SHANI DU PLESSIS

A COMPARATIVE STUDY OF SOFTWARE ARCHITECTURES IN CONSTRAINED-DEVICE IOT DEPLOYMENTS

2021

Shani du Plessis

A Comparative Study of Software Architectures in Constrained-Device IoT Deployments

MSc. in Computer Engineering

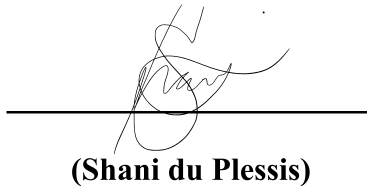
Supervisor:
Prof. Dra. Noélia Correia

Statement of Originality

A Comparative Study of Software Architectures in Constrained-Device IoT Deployments

Declaration of authorship of work: I declare to be the author of this work, which is original and unpublished. Authors and works consulted are properly cited in the text and appear in the list of references included.

Candidate:



(Shani du Plessis)

Copyright ©Shani du Plessis

The University of Algarve has the right, perpetual and without geographical boundaries, to archive and make public this work through printed copies reproduced in paper or digital form, or by any other means known or to be invented, to broadcast it through scientific repositories and allow its copy and distribution with educational or research purposes, noncommercial purposes, provided that credit is given to the author and Publisher.

Work done at the Research Center of Electronics, Optoelectronics and Telecommunications (CEOT)

Acknowledgements

First and foremost, I would like to thank Dra. Noélia Correia for her continuous support. Her support, advice and encouragement proved invaluable, not only for the completion of this research thesis, but throughout my master's degree. I am very grateful that I have had the chance to work with her and I finish this degree feeling inspired for the future.

I would like to thank my colleagues at CEOT for always lending an ear when I was faced with problems or setbacks. I am also grateful to all of the professors that have played an integral part during my time at UAlg, such as Dr. Pedro Guerreiro for his programming enthusiasm and Dr. António Ruano for his support as coordinator and director of this degree.

Lastly, I would like to thank my family for all of their support, my partner for always coming to my aid and my friends, both at UAlg and at home, who are my second family.

Abstract

Since its inception in 2009, the Internet of Things (IoT) has grown dramatically in both size and complexity. One of the areas that has seen significant developments is that of resource-constrained devices. Such devices clearly require careful engineering in order to manage resources such as energy and memory, whilst still ensuring acceptable performance. A number of aspects play a critical role in the engineering of such systems. One such aspect is the choice of software architecture. The microservices architecture appears to be a promising approach for IoT, as suggested by a number of researchers. However, limited research has been done on the implementation of microservices in IoT and resource-constrained devices, and even less research has been done to compare the microservices architecture to the monolithic architecture in such deployments.

The aim of this research thesis was to compare these two architectures in the context of IoT and constrained devices. The two architectures were compared by: energy consumption, runtime performance and memory consumption. To ensure that the results are not specific to a single programming language, each architecture was developed in three different languages: Go, Python and C++. Following a review of different asynchronous messaging protocols, Message Queuing Telemetry Transport was selected. The experiments were conducted on a Raspberry Pi 4, and a number of other hardware devices were used, including sensors, an actuator and a type C USB Tester. Two metrics were used to measure power consumption: maximum instantaneous power consumption and total power consumption. Whilst three metrics were used to measure memory consumption: maximum Resident Set Size (RSS), total RSS and central processing unit (CPU) resource usage. Each experiment was carried out 10 times in order to ensure data validity.

The power consumption results showed that the microservices architecture had, on average, 14,9% higher maximum instantaneous power consumption, whilst the total power consumption of the microservices architecture was only 3,0% greater than that of the monolithic architecture. The runtime results indicated that the microservices architecture had a longer runtime than the monolithic architecture for Go and C++, whilst the inverse was true for Python. When considering memory consumption, it was found that the maximum RSS was 37,1% greater for the microservices architecture. The total RSS results for both architectures were very similar for Go and C++, whilst microservices performed much better for Python. Lastly, the results for CPU usage showed that the monolithic architecture had, on average, 14,9% greater CPU usage than the microservices architecture. It was concluded that, for small-scale applications, the monolithic architecture had better performance across most metrics and languages. It was, however,

recommended that additional research be conducted on larger scale applications to determine the applicability of these results beyond the scope of small-scale applications. In general, there is still much room for research in this area.

Keywords: Internet of Things, resource-constrained devices, software architecture, microservices, monolithic

Resumo

A Web e a Internet das Coisas (WoT/IoT) são áreas empolgantes que, sem dúvida, continuarão a desenvolver-se nos próximos anos. À medida que vão sendo feitos novos desenvolvimentos nestas áreas, e vários tipos de objetos se tornam "Coisas", é expectável que a limitação de recursos seja cada vez mais uma preocupação. Atualmente já existem muitos dispositivos que possuem recursos limitados por vários motivos, como a sua localização em locais difíceis ou remotos (ex: sensores implantáveis ou sensores de erupção vulcânica) ou necessidade de trabalhar enquanto estão em movimento (ex: dispositivos vestíveis). Assim sendo, a necessidade de usar-se os recursos de forma eficiente será cada ver maior.

O objetivo primordial desta tese foi o de analisar a utilização de recursos por parte de uma aplicação IoT, considerando duas arquiteturas de software diferentes, implementada num dispositivo com poucos recursos. O dispositivo escolhido é um Raspberry Pi 4, dado ser um dispositivo embarcado bastante adequado para realização de testes. As arquiteturas que foram comparadas neste estudo foram: microsserviços e monolítica. Para garantir que os resultados não fossem específicos da linguagem utilizada, o desenvolvimento foi feito em três linguagens de programação: Go, Python e C++. Embora seja possível encontrar estudos que analisam como as linguagens de programação utilizam os recursos, apenas foi encontrado um estudo cujo foco é a eficiência energética, memória e tempo de execução em dispositivos com recursos limitados, não tendo sido encontrado nenhum estudo que compare o desempenho das arquiteturas de software em dispositivos com recursos limitados.

A adoção de uma arquitetura de microsserviços em ambientes WoT/IoT tem vantagens, como modularidade, flexibilidade e facilidade de manutenção. Vários estudos referem que esta arquitetura é adequada para WoT/IoT, pois compartilha muitos dos mesmos objetivos. WoT/IoT é inerentemente dinâmico e tem muitos pontos de extremidade, o que pode apresentar desafios de desenho e implementação. Uma arquitetura como microsserviços pode explorar estas caracteristicas, transformando estes desafios em vantagens. No entanto, não foi encontrada investigação que compare o desempenho da arquitetura de microsserviços com a arquitetura monolítica, especialmente no contexto
IoT, tendo sido este o foco desta tese.

A escolha do protocolo de transferência de mensagens, para comunicação entre os vários microsserviços, foi também analisada. Um protocolo de transferência leve será o mais adequado, para dispositivos que têm recursos limitados, e três opções foram consideradas em mais

profundidade: MQTT (Message Queuing Telemetry Transport), AMQP (Advanced Message Queuing Protocol) e CoAP (Constrained Application Protocol). Da análise feita, verificou-se que o MQTT é limitado na qualidade de serviço, seguranca e confiabilidade que oferece, isto quando comparado com o AMQP, sendo por isso um protocolo mais leve. Ao comparar-se MQTT e CoAP, verificou-se que ambos os protocolos oferecem vários benefícios, tendo o MQTT sido escolhido para os testes realizados.

A abordagem técnica que foi adotada é descrita em detalhe, incluindo os componentes de hardware necessários para o projeto e o software necessário para a recolha de medições. Foi ainda delineada uma metodologia experimental, a qual foi seguida de perto. Foram obtidos resultados que permitem analisar em detalhe o consumo de energia, o tempo de execução e o consumo de memória. Quanto ao consumo de energia, em específico, recolhe-se o consumo de energia instantâneo máximo e o consumo de energia total. Desta análise verificou-se que o consumo de energia instantâneo máximo da arquitetura de microserviços foi, em média, e em todas as linguagens, 14.9% maior do que o consumo obtido para a arquitetura monolítica. Verificou-se também que a linguagem Go tem o maior consumo de energia instantâneo máximo, para ambas as arquiteturas, enquanto que o Python e o C++ tiveram medidas semelhantes.

Os resultados para o consumo total de energia (durante o tempo de execução total) foram ligeiramente diferentes. Ao comparar-se as duas arquiteturas, deduziu-se que os valores de consumo de energia eram muito semelhantes e, em média, e em todas as linguagens, a arquitetura de microsserviços consumia apenas 3.0% a mais que a arquitetura monolítica. Também foi verificado que ao usar-se a arquitetura monolítica, o consumo total de energia era quase idêntico em todos as linguagens. Com a arquitetura de microserviços, o Python teve o maior consumo, seguido do Go e C++, embora os valores não tenham diferido muito. Também ficou claro que, embora o consumo de energia instantâneo máximo possa ser útil para entender os requisitos de energia de pico, não é diretamente proporcional ao consumo de energia total. Por exemplo, o Python teve o menor consumo de energia instantâneo máximo, mas o maior consumo de energia total.

A segunda parte dos resultados considerou o desempenho no que diz respeito ao tempo de execução. Considerando apenas a arquitetura, verificou-se que a quitetura de microsserviços tinha um tempo de execução maior do que a arquitetura monolítica para Go e C++, enquanto o inverso era verdadeiro para o Python, o que pode estar relacionado com a otimizacao de simultaneidade vinculada à unidade central de processamento (CPU), pelas diferentes linguagens. Ao comparar o tempo de execução das linguagens de programacao, os resultados

ficaram amplamente em linha com as expetativas. C++ teve o menor tempo de execução, seguido de perto pelo Go. O Python teve um tempo de execução significativamente mais longo, o que faz sentido já que o Python é a única linguagem interpretada que foi usada neste projeto. Foi interessante notar que o tempo de execução do Python foi muito maior ao usar-se uma arquitetura monolítica do que ao usar-se uma arquitetura de microserviços, o que não foi o caso do C++ ou Go. Com a arquitetura de microserviços, o Python teve um tempo de execução médio 319.4% maior do que o do C++, enquanto que o tempo de execuçao médio do Go foi 31.5% maior do que o do C++. Diferenças semelhantes foram observadas para a arquitetura monolítica.

O consumo de memória foi medido usando três métricas diferentes: tamanho do conjunto residente (RSS) máximo, RSS total e uso de CPU. A comparação do RSS máximo, em cada arquitetura, mostrou que o RSS máximo para a arquitetura de microserviços foi 37.1% maior do que para a arquitetura monolítica. A diferença foi especialmente significativa para Python (65.9% de diferença). Verificou-se que o Go teve um RSS máximo significativamente maior do que as outras linguagens, para ambas as arquiteturas. O Python teve o menor RSS máximo na arquitetura monolítica, enquanto que o C++ teve o menor para a arquitetura de microserviços. Os resultados para o RSS total foram muito diferentes do RSS máximo, tanto por
arquitetura como por linguagem usada. Mais concretamente, as medidas totais de RSS para Go e C++ nao diferiam muito por arquitetura, embora houvesse uma grande diferença quando comparado com o Python. Em média o RSS total foi 127.0% maior para a arquitetura monolítica do que para a arquitetura de microserviços, ao usar-se Python. Comparando por linguagem, o RSS total do Python foi significativamente maior do que para as outras duas linguagens, especialmente para a arquitetura monolítica, enquanto o Go e C++ tiveram medições RSS totais muito semelhantes.

A última métrica de consumo de memória considerada foi o uso médio da CPU. Verificou-se que a arquitetura monolítica teve, em média, 14.9% maior utilização de CPU do que a arquitetura de microserviços, e a maior diferença foi observada para o Python. Uma comparação por linguagem mostrou que o Go teve a maior utilização de
CPU, para ambas as arquiteturas. O C++ teve a segunda maior utilização de CPU, e o Python teve a menor utilização. Estas conclusões foram de encontro às expectativas, já que o Go tem processos integrados leves (rotinas Go), podendo otimizar a utilização de CPU.

Esta dissertação produziu, em geral, resultados muito interessantes, uns mais esperados que outros. Os resultados mostraram que a arquitetura monolítica teve melhor desempenho na maioria das métricas, ou seja, consumo de energia instantâneo máximo, consumo de energia total

(apenas para o Go e Python), tempo de execução geral (apenas para o Go e C++), RSS e CPU máximos. Deste modo, é possível concluir que ao implementar-se aplicações de pequena escala, em dispositivos IoT, a arquitetura monolítica pode oferecer mais benefícios. É bastante provável, no entanto, que a arquitetura de microserviços possa superar a arquitetura monolítica em aplicações de maior escala. A dimensão da aplicação deve, por isso, ser considerada ao escolher-se uma arquitetura de software.

Claramente, ainda existe muito espaço para contribuição nesta área de investigação. A investigação encontrada sobre o desempenho da arquitetura de microserviços, em comparação com a arquitetura monolítica, é limitafa e não foi encontrada investigação no contexto da IoT. Isto acaba por ser surpreendente, pois muitas empresas estão já a adotar microserviços e tem havido um aumento das pesquisas relacionadas com esta arquitetura. Assim sendo, compreender quais as vantagens e desvantagens desta arquitetura tornou-se muito pertinente. Embora esta dissertação tenha analisado a arquitetura de microsserviços, e tendo esta sido comparada com a arquitetura monolítica, considerando diferentes linguagens, a análise é feita numa escala relativamente pequena, quanto ao número de componentes de serviço, e num único dispositivo embarcado. A análise de aplicações de maior escala forneceria, certamente, perceções adicionais muito valiosas.

Palavras-chave: IoT, dispositivos com restrição de recurso, arquitetura de software, microserviços, monolítico.

Table of Contents

Acknowledgements .... *v*

Abstract .... *vi*

Resumo .... *viii*

List of Figures .... *xiv*

List of Tables .... *xvi*

Nomenclature .... *xviii*

1. Introduction and Research Objective *1*

1.1 Introduction 1

1.2 Research Objective and Expected Contribution 1

1.3 Brief Overview of Report Content 2

2. Background *3*

2.1 Constrained Devices 3

2.2 Web of Things and Internet of Things 4

2.3 Microservices Architecture, Monolithic Architecture and Messaging Protocol 5

2.4 Go, Python and C++ 7

3. Literature Review of the State of the Art *9*

3.1 Microservices vs. Monolithic as an Architecture for IoT/WoT 9

3.2 Messaging Protocol 11

3.3 Programming Language Efficiency for Embedded Devices 15

4. Technical Approach *17*

4.1 Software Architecture 17

4.2 Hardware Architecture 18

4.3 Data Collection 19

5. Methodology and Evaluation *24*

5.1 Microservices Application Development 24

5.2 Data Collection 24

5.3 Evaluation and Comparison 25

6. Results and Discussion *26*

6.1 Power Consumption 26

6.2 Runtime 29

6.3 Memory Consumption 33

7. Conclusion and Future Work *38*

8. References *42*
Appendix A – Code *46*
A.1 – Bash Code **46**
A.2 – Microservices Code **48**
A.3 – Monolithic Code **73**
Appendix B – Results *91*
B.1 – Power Results **91**
B.2 – Runtime Results **92**
B.3 – Memory Results **95**

List of Figures

Figure 1: *IETF RFC 7228 showing classification of constrained devices (KiB = 1024 bytes) [Keranen, A., Ersue, M. and Bormann, C., 2014]* 3

Figure 2: *Visual representation of the architectural difference between a monolithic application and a microservices application [Fowler, M., 2014]* 6

Figure 3: *Graphical representation of MQTT architecture [Paessler.com. 2018]* 11

Figure 4: *Graphical representation of AMQP architecture [Bahashwan, A. and Manickam, S., 2018].* 12

Figure 5: *Graphical representation of MQTT and COAP architectures [Mishra, H., 2019].* 13

Figure 6: *Graphical representation of the microservices architecture that was used in this project.* 17

Figure 7: *Graphical representation of the monolithic architecture that was used in this project.* 18

Figure 8: *Photograph showing the experimental setup of the breadboard and sensors* 19

Figure 9: *Photograph showing the experimental setup of the Type C USB Tester.* 19

Figure 10: *Snapshot of the runtime outputs when using the language specific time library (this is an example of Go)* 21

Figure 11: *Snapshot of the output when running the GNU time command for peak memory usage* 22

Figure 12: *Snapshot of the Syrupy log that was used to capture memory consumption measurements at regular intervals.* 23

Figure 13: *Bar chart showing the comparative maximum instantaneous power consumption of different languages and architectures* 27

Figure 14: *Bar chart showing the comparative total power consumption of different languages and architectures.* 28

Figure 15: *Bar chart showing the comparative overall runtime of different languages and architectures.* .. 30

Figure 16: *Box plot showing the comparative overall runtime of Go and C++ when using a microservices architecture.* 31

Figure 17: *Box plot showing the comparative overall runtime of Go and C++ when using a monolithic architecture.* 32

Figure 18: *Bar chart showing the comparative maximum RSS of different languages and architectures.* .. 34

Figure 19: *Bar chart showing the comparative total RSS of different languages and architectures.* 35

Figure 20: *Bar chart showing the comparative CPU usage of different languages and architectures.* 36

Figure A.1: *Screenshot showing the bash script to launch the Go microservices application* *46*

Figure A.2: *Screenshot showing the bash script to launch the Go monolithic application* *46*

Figure A.3: *Screenshot showing the bash script to launch the Python microservices application* *46*

Figure A.4: *Screenshot showing the bash script to launch the Python monolithic application* *47*

Figure A.5: *Screenshot showing the bash script to launch the C++ microservices application* *47*

Figure A.6: *Screenshot showing the bash script to launch the C++ monolithic application* *47*

List of Tables

Table 1: *Summary of Maximum Instantaneous Power Consumption for different languages and architectures.* 26

Table 2: *Table showing the percentage difference in maximum instantaneous power consumption between different architectures.* 27

Table 3: *Summary of Total Power Consumption for different languages and architectures.* 28

Table 4: *Table showing the percentage difference in total power consumption between different architectures.* 29

Table 5: *Table summarising the runtime measured for different languages and architectures.* 30

Table 6: *Table summarising the runtime differences between architectures.* 33

Table 7: *Table summarising the maximum RSS of different languages and architectures* 33

Table 8: *Table summarising the maximum RSS differences between architectures.* 34

Table 9: *Table summarising the total RSS of different languages and architectures* 35

Table 10: *Table summarising the total RSS differences between architectures.* 36

Table 11: *Table summarising the CPU usage of different languages and architectures* 36

Table 12: *Table summarising the CPU usage differences between architectures.* 37

Table B.1: *Table with full maximum instantaneous power results for Microservices Architecture* 91

Table B.2: *Table with full maximum instantaneous power results for Monolithic Architecture* 91

Table B.3: *Table with full total power results for Microservices Architecture* 91

Table B.4: *Table with full total power results for Monolithic Architecture* 92

Table B.5: *Table with full DHT runtime results for Microservices Architecture* 92

Table B.6: *Table with full LED runtime results for Microservices Architecture* 92

Table B.7: *Table with full PIR runtime results for Microservices Architecture* 93

Table B.8: *Table with full overall runtime results for Microservices Architecture – concurrent* 93

Table B.9: *Table with full overall runtime results for Microservices Architecture – cumulative* 93

Table B.10: *Table with full DHT runtime results for Monolithic Architecture* 94

Table B.11: *Table with full LED runtime results for Monolithic Architecture* 94

Table B.12: *Table with full PIR runtime results for Monolithic Architecture* 94

Table B.13: *Table with full overall runtime results for Monolithic Architecture* 95

Table B.14: *Table with full Maximum RSS results for Microservices Architecture* *95*

Table B.15: *Table with full Maximum RSS results for Monolithic Architecture* *95*

Table B.16: *Table with full Total RSS results for Microservices Architecture* *96*

Table B.17: *Table with full Total RSS results for Monolithic Architecture*... *96*

Table B.18: *Table with full CPU results for Microservices Architecture* *96*

Table B.19: *Table with full CPU results for Monolithic Architecture* *97*

Nomenclature

Abbreviations

Amazon SQS	:	*Amazon Simple Queue Service*
AMQP	:	*Advanced Message Queuing Protocol*
API	:	*Application Programming Interface*
CMQ	:	*Cloud Message Queue*
CoAP	:	*Constrained Application Protocol*
CPU	:	*Central Processing Unit*
DHT	:	*Digital Humidity and Temperature*
GPIO	:	*General Purpose Input/Output*
HTTP	:	*Hypertext Transfer Protocol*
IETF	:	*Internet Engineering Task Force*
IoT	:	*Internet of Things*
IP	:	*Internet Protocol*
JSON	:	*JavaScript Object Notation*
kB	:	*Kilobyte*
LED	:	*Light Emitting Diode*
MB	:	*Megabyte*
MQTT	:	*Message Queuing Telemetry Transport*
MQTT-SN	:	*Message Queuing Telemetry Transport for Sensor Networks*
QoS	:	*Quality of Service*
RAM	:	*Random Access Memory*
RAPL	:	*Running Average Power Limit*
REST	:	*Representational State Transfer*
RFC	:	*Request for Comment*
ROM	:	*Read-Only Memory*
RSS	:	*Resident Set Size*
TCP	:	*Transmission Control Protocol*
UDP	:	*User Datagram Protocol*
WoT	:	*Web of Things*

1. Introduction and Research Objective

1.1 Introduction

In 1960, the first known embedded system was used for developing the Apollo Guidance System, which marked the birth of a new world - the world of embedded devices. With the advent of the internet years later, embedded devices began to morph into what is now known as the Internet of Things (IoT) – a term coined in 1999 to describe objects that are able to communicate [Rungta, K., 2020][Lueth, K. L., 2014]. Today the term IoT encompasses a plethora of internet connected devices, from smart watches to smart fridges, from cars to eating utensils [Komiyama, N., 2017]. By 2008, there were already more "Things" connected to the internet than there were people, and it is estimated that by the end of 2021, there will be between 25 and 50 billion web-connected Things. This clearly shows the trend that has taken hold of IoT [Cook, S., 2020][Burhan, M. *et al.*, 2018].

