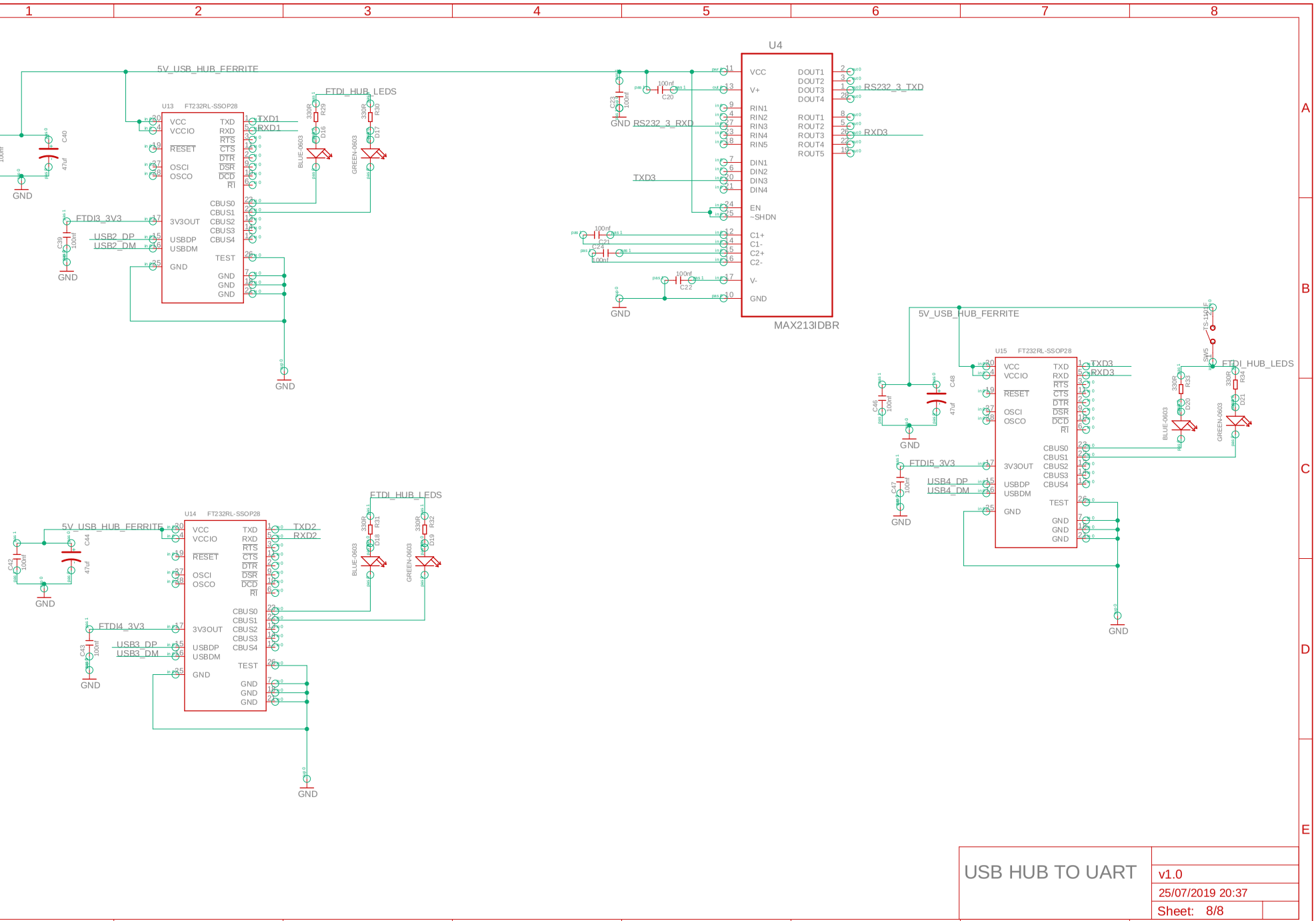
LEDs

v1.0
25/07/2019 20:37
Sheet: 5/8


1
2
3
4
5
6
7
8
A
B
C
D
E
JST-XH-07-PIN-ROUND-PAD
JST-XH-05-PIN-ROUND-PAD
SMD1206R500SF
GND
SPI Sockets
RPI RS485
RPI UART
TTL
RS232
ADC Sockets
I2C Sockets
I2C can support up to 1008 slave devices
Raspberry Pi connector sockets
v1.0
25/07/2019 20:37
Sheet: 6/8



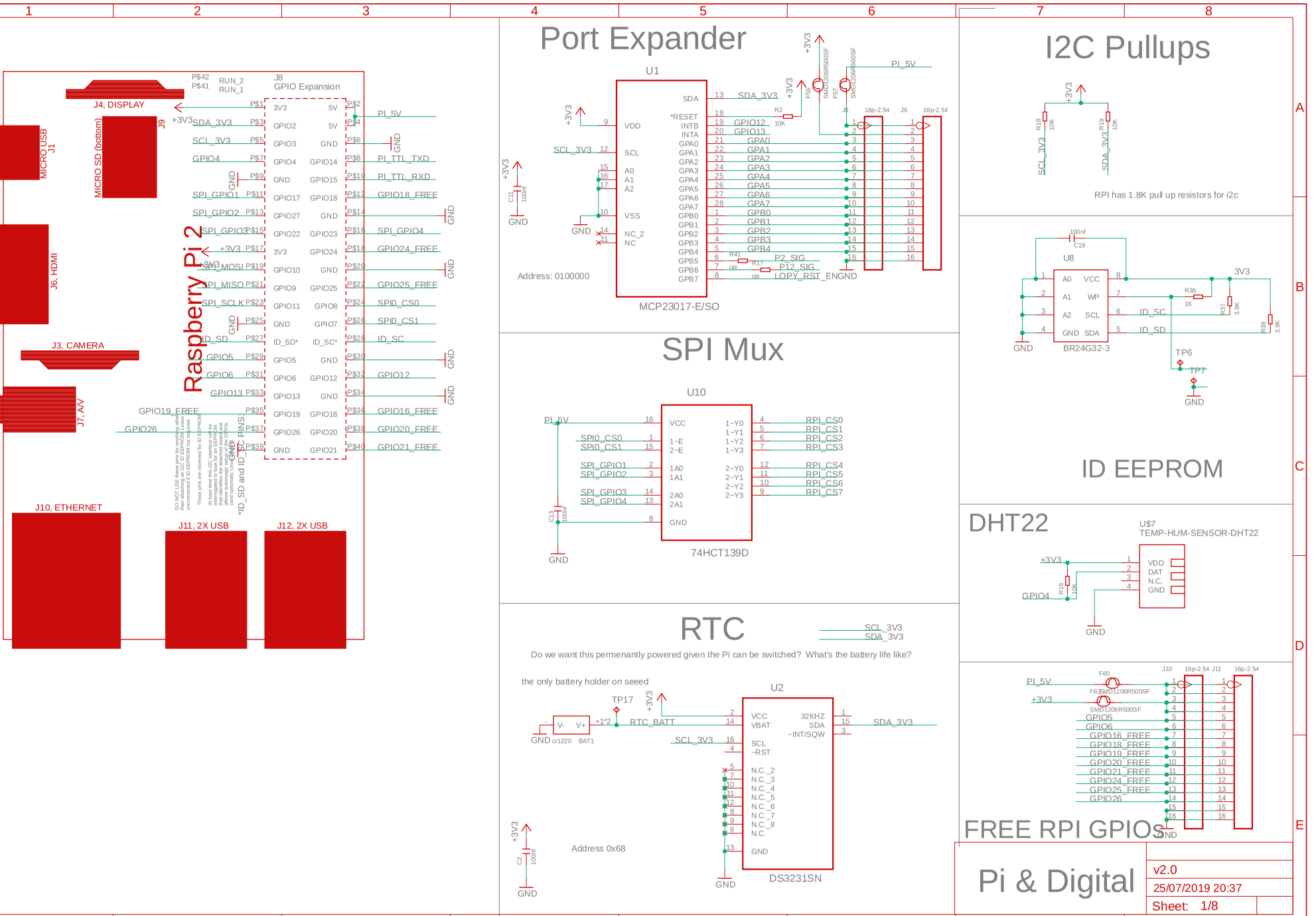
USB HUB and RS485	v1.0
	25/07/2019 20:37
	Sheet: 7/8


1
2
3
4
5
6
7
8
A
B
C
D
E
5V_USB_HUB_FERRITE
U13 FT232RL-SSOP28
VCC
VCCIO
RESET
OSCI
OSCO
3V3OUT
USBDP
USBDM
GND
TXD
RXD
RTS
CTS
DTR
DSR
DCD
RI
CBUS0
CBUS1
CBUS2
CBUS3
CBUS4
TEST
TXD1
RXD1
FTDI_HUB_LEDS
BLUE-0603
GREEN-0603
330R
FTDI3_3V3
USB2_DP
USB2_DM
U4
MAX213IDBR
VCC
V+
RIN1
RIN2
RIN3
RIN4
RIN5
DOUT1
DOUT2
DOUT3
DOUT4
ROUT1
ROUT2
ROUT3
ROUT4
ROUT5
DIN1
DIN2
DIN3
DIN4
EN
~SHDN
C1+
C1-
C2+
C2-
V-
RS232_3_TXD
RS232_3_RXD
TXD3
RXD3
U14 FT232RL-SSOP28
TXD2
RXD2
FTDI4_3V3
USB3_DP
USB3_DM
U15 FT232RL-SSOP28
TXD3
RXD3
FTDI5_3V3
USB4_DP
USB4_DM
SW5
USB HUB TO UART
v1.0
25/07/2019 20:37
Sheet: 8/8

Appendix B

PiSEB v2.0 Eagle Schematics

This appendix includes the Eagle Schematics of the second version of the PiSEB board.