With the increase in number of IoT devices, comes the diversification of the types and uses of these devices. One such group/class of devices is referred to as "constrained devices". Constrained devices are devices that have limited resources, e.g. computational power, memory or energy. A number of factors affect the management and usage of these resources, such as the type of device, the operating system, the conditions of use, the choice of programming language and the software architecture. Once a device has been selected, some of these factors can be adjusted to improve resource management, e.g. choice of programming language, whilst others cannot be altered, such as the deployment conditions in many cases. Since resource management is critical to the efficacy of constrained devices, it is an area that is well worth exploring.

1.2 Research Objective and Expected Contribution

The efficient use of resources is a critical aspect for many embedded devices. Therefore, the choice of software architecture is very important as it can have a significant impact on energy, memory and runtime performance. The aim of this research thesis was to analyse and compare the energy, memory and runtime performance of an embedded device application using two software architectures and three programming languages (to avoid language-specific results). The two software architectures that were compared are the microservices and monolithic architectures, whilst Go, C++ and Python are the three programming languages that were used.

Given the nature of resource-constrained devices, the optimization of energy and memory use, as well as runtime, is a critical concern in IoT and the Web of Things (WoT). Hence, the choice

of software architecture, programming language and programming language implementation is an important decision. Thus far, limited research has been conducted on the impact of software architecture on resource-constrained device performance (see the Section on state of the art). It is therefore expected that this research will provide valuable insights for the choice of software architecture in the field of IoT and WoT.

1.3 Brief Overview of Report Content

The remainder of this report will delve into a number of topics in more depth. Section 2 provides background on a number of key topics; Section 3 provides a literature review of the state of the art. Section 4 gives an overview of the technical approach that was taken. Section 5 discusses the methodology that was followed when conducting the experiments as well as the evaluation that was done. The results from this research thesis are shown and discussed in detail in Section 6. Finally, Sections 7 and 8 are dedicated to the conclusion and references, respectively. An appendix can be found at the end of this document.

2. Background

2.1 Constrained Devices

Constrained devices include sensors (e.g. motion or temperature sensor), actuators (e.g. LED lights or motors), aggregators, microcontrollers and many more. Constrained devices are typically devices that are built to handle a specific application purpose and are usually connected to a gateway device, which acts as the intermediate for communication to the internet. The constraints on resources of sensors, microcontrollers, etc. can range from code complexity, i.e. read-only memory (ROM)/Flash, and available random-access memory (RAM) to processing capabilities and availability of power [Nagasai, M., 2017]. Given the vast array of resource constrained devices, IETF published RFC 7228 to classify these devices into 3 classes, as shown in Figure 1.

Name	data size (e.g., RAM)	code size (e.g., Flash)
Class 0, C0	<< 10 KiB	<< 100 KiB
Class 1, C1	~ 10 KiB	~ 100 KiB
Class 2, C2	~ 50 KiB	~ 250 KiB

***Figure 1**: IETF RFC 7228 showing classification of constrained devices (KiB = 1024 bytes) [Keranen, A., Ersue, M. and Bormann, C., 2014]*

Class 0 devices have severe constraints on memory and processing capabilities; hence they are not able to communicate directly with the internet and require some gateway node in order to connect. Class 1 devices are able to run low power IoT protocols, such as Constrained Application Protocol (CoAP) and Message Queuing Telemetry Transport for Sensor Networks (MQTT-SN) running over User Datagram Protocol (UDP). Class 2 devices have constraints similar to mobile phones and, therefore, are able to run similar protocols as mobile phones. As with mobile phones, power source availability is still a major concern. Therefore, low power protocols are still preferred [Nagasai, M., 2017]. This research thesis is aimed at devices that fall into classes 1 and 2. Although Raspberry Pi 4 falls into class 2, it serves as an ideal testing device as the relative energy consumption, runtime and memory performance will be valuable information for both classes.

IETF published an additional RFC (RFC 6606), which differentiates between devices that have regular access to electricity and those that do not. The former is referred to as "power-affluent" and the latter is referred to as "power-constrained" [Gomez, C. *et al.*, 2012]. Although the

Raspberry Pi used in this project has access to power, the experiments that were done to determine energy efficiency will primarily benefit energy-constrained devices. Energy and memory efficiency are clearly critical aspects for resource-constrained devices. However, runtime can also be a critical factor for these devices. For example, a fire sensor is required to have a very fast runtime so that a client/subscriber is informed of a fire as soon as possible. Similarly, a medical or security sensor/actuator should be able to execute code quickly in order to meet functional requirements. As such, runtime performance has also been selected as a key parameter to be analysed in this project. It is important to note that real-time performance behaviours, such as runtime, depend on a number of factors, such as the operating system and any obscure background processes that may introduce noise into the measurements. In order to limit this noise, all background processes were shut off, as recommended by Hindle et al. [Hindle, A. *et al.*, 2014].

2.2 Web of Things and Internet of Things

Connectivity is an important factor within the area of Internet of Things. If devices are unable to communicate with one another over the internet, they are simply isolated Things. However, within the world of IoT, the billions of Things use a vast array of protocols, software and hardware, therefore connectivity is very challenging. The fact that there is such heterogeneity in connectivity of devices can in part be explained by the fact that in the early years of IoT, most of the attention was paid to the lower layers of the network stack in order to find ways for devices, such as wearables and kitchen items, to sense and send data.

In those early days, little attention was paid to interconnectivity and interoperability between devices, and this lack of interoperability has only increased over time as a wider range of standards and protocols has been introduced in an attempt to standardize IoT, which ironically only exacerbated the problem. One of the promising approaches/solutions to this problem is known as the Web of Things (WoT). WoT was proposed as a way to interconnect devices using the already widely implemented web. Given the wide adoption of the web, as well as the fact that the web is, for the most part, simple and open source, it appeared to be an ideal candidate for the standardization of the top layer of IoT [Bolar, T., 2020].

In addition to improved interoperability, WoT offers many benefits over its IoT counterpart, such as open and extensible standards, maintainability, loose coupling and security [Guinard, D. and Trifa, V., 2016]. Arguably one of the most important aspects of WoT is the use of machine-

understandable metadata, such as JavaScript Object Notation (JSON)[1] to describe and store information about a Thing. This concept was initially proposed in the "Building the Web of Things" book [Guinard, D. and Trifa, V., 2016] in 2016 and has since evolved into a much bigger and more well-defined concept, reflected, for example, in the W3C Thing Description [W3.org. 2019]. Within the WoT, a Thing is an abstract representation of a physical device, and the Thing Description provides information on physical device properties and statuses (e.g. temperature), possible actions, and other more complex aspects like security configurations [Parwej, Dr. Firoj & Akhtar, Nikhat & Perwej, Dr. Yusuf., 2019]. Since the research in this thesis was aimed at analysing the performance of the device itself and not on internet connectivity, the data model proposed by Guinard and Trifa (2016) was used due to its simplicity. The choice of adoption of a particular data model does not affect the usefulness and applicability of the conclusions drawn in this research thesis.

2.3 Microservices Architecture, Monolithic Architecture and Messaging Protocol

The choice of architecture is a critical consideration for embedded devices. Given the loose coupling of the WoT approach and the high volatility within WoT/IoT, microservices is clearly a standout option for software architecture. By definition, WoT has a huge deployment ecosystem and a large number of end points. Additionally, the Things are inherently dynamic whether it is spatially (e.g. wearables), in terms of time (e.g. switching on and off), in terms of purpose (e.g. what is being monitored may change) or interchangeability (e.g. one device may be changed to a different, more advanced one). This leads to very complex networks that are difficult to implement and even more difficult to maintain [Babaria, U., 2018]. These are all compelling reasons for the use of the microservices architecture.

Microservices is a software application development approach that makes use of independently deployable, modular services. The concept of a microservice is well explained by Thones, who stated that a microservice is a small application that can be deployed independently, scaled independently, tested independently and has only one responsibility [Thones, J. 2015]. Typically, each of these services is organized around a specific business or technical capability, is loosely coupled and, due to the size and modularity, highly maintainable, testable and scalable [Richardson, C., 2019].

---

[1] Widely used lightweight data interchange format

By comparison, the monolithic architecture is often considered to be the traditional approach for building applications and comprises a single code base for all its services and functionalities. This approach is often easier to develop and deploy at the outset but becomes difficult to manage as any updates or changes require accessing the whole code base [Gnatyk, R., 2018]. However, for smaller applications, end-to-end testing and debugging are often much easier for monolithic applications than for microservices applications since all application logic is contained in a single unit. Figure 2 provides a good visual explanation of the differences between a traditional monolithic application and a microservices application.

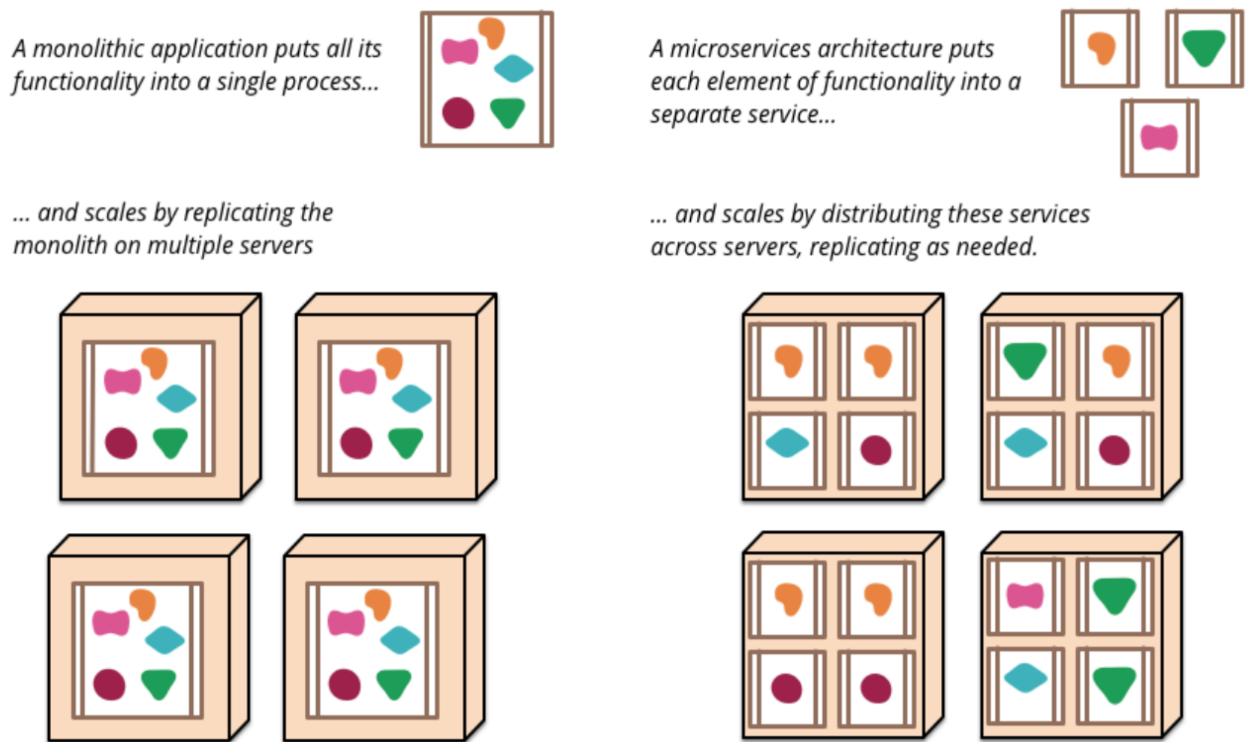


Figure 2: Visual representation of the architectural difference between a monolithic application and a microservices application [Fowler, M., 2014].

It is clear from Figure 2 that the microservices approach enables flexibility, modularity and scalability, whilst the monolithic approach offers distinct boundaries and set functionalities. The modularity of the microservices approach means that different programming languages can be used in different modules/components and that different modules/components can be distributed across different locations (embedded device, cloud, server, etc.).

Although the microservices architecture offers many clear benefits, there are also a few potential disadvantages. Individual microservice components are typically simple but the microservices ecosystem as a whole can become complex due to the number of moving parts and communication between these parts. The size of each component can also be a difficult decision as components that are too large tend towards monolithic behaviour, whilst components that are

too small may just transfer complexity from the component to the intercommunication. There are some other potential draw backs like hackability, third party dependencies, etc. [Atchison, L., Wieldt, T. and Paul, F., 2018]. Evidently, the monolithic architecture also has a few disadvantages such as scalability, which can only be done for the entire application, not individual functionalities. Large monolithic applications also become very difficult to understand and it can be difficult to anticipate the impact of an update/change on the application. Overall, the microservices architecture appears to be a promising option for IoT/WoT and embedded devices. However, further research is required to verify this.

An obvious question that arises after inspection of Figure 2 is about how the components communicate in the microservices architecture. Components communicate using "messages", which is a broad term for any inter-process communication protocol such as Hypertext Transfer Protocol (HTTP), Transmission Control Protocol (TCP), Advanced Message Queuing Protocol (AMQP), etc. Most frequently, communication between microservice components is designed to be flexible and event driven. The two most common types of protocols are HTTP and lightweight asynchronous messaging protocols, e.g. AMQP [De la Torre, C., Wagner, B. and Rousos, M., 2020]. Due to the resource constrained nature of many embedded devices, it was decided that a lightweight messaging protocol, instead of HTTP, would be used in this project.

Asynchronous messaging protocols are generally event driven. With this approach, microservices communicate by exchanging messages via a bus. This approach further enables loose coupling and also means that service discovery is not needed. There are a number of different asynchronous messaging protocols, such as Amazon Simple Queue Service (Amazon SQS), CoAP, which can also operate in synchronous mode, Cloud Message Queue (CMQ), AMQP and Message Queuing Telemetry Transport (MQTT). Although each of these protocols have their own merits, CoAP, AMQP and MQTT are three of the most widely used publish-subscribe messaging protocols and, therefore, the choice of messaging protocol for this project was limited to these three protocols. CoAP, AMQP and MQTT are discussed in further detail in Section 3.2.

2.4 Go, Python and C++

The following three programming languages were selected for comparison in this research thesis: Go, Python and C++. These languages were selected as two are compiled languages (Go and C++) and one is an interpreted language (Python). Go is a compiled language but is programmed similarly to a dynamic-typed interpreted language [Go Documentation., 2016]. Compiled languages have the advantage that the source code is translated into machine code by a compiler.

This results in very efficient machine code that can be executed many times. By comparison, interpreted languages must be parsed, interpreted, and executed every time that the program is run. Hence the overhead of translating from source code to machine code is incurred every time. Interpreted languages are, however, usually more flexible and offer some advantages such as dynamic typing and less lines of code [Blokdyk, G., 2018].

C++ is a general-purpose programming language that was released in 1985 and is an extension of the C programming language. C++ was designed with a focus on system programming and embedded/resource-constrained systems, hence performance, efficiency and flexibility were key design considerations [Stroustrup, B., 2013]. Go was first released in 2009 and was designed by Google engineers as a modern approach to today's software engineering paradigm, i.e. scalable and cloud-based. Go is built for concurrency, can be compiled on most machines and is simple to learn [Biggs, J., & Popper, B., 2020]. Python is an interpreted high-level, general-purpose programming language that was released in 1991. It was designed with an emphasis on code readability and simplicity. It fully supports object-oriented and structured programming and is dynamically typed [Kuhlman, D., 2011]. Although the primary focus of this thesis was on comparing software architectures, this research should also provide some interesting insights into the comparison of runtime performance, energy consumption and memory consumption of these languages.

3. Literature Review of the State of the Art

3.1 Microservices vs. Monolithic as an Architecture for IoT/WoT

As discussed in the introduction, given the distributed and dynamic nature of IoT and WoT, the microservices architecture stands out as a highly promising approach in many contexts. In fact, as early as 2005 the term "Micro-Web-Services" was introduced by Peter Rodgers (2005) at the Web Services Edge conference [Rodgers, P., 2005]. Over recent years, substantial research has been done to determine the aptness of the microservices architecture for IoT and WoT. Santana, Alencar and Prazeres (2018) conducted a thorough review of the use of the microservices architecture to solve many of the problems faced in the field of IoT. The authors carried out a systematic mapping of 18 studies and produced an overview of the state of the art of application of the microservices architecture in IoT and WoT. They found that across all of the analysed works, the primary focus was on the design phase and that there was still a need for investigation into implementation, evaluation and operation [Santana, C., Alencar, B. and Prazeres, C., 2018].

In-line with the findings of Santana et al. (2018), very few studies could be found that compared the implementation and performance, on a technical level, of the monolithic architecture to the microservices architecture in the context of IoT/WoT. One study, conducted by Al-Debagey and Martinek (2018), compared the two architectures for a web application, although no resource constrained devices were used. The study aimed at comparing the throughput[2] and response time[3] of the two architectures. The results indicated that the two architectures had a similar performance in both metrics. However, when there were only a small number of requests/clients (1000 or less threads), the monolithic architecture had a better throughput. With 2000 threads or more, the microservices architecture performed slightly better. The response times of the two architectures were almost identical regardless of the number of threads [Al-Debagy, O. and Martinek, P., 2018].

Another study, conducted by Tapia et al. (2020), was not related to IoT but provided some valuable insights into the performance of the monolithic and microservices architectures. Following a number of tests, a comparative analysis was done of the results. It was found that the monolithic architecture used less central processing unit (CPU) resources, whilst the microservices architecture consumed less memory. It was also discussed that a monolithic architecture can be more efficient, and incur less overhead, but primarily for small-scale

[2] Number of requests that the application could handle per second

[3] Time that elapses between the request and the response

applications [Tapia, F. *et al.*, 2020]. It can be concluded from these two studies that, in terms of non-IoT applications, both architectures have benefits and drawbacks. In the context of IoT/WoT, a number of studies were conducted that provided some insights into the architectures, albeit in a non-technical sense. These studies are outlined below.

Zeiner, Goller, Jiménez, Salmhofer and Haas (2016) conducted a study that focused on a WoT platform based on the microservices architecture. In this study, the authors built a Web of Things platform that implemented a REST interface using a JSON data format, with the aim of building a responsive, resilient, flexible and message driven system. The findings from the study indicated that the use of the microservices architecture allows the flexibility and scalability that is required for a WoT platform. They also found that maintenance of and changes to the platform were much simpler (compared to a monolithic architecture) and did not compromise the performance of the platform due to the modularity of the microservice components [Zeiner, H. *et al.* 2016].

Another study, conducted by Mena, Criado, Iribarne and Corral (2020), investigated the use of a digital representation of a device based on microservices and the WoT framework, named Digital Dice, to solve the problems faced by resource-limited embedded devices. The authors of this study discussed the advantages of using WoT, such as the Thing Description that is used to define Things in a standard way. They also discussed the advantages of using a microservices architecture, such as the ability to break down complex interactions so that there is one microservice for each interaction. The authors concluded that the use of microservices within the WoT framework enabled the system to achieve flexibility and robustness, and also allowed for easier maintenance and development [Mena, M. *et al.* 2020].

In a study by Butzin, Golatowski and Timmermann (2016), the authors discussed the similarities in goals of microservices and IoT, which included lightweight communication, independently deployable software and minimizing centralized management. The authors concluded that, although microservices and IoT approach their goals from different directions, they share many common goals [Butzin, B., Golatowski, F. and Timmermann, D., 2016]. A study by Santana et al. (2019) proposed the use of microservices to improve the reliability of IoT applications [Santana, C. *et al.,* 2019]. Huang, Lu, Walenstein and Medhi (2017) proposed reconceiving IoT's fundamental unit of construction (a "Thing") as a microservice and argued that the microservices approach for IoT can improve many aspects such as API gateways, distribution of services, uniform service discovery and access control [Lu, D. *et al.*, 2017]. Overall, the use of microservices in the field of IoT and WoT gave favourable results in all of the studies that were reviewed. However, there is no technical evidence to support the use of the microservices

architecture over the monolithic architecture in the context of IoT and resource-constrained devices. Hence it was decided that the performance of the two architectures would be compared on a technical level in this research thesis.

3.2 Messaging Protocol

As mentioned in the Introduction Section, there is an abundance of potential messaging protocols. Given that this project aimed to implement an IoT application on a resource constrained device, only lightweight messaging protocols were explored. Three messaging protocols were identified as options for this project: MQTT, AMQP and CoAP. Since MQTT and AMQP are the most closely related protocols out of the three, they are discussed and compared first. A number of comparative reviews have been done between different messaging protocols for IoT, and in particular about the differences between MQTT and AMQP.

MQTT was first authored in 1999 by Andy Stanford-Clark and Arlen Nipper and was later standardised by OASIS, in 2013. MQTT is a standard messaging protocol for IoT and is designed as an extremely lightweight publish-subscribe messaging transport [Mqtt.org. 2020]. MQTT is described well by Figure 3. Essentially, each resource is an individual and separate component and each of these components communicates directly with an MQTT-Broker. The publisher component publishes any relevant information, e.g. temperature, to the MQTT queue and the subscriber component then subscribes to relevant information from the queue. Although only 1-way communication is shown on the publisher side in Figure 3, 2-way communication with the MQTT-broker is possible for all components [Paessler.com. 2018].

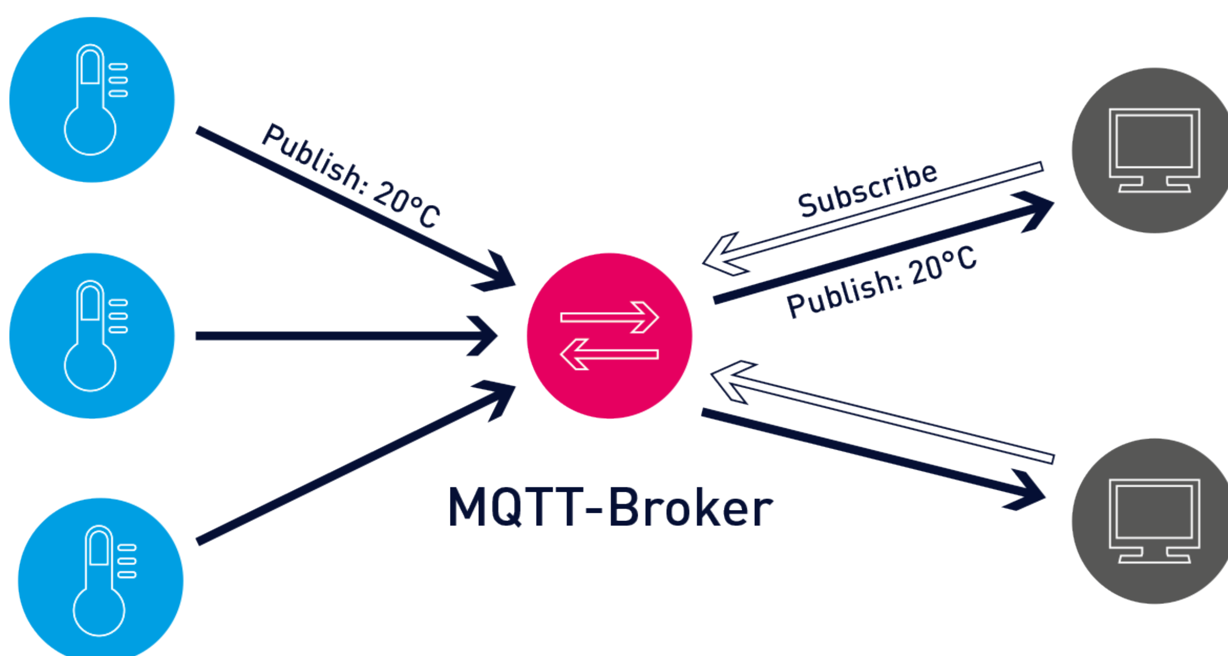


Figure 3: Graphical representation of MQTT architecture [Paessler.com. 2018]

Originally, MQTT was designed for use in devices with unreliable network resources, e.g. in remote locations, but it is now more widely adopted. MQTT runs on top of TCP/IP and was designed to be an event-driven protocol, hence there is no ongoing data transmission [Paessler.com. 2018]. In addition to minimizing number of transmissions, the transmitted messages are also small and tightly defined. Each message has a header of only 2 bytes and there are three possible quality of service (QoS) levels based on the desired balance between data transmission minimization and reliability maximization.

AMQP is an open standard protocol by OASIS that was released in 2011. AMQP also follows the publish-subscribe paradigm and was initially designed to enable interoperability between different devices that have different internal systems [Dizdarević, J., Carpio, F., Jukan, A. and Masip-Bruin, X., 2019]. AMQP has substantially more functionality than MQTT. It allows for message orientation, security, routing and switching reliability. The top-level architecture of AMQP is quite similar to that of MQTT, as can be seen in Figure 4.

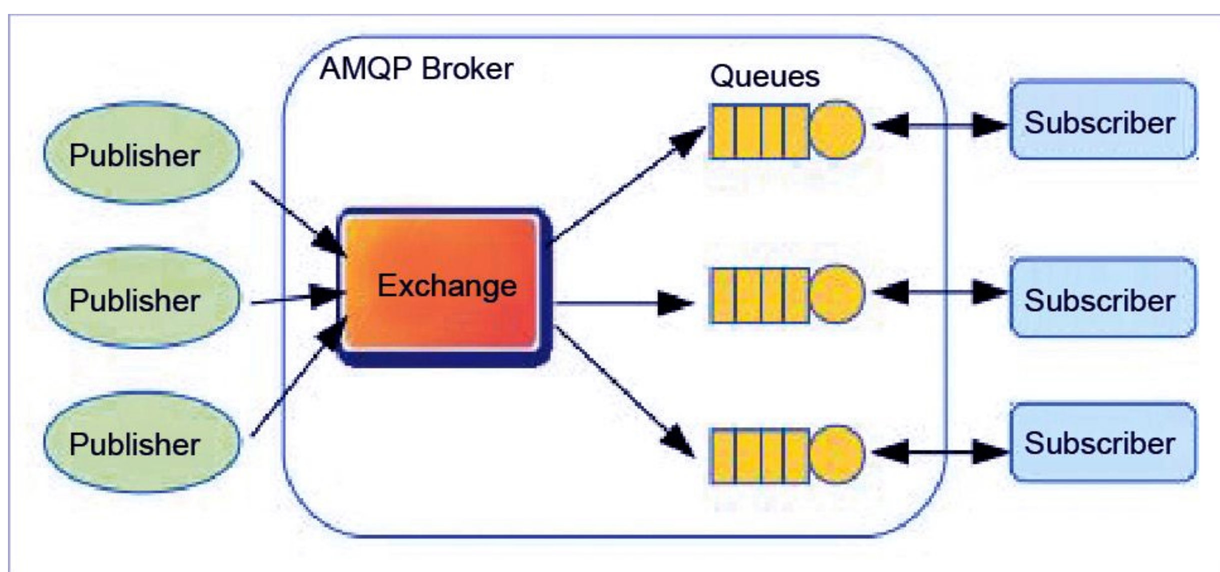


Figure 4: *Graphical representation of AMQP architecture [Bahashwan, A. and Manickam, S., 2018].*

In comparison to the MQTT architecture, it can be seen that the AMQP broker is split into two parts: exchange and queues. The exchange receives messages from the publisher and routes each message to the correct subscriber queue. There are a number of other key differences as well, which are outlined in a number of comparative studies. The message overhead is one such example. MQTT has a smaller message header size (2 bytes vs 8 bytes) and MQTT has a small and defined payload, whilst AMQP's payload is more flexible. AMQP has full cache and proxy support and substantially more security standards. Additionally, the QoS offered by AMQP is

superior to that of MQTT [Dizdarević, J. et al., 2019] [Bahashwan, A. and Manickam, S., 2018] [Al-Masri, E. et al., 2020] [MQTT, A., 2019].

Although AMQP has many benefits and is widely adopted in IoT devices that prioritise flexibility and reliability, all of the studies concur that this protocol is not well suited to constrained environments. According to Dizdarević et al. (2019), with all of the features that AMQP offers, it has relatively high power, processing and memory requirements. Hence it is better suited to a system that is not bandwidth, power or latency restricted [Dizdarević, J. et al., 2019]. Al-Masri et al. also found in their study that MQTT had much lower memory, CPU and power consumption than AMQP [Al-Masri, E. et al., 2020]. Based on these findings, and the fact that this project aimed to optimize resource usage in constrained devices, MQTT is better suited to this project than AMQP. Therefore, the next comparison to be made is between MQTT and CoAP.

MQTT and CoAP have a number of fundamental differences. MQTT is a many-to-many communication protocol for passing messages through a central broker, to and from multiple clients. MQTT is ideally used as a communications bus for live data. Although CoAP can operate in different modes, it is primarily a one-to-one protocol that allows direct communication between devices on the same constrained network. Hence it is better suited for resource creation and management on devices. Additionally, CoAP is commonly referred to as a request-response protocol, whilst MQTT is a publish-subscribe protocol [Jaffey, T., 2014]. Figure 5 provides a good visual representation of the architectural differences.

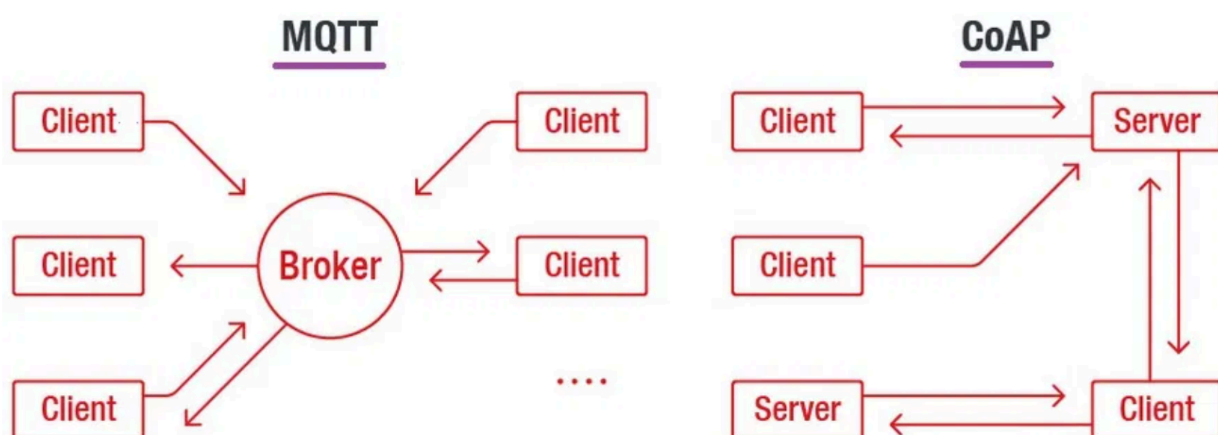


***Figure 5**: Graphical representation of MQTT and COAP architectures [Mishra, H., 2019].*

Some additional benefits offered by MQTT include time, space and synchronization decoupling. Time decoupling is mentioned as nodes can publish information regardless of the state of other nodes. Hence sleep or low power modes, which are often critical to power constrained devices, can still be employed without affecting communication. Space decoupling is achieved as nodes only need to know the IP-address of the broker node, not the addresses of all other nodes that

publish or subscribe to information. Synchronization decoupling is achieved as messages are only retrieved from the message queue once the subscriber node has completed all existing operations. This improves resource management of devices and allows for sleepy states [Stansberry, J., 2015].

CoAP does, however, offer a number of advantages over MQTT, such as message metadata. MQTT can be used by any client for any purpose, but clients must know the correct message formats to be able to communicate. By comparison, CoAP allows for content negotiation and discovery. Another advantage of CoAP is that it uses UDP as the transport protocol, whilst MQTT typically uses TCP as the transport protocol, although more recent developments have allowed for operation over UDP e.g. MQTT-SN. UDP's connectionless datagrams have less overhead, which allows devices to remain in lower power modes for longer periods of time, thereby conserving power. This advantage comes with a downside, however, as UDP is inherently, and intentionally, much less reliable than TCP [Mishra, H., 2019][Stansberry, J., 2015].

A number of studies have also been conducted to compare the technical performance of MQTT and CoAP. One such study was conducted by Van der Westhuizen and Hancke (2018) to compare communication delay and network traffic on both resource constrained devices and non-resource constrained devices. The researchers found that MQTT and CoAP had very similar communication delays when both the client and broker/server were on the same network. When the client was connected to an external network, however, MQTT had smaller delays than CoAP, and it was expected that the difference in delay would increase with increased packet size. Additionally, it was found that the architecture of MQTT was better suited when the same messages were forwarded to multiple clients/subscribers and that MQTT was generally simpler to implement. However, it was found that, on average, CoAP had smaller packet sizes and no keepalive messages, which led to lower power consumption [Van der Westhuizen, H. W. and Hancke, G. P., 2018].

In a study conducted by Naik (2017), a comparative analysis of MQTT, CoAP, AMQP and HTTP was done for IoT systems. The study found that CoAP incurs the lowest message overhead and power consumption, followed by MQTT. The study also emphasized the fact that both CoAP and MQTT are designed for low bandwidth and resource constrained devices; and can both be used on an 8-bit controller with less than 1 kilobyte (kB) of memory. The author also highlighted that, on average across different studies, CoAP consumes slightly less power and resources than MQTT. Additionally, the study concluded that MQTT consumed slightly higher bandwidth than CoAP and that MQTT offered greater reliability, whilst both MQTT and CoAP had the lowest

interoperability scores. Finally, the protocols were also ranked on their usage[4] in IoT, where MQTT was ranked highest amongst the four, followed by AMQP [Naik, N., 2017].

It is clear from the available research that both MQTT and CoAP offer a number of advantages and disadvantages, and that the choice of protocol is highly dependent on the usage scenario. One additional factor played a pivotal role in deciding which protocol to use for this research thesis, which was the language support for the protocols and the ease of implementation. Since the primary objective of this thesis was to compare software architectures, not application layer protocols, MQTT was selected by preference.

3.3 Programming Language Efficiency for Embedded Devices

Three different programming languages were used in this research thesis, primarily to ensure that the results obtained from this comparative study were not language specific. However, the choice of programming language can also have a significant impact on resource management. A number of studies have been conducted to determine the impact of programming language on runtime, memory and energy performance. However, to date no research has yet been conducted that compares the three programming languages that have been selected across different software architectures and on an embedded device. In fact, very limited research has been conducted on the choice of programming language for resource constrained devices in general.

Two studies were identified as being relevant to the research topic in this project. The first study was conducted by Pereira et al. (2017) to analyse energy efficiency across different programming languages. The primary focus was on energy efficiency, but runtime and memory-usage were also analysed by the authors and the analysis was carried out on a personal computer, i.e. a non-resource constrained device. Twenty-seven programming languages were compared by running ten benchmark problems from the Computer Language Benchmark Game framework on a desktop in each language. Amongst the twenty-seven languages were Rust, C, C++, Java, Python and Go. The energy consumption was measured using running average power limit (RAPL) function calls, whilst the runtime and memory were measured using the *time* tool that is available in all Unix-based systems [Pereira, R. *et al.,* 2017].

The results obtained in the study led the authors to the conclusion that there is no concrete optimal programming language from the perspective of runtime, energy efficiency and memory

[4] In the referenced study, usage referred to the degree of adoption of each protocol in industry, i.e. is it commonly used or not. This ranking is not specifically related to embedded devices, but rather to IoT systems in general.

performance. When considering each metric individually, C, Rust and C++ performed the best for both runtime and energy efficiency. However, Pascal, Go and C performed the best in terms of memory efficiency. It is clear from the research that C++ and Go performed much better than Python in all three metrics. This study provided some very interesting insights into programming language performance for personal computers but did not touch on constrained devices. It did, however, offer inspiration on how to conduct the experiments and, specifically, how to measure experimental values [Pereira, R. *et al.,* 2017].

The second study that was of particular interest was conducted by Georgiou et al. (2017) to analyse the energy consumption of 14 programming languages on a portable personal computer and on a Raspberry Pi 3b. The authors used data from Rosetta Code Repository, which is a publicly available programming chrestomathy, to conduct an empirical study. The programming languages that were chosen were a combination of compiled, semi-compiled and interpreted programming languages that included C, C++, Java, Go, Rust and Python. Ultimately, the languages that were chosen were a subset of the languages chosen for the study by Pereira et al. (2017). The main difference was that the study by Georgiou et al. (2017) only analysed energy efficiency (not memory or runtime) and analysed the languages on two hardware devices (compared to one in the study by Pereira et al.) [Georgiou, S., Kechagia, M. and Spinellis, D., 2017].

Georgiou et al. (2017) analysed the performance of the programming languages by running a selection of well-known tasks in each programming language, such as array-concatenation, url-encoding and sorting algorithms. The authors also discussed, in some detail, about the different possible methods for measuring energy consumption. Two methods were presented: a direct method using hardware components, which offers coarse-grained measurements and low sampling rate, and an indirect method that uses software components, which often suffers from inaccuracy. The authors opted to use the direct method, and prior to measurement of energy consumption, the computer system was rebooted and allowed to reach stable condition. The results from these experiments indicated that VB.NET and Swift were the most energy inefficient programming languages, whilst Go was the most energy efficient, followed closely by C and C++ [Georgiou, S., Kechagia, M. and Spinellis, D., 2017].

4. Technical Approach

4.1 Software Architecture

As per the objective of this project (see Section 1.2), six applications were developed. Three applications were developed using a microservices architecture (in three different languages), which can be seen in Figure 6. The other three applications were developed using a monolithic architecture. The microservices applications comprise four distinct microservices, which communicate using MQTT, an event-driven lightweight messaging protocol. The first three microservices ran on the Raspberry Pi and each microservice was responsible for a dedicated sensor or actuator service.

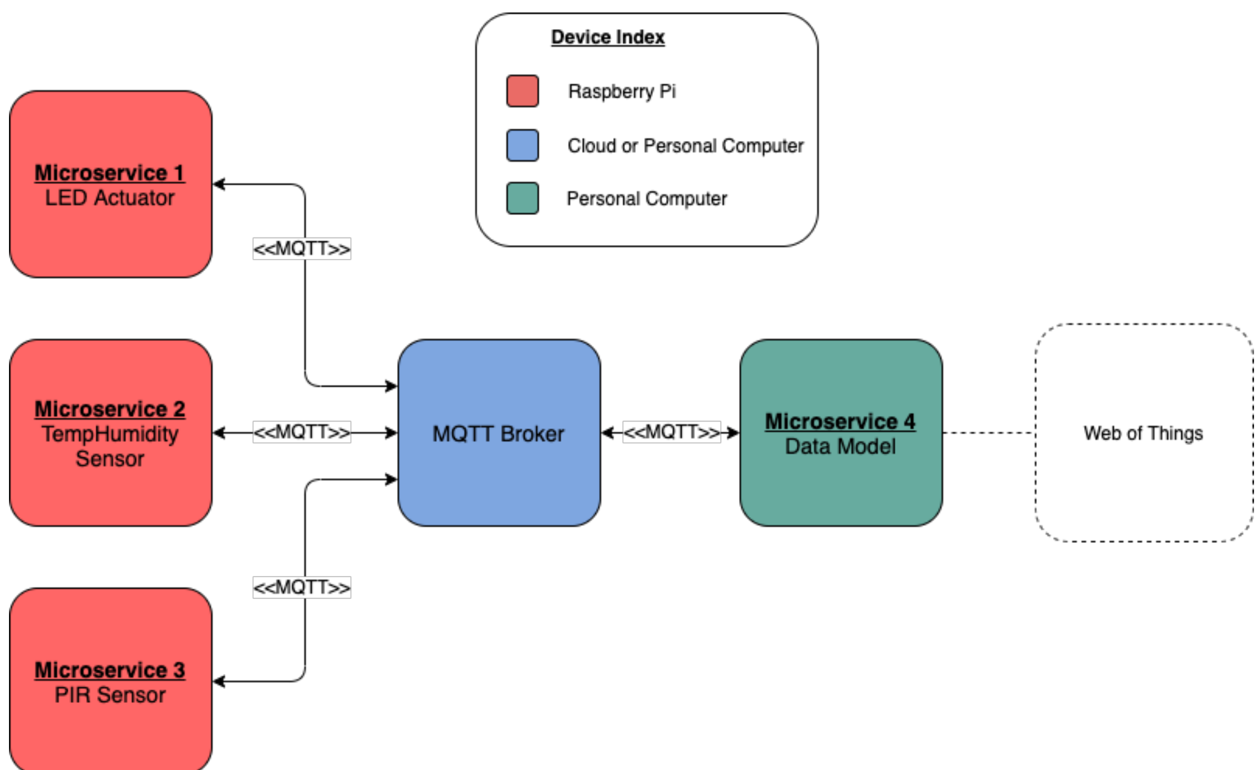


Figure 6: Graphical representation of the microservices architecture that was used in this project.

The MQTT broker, which was responsible for the message queue, ran on a personal computer using Eclipse Mosquitto, although it can also be run on the cloud [Vmware, 2020][*Eclipse Mosquitto*, 2018]. The fourth microservice was responsible for maintaining a data model of the LED actuator, temperature/humidity sensor and PIR sensor. This last microservice ran on a personal computer. Microservices 1, 2 and 3 were developed three times, using the three programming languages outlined in Section 2.4. Microservice 4 was only written once using Python since the first microservices application was developed using Python.

The monolithic architecture can be seen in Figure 7. With this architecture, all of the server-side logic (actuators and sensors) are contained in a single unit/component. The server application and client application also communicated using MQTT and an MQTT broker.