Port Expander

SPI Mux

RTC

I2C Pullups

RPI has 1.8K pull up resistors for i2c

ID EEPROM

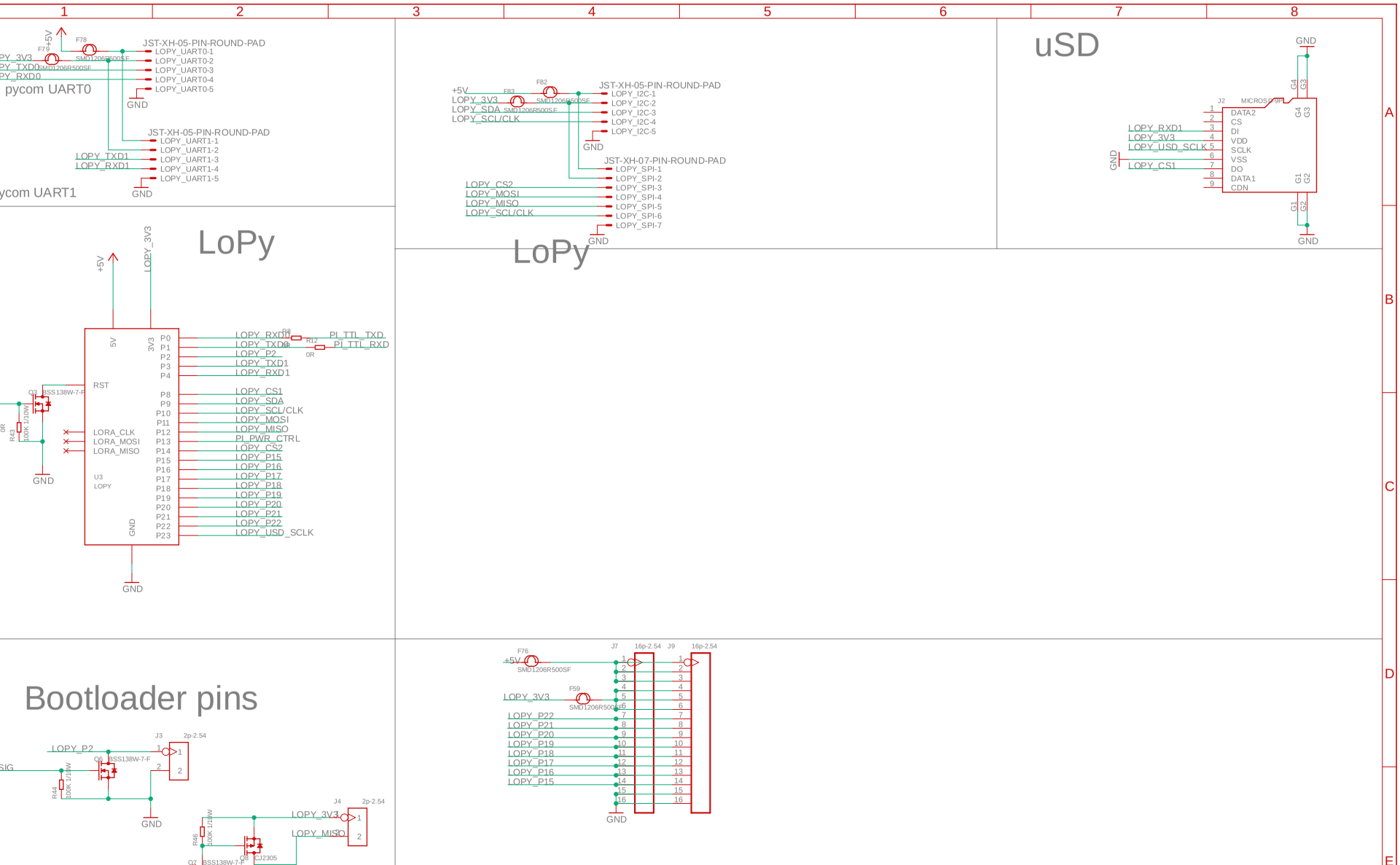
DHT22

FREE RPI GPIOs

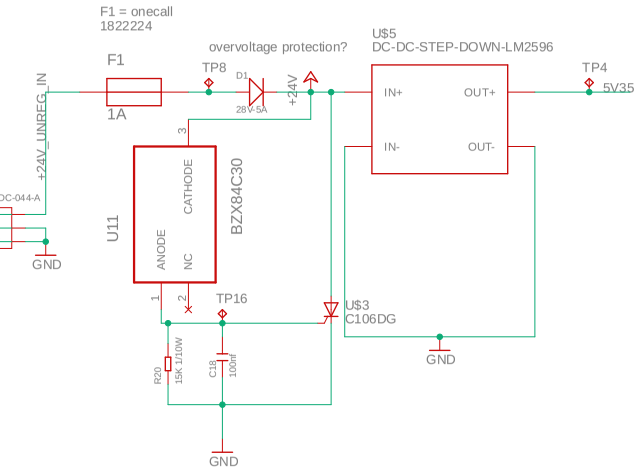
Pi & Digital

v2.0	
25/07/2019 20:37	
Sheet: 1/8	

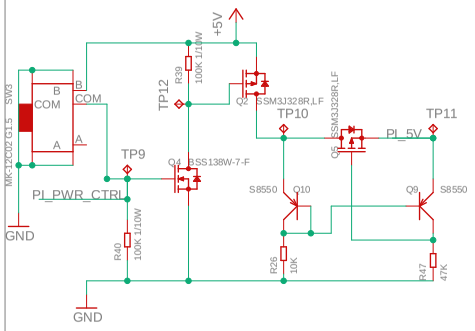
LoPy UART


1
2
3
4
5
6
7
8
A
B
C
D
E
pycom UART0
JST-XH-05-PIN-ROUND-PAD
LOPY_UART0-1
LOPY_UART0-2
LOPY_UART0-3
LOPY_UART0-4
LOPY_UART0-5
GND
JST-XH-05-PIN-ROUND-PAD
LOPY_UART1-1
LOPY_UART1-2
LOPY_UART1-3
LOPY_UART1-4
LOPY_UART1-5
LOPY_TXD1
LOPY_RXD1
GND
ycom UART1
LoPy
+5V
LOPY_3V3
P0
P1
P2
P3
P4
P8
P9
P10
P11
P12
P13
P14
P15
P16
P17
P18
P19
P20
P21
P22
P23
RST
LORA_CLK
LORA_MOSI
LORA_MISO
LOPY_RXD1
LOPY_TXD1
LOPY_P2
LOPY_CS1
LOPY_SDA
LOPY_SCL/CLK
LOPY_MOSI
LOPY_MISO
PI_PWR_CTRL
LOPY_CS2
LOPY_P15
LOPY_P16
LOPY_P17
LOPY_P18
LOPY_P19
LOPY_P20
LOPY_P21
LOPY_P22
LOPY_USD_SCLK
PI_TTL_TXD
PI_TTL_RXD
BSS138W-7-F
GND
JST-XH-05-PIN-ROUND-PAD
LOPY_I2C-1
LOPY_I2C-2
LOPY_I2C-3
LOPY_I2C-4
LOPY_I2C-5
LOPY_SDA
LOPY_SCL/CLK
JST-XH-07-PIN-ROUND-PAD
LOPY_SPI-1
LOPY_SPI-2
LOPY_SPI-3
LOPY_SPI-4
LOPY_SPI-5
LOPY_SPI-6
LOPY_SPI-7
LOPY_CS2
LOPY_MOSI
LOPY_MISO
LOPY_SCL/CLK
LoPy
uSD
MICROSD-SPI
DATA2
CS
DI
VDD
SCLK
VSS
DO
DATA1
CDN
LOPY_RXD1
LOPY_3V3
LOPY_USD_SCLK
LOPY_CS1
Bootloader pins
2p-2.54
LOPY_P2
LOPY_3V3
LOPY_MISO
16p-2.54
LOPY_P22
LOPY_P21
LOPY_P20
LOPY_P19
LOPY_P18
LOPY_P17
LOPY_P16
LOPY_P15