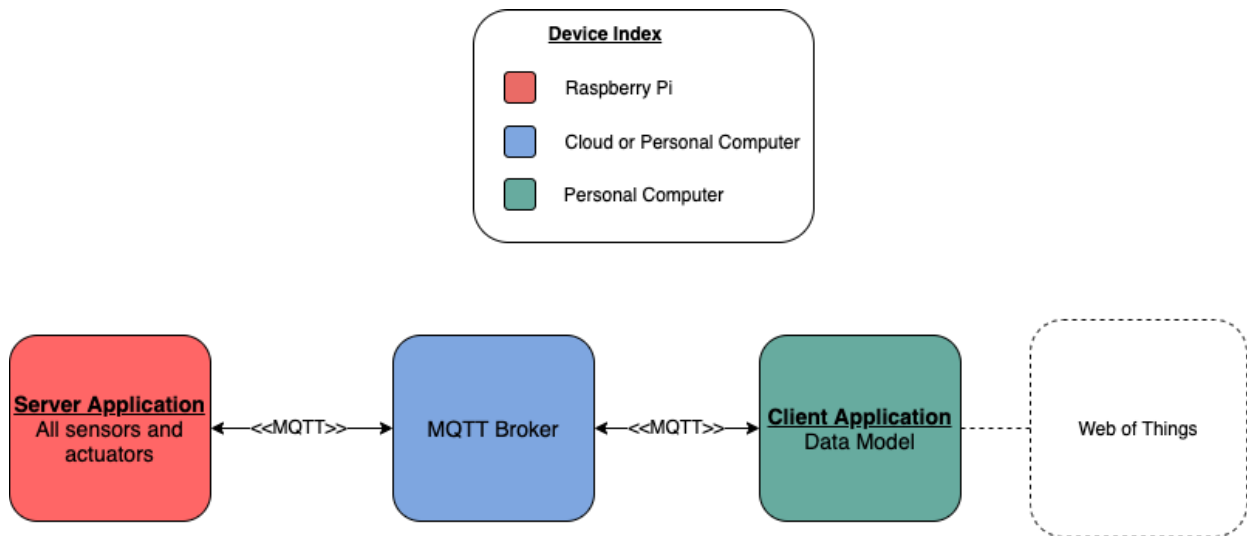


Figure 7: Graphical representation of the monolithic architecture that was used in this project.

4.2 Hardware Architecture

The following hardware was used for the experiments:

- Raspberry Pi 4 with 4GB RAM & 16GB MicroSD (2020 Model)
- MacBook Pro 13-inch, 2017 with a 2,3 GHz Dual-Core Intel Core i5 processor and 8GB RAM
- Type C USB Tester DC Digital Voltmeter and Ammeter
- DHT22 5V DTL 2% MOD temperature and humidity sensor
- Passive Infrared (PIR) sensor 240''
- LED 3mm red
- Arduino breadboard and wires

The setup of the breadboard and sensors is shown in Figure 8, whilst the Type C USB tester is shown in Figure 9.

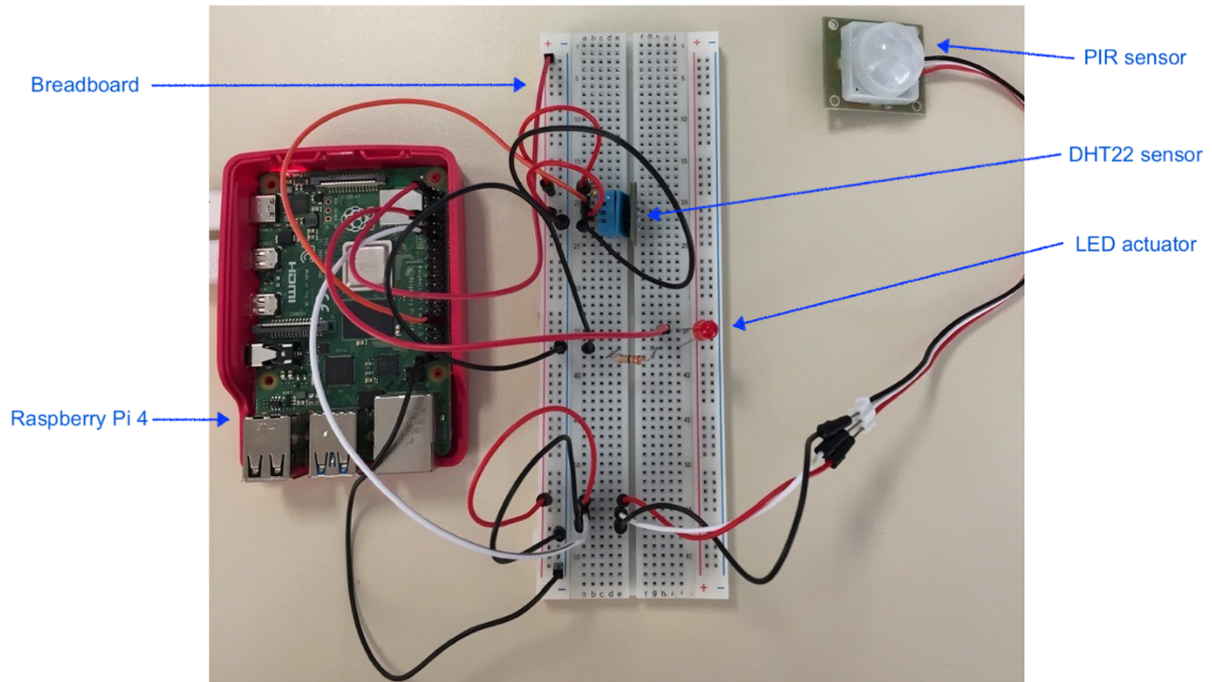


***Figure 8**: Photograph showing the experimental setup of the breadboard and sensors*

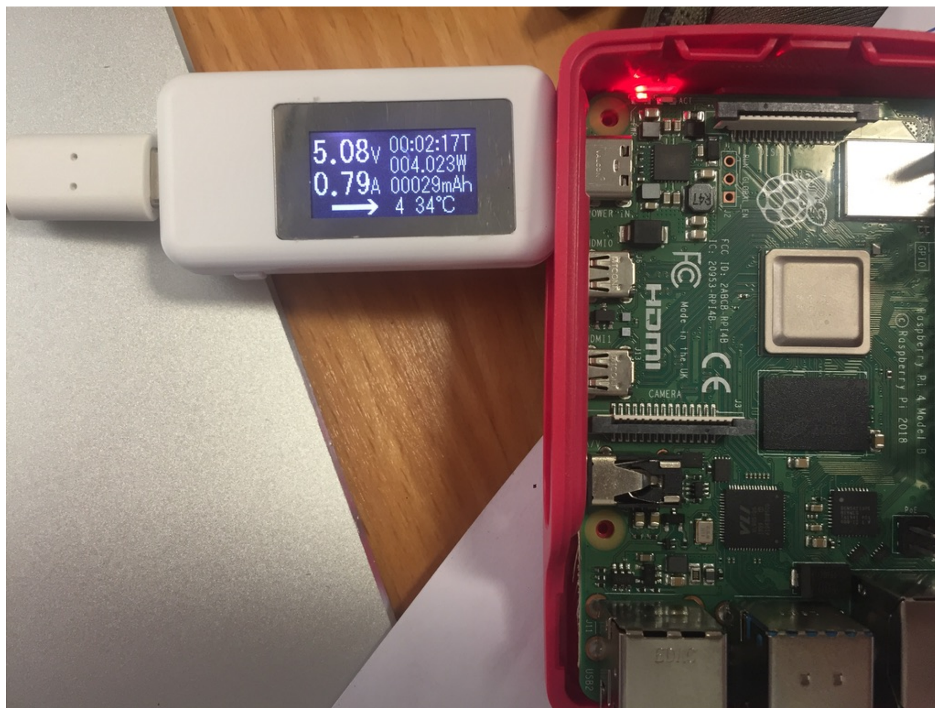


***Figure 9**: Photograph showing the experimental setup of the Type C USB Tester.*

4.3 Data Collection

A relatively large amount of data was collected for this project. Each of the datasets described below were collected six times, twice for each of the three languages. Additionally, each dataset was collected 10 times to ensure data validity. Hence a total of 180 datasets were collected.

4.3.1 Power Consumption

Two different power consumption measurements were taken. Both of these measurements were taken using hardware (see Figure 9) since the software alternative (Powerstat) yielded inaccurate results. The first measurement that was taken was the instantaneous maximum or peak power consumption in Watt. The maximum power consumption was measured as this gives an idea of potential load spikes, which can be a limiting factor when energy harvesting is used, as is the case for many resource-constrained devices. The instantaneous maximum power consumption was measured using the type C USB tester as depicted in Figure 9.

The second measurement that was taken was the overall power consumption in Watt. This was measured by resetting the device readings to 0 and then executing the program. Upon completion of the program, the total runtime and electric charge, in milliamp hours (mAh), were captured. These readings were then converted to Watts using:

$$E_W = \frac{Q_{mAh} \times V_v}{1000 \times t_h} \quad (1)$$

Where E_w is the energy consumption in Watts,

Q_{mAh} is the electric charge in mAh,

V_v is the voltage in Volts,

t_h is the duration in hours

4.3.2 Runtime

Program runtime is an important performance metric as it gives an indication of the speed of execution of a program. Runtime refers to the period of time from the moment that a program is executed, until the program ends/is closed. During this time, the program is loaded into RAM which includes both the executable file and any libraries or other files that are referenced by the program. When the program ends, the memory used by the program is freed for use by other programs/processes. In the area of IoT, runtime is an important consideration as low latency is often a key application requirement, e.g. medical devices, fire sensors, etc. Additionally, program runtime is often critical for resource-constrained devices as it dictates the duration during which resources are used, and therefore the total resource usage. Some programs may have low instantaneous power and memory consumptions, but long runtimes, or vice versa. Therefore, it is important to consider runtime when comparing resource consumption. It is also important to note that runtime is highly dependent on the operating system that is used by a device. Since the

same device, and operating system[5], was used for all experiments in this research thesis, runtimes could be compared directly.

Runtime was measured for each IoT service, when using the microservices architecture, as well as for overall program execution (time until all programs came to an end), for both architectures. The measurements were taken using the time library that was available for each language, i.e. C time library for C++, package time for Go and time module for Python. The measurements were then recorded in a .txt file. To ensure that these times were accurate, they were compared to the runtimes measured by the GNU time tool, and it was found that they were identical up to 100th of a second. Figure 10 shows a snapshot of the individual microservices runtimes that were captured.

```
1   PIR runtime = 5.938340038
2   Humidity and temperature runtime = 15.23761147
3   LED subsriber runtime = 113.274438286
4   PIR runtime = 7.756507411
5   Humidity and temperature runtime = 15.750742348
6   LED subsriber runtime = 114.44261036
7   PIR runtime = 6.960154441
8   Humidity and temperature runtime = 15.626923645
9   LED subsriber runtime = 102.263831264
10  PIR runtime = 7.796252434
11  Humidity and temperature runtime = 15.438289957
12  LED subsriber runtime = 125.653936215
13  PIR runtime = 8.015682016
14  Humidity and temperature runtime = 16.005173121
15  LED subsriber runtime = 115.170467108
16  PIR runtime = 7.7427028700000005
17  Humidity and temperature runtime = 15.25102536
18  LED subsriber runtime = 109.757388464
19  PIR runtime = 7.38978775
20  Humidity and temperature runtime = 15.612205738
21  LED subsriber runtime = 122.572531051
22  PIR runtime = 8.015360485
23  Humidity and temperature runtime = 15.577662641
24  LED subsriber runtime = 102.362130863
25  PIR runtime = 8.675664908
26  Humidity and temperature runtime = 16.644856813
27  LED subsriber runtime = 88.945034573
28  PIR runtime = 8.051036707
29  Humidity and temperature runtime = 16.289538924
30  LED subsriber runtime = 82.220552331
```

Figure 10*: Snapshot of the runtime outputs when using the language specific time library (this is an example of Go)*

[5] The Raspberry Pi used in this research thesis runs the Raspbian GNU/Linux version 10 operating system

4.3.3 Memory Consumption

Three different memory consumption measurements were taken for this project. The first was the maximum resident set size (RSS) in kB. RSS is the amount of main memory (resident RAM) that a process is using at the time of measurement. This does not include swapped or otherwise non-resident RAM. Therefore, the maximum RSS measurement indicates the peak RAM that was used by a process, which provides valuable information as constrained devices usually have limitations on available RAM. The maximum RSS was measured using the GNU time command that is available in Linux. Figure 11 shows the output of the GNU time command that was saved to a .txt file after each run.

```
1   Command being timed: "./pirPub 10.35.0.229"
2   User time (seconds): 0.29
3   System time (seconds): 0.07
4   Percent of CPU this job got: 75%
5   Elapsed (wall clock) time (h:mm:ss or m:ss): 0:00.48
6   Average shared text size (kbytes): 0
7   Average unshared data size (kbytes): 0
8   Average stack size (kbytes): 0
9   Average total size (kbytes): 0
10  Maximum resident set size (kbytes): 3540
11  Average resident set size (kbytes): 0
12  Major (requiring I/O) page faults: 4
13  Minor (reclaiming a frame) page faults: 249
14  Voluntary context switches: 13
15  Involuntary context switches: 4
16  Swaps: 0
17  File system inputs: 592
18  File system outputs: 8
19  Socket messages sent: 0
20  Socket messages received: 0
21  Signals delivered: 0
22  Page size (bytes): 4096
23  Exit status: 0
```

Figure 11*: Snapshot of the output when running the GNU time command for peak memory usage*

The second memory consumption measurement that was taken was the overall memory consumption from the start to the end of the program execution. This was also measured by RSS in kB but was done using Syrupy, which is a Python script that takes regular snapshots of the memory to dynamically build a profile of the memory consumption of a program [Sukumaran, J., 2020]. Snapshots are taken at intervals of 1 second, therefore the total RSS can be calculated by summing RSS over the total runtime of the program. Figure 12 provides a snapshot of the Syrupy log.

1	PID	DATE	TIME	ELAPSED	RSS	VSIZE	CMD
2	1215	2021-05-23	11:30:38	00:00	2448	7676	bash runGoS
3	1215	2021-05-23	11:30:39	00:01	2448	7676	bash runGoS
4	1215	2021-05-23	11:30:40	00:02	2448	7676	bash runGoS
5	1215	2021-05-23	11:30:41	00:03	2448	7676	bash runGoS
6	1215	2021-05-23	11:30:42	00:04	2448	7676	bash runGoS
7	1215	2021-05-23	11:30:43	00:05	2448	7676	bash runGoS
8	1215	2021-05-23	11:30:44	00:06	2448	7676	bash runGoS
9	1215	2021-05-23	11:30:45	00:07	2448	7676	bash runGoS
10	1215	2021-05-23	11:30:47	00:09	2448	7676	bash runGoS
11	1215	2021-05-23	11:30:48	00:10	2448	7676	bash runGoS
12	1215	2021-05-23	11:30:49	00:11	2448	7676	bash runGoS
13	1215	2021-05-23	11:30:50	00:12	2448	7676	bash runGoS
14	1215	2021-05-23	11:30:51	00:13	2448	7676	bash runGoS
15	1215	2021-05-23	11:30:52	00:14	2448	7676	bash runGoS
16	1215	2021-05-23	11:30:53	00:15	2448	7676	bash runGoS
17	1215	2021-05-23	11:30:54	00:16	2448	7676	bash runGoS
18	1215	2021-05-23	11:30:55	00:17	2448	7676	bash runGoS
19	1215	2021-05-23	11:30:56	00:18	2448	7676	bash runGoS
20	1215	2021-05-23	11:30:57	00:19	2448	7676	bash runGoS
21	1215	2021-05-23	11:30:58	00:20	2448	7676	bash runGoS
22	1215	2021-05-23	11:30:59	00:21	2448	7676	bash runGoS
23	1215	2021-05-23	11:31:00	00:22	2448	7676	bash runGoS
24	1215	2021-05-23	11:31:01	00:23	2448	7676	bash runGoS

***Figure 12**: Snapshot of the Syrupy log that was used to capture memory consumption measurements at regular intervals.*

The third memory consumption measurement that was taken was the average CPU usage (%) that was required to execute a program. Since the Raspberry Pi that was used in this project has 4 cores, the CPU usage often exceeded 100%. It is also interesting to note that the CPU usage is calculated by determining the total scheduling time as a percentage of the total runtime, e.g. if a process was scheduled for 1ms and the overall runtime was 10ms, then the CPU usage was 10%. The CPU usage output can be seen in Figure 11.

5. Methodology and Evaluation

This research thesis was carried out in three parts, as outlined below.

5.1 Microservices Application Development

This phase accounted for the largest portion of time as the applications had to be developed in three different languages, using two different architectures. The first application was developed in Python using a microservices architecture and the next was developed in Python using a monolithic architecture. The microservices and monolithic applications were then developed using C++ and lastly using Go. Overall, C++ and Go proved the hardest to develop. This was because there is very little information available for programming of embedded devices with C++ and Go.

Embedded devices are usually programmed using C if very resource-constrained or Python otherwise. The reading of temperature and humidity data from the digital humidity and temperature (DHT) sensor was especially difficult as there are no C++ or Go libraries for this. In the end, a C library (WiringPi) had to be used for signal processing[6]. Overall, the applications were much easier to build using Python.

5.2 Data Collection

Once the source code was written for all applications, the data collection phase began. 6 sets of data were collected 10 times for validation, as described in Section 4.3. Each dataset was only collected 10 times since the power measurements were taken manually (the type C USB tester does not have capabilities to store or send data). It is important to note that for compiled languages (C++ and Go), the programs were compiled before running the experiments.

Each experiment was conducted using the following steps:

1. The Raspberry Pi was rebooted and then allowed 1 minute to reach stable condition.

[6] Data from the DHT22 sensor is sent by transmitting signals of varying length (milliseconds or microseconds) to indicate 8-bit values. The first 8 bits correspond to the integral humidity value, the second 8 bits to the decimal humidity value, the third to the integral temperature value, the fourth to the decimal temperature value and the last 8 bits to the checksum [UUGear, 2018].

2. The type C USB tester was reset and then the program/s was/were executed. For microservices, where there were a number of separate programs, the programs were executed in parallel using a bash file (see Appendix A.1).
3. The power consumption, runtime and memory consumption were collected and stored in .txt or .xslx files.
4. The above steps were then repeated ten times for data validation.

5.3 Evaluation and Comparison

The last part of the project involved the evaluation and comparison of the data that was collected. The processing of the data was primarily done using the Pandas library in Python [Pandas, 2021]. It was decided that the results would be presented in two formats: as bar charts and as tables. Some box plots were also created where deemed useful.

The results that were obtained could be compared directly, i.e. no additional metrics, like root mean squared error, were needed. For all readings (power consumption, runtime, and memory consumption) except CPU usage, low values correspond to high performance. The results are presented and discussed in the next Section.

6. Results and Discussion

6.1 Power Consumption

As explained in Sections 4 and 5, power consumption measurements were taken using a type C USB tester. Two types of power consumption readings were taken during each run: the maximum instantaneous power consumption and the total power consumption for the program to finish running. Both measurements are provided in Watts. It is important to note that the power consumption measurements include the power required to operate the device. Although many of the power readings were between 3,2 and 4,2 Watts, the idle power consumption of a Raspberry Pi 4 is typically just over 3 Watts. However, the idle power consumption was near-identical for each run, therefore the total power consumption values could be compared. The maximum instantaneous power consumption results are summarized in Table 1 and Figure 13 below. The full results can be seen in Appendix B.

Table 1: Summary of maximum instantaneous power consumption for different languages and architectures.

Mean Power Consumption (Watt)	Median Power consumption (Watt)	Language	Architecture
4,21	4,20	Go	Microservices
3,65	3,65	Go	Monolithic
3,93	3,94	Python	Microservices
3,20	3,20	Python	Monolithic
3,93	3,94	Cpp	Microservices
3,43	3,42	Cpp	Monolithic

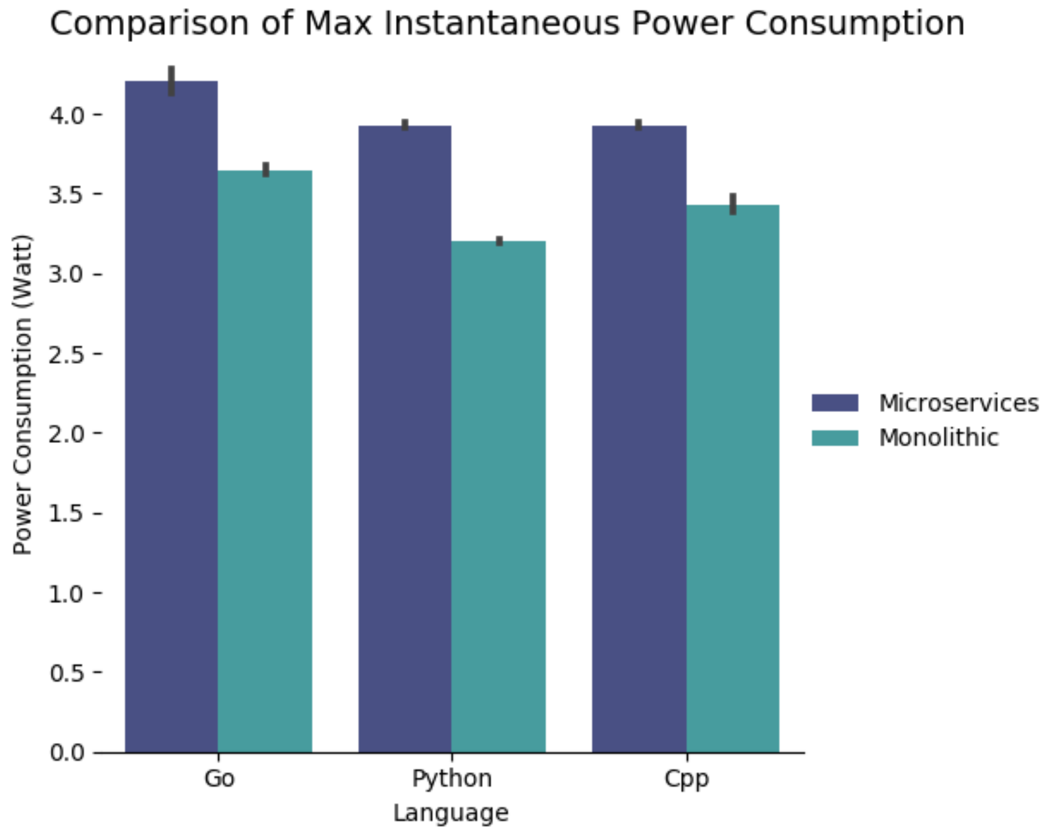


***Figure 13**: Bar chart showing the comparative maximum instantaneous power consumption of different languages and architectures*

It can be seen from Figure 13 that the microservices architecture consistently had a higher maximum instantaneous power consumption than the monolithic architecture, for all languages. This difference in power consumption can also be seen in Table 2, which shows that the average difference between microservices and monolithic power consumption (across all languages) was -14,9%.

Table 2: Table showing the percentage difference in maximum instantaneous power consumption (Watt) between different architectures.

Language	Microservices Max Power (Watt)	Monolithic Max Power (Watt)	Difference
Go	4,21	3,65	-13,33%
Python	3,93	3,20	-18,60%
Cpp	3,93	3,43	-12,75%
Average			-14,89%

When considering power consumption by language, the difference is less remarkable. However, Go had the highest maximum instantaneous power consumption, followed by C++ and then by Python. C++ and Python had similar measurements for the microservices architecture.

Table 3 and Figure 14 show a summary of the total power consumption (over the entire runtime).

Table 3: Summary of total power consumption for different languages and architectures.

Mean Total Power Consumption (Watts)	Median Total Power Consumption (Watts)	Language	Architecture
3,67	3,69	Go	Microservices
3,53	3,53	Go	Monolithic
3,74	3,74	Python	Microservices
3,49	3,49	Python	Monolithic
3,50	3,55	Cpp	Microservices
3,55	3,55	Cpp	Monolithic

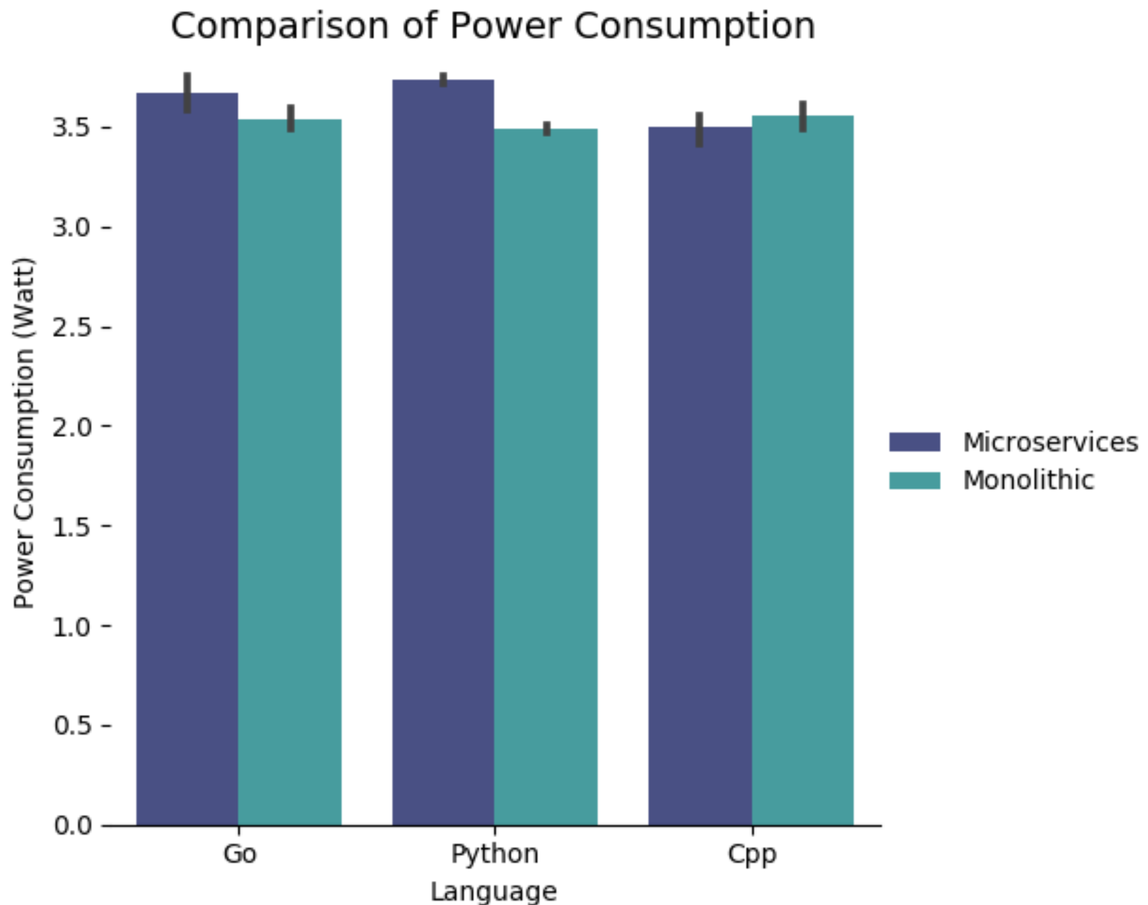


Figure 14: Bar chart showing the comparative total power consumption of different languages and architectures.

Compared to the maximum instantaneous power consumption, the total power consumption measurements do not differ much by architecture or language, which can also be seen in Table 4. The microservices architecture consumed slightly more power for Go and Python but consumed slightly less for C++. When considering the microservices architecture, Python consumed the most, followed by Go and then by C++. For the monolithic architecture, the values were almost the same.

Table 4: Table showing the percentage difference in total power consumption (Watt) between different architectures.

Language	Microservices Total Power (Watt)	Monolithic Total Power (Watt)	Difference
Go	3,67	3,53	-3,75%
Python	3,74	3,49	-6,72%
Cpp	3,50	3,55	1,55%
Average			-2,97%

One of the most interesting findings was that the order of maximum instantaneous power consumption (by language) was different to the order of total power consumption. For example, Python had the lowest maximum instantaneous power consumption but the highest total power consumption. This could be explained by the fact that Python has a much longer runtime than Go and C++ (see Section 6.2), hence Python does not have a high peak consumption but rather consumes at a stable level for a longer period of time. It is also interesting to note that there was a much bigger difference between the two architectures for maximum instantaneous power consumption than for total power consumption. On average, the monolithic architecture consumed slightly less power for both metrics.

6.2 Runtime

Runtime (duration from start to end of program execution) was measured using language-specific time libraries, e.g. C time library for C++, and the GNU time command. Four different runtimes were measured for the microservices architecture - one for each sensor/actuator and one for overall runtime. For the monolithic architecture, the overall runtime was measured. For comparison, two overall microservices runtimes were used. The first is the cumulative microservices runtime, which was calculated by summing the runtimes of all of the individual microservices applications. The second was the overall “concurrent” runtime that was measured from the execution of the bash script until the last microservice program came to an end. The runtime results are summarized in Table 5 and Figure 15.

Table 5: Table summarising the runtime measured for different languages and architectures.

Mean Runtime (s)	Median Runtime (s)	Language	Architecture
131,04	133,60	Go	Microservices - Cumulative
119,10	119,00	Go	Microservices
112,14	112,15	Go	Monolithic
673,70	663,35	Python	Microservices - Cumulative
380,70	381,00	Python	Microservices
621,47	623,46	Python	Monolithic
97,49	97,84	Cpp	Microservices - Cumulative
90,60	89,00	Cpp	Microservices
82,30	81,85	Cpp	Monolithic

** "Microservices – Cumulative" indicates the total runtime as a sum of the individual microservices runtimes. "Microservices" is the total runtime for all microservices concurrently.*

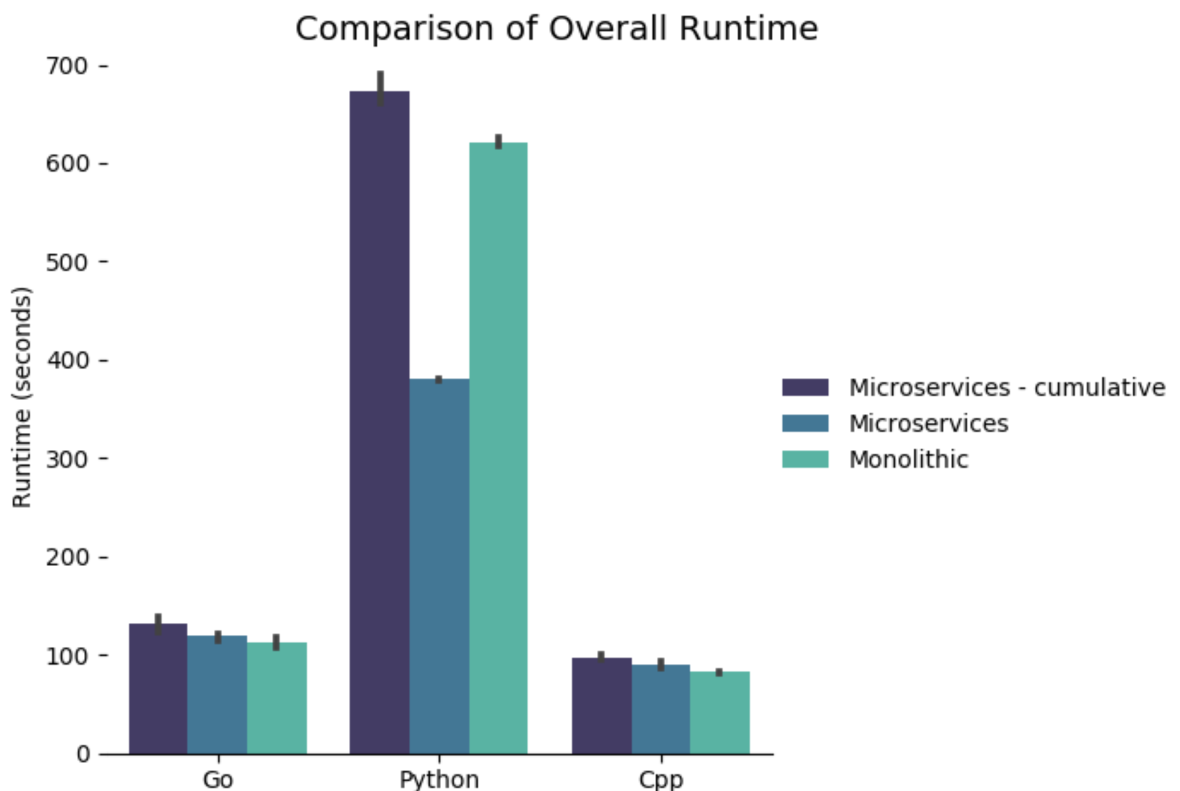


***Figure 15**: Bar chart showing the comparative overall runtime of different languages and architectures.*

Table 5 and Figure 15 clearly show that Python had a substantially longer runtime than the other languages, for both architectures and with both cumulative and concurrent runtimes. This is not surprising since Python is an interpreted language (see Section 2.4). It was interesting to see that the concurrent microservices runtime was substantially smaller than that of the monolithic architecture for Python, which could be indicative of sub-optimal CPU-bound concurrency in Python. However, further research would be needed to make any conclusions on this. The results for Go and C++ were contrary to those of Python as the runtimes were slightly shorter with the

monolithic architecture than with the microservices architecture. Comparing by language, it can be seen that Go had a slightly longer runtime than C++. In order to compare these two languages more closely, two box plots were generated – one for each architecture. These are shown in Figures 16 and 17.

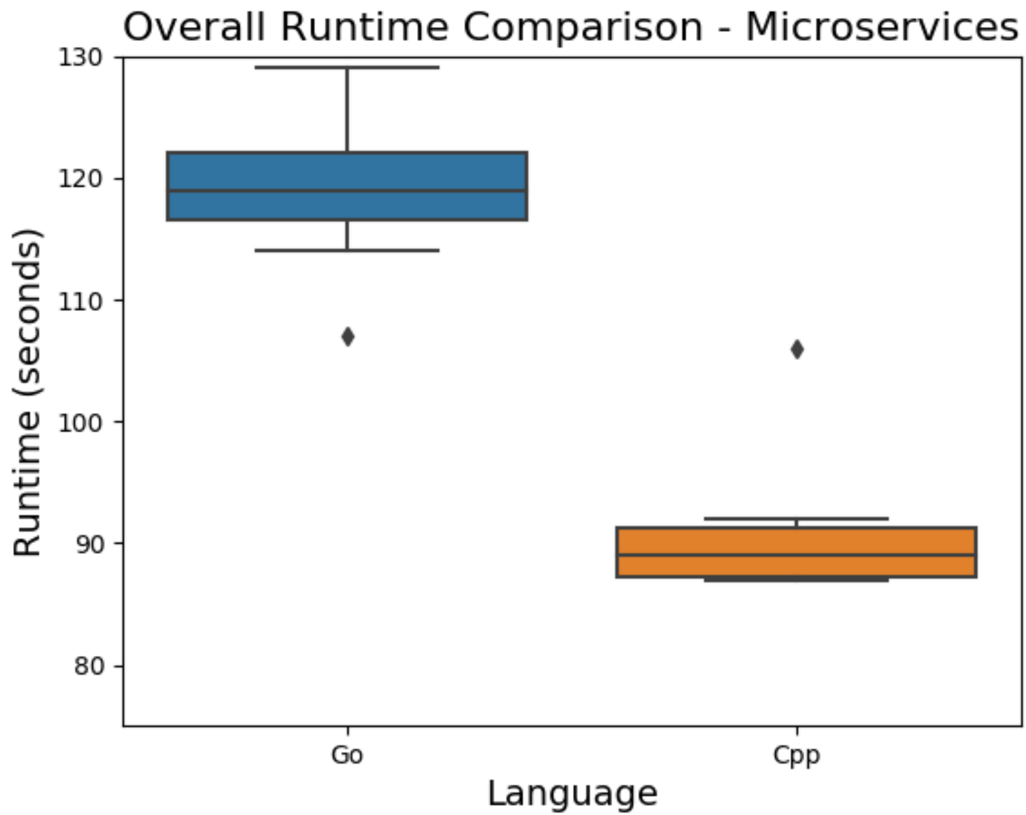


***Figure 16**: Box plot showing the comparative overall runtime of Go and C++ when using a microservices architecture.*

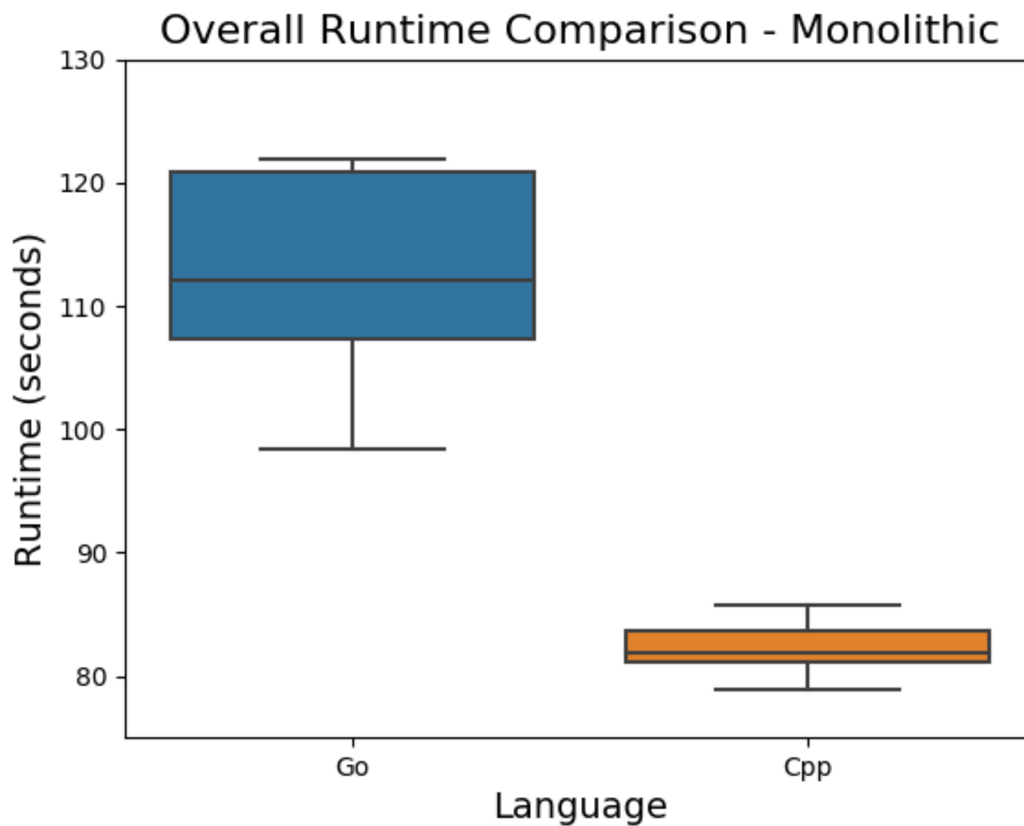


Figure 17*: Box plot showing the comparative overall runtime of Go and C++ when using a monolithic architecture.*

Figures 16 and 17 show that Go had a slightly longer runtime than C++ for both architectures. When using a microservices architecture, Go had a median runtime of 119,0 seconds whilst C++ had a median runtime of 89,0 seconds, which gives a difference of -25,2%. For the monolithic architecture, Go had a median runtime of 112,2 seconds whilst C++ had a median runtime of 81,9 seconds, which gives a difference of -27%. Additionally, the box plots show that Go had a larger spread of runtime values than C++. The fact that Go had a longer runtime than C++ could be explained by the fact that C code had to be wrapped in Go code in order to access the general-purpose input/output (GPIO) pins for DHT22 readings. Compared to the Python runtime, however, Go had a similar runtime to C++.

Table 6 summarizes the difference in mean runtimes between the different architectures. It can be seen that the monolithic architecture had a shorter runtime for Go and C++, whilst the microservices architecture had a shorter runtime for Python.

Table 6: Table summarising the runtime differences between architectures.

Language	Microservices Runtime (s)	Monolithic Runtime (s)	Difference
Go	119,10	112,14	-5,84%
Python	380,70	621,47	63,24%
Cpp	90,60	82,30	-9,16%
Average			16,08%

As discussed in the literature review (Section 3.1), very limited research has been done to compare the performance of the monolithic and microservices architectures on a technical level. However, the research conducted by Al-Debagy and Martinek (2018), although not directly related, yielded results that correlate to the results from this research thesis. The authors found that, for a small number of threads, the monolithic architecture had a shorter response time than the microservices architecture. For a larger number of threads (greater than 1000), the microservices architecture had a shorter response time [Al-Debagy, O. and Martinek, P., 2018]. Although the impact of number of threads was not researched in this project, and response time is not directly comparable to runtime, it is clear that with a small number of threads, the monolithic architecture had a shorter runtime in this research thesis and therefore the runtime findings are, at least in part, in line with the finding of Al-Debagy and Martinek (2018).

6.3 Memory Consumption

Memory consumption was measured using three different metrics. The first was the maximum RSS in kB (see Section 4.3.3 for more information), which was measured through the GNU time command. The results for maximum RSS are summarized in Table 7 and Figure 18.