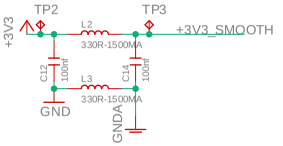
DC&DC



Pi Power Control

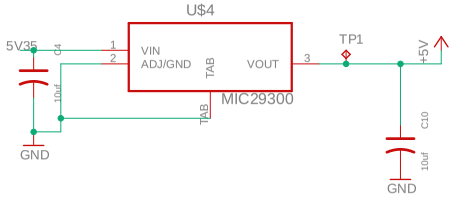


Pi filter for 3V3

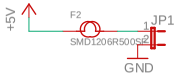


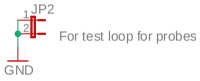
LDO

TP4 & TP1 should be close together



Needs good ground plane for termal issues

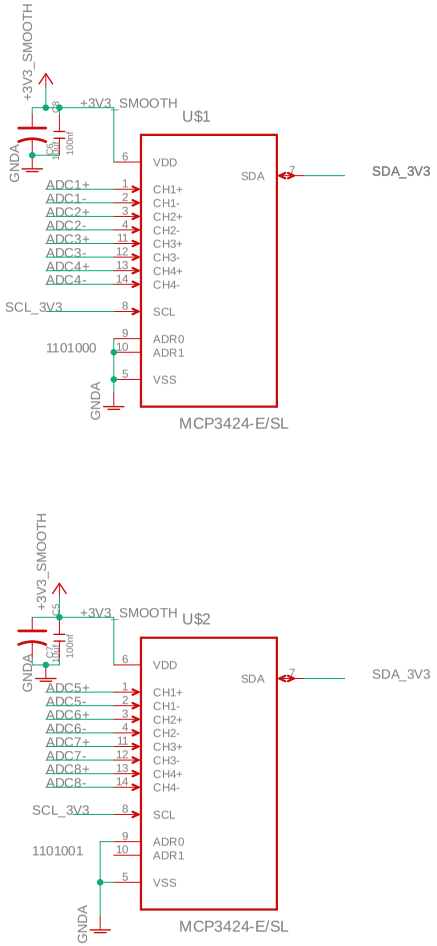




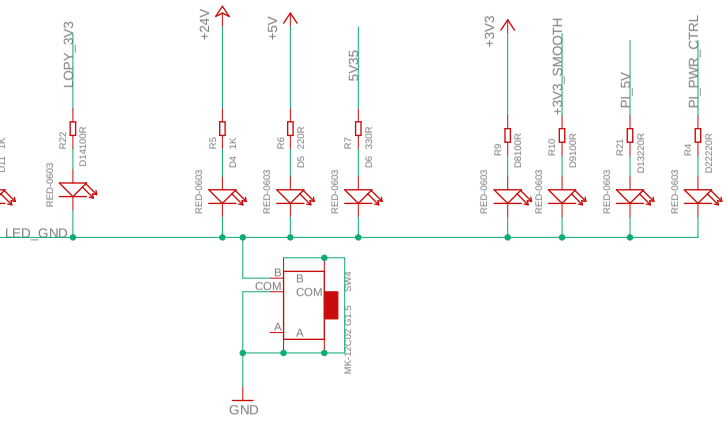
For test loop for probes

PSU

v2.0
25/07/2019 20:37
Sheet: 3/8



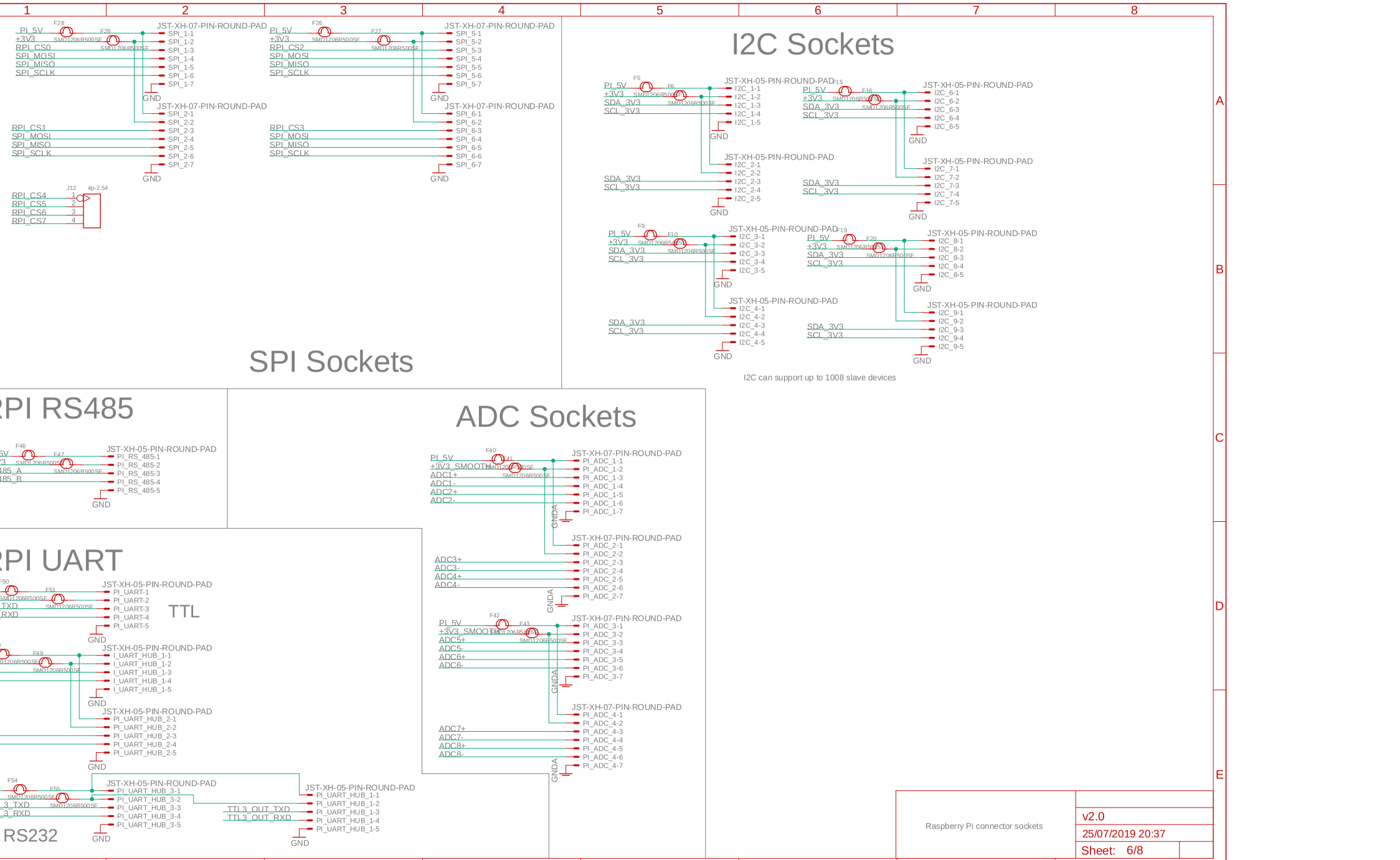
ADC	
	v2.0
	25/07/2019 20:37
	Sheet: 4/8