Table 7: Table summarising the maximum RSS of different languages and architectures

Mean Max RSS (kB)	Median Max RSS (kB)	Language	Architecture
97637,20	98628,00	Go	Microservices
83097,50	89543,00	Go	Monolithic
40030,90	40090,00	Python	Microservices
13651,40	13657,50	Python	Monolithic
21824,00	21822,00	Cpp	Microservices
15150,40	15150,00	Cpp	Monolithic

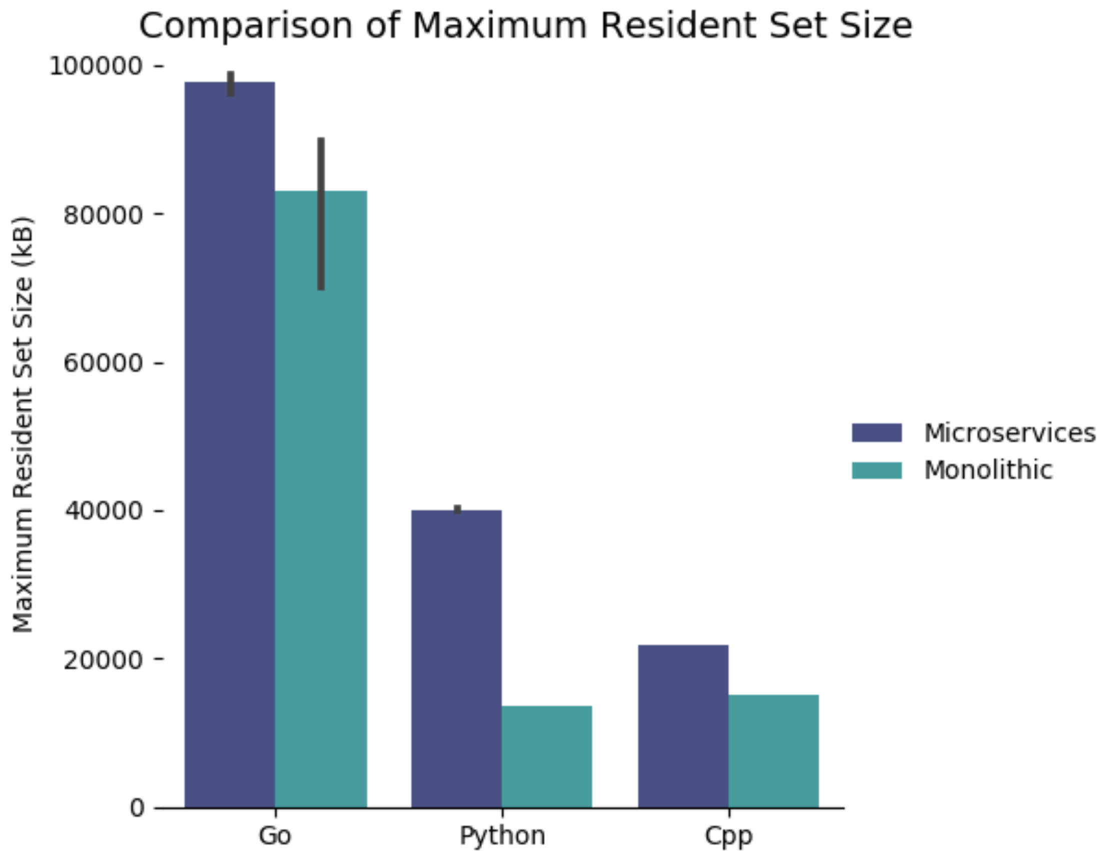


***Figure 18**: Bar chart showing the comparative maximum RSS of different languages and architectures.*

It is clear from both Table 7 and Figure 18 that, for all 3 languages, the monolithic architecture had a significantly smaller maximum RSS. Table 8 shows that the average difference in maximum RSS between the microservices architecture and the monolithic architecture was -37,12%. When comparing by language, it can be seen that the mean maximum RSS for Go was much larger that for Python and C++. This result was surprising as it was expected that Go and C++ would have similar memory consumption measurements. In order to gain a better understanding, the total RSS should also be considered.

Table 8: Table summarising the maximum RSS differences between architectures.

Language	Microservices Max RSS (kB)	Monolithic Max RSS (kB)	Difference
Go	97637,20	83097,50	-14,89%
Python	40030,90	13651,40	-65,90%
Cpp	21824,00	15150,40	-30,58%
Average			-37,12%

The second memory consumption metric was the total RSS, which was measured using a Syrupy script (see Section 4.3.3 for more information). Using Syrupy memory snapshots, a profile was built of the memory consumption of each program. The results are summarized in Table 9 and Figure 19.

Table 9: Table summarising the total RSS of different languages and architectures

Mean Total RSS (MB)	Median Total RSS (MB)	Language	Architecture
190,94	188,50	Go	Microservices
194,66	194,66	Go	Monolithic
584,19	586,80	Python	Microservices
1325,93	1323,42	Python	Monolithic
185,17	187,78	Cpp	Microservices
180,30	177,90	Cpp	Monolithic

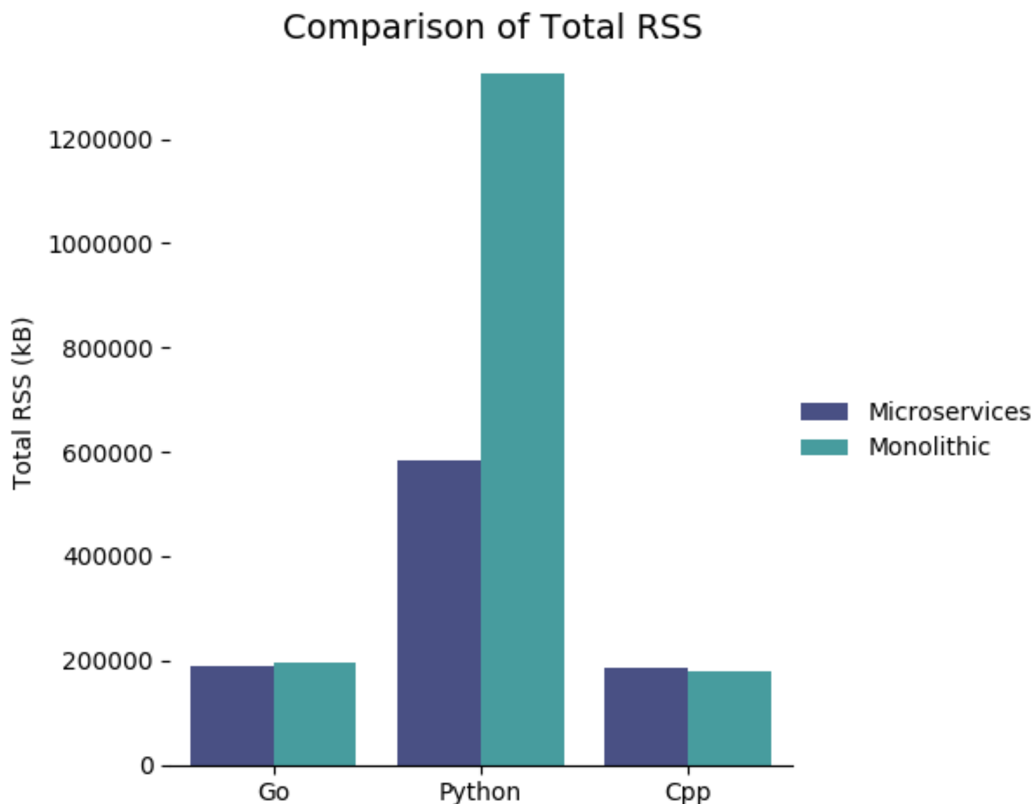


Figure 19: Bar chart showing the comparative total RSS of different languages and architectures.

The total RSS results were more in line with expectations (in contrast to the maximum RSS results). Figure 19 shows that Go and C++ had similar mean total RSS measurements, whereas Python had a greater mean total RSS for both architectures. It is also interesting to note that, when using Python, the microservices architecture had a much smaller mean total RSS than the monolithic architecture, whilst the total RSS values for Go and C++ did not differ much by architecture. This deduction is reinforced by the differences shown in Table 10, which shows that the total RSS for Python was 127% greater when using the monolithic architecture than with the microservices architecture. For the other two languages, the differences were negligible.

Table 10: Table summarising the total RSS differences between architectures.

Language	Microservices Total RSS (MB)	Monolithic Total RSS (MB)	Difference
Go	190,94	194,66	1,94%
Python	584,19	1325,93	126,97%
Cpp	185,17	180,30	-2,63%
Average			42,09%

The third, and last, memory consumption metric was the average CPU usage in percentage. This was also measured using the GNU time command. Table 11 and Figure 20 summarize the results that were obtained.

Table 11: Table summarising the CPU usage of different languages and architectures

Mean CPU (%)	Median CPU (%)	Language	Architecture
98,10	97,83	Go	Microservices
110,40	107,00	Go	Monolithic
65,30	65,33	Python	Microservices
83,10	83,00	Python	Monolithic
94,03	94,00	Cpp	Microservices
98,70	99,00	Cpp	Monolithic

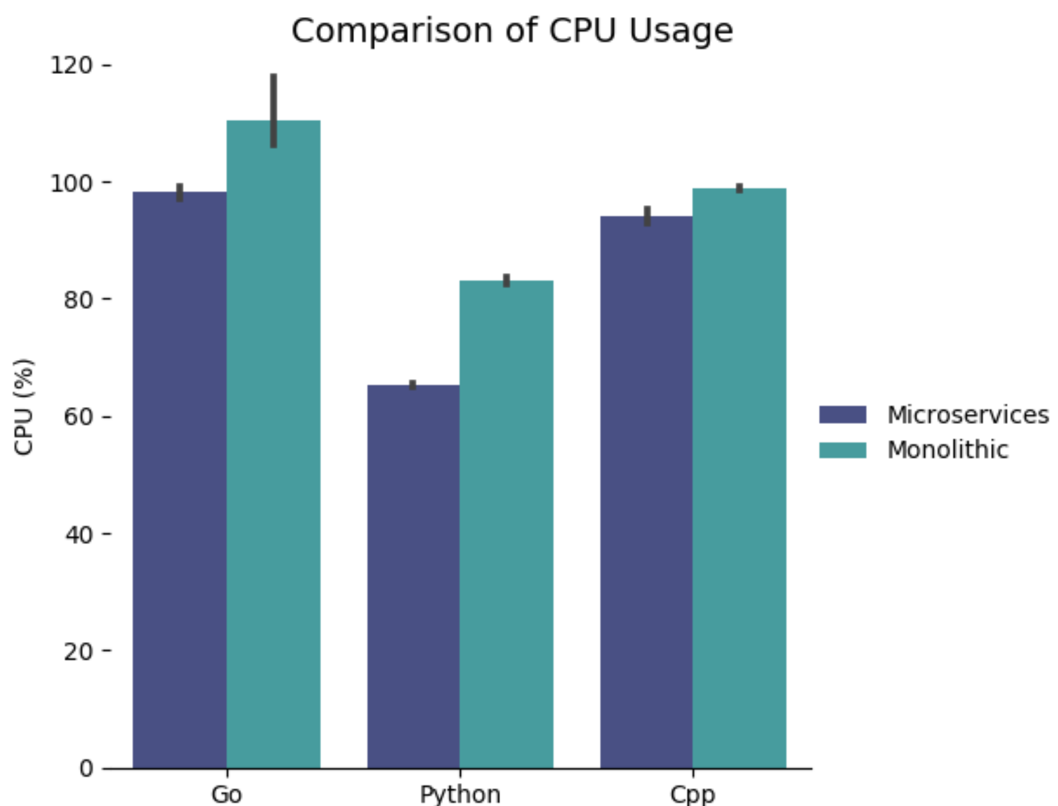


Figure 20: *Bar chart showing the comparative CPU usage of different languages and architectures.*

The results shown in Table 11 and Figure 20 show that, regardless of language, the monolithic architecture had higher CPU usage than the microservices architecture. On average the monolithic architecture had 14,9% higher CPU usage. When comparing by languages, it was found that Go had the highest average CPU usage, followed by C++ and then by Python. This result is in line with expectations since Go has built in light-weight threads (called go routines) and, with more recent versions of Go, Go routines automatically use all CPU cores that are available, if required by the process. Since the Raspberry Pi used in this research thesis has 4 cores, higher CPU usage, e.g. 110%, may indicate better use of resources. However, this would need to be studied further to make any conclusions.

Table 12: Table summarising the CPU usage differences between architectures.

Language	Microservices CPU (%)	Monolithic CPU (%)	Difference
Go	98,10	110,40	12,54%
Python	65,30	83,10	27,26%
Cpp	94,03	98,70	4,97%
Average			14,92%

7. Conclusion and Future Work

The Web and Internet of Things (WoT/IoT) is an exciting field that will no doubt continue to develop over the years to come. As further developments are made, and more strange and wonderful objects become “Things”, the limitations on resources will likely grow. In 2021 there are already many devices that face these constraints due to a variety of reasons such as their remote placement (e.g. implantable sensors or volcanic eruption sensors) or the need to work whilst in motion (e.g. wearable sensors). Hence there is a great need for efficient resource usage in resource-constrained devices.

The aim of this research thesis was to compare the impact on resource usage of two different software architectures when implementing an IoT application on a resource-constrained device. The device that was chosen for this thesis is a Raspberry Pi 4 as it is an excellent embedded device on which to conduct experiments. The two different architectures that were compared in this study were: microservices and monolithic. In order to ensure that the results were not language specific, the architectures were developed in three programming languages: Go, Python and C++. Although a number of studies were found that compare the resource usage of different programming languages, only one study could be found that focused on resource usage in resource-constrained devices and no studies could be found that compared the performance of the two different architectures in resource-constrained devices.

The microservices architecture offers many benefits for WoT and IoT, such as modularity, flexibility and maintainability. A number of studies were found that concluded that the microservices architecture is well suited for WoT and IoT as it shares many of the same goals. WoT/IoT is inherently dynamic and has many endpoints, which can present a lot of challenges to design and implementation. An architecture like microservices can exploit these characteristics, turning the challenges into advantages. However, very little research has been done to compare the technical performance of the microservices architecture to the monolithic architecture, especially in the context of IoT. Therefore, a technical comparison of these two architectures was made in this research thesis.

The choice of messaging protocol for communication between microservices/components was also reviewed and discussed. It was found that a lightweight non-HTTP messaging protocol was best suited to resource-constrained devices. Three options were considered in depth: MQTT, AMQP and CoAP. A number of studies were analysed, and it was found that MQTT offers less features (e.g. quality of service, security, reliability) than AMQP and is therefore more

lightweight. When comparing MQTT and CoAP, it was found that both protocols offer many benefits. It was decided that MQTT would be used for this thesis based on preference.

The technical approach was outlined in detail, including the hardware components that were required for the project, and the software that was needed for the collection of measurements. A three-part experimental methodology was also outlined, which was closely followed. The results obtained during this research were summarized in a number of tables and charts and were discussed in detail. The results Section was split into three main parts: power consumption, runtime and memory consumption. Two different power consumption readings were taken for the first part of the results: maximum instantaneous power consumption and total power consumption. It was found that the maximum instantaneous power consumption of the microservices architecture was, on average across all languages, 14,9% higher than for the monolithic architecture. It was also found that Go had the highest maximum instantaneous power consumption, for both architectures, whilst Python and C++ had similar measurements.

The results for total power consumption (over the full runtime) were slightly different. When comparing the two architectures, it was found that the power consumption values were very similar and on average, across all languages, the microservices architecture consumed only 3,0% more than the monolithic architecture. It was also found that, when using the monolithic architecture, total power consumption was almost identical for all languages. With the microservices architecture, Python had the highest consumption, followed by Go and the C++, although the values did not differ by much. It was also clear that, although maximum instantaneous power consumption can be useful to understand peak power requirements, it is not directly proportional to total power consumption, e.g. Python had the smallest maximum instantaneous power consumption but the greatest total power consumption.

The second part of the results considered runtime performance. It was found that the microservices architecture had a longer runtime than the monolithic architecture for Go and C++, whilst the inverse was true for Python, which could be related to CPU-bound concurrency optimization of the different languages. When comparing runtime performance of the programming languages, the results were largely in line with expectations. C++ had the shortest runtime, followed closely by Go. Python had a significantly longer runtime, which makes sense since Python is the only interpreted language that was used in this project. It was interesting to note that the Python runtime was much longer when using a monolithic architecture than when using a microservices architecture, which was not the case for C++ or Go. With the microservices architecture, Python had a mean runtime that was 319,4% greater than that of C++, whilst Go's

mean runtime was 31,5% greater than that of C++. Similar differences were observed for the monolithic architecture.

Memory consumption was measured using three different metrics: maximum RSS, total RSS and CPU usage. A comparison of maximum RSS by architecture showed that maximum RSS for the microservices architecture was 37,1% greater than for the monolithic architecture. The difference was especially significant for Python (65,9% difference). It was found that Go had a significantly greater maximum RSS than the other languages, for both architectures. Python had the smallest maximum RSS for the monolithic architecture, whilst C++ had the smallest for the microservices architecture. The results for total RSS were very different from maximum RSS, both by architecture and by language. The total RSS measurements for Go and C++ did not differ much by architecture, whilst there was a big difference for Python. On average, the total RSS was 127,0% greater for the monolithic architecture than the microservices architecture when using Python. Comparing by language, the total RSS of Python was significantly greater than for the other two languages, especially for the monolithic architecture, whilst Go and C++ had very similar total RSS measurements.

The last memory consumption metric that was considered was the average CPU usage. It was found that the monolithic architecture had, on average, 14,9% higher CPU usage than the microservices architecture and the biggest difference was observed for Python. A comparison by language showed that Go had the greatest CPU usage, for both architectures. C++ had the second highest CPU usage and Python had the lowest usage. These findings were in line with expectations since Go has built-in light-weight threads (Go routines) and therefore can optimize CPU usage.

Overall, this research thesis yielded some very interesting results, some of which were expected whilst others were not. The results showed that the monolithic architecture had better performance in most metrics, i.e. maximum instantaneous power consumption, total power consumption (only for Go and Python), overall runtime (only for Go and C++), maximum RSS and CPU. Therefore, it could be concluded that, when deploying small scale applications on IoT devices, the monolithic architecture may offer more benefits. It is quite likely, however, that the microservices architecture could outperform the monolithic architecture with larger scale applications. The size of the application should therefore be considered when choosing a software architecture.

Clearly there is still substantial room for contribution in this area of research. Very limited research has been done on the performance of the microservices architecture compared to the monolithic architecture and no such research could be found in the context of IoT. This is surprising since many corporations are moving towards microservices, and significant research is being done on the use of this architecture. It is therefore important to understand what advantages and disadvantages this architecture could introduce. Although this study made a technical comparison of the two architectures and of different languages, it was done on small scale and on a single embedded device. Additional research on a larger scale would provide valuable insights.

8. References

Al-Debagy, O. and Martinek, P. (2018) "A comparative review of microservices and monolithic architectures," in *2018 IEEE 18th International Symposium on Computational Intelligence and Informatics (CINTI)*. IEEE.

Al-Masri, E., Kalyanam, K., Batts, J., Kim, J., Singh, S., Vo, T. and Yan, C., 2020. Investigating Messaging Protocols for the Internet of Things (IoT). *IEEE Access*, 8, pp.94880-94911.

Atchison, L., Wieldt, T. and Paul, F., 2018. Microservices: What They Are And How They Work. [online] New Relic Blog. Available at: <https://blog.newrelic.com/technology/microservices-what-they-are-why-to-use-them/> [Accessed 16 January 2021].

Babaria, U., 2018. *Why Iot Development Needs Microservices And Containerization*. [online] Einfochips.com. Available at: <https://www.einfochips.com/blog/why-iot-development-needs-microservices-and-containerization/> [Accessed 8 January 2021].

Bahashwan, A. and Manickam, S., 2018. A Brief Review of Messaging Protocol Standards for Internet of Things (IoT). *Journal of Cyber Security and Mobility*, 8(1), pp.1-14.

Biggs, J., & Popper, B. (2020). What's so great about Go? [online] Stackoverflow.blog. Available at: <https://stackoverflow.blog/2020/11/02/go-golang-learn-fast-programming-languages/> [Accessed: 29 June 2021].

Blokdyk, G. (2018). *IBM docs: Complete self-assessment guide*. North Charleston, SC: Createspace Independent Publishing Platform.

Bolar, T., 2020. *Web Of Things Over Iot And Its Applications*. [online] InfoQ. Available at: <https://www.infoq.com/articles/web-of-things-iot-apps/#:~:text=What%20is%20Internet%20of%20Things,and%2For%20other%20connected%20devices.> [Accessed 8 January 2021].

Burhan, M. *et al.* (2018) "IoT elements, layered architectures and security issues: A comprehensive survey," *Sensors (Basel, Switzerland)*, 18(9). doi: 10.3390/s18092796.

Butzin, B., Golatowski, F. and Timmermann, D. (2016) "Microservices approach for the internet of things," in *2016 IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA)*. IEEE.

Cook, S., 2020. *60+ Iot Statistics, Facts And Trends [2020 Edition] | Comparitech*. [online] Comparitech. Available at: <https://www.comparitech.com/internet-providers/iot-statistics/> [Accessed 23 December 2020].

De la Torre, C., Wagner, B. and Rousos, M., 2020. NET Microservices: Architecture For Containerized .NET Applications. 1st ed. Washington: Microsoft Developer Division.

Dizdarević, J., Carpio, F., Jukan, A. and Masip-Bruin, X., 2019. A Survey of Communication Protocols for Internet of Things and Related Challenges of Fog and Cloud Computing Integration. *ACM Computing Surveys*, 51(6), pp.1-29.

Eclipse Mosquitto (2018) [online] Mosquitto.org. Available at: <https://mosquitto.org/> [Accessed: January 25, 2021].

Fowler, M., 2014. Microservices. [online] martinfowler.com. Available at: <https://martinfowler.com/articles/microservices.html> [Accessed 17 December 2020].

Georgiou, S., Kechagia, M. and Spinellis, D. (2017) "Analyzing programming languages' energy consumption: An empirical study," in *Proceedings of the 21st Pan-Hellenic Conference on Informatics*. New York, NY, USA: ACM.

Gnatyk, R. (2018). Microservices vs Monolith: which architecture is the best choice for your business? [online] N-ix.com. Available at: <https://www.n-ix.com/microservices-vs-monolith-which-architecture-best-choice-your-business/> [Accessed: 29 June 2021].

Go Documentation. (2016). [online] Golang.org. Available at: <https://golang.org/doc/> [Accessed: 29 June 2021].

Gomez, C. *et al.* (2012) "Problem statement and requirements for IPv6 over low-power wireless personal area network (6LoWPAN) routing." [online] IETF.org. Available at: https://tools.ietf.org/html/rfc6606 [Accessed: 25 January 2021].

Guinard, D. and Trifa, V., 2016. *Building The Web Of Things*. 1st ed. New York: Manning.

Hindle, A. *et al.* (2014) "GreenMiner: a hardware based mining software repositories software energy consumption framework," in *Proceedings of the 11th Working Conference on Mining Software Repositories - MSR 2014*. New York, New York, USA: ACM Press.

Jaffey, T. (2014). MQTT and CoAP, IoT Protocols. [online] Eclipse.org. Available at: <https://www.eclipse.org/community/eclipse_newsletter/2014/february/article2.php> [Accessed: 15 June 2021].

Keranen, A., Ersue, M. and Bormann, C. (2014) "Terminology for constrained-node networks," *Internet Engineering Task Force*. [online] IETF.org. Available at: <https://tools.ietf.org/html/rfc7228> [Accessed: 25 January 2021].

Komiyama, N., 2017. *Fork Hides Noodle-Slurping Sounds*. [online] The Japan Times. Available at: <https://www.japantimes.co.jp/news/2017/10/28/national/media-national/fork-hides-noodle-slurping-sounds/> [Accessed 23 December 2020].

Kuhlman, D. (2011). A python book: Beginning python, advanced python, and python exercises. Platypus Global Media.

Lu, D. et al. (2017) "A Secure Microservice Framework for IoT," in 2017 IEEE Symposium on Service-Oriented System Engineering (SOSE). IEEE.

Lueth, K. L. (2014) Why it is called Internet of Things: Definition, history, disambiguation. [online] Iot-analytics.com. Available at: <https://iot-analytics.com/internet-of-things-definition/> [Accessed: 27 January 2021].

Mena, M. *et al.* (2020) "WoTnectivity: A communication pattern for different web of things connection protocols," in *2020 IEEE 44th Annual Computers, Software, and Applications Conference (COMPSAC)*. IEEE.

Mishra, H. (2019) *COAP vs MQTT*. [online] Iotbyhvm.ooo. Available at: <https://iotbyhvm.ooo/coap-vs-mqtt/> [Accessed: 5 February 2021].

Morel, A., 2019. *AMQP Vs MQTT | Top 14 Differences To Learn With Infographics*. [online] EDUCBA. Available at: <https://www.educba.com/amqp-vs-mqtt/> [Accessed 17 January 2021].

Mqtt.org. 2020. *MQTT - The Standard For Iot Messaging*. [online] Available at: <https://mqtt.org> [Accessed 17 January 2021].

Nagasai, M. (2017) Classification of IoT Devices. [online] Cisoplatform.com. Available at: <https://www.cisoplatform.com/profiles/blogs/classification-of-iot-devices> [Accessed: 25 January 2021].

Naik, N. (2017) "Choice of effective messaging protocols for IoT systems: MQTT, CoAP, AMQP and HTTP," in *2017 IEEE International Systems Engineering Symposium (ISSE)*. IEEE.

Paessler.com. 2018. *What Is MQTT? Definition And Details*. [online] Available at: <https://www.paessler.com/it-explained/mqtt> [Accessed 17 January 2021].

Parwej, Dr. Firoj & Akhtar, Nikhat & Perwej, Dr. Yusuf. (2019). An Empirical Analysis of Web of Things (WoT). Volume 10. Page 1-9. 10.26483/ijarcs.v10i3.

Pereira, R. *et al.* (2017) "Energy efficiency across programming languages: how do energy, time, and memory relate?," in *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering*. New York, NY, USA: ACM.

Richardson, C., 2019. What Are Microservices?. [online] microservices.io. Available at: <https://microservices.io/> [Accessed 16 December 2020].

Rodgers, P., 2005. "Service-Oriented Development on NetKernel- Patterns, Processes & Products to Reduce System Complexity Web Services Edge 2005 East: CS-3". *CloudComputingExpo 2005*. SYS-CON TV.

Rungta, K. (2020) *Embedded systems tutorial: What is, types, history & examples.* [online] Guru99.com. Available at: <https://www.guru99.com/embedded-systems-tutorial.html> [Accessed: 27 January 2021].

Santana, C., Alencar, B. and Prazeres, C. (2018) "Microservices: A mapping study for internet of things solutions," in *2018 IEEE 17th International Symposium on Network Computing and Applications (NCA)*. IEEE.

Santana, C. *et al.* (2019) "A reliable architecture based on reactive microservices for IoT applications," in *Proceedings of the 25th Brazillian Symposium on Multimedia and the Web*. New York, NY, USA: ACM.

Stansberry, J. (2015) *MQTT and CoAP: Underlying Protocols for the IoT.* [online] Electronicdesign.com. Available at: <https://www.electronicdesign.com/technologies/iot/article/21800998/silicon-labs-mqtt-and-coap-underlying-protocols-for-the-iot> [Accessed: 5 February 2021].

Stroustrup, B. (2013). *The C++ programming language* (4th ed.). Boston, MA: Addison-Wesley Educational.

Sukumaran, J. (2020). *Syrupy: System Resource Usage Profiler*. [online] github.com. Available at: <https://github.com/jeetsukumaran/Syrupy> [Accessed 21 February 2021].

Tapia, F. *et al.* (2020) “From monolithic systems to microservices: A comparative study of performance,” *Applied sciences (Basel, Switzerland)*, 10(17), p. 5797.

Thones, J. (2015) “Microservices,” *IEEE software*, 32(1), pp. 116–116.

UUGear. (2018). DHT11 Humidity & Temperature Sensor Module. [online] Uugear.com. Available at: <http://www.uugear.com/portfolio/dht11-humidity-temperature-sensor-module/> [Accessed 15 July 2021].

Van der Westhuizen, H. W. and Hancke, G. P. (2018) “Practical comparison between COAP and MQTT - sensor to server level,” in *2018 Wireless Advanced (WiAd)*. IEEE.

VMware (2020) *Messaging that just works — RabbitMQ*. [online] Rabbitmq.com. Available at: <https://www.rabbitmq.com/> [Accessed: January 25 January 2021].

W3.org. 2019. *Thing Description (TD) Ontology*. [online] w3.org. Available at: <https://www.w3.org/2019/wot/td> [Accessed 8 January 2021].

Zeiner, H. *et al.* (2016) “SeCoS: Web of Things platform based on a microservices architecture and support of time-awareness,” *E & I*, 133(3), pp. 158–162.

Appendix A – Code

The code that was used for this project is shown in this appendix.

A.1 – Bash Code

Bash scripts were used to launch Go, Python or C++ programs on the Raspberry Pi.

```
1   #Script to run 3 go scripts in parallel on the publisher side.
2   #Run by using the following line in the correct directory in terminal: bash runCppScripts <ipaddress of
    raspberry pi>
3   #$1 represents the passed argument (ipaddress).
4
5   start=$(date +%s.%N)
6
7   /usr/bin/time -v -o memoryResultsGoLong.txt -a ./humTempPub $1 &
8   /usr/bin/time -v -o memoryResultsGoLong.txt -a ./pirPub $1 &
9   /usr/bin/time -v -o memoryResultsGoLong.txt -a ./ledSub $1;
10
11  end=$(date +%s.%N)
12
13  awk -v var1="$start" -v var2="$end" 'BEGIN {print "Overall runtime: " + var2 - var1}'
```

Figure A.1*: Screenshot showing the bash script to launch the Go microservices application*

```
1   #Script to run 3 go scripts in parallel on the publisher side.
2   #Run by using the following line in the correct directory in terminal: bash runCppScripts
    <ipaddress of raspberry pi>
3   #$1 represents the passed argument (ipaddress).
4
5   start=$(date +%s.%N)
6
7   /usr/bin/time -v -o memoryResultsGoMonoLong.txt -a ./allServices $1;
8
9   end=$(date +%s.%N)
10
11  awk -v var1="$start" -v var2="$end" 'BEGIN {print "Overall runtime: " + var2 - var1}'
```

Figure A.2*: Screenshot showing the bash script to launch the Go monolithic application*

```
1   #Script to run 3 python scripts in parallel on the publisher side.
2   #Run by using the following line in the correct directory in terminal: bash runPythonScripts <ipaddress of
    raspberry pi>
3   #$1 represents the passed argument (ipaddress).
4
5   start=$(date +%s.%N)
6
7   /usr/bin/time -v -o memoryResultsPythonLong.txt -a python3 humidityTemperatureMicroservicePublisher.py $1 &
8   /usr/bin/time -v -o memoryResultsPythonLong.txt -a python3 pirMicroservicePublisher.py $1 &
9   /usr/bin/time -v -o memoryResultsPythonLong.txt -a python3 ledMicroserviceSubscriber.py $1;
10
11  end=$(date +%s.%N)
12
13  awk -v var1="$start" -v var2="$end" 'BEGIN {print "Overall runtime: " + var2 - var1}'
```

Figure A.3*: Screenshot showing the bash script to launch the Python microservices application*

```
1   #Script to run python script on the publisher side.
2   #Run by using the following line in the correct directory in terminal: bash runPythonScripts
    <ipaddress of raspberry pi>
3   #$1 represents the passed argument (ipaddress).
4
5   start=$(date +%s.%N)
6
7   /usr/bin/time -v -o memoryResultsPythonMonoLong.txt -a python3 allServices.py $1
8
9   end=$(date +%s.%N)
10
11  awk -v var1="$start" -v var2="$end" 'BEGIN {print "Overall runtime: " + var2 - var1}'
```

Figure A.4: *Screenshot showing the bash script to launch the Python monolithic application*

```
1   #Script to run 3 python scripts in parallel on the publisher side.
2   #Run by using the following line in the correct directory in terminal: bash runCppScripts
    <ipaddress of raspberry pi>
3   #$1 represents the passed argument (ipaddress).
4
5   start=$(date +%s.%N)
6
7   /usr/bin/time -v -o memoryResultsCppLong.txt -a ./humTempPub $1 &
8   /usr/bin/time -v -o memoryResultsCppLong.txt -a ./pirPub $1 &
9   /usr/bin/time -v -o memoryResultsCppLong.txt -a ./ledSub $1;
10
11  end=$(date +%s.%N)
12
13  awk -v var1="$start" -v var2="$end" 'BEGIN {print "Overall runtime: " + var2 - var1}'
```

Figure A.5: *Screenshot showing the bash script to launch the C++ microservices application*

```
1   #Script to run 3 python scripts in parallel on the publisher side.
2   #Run by using the following line in the correct directory in terminal: bash runCppScripts
    <ipaddress of raspberry pi>
3   #$1 represents the passed argument (ipaddress).
4
5   start=$(date +%s.%N)
6
7   /usr/bin/time -v -o memoryResultsCppMonoLong.txt -a ./allServices $1
8
9   end=$(date +%s.%N)
10
11  awk -v var1="$start" -v var2="$end" 'BEGIN {print "Overall runtime: " + var2 - var1}'
```

Figure A.6: *Screenshot showing the bash script to launch the C++ monolithic application*

A.2 – Microservices Code

Go Code – Temperature and Humidity Sensor

```go
package main

import "C"

import (
	"encoding/json"
	"fmt"
	"io/ioutil"
	"log"
	"os"
	"os/signal"
	"strconv"
	"strings"
	"syscall"
	"time"

	mqtt "github.com/eclipse/paho.mqtt.golang"
)

var sessionStatus bool = true
var counter int = 0
var start = time.Now()
var dhtStart = time.Now()
var dhtEnd = time.Now()
var dhtDuration float64
var TOPIC_H string = "Humidity"
var TOPIC_T string = "Temperature"
var ADDRESS string
var PORT = 1883
var temperatureReading float32 = 0
var humidityReading float32 = 0

type tempStruct struct {
	Temp float32
	Unit string
}

type humStruct struct {
	Humidity float32
	Unit     string
}

type reading interface {
	structToJSON() []byte
}

// Function to handle receipt of a message
var messagePubHandler mqtt.MessageHandler = func(client mqtt.Client, msg mqtt.Message) {
	fmt.Println("Message received")
}

```

```
52  // Function to publish a message
53  func publish(client mqtt.Client) {
54      if !sessionStatus {
55          doneString := "{\"Done\": \"True\"}"
56          client.Publish(TOPIC_T, 0, false, doneString)
57          client.Publish(TOPIC_H, 0, false, doneString)
58          return
59      }
60      dhtEnd = time.Now()
61      dhtDuration = dhtEnd.Sub(dhtStart).Seconds()
62      if (temperatureReading == 0 && humidityReading == 0) || dhtDuration > 1 {
63          C.read_dht_data()
64          dhtStart = time.Now()
65          byteSlice, readErr := ioutil.ReadFile("reading.txt")
66          if readErr != nil {
67              log.Fatal(readErr)
68          }
69          mySlice := byteSliceToIntSlice(byteSlice)
70          if mySlice[0] != 0 && mySlice[2] != 0 {
71              temperatureReading = float32(mySlice[2] + (mySlice[3] / 10))
72              humidityReading = float32(mySlice[0] + (mySlice[1] / 10))
73          }
74      }
75      currentTemperature := tempStruct{
76          Temp: temperatureReading,
77          Unit: "C",
78      }
79      currentHumidity := humStruct{
80          Humidity: humidityReading,
81          Unit:     "%",
82      }
83      jsonTemperature := currentTemperature.structToJSON()
84      jsonHumidity := currentHumidity.structToJSON()
85      client.Publish(TOPIC_T, 0, false, string(jsonTemperature))
86      client.Publish(TOPIC_H, 0, false, string(jsonHumidity))
87      return
88  }
89
90  func getJSON(r reading) []byte {
91      return r.structToJSON()
92  }
```

```
94   func byteSliceToIntSlice(bs []byte) []int {
95       strings := strings.Split(string(bs), ",")
96       result := make([]int, len(strings))
97       for i, s := range strings {
98           if len(s) == 0 {
99               continue
100          }
101          n, convErr := strconv.Atoi(s)
102          if convErr != nil {
103              log.Fatal(convErr)
104          }
105          result[i] = n
106      }
107      return result
108  }
109
110  func (ts tempStruct) structToJSON() []byte {
111      jsonReading, jsonErr := json.Marshal(ts)
112      if jsonErr != nil {
113          log.Fatal(jsonErr)
114      }
115      return jsonReading
116  }
117
118  func (ts humStruct) structToJSON() []byte {
119      jsonReading, jsonErr := json.Marshal(ts)
120      if jsonErr != nil {
121          log.Fatal(jsonErr)
122      }
123      return jsonReading
124  }
125
126  var connectHandler mqtt.OnConnectHandler = func(client mqtt.Client) {
127      fmt.Println("Connected")
128  }
129
130  var connectLostHandler mqtt.ConnectionLostHandler = func(client mqtt.Client, err error) {
131      fmt.Printf("Connection lost: %v", err)
132  }
```

```
134  func saveResultToFile(filename string, result string) {
135      file, errOpen := os.OpenFile(filename, os.O_APPEND|os.O_CREATE|os.O_WRONLY, 0644)
136      if errOpen != nil {
137          log.Fatal(errOpen)
138      }
139      byteSlice := []byte(result)
140      _, errWrite := file.Write(byteSlice)
141      if errWrite != nil {
142          log.Fatal(errWrite)
143      }
144  }
145
```

```
146 func main() {
147     // Retrieve the IP address
148     if len(os.Args) <= 1 {
149         fmt.Println("IP address must be provided as a command line argument")
150         os.Exit(1)
151     }
152     ADDRESS = os.Args[1]
153     fmt.Println(ADDRESS)
154 
155     // End program with ctrl-C
156     c := make(chan os.Signal, 1)
157     signal.Notify(c, os.Interrupt, syscall.SIGTERM)
158     go func() {
159         <-c
160         os.Exit(0)
161     }()
162 
163     // Creat MQTT client
164     opts := mqtt.NewClientOptions()
165     opts.AddBroker(fmt.Sprintf("tcp://%s:%d", ADDRESS, PORT))
166     opts.OnConnect = connectHandler
167     opts.OnConnectionLost = connectLostHandler
168     client := mqtt.NewClient(opts)
169     if token := client.Connect(); token.Wait() && token.Error() != nil {
170         panic(token.Error())
171     }
172     // Publish to topic
173     numIterations := 100000
174     for i := 0; i < numIterations; i++ {
175         if i == numIterations-1 {
176             sessionStatus = false
177         }
178         publish(client)
179     }
180 
181     // Disconnect and save results to file
182     client.Disconnect(100)
183     end := time.Now()
184     duration := end.Sub(start).Seconds()
185     resultString := fmt.Sprint("Humidity and temperature runtime = ", duration, "\n")
186     saveResultToFile("piResultsGoLong.txt", resultString)
187     fmt.Println("Humidity and temperature runtime at end =", duration)
188 }
```