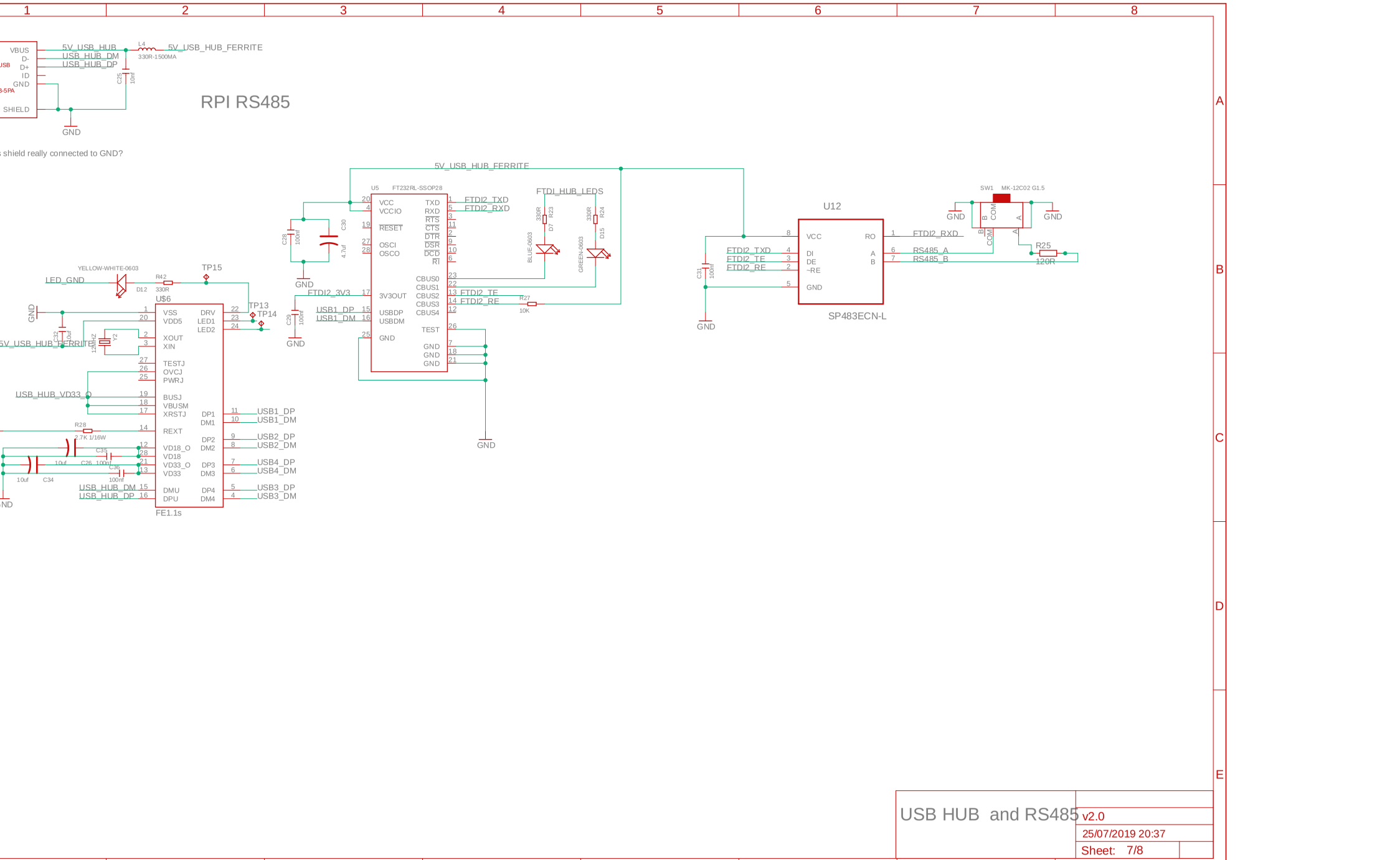


H1 MOUNT-HOLE3.0
H2 MOUNT-HOLE3.0
H3 MOUNT-HOLE3.0
H4 MOUNT-HOLE3.0
H5 MOUNT-HOLE3.0
H6 MOUNT-HOLE3.0

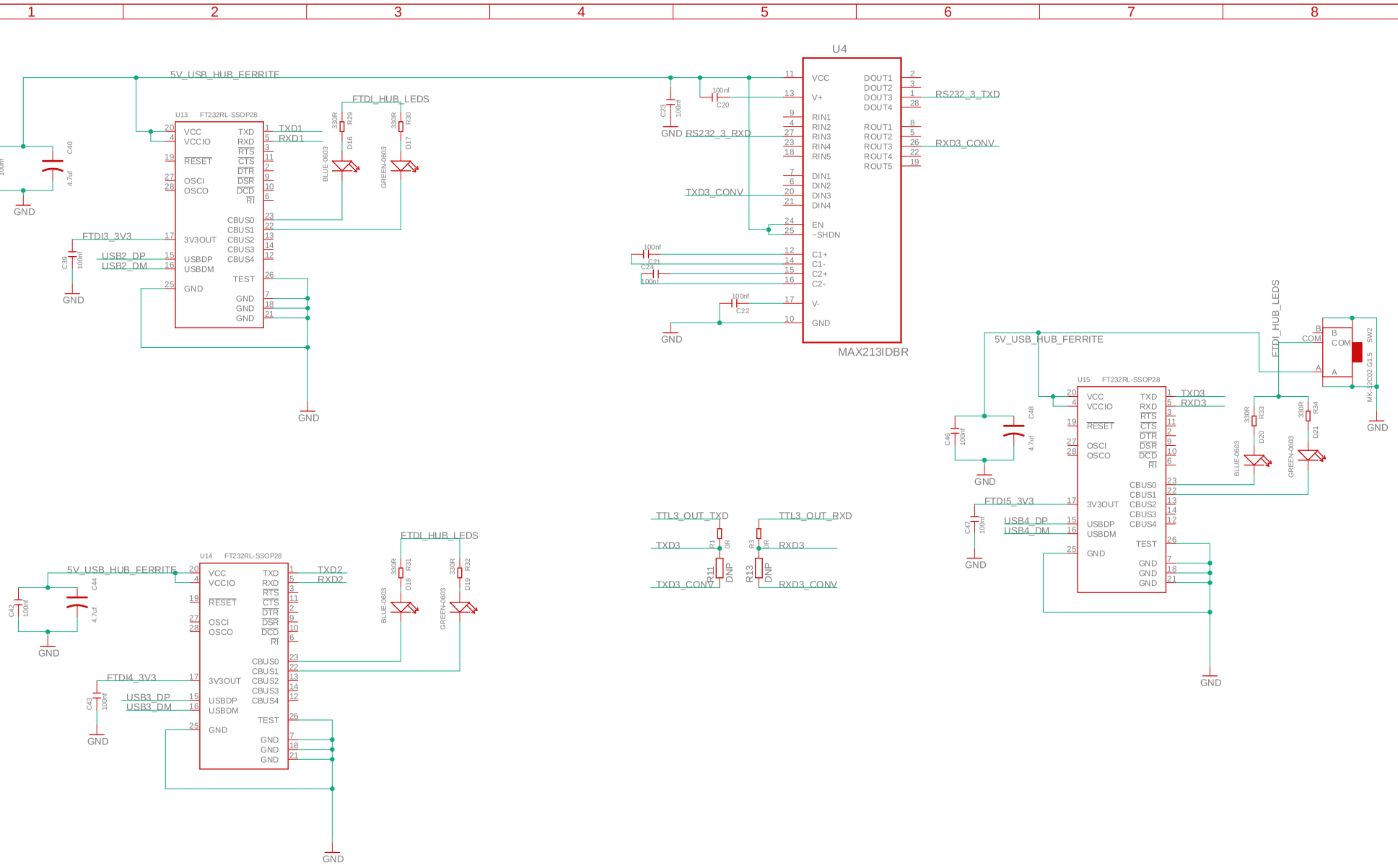
LEDs

v2.0
25/07/2019 20:37
Sheet: 5/8


1
2
3
4
5
6
7
8
A
B
C
D
E
SPI Sockets
PI_5V
+3V3
RPI_CS0
SPI_MOSI
SPI_MISO
SPI_SCLK
GND
JST-XH-07-PIN-ROUND-PAD
SPI_1-1
SPI_1-2
SPI_1-3
SPI_1-4
SPI_1-5
SPI_1-6
SPI_1-7
RPI_CS1
SPI_2-1
SPI_2-2
SPI_2-3
SPI_2-4
SPI_2-5
SPI_2-6
SPI_2-7
RPI_CS2
SPI_5-1
SPI_5-2
SPI_5-3
SPI_5-4
SPI_5-5
SPI_5-6
SPI_5-7
RPI_CS3
SPI_6-1
SPI_6-2
SPI_6-3
SPI_6-4
SPI_6-5
SPI_6-6
SPI_6-7
J12
4p-2.54
RPI_CS4
RPI_CS5
RPI_CS6
RPI_CS7
I2C Sockets
SDA_3V3
SCL_3V3
JST-XH-05-PIN-ROUND-PAD
I2C_1-1
I2C_1-2
I2C_1-3
I2C_1-4
I2C_1-5
I2C_2-1
I2C_2-2
I2C_2-3
I2C_2-4
I2C_2-5
I2C_3-1
I2C_3-2
I2C_3-3
I2C_3-4
I2C_3-5
I2C_4-1
I2C_4-2
I2C_4-3
I2C_4-4
I2C_4-5
I2C_6-1
I2C_6-2
I2C_6-3
I2C_6-4
I2C_6-5
I2C_7-1
I2C_7-2
I2C_7-3
I2C_7-4
I2C_7-5
I2C_8-1
I2C_8-2
I2C_8-3
I2C_8-4
I2C_8-5
I2C_9-1
I2C_9-2
I2C_9-3
I2C_9-4
I2C_9-5
I2C can support up to 1008 slave devices
RPI RS485
PI_RS_485-1
PI_RS_485-2
PI_RS_485-3
PI_RS_485-4
PI_RS_485-5
ADC Sockets
+3V3_SMOOTH
ADC1+
ADC1-
ADC2+
ADC2-
ADC3+
ADC3-
ADC4+
ADC4-
ADC5+
ADC5-
ADC6+
ADC6-
ADC7+
ADC7-
ADC8+
ADC8-
GNDA
JST-XH-07-PIN-ROUND-PAD
PI_ADC_1-1
PI_ADC_1-2
PI_ADC_1-3
PI_ADC_1-4
PI_ADC_1-5
PI_ADC_1-6
PI_ADC_1-7
PI_ADC_2-1
PI_ADC_2-2
PI_ADC_2-3
PI_ADC_2-4
PI_ADC_2-5
PI_ADC_2-6
PI_ADC_2-7
PI_ADC_3-1
PI_ADC_3-2
PI_ADC_3-3
PI_ADC_3-4
PI_ADC_3-5
PI_ADC_3-6
PI_ADC_3-7
PI_ADC_4-1
PI_ADC_4-2
PI_ADC_4-3
PI_ADC_4-4
PI_ADC_4-5
PI_ADC_4-6
PI_ADC_4-7
RPI UART
TTL
PI_UART-1
PI_UART-2
PI_UART-3
PI_UART-4
PI_UART-5
JST-XH-05-PIN-ROUND-PAD
I_UART_HUB_1-1
I_UART_HUB_1-2
I_UART_HUB_1-3
I_UART_HUB_1-4
I_UART_HUB_1-5
PI_UART_HUB_2-1
PI_UART_HUB_2-2
PI_UART_HUB_2-3
PI_UART_HUB_2-4
PI_UART_HUB_2-5
PI_UART_HUB_3-1
PI_UART_HUB_3-2
PI_UART_HUB_3-3
PI_UART_HUB_3-4
PI_UART_HUB_3-5
TTL3_OUT_TXD
TTL3_OUT_RXD
RS232
Raspberry Pi connector sockets
v2.0
25/07/2019 20:37
Sheet: 6/8



USB HUB and RS485	v2.0
	25/07/2019 20:37
	Sheet: 7/8


U4
MAX213IDBR
U13 FT232RL-SSOP28
U14 FT232RL-SSOP28
U15 FT232RL-SSOP28
5V_USB_HUB_FERRITE
FTDI_HUB_LEDS
RS232_3_TXD
RS232_3_RXD
RXD3_CONV
TXD3_CONV
TXD1
RXD1
TXD2
RXD2
TXD3
RXD3
FTDI3_3V3
FTDI4_3V3
FTDI5_3V3
USB2_DP
USB2_DM
USB3_DP
USB3_DM
USB4_DP
USB4_DM
TTL3_OUT_TXD
TTL3_OUT_RXD
GND
USB HUB TO UART
v2.0
25/07/2019 20:37
Sheet: 8/8

References

[1] (2019). meta-virtualization. http://git.yoctoproject.org/cgit/cgit.cgi/meta-virtualization/. Accessed on 16 May 2019.

[2] Bug Labs, Inc (2017). freeboard. https://freeboard.io. Accessed on 28 April 2019.

[3] Abankwa, N. O., Bowker, J., Johnston, S. J., Scott, M., and Cox, S. J. (2018). Estimating the Longitudinal Center of Flotation of a Vessel in Waves Using Acceleration Measurements. *IEEE Sensors Journal*, 18(20):8426–8435.

[4] Aberer, K., Hauswirth, M., and Salehi, A. (2007). Infrastructure for data processing in large-scale interconnected sensor networks. Technical report, IEEE Computer Society.

[5] Aberer, K., Sathe, S., Chakraborty, D., Martinoli, A., Barrenetxea, G., Faltings, B., and Thiele, L. (2010). OpenSense: open community driven sensing of environment. In *Proceedings of the ACM SIGSPATIAL International Workshop on GeoStreaming*, pages 39–42. ACM.

[6] advancedtelematic (2019). meta-updater. https://github.com/advancedtelematic/meta-updater. Accessed on 16 May 2019.

[7] Air-Go (2019). Air-go. *http://air-go.es*. Accessed on 27 April 2019.

[8] AirPi (2019). Airpi. *http://airpi.es/*. Accessed on 31 March 2019.

[9] Anderson, J. C., Lehnardt, J., and Slater, N. (2010). *CouchDB: the definitive guide*. O'Reilly Media, Inc.

[10] Apetroaie-Cristea, M. (2019a). meta-remote-management. http://doi.org/10.5281/zenodo.2543575. Accessed on 16 May 2019.

[11] Apetroaie-Cristea, M. (2019b). meta-remote-management-raspberrypi. http://doi.org/10.5281/zenodo.2542820. Accessed on 16 May 2019.

[12] Apetroaie-Cristea, M., Basford, P., Tiniuc, A., Johnston, S., and Cox, S. (2019). PiSEB PCB Schematics. https://doi.org/10.5281/zenodo.2545262. Accessed on 16 May 2019.

[13] Apetroaie-Cristea, M., Scott, M., Johnston, S. J., and Cox, S. J. (2016). Indoor localisation system based on low-cost commodity hardware. In *Proceedings of the 2016 ACM International Joint Conference on Pervasive and Ubiquitous Computing: Adjunct*, pages 13–16. ACM.

[14] Arnold, M., Hoydis, J., and ten Brink, S. (2019). Novel Massive MIMO Channel Sounding Data applied to Deep Learning-based Indoor Positioning. In *SCC 2019; 12th International ITG Conference on Systems, Communications and Coding*, pages 1–6. VDE.

[15] Ashton, K. et al. (2009). That 'Internet of Things' thing. *RFID Journal*, 22(7):97–114.

[16] Atzori, L., Iera, A., and Morabito, G. (2010). The Internet of Things: A survey. In *Computer networks*, volume 54, pages 2787–2805. Elsevier.

[17] Augustin, A., Yi, J., Clausen, T., and Townsley, W. (2016). A study of LoRa: Long range & low power networks for the Internet of Things. *Sensors*, 16(9):1466.

[18] Bahena, V. R. (2014). Embedded distributed systems: A case of study with Clear Linux project for Intel R Architecture. Intel.

[19] Bai, H. and Shi, G. (2007). Gas sensors based on conducting polymers. *Sensors*, 7(3):267–307.

[20] Bakıcı, T., Almirall, E., and Wareham, J. (2013). A smart city initiative: the case of Barcelona. *Journal of the Knowledge Economy*, 4(2):135–148.

[21] balena.io (2017). balena.io. `https://balena.io/`. Accessed on 26 April 2019.

[22] Beckman, P., Sankaran, R., Catlett, C., Ferrier, N., Jacob, R., and Papka, M. (2016). Waggle: An open sensor platform for edge computing. In *2016 IEEE SENSORS*, pages 1–3. IEEE.

[23] Bell, S., Cornford, D., and Bastin, L. (2015). How good are citizen weather stations? Addressing a biased opinion. *Weather*, 70(3):75–84.

[24] Bergen, M. H., Schaal, F. S., Klukas, R., Cheng, J., and Holzman, J. F. (2018). Toward the implementation of a universal angle-based optical indoor positioning system. *Frontiers of Optoelectronics*, 11(2):116–127.

[25] Bernstein, D. (2014). Containers and cloud: From LXC to Docker to Kubernetes. *IEEE Cloud Computing*, 1(3):81–84.

[26] Bielsa, A. (2012). Smart City project in Serbia for environmental monitoring by Public Transportation. `http://www.libelium.com/smart_city_environmental_parameters_public_transportation_waspmote/`. Accessed on 30 March 2019.

[27] Blair, S., Matheson, N., Munro, R., and Booth, C. (2019). A new platform for validating real-time, large-scale WAMPAC systems. In *PAC World Conference 2019*.

[28] Brewer, E. A. (2015). Kubernetes and the path to cloud native. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, pages 167–167. ACM.

[29] Brookes, D., Stedman, J., Kent, A., King, R., Venfield, H., Cooke, S., Lingard, J., Vincent, K., Bush, T., and Abbott, J. (2013). Technical report on UK supplementary assessment under the Air Quality Directive (2008/50/EC), the Air Quality Framework Directive (96/62/EC) and Fourth Daughter Directive (2004/107/EC) for 2011. *Report for the Department for Environment, Food and Rural Affairs, Welsh Government, the Scottish Government and the Department of the Environment for Northern Ireland. Available online at: http://ukair. defra. gov. uk/assets/documents/reports/cat09/1312231525_ AQD_ DD4_ 2012mapsrepv0. p df.*

[30] Brunekreef, B. and Holgate, S. T. (2002). Air pollution and health. In *The Lancet*, volume 360, pages 1233–1242. Elsevier.

[31] Bulot, F. M., Johnston, S. J., Basford, P. J., Easton, N. H., Apetroaie-Cristea, M., Foster, G. L., Morris, A. K., Cox, S. J., and Loxham, M. (2019). Long-term field comparison of multiple low-cost particulate matter sensors in an outdoor urban environment. *Scientific reports*, 9(1):7497.

[32] Caragliu, A., Del Bo, C., and Nijkamp, P. (2011). Smart cities in Europe. *Journal of urban technology*, 18(2):65–82.

[33] Carmichael, L. and Lambert, C. (2011). Governance, knowledge and sustainability: the implementation of EU directives on air quality in Southampton. *Local Environment*, 16(2):181–191.

[34] Carr, J. and Doleac, J. L. (2016). The geography, incidence, and underreporting of gun violence: new evidence using ShotSpotter data. *Incidence, and Underreporting of Gun Violence: New Evidence Using Shotspotter Data (April 26, 2016)*.

[35] Carruthers, K. (2016). Internet of Things and Beyond: Cyber-Physical Systems. `http://iot.ieee.org/newsletter/may-2016/internet-of-things-and-beyond-cyber-physical-systems.html`. Accessed on 25 April 2019.

[36] Catlett, C. E., Beckman, P. H., Sankaran, R., and Galvin, K. K. (2017). Array of things: a scientific research instrument in the public way: platform design and early lessons learned. In *Proceedings of the 2nd International Workshop on Science of Smart City Operations and Platforms Engineering*, pages 26–33. ACM.

[37] Chen, M., Mao, S., and Liu, Y. (2014). Big data: A survey. In *Mobile Networks and Applications*, volume 19, pages 171–209. Springer.

[38] Chodorow, K. (2013). *MongoDB: the definitive guide*. O'Reilly Media, Inc.

[39] Codd, E. F. (1970). A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387.

[40] Cohen, J. E. and Tilman, D. (1996). Biosphere 2 and biodiversity: The lessons so far. *Science*, 274(5290):1150.

[41] Coleman, E., Goldstein, B., and Dyson, L. (2013). Lessons from the london datastore. *Beyond transparency: open data and the future of civic innovation*, pages 39–50.

[42] Cox, S. J., Cox, J. T., Boardman, R. P., Johnston, S. J., Scott, M., and O'Brien, N. S. (2014). Iridis-pi: a low-cost, compact demonstration cluster. *Cluster Computing*, 17(2):349–358.

[43] Davis, N. (2015). Wanted! An army of citizen scientists to tackle air pollution. *The Guardian*. Online at `https://www.theguardian.com/environment/2015/aug/30/citizen-scientists-tackle-air-pollution`, Accessed on 22 December 2019.

[44] Devarakonda, S., Sevusu, P., Liu, H., Liu, R., Iftode, L., and Nath, B. (2013). Real-time air quality monitoring through mobile sensing in metropolitan areas. In *Proceedings of the 2nd ACM SIGKDD International Workshop on Urban Computing*, page 15. ACM.

[45] Dohr, A., Modre-Opsrian, R., Drobics, M., Hayn, D., and Schreier, G. (2010). The Internet of Things for ambient assisted living. In *Information technology: new generations (ITNG), 2010 Seventh International Conference*, pages 804–809. IEEE.

[46] Dolstra, E. and Löh, A. (2008). NixOS: A purely functional Linux distribution. In *ACM Sigplan Notices*, volume 43, pages 367–378. ACM.

[47] Edwards, C. (2013). Not-so-humble Raspberry Pi gets big ideas. *Engineering & Technology*, 8(3):30–33.

[48] Ely, D., Savage, S., and Wetherall, D. (2001). Alpine: A user-level infrastructure for network protocol development. In *USITS*, volume 1, pages 15–15.

[49] Eroglu, Y., Erden, F., and Guvenc, I. (2019). Adaptive Kalman Tracking for Indoor Visible Light Positioning. *arXiv preprint arXiv:1909.12985*.

[50] European Comission (2016). Amsterdam is the European Capital of Innovation 2016. `http://ec.europa.eu/research/index.cfm?pg=newsalert&year=2016&na=na-080416`. Accessed on 26 April 2019.

[51] Fainelli, F. (2008). The OpenWrt development framework. In *Proceedings of the Free and Open Source Software Developers European Meeting*.

[52] Feng, Z. (2011). Research on water-saving irrigation automatic control system based on Internet of Things. In *Electric Information and Control Engineering (ICEICE), 2011 International Conference on*, pages 2541–2544. IEEE.

[53] Forsström, S. and Jennehag, U. (2017). A performance and cost evaluation of combining OPC-UA and Microsoft Azure IoT Hub into an industrial Internet-of-Things system. In *2017 Global Internet of Things Summit (GIoTS)*, pages 1–6. IEEE.

[54] Fukuda-Parr, S. (2016). From the Millennium Development Goals to the Sustainable Development Goals: shifts in purpose, concept, and politics of global goal setting for development. *Gender & Development*, 24(1):43–52.

[55] Gartner (2017). Gartner Says 8.4 Billion Connected "Things" Will Be in Use in 2017, Up 31 Percent From 2016. Accessed on 26 April 2019.

[56] Geissler, J.-B., Tricarico, L., and Vecchio, G. (2017). The construction of a trading zone as political strategy: a review of London Infrastructure Plan 2050. In *GLA Intelligence*.

[57] GeSI (2019). DigitalAccessIndex. *Online:* *https://digitalaccessindex-sdg.gesi.org/*.

[58] Gomez, C., Oller, J., and Paradells, J. (2012). Overview and evaluation of bluetooth low energy: An emerging low-power wireless technology. *Sensors*, 12(9):11734–11753.

[59] Greenspan, J. and Bulger, B. (2001). *MySQL/PHP database applications*. John Wiley & Sons, Inc.

[60] Guth, J., Breitenbücher, U., Falkenthal, M., Leymann, F., and Reinfurt, L. (2016). Comparison of IoT platform architectures: A field study based on a reference architecture. In *2016 Cloudification of the Internet of Things (CIoT)*, pages 1–6. IEEE.

[61] Haines, N. (2015). The Future of Ubuntu. In *Beginning Ubuntu for Windows and Mac Users*, pages 205–209. Springer.

[62] Hart, J. K. and Martinez, K. (2006). Environmental Sensor Networks: A revolution in the earth system science? *Earth-Science Reviews*, 78(3):177–191.

[63] He, J., Liu, H., and Salvo, A. (2019). Severe air pollution and labor productivity: Evidence from industrial towns in China. *American Economic Journal: Applied Economics*, 11(1):173–201.

[64] Hendrawan, I. N. R. and Arsa, I. G. N. W. (2017). Zolertia Z1 energy usage simulation with Cooja simulator. In *2017 1st International Conference on Informatics and Computational Sciences (ICICoS)*, pages 147–152. IEEE.

[65] Hightower, K., Burns, B., and Beda, J. (2017). *Kubernetes: up and running: dive into the future of infrastructure*. " O'Reilly Media, Inc.".

[66] Ho, E. (2017). Smart subjects for a Smart Nation? Governing (smart) mentalities in Singapore. In *Urban Studies*, volume 54, pages 3101–3118. SAGE Publications Sage UK: London, England.

[67] Jackson, K. R., Ramakrishnan, L., Muriki, K., Canon, S., Cholia, S., Shalf, J., Wasserman, H. J., and Wright, N. J. (2010). Performance analysis of high performance computing applications on the amazon web services cloud. In *2nd IEEE international conference on cloud computing technology and science*, pages 159–168. IEEE.

[68] Johnston, S., Apetroaie-Cristea, M., Scott, M., and Cox, S. (2016a). Applicability of commodity, low cost, single board computers for Internet of Things devices. In *World Forum on Internet of Things (WF-IoT) 2016*, pages 1–6.

[69] Johnston, S. J., Apetroaie-Cristea, M., Scott, M., and Cox, S. J. (2016b). Applied Internet of Things. In Buyya, R. and Dastjerdi, A. V., editors, *Internet of Things Principles and Paradigms*, chapter 15, pages 277–297. Elsevier, 50 Hampshire Street, 5th Floor, Cambridge, MA 02139, USA.

[70] Junyan, L., Shiguo, X., and Yijie, L. (2009). Application research of embedded database SQLite. In *2009 International Forum on Information Technology and Applications*, volume 2, pages 539–543. IEEE.

[71] Kakakhel, S. R. U., Mukkala, L., Westerlund, T., and Plosila, J. (2018). Virtualization at the network edge: A technology perspective. In *2018 Third International Conference on Fog and Mobile Edge Computing (FMEC)*, pages 87–92. IEEE.

[72] Kelly, K., Whitaker, J., Petty, A., Widmer, C., Dybwad, A., Sleeth, D., Martin, R., and Butterfield, A. (2017). Ambient and laboratory evaluation of a low-cost particulate matter sensor. *Environmental Pollution*, 221:491–500.

[73] Kinney, P. et al. (2003). Zigbee technology: Wireless control that simply works. In *Communications Design Conference*, volume 2, pages 1–7.

[74] Koponen., L. (2015). Raspberry Pi controller. `http://rpc.gehennom.org/`. Accessed on 6 April 2019.

[75] Krishnan, S. and Gonzalez, J. L. U. (2015). *Building Your Next Big Thing with Google Cloud Platform: A Guide for Developers and Enterprise Architects*. Springer.

[76] Kumar, P., Morawska, L., Martani, C., Biskos, G., Neophytou, M., Di Sabatino, S., Bell, M., Norford, L., and Britter, R. (2015). The rise of low-cost sensing for managing air pollution in cities. *Environment International*, 75:199–205.

[77] Kumar, S., Gil, S., Katabi, D., and Rus, D. (2014). Accurate indoor localization with zero start-up cost. In *Proceedings of the 20th Annual International Conference on Mobile computing and networking*, pages 483–494. ACM.

[78] Lashof, D. A. and Ahuja, D. R. (1990). Relative contributions of greenhouse gas emissions to global warming. *Nature*, 344(6266):529–531.

[79] Lechelt, Z., Rogers, Y., Marquardt, N., and Shum, V. (2016). ConnectUs: A New Toolkit for Teaching about the Internet of Things. In *Proceedings of the 2016 CHI Conference Extended Abstracts on Human Factors in Computing Systems*, pages 3711–3714. ACM.

[80] Lee, J. H., Hancock, M. G., and Hu, M.-C. (2014). Towards an effective framework for building smart cities: Lessons from Seoul and San Francisco. *Technological Forecasting and Social Change*, 89:80–99.

[81] Lekić, M. and Gardašević, G. (2018). IoT sensor integration to Node-RED platform. In *2018 17th International Symposium INFOTEH-JAHORINA (INFOTEH)*, pages 1–5. IEEE.

[82] Levis, P., Madden, S., Polastre, J., Szewczyk, R., Whitehouse, K., Woo, A., Gay, D., Hill, J., Welsh, M., Brewer, E., et al. (2005). TinyOS: An operating system for sensor networks. In *Ambient Intelligence*, pages 115–148. Springer.

[83] Lewis, A. C., Lee, J. D., Edwards, P. M., Shaw, M. D., Evans, M. J., Moller, S. J., Smith, K. R., Buckley, J. W., Ellis, M., Gillot, S. R., et al. (2016). Evaluating the performance of low cost chemical sensors for air pollution research. *Faraday discussions*, 189:85–103.

[84] Li, Y. and Manoharan, S. (2013). A performance comparison of SQL and NoSQL databases. In *Communications, Computers and Signal Processing (PACRIM), 2013 IEEE Pacific Rim Conference on*, pages 15–19. IEEE.

[85] Libelium (2016). Mobile monitoring system: Vehicles with sensors to control air quality in Glasgow. `http://www.libelium.com/mobile-monitoring-system-vehicles-with-sensors-to-control-air-quality-in-glasgow/`. Accessed on 30 March 2019.

[86] Libelium (2017a). Smart City project in Ljubljana Shopping and Business Centre to follow its Green Mission strategy. `http://www.libelium.com/smart-city-project-in-ljubljana-shopping-and-business-centre-to-follow-its-green-mission-strategy/`. Accessed on 30 March 2019.

[87] Libelium (2017b). Snow and ice monitoring in UK winter highways for a Smart Road management. `http://www.libelium.com/snow-and-ice-monitoring-in-uk-winter-highways-for-a-smart-road-management/`. Accessed on 30 March 2019.

[88] Libelium Comunicaciones Distribuidas S.L (2017). Libelium. `http://www.libelium.com/`. Accessed on 30 March 2019.

[89] Liu, H., Darabi, H., Banerjee, P., and Liu, J. (2007). Survey of wireless indoor positioning techniques and systems. *Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on*, 37(6):1067–1080.

[90] Ltd., R. P. T. (2019). ADD-ON BOARDS AND HATs. *Online at* ***https://github.com/raspberrypi/hats***. *Accessed on 11 April 2019.*

[91] M. Kurtzer, G. (2016). Singularity 2.1.2 - Linux application and environment containers for science. 10.5281/zenodo.60736.

[92] Madhavapeddy, A. and Scott, D. J. (2013). Unikernels: Rise of the virtual library operating system. *Queue*, 11(11):30.

[93] Manolakis, D. E. (1996). Efficient solution and performance analysis of 3-D position estimation by trilateration. *Aerospace and Electronic Systems, IEEE Transactions*, pages 1239–1248.

[94] Mariakakis, A. T., Sen, S., Lee, J., and Kim, K. (2014). Sail: Single access point-based indoor localization. In *Proceedings of the 12th annual international conference on Mobile systems, applications, and services*, pages 315–328. ACM.

[95] Marksteiner, S., Jiménez, V. J. E., Valiant, H., and Zeiner, H. (2017). An overview of wireless IoT protocol security in the smart home domain. In *2017 Internet of Things Business Models, Users, and Networks*, pages 1–8. IEEE.

[96] Maxwell, Jake and Purnell, Newley (2016). Singapore Is Taking the 'Smart City' to a Whole New Level. `https://www.wsj.com/articles/singapore-is-taking-the-smart-city-to-a-whole-new-level-1461550026`. Accessed on 4 April 2019.

[97] Mayer, H. (1999). Air pollution in cities. *Atmospheric environment*, 33(24):4029–4037.

[98] Meffert, C., Clark, D., Baggili, I., and Breitinger, F. (2017). Forensic State Acquisition from Internet of Things (FSAIoT): A general framework and practical approach for IoT forensics through IoT device state acquisition. In *Proceedings of the 12th International Conference on Availability, Reliability and Security*, page 56. ACM.

[99] Mender (2017). Over-the-air software updates for embedded Linux devices. `https://mender.io/`. Accessed on 26 April 2019.

[100] Merkel, D. (2014). Docker: lightweight Linux containers for consistent development and deployment. *Linux Journal*, 2014(239):2.

[101] Miorandi, D., Sicari, S., De Pellegrini, F., and Chlamtac, I. (2012). Internet of things: Vision, applications and research challenges. *Ad Hoc Networks*, 10(7):1497–1516.

[102] Mishra, S. and Beaulieu, A. (2004). *Mastering Oracle SQL: Putting Oracle SQL to Work*. O'Reilly Media, Inc.

[103] Mocevicius, R. (2015). *CoreOS Essentials*. Packt Publishing Ltd.

[104] Momjian, B. (2001). *PostgreSQL: Introduction and Concepts*, volume 192. Addison-Wesley New York.

[105] Mora, L. and Bolici, R. (2015). How to become a smart city: Learning from amsterdam. In *International conference on Smart and Sustainable Planning for Cities and Regions*, pages 251–266. Springer.

[106] Moran, D., Bychkov, E., Sherman, I., and Ron, U. (2013). Foldable mobile phone. US Patent 8,406,826.

[107] Nahrstedt, K., Lopresti, D., Zorn, B., Drobnis, A. W., Mynatt, B., Patel, S., and Wright, H. V. (2016). Smart Communities Internet of Things. *arXiv preprint arXiv:1604.02028*.

[108] Nasar, M. and Kausar, M. A. (2019). Suitability Of Influxdb Database For Iot Applications. *International Journal of Innovative Technology and Exploring Engineering*, 8(10):1850–1857.

[109] Negus, C. (2015). *Docker Containers (includes Content Update Program): Build and Deploy with Kubernetes, Flannel, Cockpit, and Atomic*. Prentice Hall Press.

[110] Nijholt, A. (2008). Google home: Experience, support and re-experience of social home activities. *Information Sciences*, 178(3):612–630.

[111] Ono, T., Lida, K., and Yamazaki, S. (2017). Achieving sustainable development goals (SDGs) through ICT services. *FUJITSU Sci. Tech. J*, 53(6):17–22.

[112] Open Data Institute (2016). Breathe Heathrow: Democratising air data to meet local needs. `http://theodi.org/summer-showcase-breathe-heathrow`. Accessed on 4 April 2019.

[113] OpenStreetMap Contributors (2019). OpenStreetMap.

[114] Pfeil, M., Bartoschek, T., and Wirwahn, J. A. (2015). OPENSENSEMAP-A Citizen Science Platform For Publishing And Exploring Sensor Data as Open Data. In *Free and Open Source Software for Geospatial (FOSS4G) Conference Proceedings*, volume 15, page 39.

[115] Platform for Accelerating the Circular Economy (PACE) and the UN E-Waste Coalition (2019). A New Circular Vision for Electronics. *Online*.

[116] Plotly (2016). Plotly. `https://plot.ly/`. Accessed on 1 May 2019.

[117] Poulter, A. J., Johnston, S. J., and Cox, S. J. (2016). SRUP: The secure remote update protocol. In *Internet of Things (WF-IoT), 2016 IEEE 3rd World Forum on*, pages 42–47. IEEE.

[118] Prometheus (2016). Prometheus. `https://prometheus.io/`. Accessed on 26 April 2019.

[119] Purington, A., Taft, J. G., Sannon, S., Bazarova, N. N., and Taylor, S. H. (2017). Alexa is my new BFF: social roles, user satisfaction, and personification of the amazon echo. In *Proceedings of the 2017 CHI Conference Extended Abstracts on Human Factors in Computing Systems*, pages 2853–2859. ACM.

[120] Rappaport, T. S. et al. (1996). *Wireless communications: principles and practice*, volume 2. Prentice Hall PTR New Jersey.

[121] Raymond, S. A., Gordon, G. E., and Singer, D. B. (1998). Health monitoring system. US Patent 5,778,882.

[122] Recupero, D. R., Presutti, V., Consoli, S., Gangemi, A., and Nuzzolese, A. G. (2015). Sentilo: frame-based sentiment analysis. *Cognitive Computation*, 7(2):211–225.

[123] rkt (2017). rkt. `https://github.com/rkt/rkt`. Accessed on 26 April 2019.

[124] Roberts, S., Arseneault, L., Barratt, B., Beevers, S., Danese, A., Odgers, C. L., Moffitt, T. E., Reuben, A., Kelly, F. J., and Fisher, H. L. (2019). Exploration of NO2 and PM2. 5 air pollution and mental health problems using high-resolution data in London-based children from a UK longitudinal cohort study. *Psychiatry research*, 272:8–17.

[125] Salvador, O. and Angolini, D. (2014). *Embedded Linux Development with Yocto Project*. Packt Publishing Ltd.

[126] Samteladze, N. and Christensen, K. (2012). Delta: Delta encoding for less traffic for apps. In *37th Annual IEEE Conference on Local Computer Networks*, pages 212–215. IEEE.

[127] Sayahi, T., Butterfield, A., and Kelly, K. (2019). Long-term field evaluation of the plantower pms low-cost particulate matter sensors. *Environmental pollution*, 245:932–940.

[128] Schwab, K. (2015). The Fourth Industrial Revolution: what it means and how to respond. World Economic Forum.

[129] Scuro, C., Sciammarella, P. F., Lamonaca, F., Olivito, R. S., and Carni, D. L. (2018). IoT for structural health monitoring. *IEEE Instrumentation & Measurement Magazine*, 21(6):4–14.

[130] Seaton, A., Godden, D., MacNee, W., and Donaldson, K. (1995). Particulate air pollution and acute health effects. *The lancet*, 345(8943):176–178.

[131] seed (2017). Grove System. `http://wiki.seeed.cc/Grove_System/`. Accessed on 27 April 2019.

[132] Shacklette, M. (1995). Linux Operating System. *Handbook of Computer Networks: LANs, MANs, WANs, the Internet, and Global, Cellular, and Wireless Networks, Volume 2*, pages 78–90.

[133] Shchekotov, M. (2014). Indoor localization method based on Wi-Fi trilateration technique. In *Proceeding of the 16th conference of fruct association.*

[134] Sigoure, B. (2010). OpenTSDB: The distributed, scalable time series database. *Proc. OSCON*, 11.

[135] Singh, R., Gehlot, A., Gupta, L. R., Singh, B., and Swain, M. (2019). Internet of Things with Raspberry Pi and Arduino.

[136] Southern Water (2017). Your AMR meter. `https://www.southernwater.co.uk/your-amr-meter`. Accessed on 30 March 2019.

[137] spybike (2014). Anti-theft GPS tracking devices for Bicycles. `http://www.integratedtrackers.com/`. Accessed on 11 April 2019.

[138] Stackowiak, R. (2019). IoT Central and Solution Accelerators. In *Azure Internet of Things Revealed*, pages 119–143. Springer.

[139] SWupdate (2010). SWupdate. *Online*. `https://github.com/sbabic/swupdate`. Accessed on 26 April 2019.

[140] The Freecycle Network (2017). Freecycle UK. `https://www.freecycle.org/`. Accessed on 31 March 2019.

[141] The Open Knowledge Lab (2017). Fine Dust Sensor for Citizen Science Project. `http://luftdaten.info/`. Accessed on 27 April 2019.

[142] The Things Network (2017). The Things Network. `https://www.thethingsnetwork.org/`. Accessed on 28 April 2019.

[143] TP-Link (2017). 2.4GHz 5dBi Indoor Omni-directional Antenna. Technical report.

[144] Turnbull, J. (2014). *The Docker Book: Containerization is the new virtualization.* James Turnbull.

[145] Vangelista, L., Zanella, A., and Zorzi, M. (2015). Long-range IoT technologies: The dawn of LoRa. In *Future Access Enablers of Ubiquitous and Intelligent Infrastructures*, pages 51–58. Springer.

[146] Vasisht, D., Kumar, S., and Katabi, D. (2016). Decimeter-level localization with a single WiFi access point. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 165–178.

[147] Vavilapalli, V. K., Murthy, A. C., Douglas, C., Agarwal, S., Konar, M., Evans, R., Graves, T., Lowe, J., Shah, H., Seth, S., et al. (2013). Apache Hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th annual Symposium on Cloud Computing*, page 5. ACM.

[148] Velosa, A., Schulte, W. R., and Benoit, J. L. (2016). Hype cycle for the internet of things, 2016.

[149] Walters, C., Poo-Caamaño, G., and German, D. M. (2013a). The future of continuous integration in GNOME. In *Proceedings of the 1st International Workshop on Release Engineering*, pages 33–36. IEEE Press.

[150] Walters, C., Poo-Caamaño, G., and German, D. M. (2013b). The future of continuous integration in GNOME. In *Proceedings of the 1st International Workshop on Release Engineering*, pages 33–36. IEEE Press.

[151] Walton, H., Dajnak, D., Beevers, S., Williams, M., Watkiss, P., and Hunt, A. (2015). Understanding the health impacts of air pollution in London. *London: Kings College London, Transport for London and the Greater London Authority*.

[152] Wang, Y. and Shao, L. (2017). Understanding occupancy pattern and improving building energy efficiency through Wi-Fi based indoor positioning. *Building and Environment*, 114:106–117.

[153] Weiser, M. (1991). The computer for the 21st century. *Scientific American*, 265(3):94–104.

[154] Weiser, M. (1993). Some computer science issues in ubiquitous computing. *Communications of the ACM*, 36(7):75–84.

[155] Wick, A. (2012). The HaLVM: A simple platform for simple platforms. *Xen Summit Talk, August*.

[156] Wilder, B. (2012). *Cloud architecture patterns: using Microsoft Azure*. O'Reilly Media, Inc.

[157] World Health Organization (2006). *Air quality guidelines: global update 2005: particulate matter, ozone, nitrogen dioxide, and sulfur dioxide*. World Health Organization.

[158] Yadav, V. and Yadav, V. (2015). Challenging and oppourtunities of Project Ara. *IT INTELLIGENCE INNOVATIONS-2015*, page 1.

[159] Zuniga, J. C. and Ponsard, B. (2016). Sigfox system description. *LPWAN@ IETF97, Nov. 14th*, 25.