The following code is the C code that has been wrapped in order to access the GPIO pins.

```
3   // #cgo LDFLAGS: -lwiringPi
4   // #include <wiringPi.h>
5   // #include <stdio.h>
6   // #include <stdlib.h>
7   // #include <stdint.h>
8   // #include <string.h>
9   // #include <time.h>
10  // #include <unistd.h>
11  // #define MAX_TIMINGS  85
12  // #define DHT_PIN      7   /* GPIO-4 */
13  // int data[5] = { 0, 0, 0, 0, 0 };
14  // double read_dht_data()
15  // {
16  //  clock_t begin = clock();
17  //  wiringPiSetup();
18  //  uint8_t laststate   = HIGH;
19  //  uint8_t counter     = 0;
20  //  uint8_t j           = 0, i;
21  //  data[0] = data[1] = data[2] = data[3] = data[4] = 0;
22  //  /* pull pin down for 18 milliseconds */
23  //  pinMode( DHT_PIN, OUTPUT );
24  //  digitalWrite( DHT_PIN, LOW );
25  //  delay( 18 );
26  //  /* prepare to read the pin */
27  //  digitalWrite( DHT_PIN, HIGH);
28  //  delayMicroseconds( 40 );
29  //  pinMode( DHT_PIN, INPUT );
30  //  /* detect change and read data */
31  //  for ( i = 0; i < MAX_TIMINGS; i++ )
32  //  {
33  //      counter = 0;
34  //      while ( digitalRead( DHT_PIN ) == laststate )
35  //      {
36  //          counter++;
37  //          delayMicroseconds( 2 );
38  //          if ( counter == 255 )
39  //              break;
40  //      }
41  //      laststate = digitalRead( DHT_PIN );
42  //      if ( counter == 255 ){
43  //          break;
```

```
44    //        }
45    //        /* ignore first 3 transitions */
46    //        if ( (i >= 4) && (i % 2 == 0) )
47    //        {
48    //            /* shove each bit into the storage bytes */
49    //            data[j / 8] <<= 1;
50    //            if ( counter > 16 )
51    //                data[j / 8] |= 1;
52    //            j++;
53    //        }
54    //    }
55    //    /*
56    //     * check we read 40 bits (8bit x 5 ) + verify checksum in the last byte
57    //     * print it out if data is good
58    //     */
59    //    if ( (j >= 40) &&
60    //         (data[4] == ( (data[0] + data[1] + data[2] + data[3]) & 0xFF) ) )
61    //    {
62    //        FILE *f = fopen("reading.txt", "w");
63    //        if (f == NULL)
64    //        {
65    //            printf("Error opening file!\n");
66    //            exit(1);
67    //        }
68    //        fprintf(f, "%d,%d,%d,%d,%d", data[0], data[1], data[2], data[3], data[4]);
69    //        fclose(f);
70    //        clock_t end = clock();
71    //        double time_spent = (double)(end - begin) / CLOCKS_PER_SEC;
72    //        return time_spent;
73    //    } else  {
74    //        FILE *f = fopen("reading.txt", "w");
75    //        if (f == NULL)
76    //        {
77    //            printf("Error opening file!\n");
78    //            exit(1);
79    //        }
80    //        fprintf(f, "%d,%d,%d,%d,%d", data[0], data[1], data[2], data[3], data[4]);
81    //        fclose(f);
82    //        return data[0];
83    //    }
84    // }
```

```go
1   package main
2
3   import (
4       "encoding/json"
5       "fmt"
6       "log"
7       "os"
8       "os/signal"
9       "syscall"
10      "time"
11
12      mqtt "github.com/eclipse/paho.mqtt.golang"
13      rpio "github.com/stianeikeland/go-rpio"
14  )
15
16  var sessionStatus bool = true
17  var counter int = 0
18  var start = time.Now()
19  var TOPIC string = "PIR"
20
21  type pirStruct struct {
22      PIR bool
23  }
24
25  func saveResultToFile(filename string, result string) {
26      file, errOpen := os.OpenFile(filename, os.O_APPEND|os.O_CREATE|os.O_WRONLY, 0644)
27      if errOpen != nil {
28          log.Fatal(errOpen)
29      }
30      byteSlice := []byte(result)
31      _, errWrite := file.Write(byteSlice)
32      if errWrite != nil {
33          log.Fatal(errWrite)
34      }
35  }
36
37  // Function to handle receipt of a message
38  var messagePubHandler mqtt.MessageHandler = func(client mqtt.Client, msg mqtt.Message) {
39      fmt.Println("Message received")
40  }
41
```

```
42  //Function to publish a message
43  func publish(client mqtt.Client) {
44      if !sessionStatus {
45          doneString := "{\"Done\": \"True\"}"
46          client.Publish(TOPIC, 0, false, doneString)
47          return
48      } else {
49          pirPin := rpio.Pin(17)
50          pirPin.Input()
51          readValue := pirPin.Read()
52          var pirReading bool
53          if int(readValue) == 1 {
54              pirReading = true
55          } else {
56              pirReading = false
57          }
58          currentPIR := pirStruct{
59              PIR: pirReading,
60          }
61          jsonPIR := currentPIR.structToJSON()
62          client.Publish(TOPIC, 0, false, string(jsonPIR))
63          return
64      }
65  }
66
67  func (ps pirStruct) structToJSON() []byte {
68      jsonReading, jsonErr := json.Marshal(ps)
69      if jsonErr != nil {
70          log.Fatal(jsonErr)
71      }
72      return jsonReading
73  }
74
75  var connectHandler mqtt.OnConnectHandler = func(client mqtt.Client) {
76      fmt.Println("Connected")
77  }
78
79  var connectLostHandler mqtt.ConnectionLostHandler = func(client mqtt.Client, err error) {
80      fmt.Printf("Connection lost: %v", err)
81  }
82
```

```go
83  var ADDRESS string
84  var PORT = 1883
85
86  func main() {
87      // Retrieve IP address
88      if len(os.Args) <= 1 {
89          fmt.Println("IP address must be provided as a command line argument")
90          os.Exit(1)
91      }
92      ADDRESS = os.Args[1]
93      fmt.Println(ADDRESS)
94
95      // Check that RPIO opened correctly
96      if err := rpio.Open(); err != nil {
97          fmt.Println(err)
98          os.Exit(1)
99      }
100
101     // End program with ctrl-C
102     c := make(chan os.Signal, 1)
103     signal.Notify(c, os.Interrupt, syscall.SIGTERM)
104     go func() {
105         <-c
106         os.Exit(0)
107     }()
108
109     // Creat MQTT client
110     opts := mqtt.NewClientOptions()
111     opts.AddBroker(fmt.Sprintf("tcp://%s:%d", ADDRESS, PORT))
112     opts.SetClientID("go_mqtt_client_pir")
113     opts.SetDefaultPublishHandler(messagePubHandler)
114     opts.OnConnect = connectHandler
115     opts.OnConnectionLost = connectLostHandler
116     client := mqtt.NewClient(opts)
117     if token := client.Connect(); token.Wait() && token.Error() != nil {
118         panic(token.Error())
119     }
120
```

```go
121        // Publish to topic
122        numIterations := 100000
123        for i := 0; i < numIterations; i++ {
124            if i == numIterations-1 {
125                sessionStatus = false
126            }
127            publish(client)
128        }
129
130        // Disconnect
131        client.Disconnect(100)
132        end := time.Now()
133        duration := end.Sub(start).Seconds()
134        resultString := fmt.Sprint("PIR runtime = ", duration, "\n")
135        saveResultToFile("piResultsGoLong.txt", resultString)
136        fmt.Println("PIR runtime = ", duration)
137    }
```

Go Code – LE

```go
1    package main
2
3    import (
4        "encoding/json"
5        "fmt"
6        "log"
7        "os"
8        "os/signal"
9        "strings"
10       "syscall"
11       "time"
12
13       mqtt "github.com/eclipse/paho.mqtt.golang"
14       rpio "github.com/stianeikeland/go-rpio"
15   )
16
17   var sessionStatus bool = true
18   var counter int = 0
19   var start = time.Now()
20   var TOPIC string = "LED"
21
22   type ledStruct struct {
23       LED_1 bool
24       GPIO  int
25   }
26
27   func saveResultToFile(filename string, result string) {
28       file, errOpen := os.OpenFile(filename, os.O_APPEND|os.O_CREATE|os.O_WRONLY, 0644)
29       if errOpen != nil {
30           log.Fatal(errOpen)
31       }
32       byteSlice := []byte(result)
33       _, errWrite := file.Write(byteSlice)
34       if errWrite != nil {
35           log.Fatal(errWrite)
36       }
37   }
38
```

```
39  // Function to handle receipt of message
40  var messagePubHandler mqtt.MessageHandler = func(client mqtt.Client, msg mqtt.Message) {
41      counter++
42      if counter == 1 {
43          start = time.Now()
44      }
45      var led ledStruct
46      ledPin := rpio.Pin(12)
47      if strings.Contains(string(msg.Payload()), "Done") {
48          sessionStatus = false
49          ledPin.Output()
50          ledPin.Low()
51          end := time.Now()
52          duration := end.Sub(start).Seconds()
53          resultString := fmt.Sprint("LED subsriber runtime = ", duration, "\n")
54          saveResultToFile("piResultsGoLong.txt", resultString)
55          fmt.Println("LED subsriber runtime = ", duration)
56      } else {
57          json.Unmarshal([]byte(msg.Payload()), &led)
58          ledPin = rpio.Pin(led.GPIO)
59          ledPin.Output()
60          if led.LED_1 {
61              ledPin.High()
62          } else {
63              ledPin.Low()
64          }
65      }
66  }
67
68  var connectHandler mqtt.OnConnectHandler = func(client mqtt.Client) {
69      fmt.Println("Connected")
70  }
71
72  var connectLostHandler mqtt.ConnectionLostHandler = func(client mqtt.Client, err error) {
73      fmt.Printf("Connection lost: %v", err)
74  }
75
76  var ADDRESS string
77  var PORT = 1883
78
```

```
79  func main() {
80
81      // Save the IP address
82      if len(os.Args) <= 1 {
83          fmt.Println("IP address must be provided as a command line argument")
84          os.Exit(1)
85      }
86      ADDRESS = os.Args[1]
87      fmt.Println(ADDRESS)
88
89      // Check that RPIO opened correctly
90      if err := rpio.Open(); err != nil {
91          fmt.Println(err)
92          os.Exit(1)
93      }
94
95      // End program with ctrl-C
96      c := make(chan os.Signal, 1)
97      signal.Notify(c, os.Interrupt, syscall.SIGTERM)
98      go func() {
99          <-c
100         os.Exit(0)
101     }()
102
103     // Creat MQTT client
104     opts := mqtt.NewClientOptions()
105     opts.AddBroker(fmt.Sprintf("tcp://%s:%d", ADDRESS, PORT))
106     opts.SetClientID("go_mqtt_client_led")
107     opts.SetDefaultPublishHandler(messagePubHandler)
108     opts.OnConnect = connectHandler
109     opts.OnConnectionLost = connectLostHandler
110     client := mqtt.NewClient(opts)
111     if token := client.Connect(); token.Wait() && token.Error() != nil {
112         panic(token.Error())
113     }
114
115     // Subscribe to topic
116     token := client.Subscribe(TOPIC, 1, nil)
117     token.Wait()
118
119     // Stay in loop to receive message
120     for sessionStatus { //Do nothing
121     }
122
123     // Disconnect
124     client.Disconnect(100)
125 }
```

Python Code – Temperature and Humidity

```
1   import time
2   import board
3   import adafruit_dht
4   import paho.mqtt.publish as publish
5   import json
6   import sys
7
8   start = time.time()
9   startDht = time.time()
10  endDht = time.time()
11  humidity = 0
12  temperature = 0
13
14  MQTT_SERVER = sys.argv[1]
15
16  # Initial the dht device, with data pin connected to:
17  dhtDevice = adafruit_dht.DHT11(board.D4)
18  count = 0
19  while count < 100000:
20      try:
21          endDht = time.time()
22          if(humidity == 0 and temperature == 0) or (endDht-startDht) > 1:
23              humidity = dhtDevice.humidity # Get current humidity from dht11
24              temperature = dhtDevice.temperature # Get current temperature from dht11
25              startDht = time.time()
26          hum_json = {"Humidity": humidity, "Unit": "%"}
27          publish.single("Humidity", json.dumps(hum_json), hostname=MQTT_SERVER)
28          temp_json = {"Temp": temperature, "Unit": "C"}
29          publish.single("Temperature", json.dumps(temp_json), hostname=MQTT_SERVER)
30          count += 1
31      except RuntimeError as error:  # Errors happen fairly often, DHT's are hard to read, just keep going
32          error.args[0]
33
34  publish.single("Humidity", json.dumps({"Done": True}), port=1883, hostname=MQTT_SERVER)
35  publish.single("Temperature", json.dumps({"Done": True}), port=1883, hostname=MQTT_SERVER)
36  end = time.time()
37  print("Humidity and temperature runtime = " + str(end-start))
38  with open("piResultsPythonLong.txt", "a") as myfile:
39      myfile.write("Humidity and temperature publisher runtime = " + str(end-start) + "\n")
```

Python Code – PIR

```python
import time
import paho.mqtt.publish as publish
import json
import RPi.GPIO as GPIO
import sys

start = time.time()

MQTT_SERVER = sys.argv[1]
MQTT_PATH = "PIR"

# Initial the pir device, with data pin connected to 17:
GPIO.setmode(GPIO.BCM)
GPIO.setup(17, GPIO.IN)
presence = False

count = 0
while count < 100000:
    try:
        presence = GPIO.input(17)
        temp_json = {"PIR": presence}
        # Publish message
        publish.single(MQTT_PATH, json.dumps(temp_json), port=1883, hostname=MQTT_SERVER)
    except RuntimeError as error:  # Errors happen fairly often, DHT's are hard to read, just keep going
        print(error.args[0])
    count += 1

# Publish message to end program
publish.single(MQTT_PATH, json.dumps({"Done": True}), port=1883, hostname=MQTT_SERVER)
end = time.time()
print("PIR publisher runtime = " + str(end-start))
with open("piResultsPythonLong.txt", "a") as myfile:
    myfile.write("PIR publisher runtime = " + str(end - start) + "\n")
```

Python Code – LED

```python
import paho.mqtt.client as mqtt
import json
import RPi.GPIO as GPIO
import time
import sys

MQTT_SERVER = sys.argv[1]
MQTT_PATH = "LED"
GPIO.setmode(GPIO.BCM)
GPIO.setwarnings(False)
num_messages = 0
start = time.time()
pin = 0

def on_connect(client, userdata, flags, rc):
    print("Connected. Result code: "+ str(rc))
    client.subscribe(MQTT_PATH)

```

```
19  # The on_message function runs once a message is received from the broker
20  def on_message(client, userdata, msg):
21      global num_messages
22      global start
23      global pin
24      num_messages += 1
25      if num_messages == 1:
26          start = time.time()
27      received_json = json.loads(msg.payload) #convert the string to json object
28      if "Done" in received_json: # Save runtime to file
29          client.loop_stop()
30          client.disconnect()
31          end = time.time()
32          timer = end - start
33          with open("piResultsPythonLong.txt", "a") as myfile:
34              myfile.write("LED subscriber runtime = " + str(timer) + "\n")
35          print("LED subscriber runtime = " + str(timer) + "\n");
36          GPIO.output(pin, GPIO.LOW)
37
38      else: # change led status
39          led_1_status = received_json["LED_1"]
40          pin = received_json["GPIO"]
41          GPIO.setup(pin, GPIO.OUT)
42          if led_1_status:
43              GPIO.output(pin, GPIO.HIGH)
44          else:
45              GPIO.output(pin, GPIO.LOW)
46
47  client = mqtt.Client()
48  client.on_connect = on_connect
49  client.on_message = on_message
50  client.connect(MQTT_SERVER, 1883, 60)
51  client.loop_forever()
```

C++ Code – Temperature and Humidity

```
1   extern "C" {
2       #include <wiringPi.h>
3       #include <stdio.h>
4       #include <stdlib.h>
5       #include <string.h>
6       #include "MQTTClient.h"
7   }
8   #include <csignal>
9   #include <iostream>
10  #include "include/rapidjson/document.h"
11  #include "include/rapidjson/stringbuffer.h"
12  #include "include/rapidjson/prettywriter.h"
13  #include "include/rapidjson/writer.h"
14  #include <chrono>
15  #include <fstream>
16  #include <typeinfo>
17
18  // MQTT variables
19  #define CLIENTID    "hum_temp_client"
20  #define TOPIC_T       "Temperature"
21  #define TOPIC_H       "Humidity"
22  #define QOS         0
23  #define TIMEOUT     10000L
24
25  char* ADDRESS;
26
27  // Pi dht11 variables
28  #define MAXTIMINGS  85
29  #define DHTPIN      7
30  int dht11_dat[5] = { 0, 0, 0, 0, 0 }; //first 8bits is for humidity integral value, second 8bits for humidity decimal,
31
32  // RapidJson variables
33  using namespace rapidjson;
34  using namespace std::chrono;
35
```

```
36  // Function to publish message
37  int publish_message(std::string str_message, const char *topic, MQTTClient client){
38      // Initializing components for MQTT publisher
39      MQTTClient_message pubmsg = MQTTClient_message_initializer;
40      MQTTClient_deliveryToken token;
41
42      // Updating values of pubmsg object
43      char *message = new char[str_message.length() + 1];
44      strcpy(message, str_message.c_str());
45      pubmsg.payload = message;
46      pubmsg.payloadlen = (int)std::strlen(message);
47      pubmsg.qos = QOS;
48      pubmsg.retained = 0;
49
50      MQTTClient_publishMessage(client, topic, &pubmsg, &token); // Publish the message
51      int rc = MQTTClient_waitForCompletion(client, token, TIMEOUT);
52      return rc;
53  }
54
55  std::string json_to_string(const rapidjson::Document& doc){
56      //Serialize JSON to string for the message
57      rapidjson::StringBuffer string_buffer;
58      string_buffer.Clear();
59      rapidjson::PrettyWriter<rapidjson::StringBuffer> writer(string_buffer);
60      doc.Accept(writer);
61      return std::string(string_buffer.GetString());
62  }
63
```

```
64  // Reading of the dht11 is rather complex in C/C++. See this site that explains how readings are made: http://www.uugea
65  int* read_dht11_dat()
66  {
67      auto start1 = high_resolution_clock::now();
68      uint8_t laststate   = HIGH;
69      uint8_t counter     = 0;
70      uint8_t j       = 0, i;
71      dht11_dat[0] = dht11_dat[1] = dht11_dat[2] = dht11_dat[3] = dht11_dat[4] = 0;
72
73      // pull pin down for 18 milliseconds. This is called “Start Signal” and it is to ensure DHT11 has detected the sigr
74      pinMode( DHTPIN, OUTPUT );
75      digitalWrite( DHTPIN, LOW );
76      delay( 18 );
77
78      // Then MCU will pull up DATA pin for 40us to wait for DHT11’s response.
79      digitalWrite( DHTPIN, HIGH );
80      delayMicroseconds( 40 );
81
82      // Prepare to read the pin
83      pinMode( DHTPIN, INPUT );
84
85      // Detect change and read data
86      for ( i = 0; i < MAXTIMINGS; i++ )
87      {
88          counter = 0;
89          while ( digitalRead( DHTPIN ) == laststate )
90          {
91              counter++;
92              delayMicroseconds( 1 );
93              if ( counter == 255 )
94              {
95                  break;
96              }
97          }
98          laststate = digitalRead( DHTPIN );
99
100         if ( counter == 255 )
101             break;
102
```

```
103             // Ignore first 3 transitions
104             if ( (i >= 4) && (i % 2 == 0) )
105             {
106                 // Add each bit into the storage bytes
107                 dht11_dat[j / 8] <<= 1;
108                 if ( counter > 16 )
109                     dht11_dat[j / 8] |= 1;
110                 j++;
111             }
112         }
113 
114         // Check that 40 bits (8bit x 5 ) were read + verify checksum in the last byte
115         if ( (j >= 40) && (dht11_dat[4] == ( (dht11_dat[0] + dht11_dat[1] + dht11_dat[2] + dht11_dat[3]) & 0xFF) ) )
116         {
117             return dht11_dat; // If all ok, return pointer to the data array
118         } else  {
119             return dht11_dat; //If there was an error, set first array element to -1 as flag to main function
120         }
121     }
122 
123     int main(int argc, char* argv[])
124     {
125         auto start = high_resolution_clock::now(); // Starting timer
126         std::string input = argv[1]; // IP address as command line argument to avoid hard coding
127         input.append(":1883"); // Append MQTT port
128 
129         char char_input[input.length() + 1];
130         strcpy(char_input, input.c_str());
131         ADDRESS = char_input;
132         double temperature = 0;
133         double humidity = 0;
134 
135         MQTTClient client;
136         MQTTClient_connectOptions conn_opts = MQTTClient_connectOptions_initializer;
137         int rc;
138 
139         MQTTClient_create(&client, ADDRESS, CLIENTID, MQTTCLIENT_PERSISTENCE_NONE, NULL);
140         conn_opts.keepAliveInterval = 20;
141         conn_opts.cleansession = 1;
142 
```

```
143     if ((rc = MQTTClient_connect(client, &conn_opts)) != MQTTCLIENT_SUCCESS)
144     {
145         printf("Failed to connect, return code %d\n", rc);
146         exit(EXIT_FAILURE);
147     } else{
148         printf("Connected. Result code %d\n", rc);
149     }
150
151     wiringPiSetup(); // Required for wiringPi
152
153     int count = 0;
154     int num_iterations = 100000;
155     auto dhtStart = high_resolution_clock::now();
156     auto dhtEnd = high_resolution_clock::now();
157     std::chrono::duration<double> dhtTimer;
158
159     while(count <= num_iterations) {
160         dhtEnd = high_resolution_clock::now();
161         dhtTimer = dhtEnd - dhtStart;
162
163         if((temperature == 0 && humidity == 0) || dhtTimer > (std::chrono::seconds(1))) { //need to get values from
164             int *readings = read_dht11_dat();
165             dhtStart = high_resolution_clock::now();
166             int counter = 0;
167             while (readings[0] == -1 && counter < 5) {
168                 readings = read_dht11_dat(); // Errors frequently occur when reading dht sensor. Keep reading until values are ret
169                 counter = counter + 1;
170             }
171             if (counter == 5) {
172                 std::cout << "Problem with DHT11 sensor. Check Raspberry Pi \n";
173                 return 1;
174             }
175             humidity = readings[0] + (readings[1] / 10);
176             temperature = readings[2] + (readings[3] / 10);
177         }
178
179         if(count == num_iterations){
180             rapidjson::Document document_done;
181             document_done.SetObject();
182             rapidjson::Document::AllocatorType& allocator1 = document_done.GetAllocator();
183             document_done.AddMember("Done", true, allocator1);
```

```
184             std::string pub_message_done = json_to_string(document_done);
185             rc = publish_message(pub_message_done, TOPIC_T, client);
186             rc = publish_message(pub_message_done, TOPIC_H, client);
187         }
188         else {
189             //Create JSON DOM document object for humidity
190             rapidjson::Document document_humidity;
191             document_humidity.SetObject();
192             rapidjson::Document::AllocatorType &allocator2 = document_humidity.GetAllocator();
193             document_humidity.AddMember("Humidity", humidity, allocator2);
194             document_humidity.AddMember("Unit", "%", allocator2);
195 
196             //Create JSON DOM document object for temperature
197             rapidjson::Document document_temperature;
198             document_temperature.SetObject();
199             rapidjson::Document::AllocatorType &allocator3 = document_temperature.GetAllocator();
200             document_temperature.AddMember("Temp", temperature, allocator3);
201             document_temperature.AddMember("Unit", "C", allocator3);
202             try {
203                 std::string pub_message_humidity = json_to_string(document_humidity);
204                 rc = publish_message(pub_message_humidity, TOPIC_H, client);
205                 std::string pub_message_temperature = json_to_string(document_temperature);
206                 rc = publish_message(pub_message_temperature, TOPIC_T, client);
207             } catch (const std::exception &exc) {
208                 // catch anything thrown within try block that derives from std::exception
209                 std::cerr << exc.what();
210             }
211         }
212         count = count + 1;
213     }
214 
215     // End of loop. Stop MQTT and calculate runtime
216     MQTTClient_disconnect(client, 1000);
217     MQTTClient_destroy(&client);
218     auto end = high_resolution_clock::now();
219 
220     std::chrono::duration<double> timer = end-start;
221     std::ofstream outfile;
222     outfile.open("piResultsCppLong.txt", std::ios_base::app); // append to the results text file
223     outfile << "Humidity and temperature publisher runtime = " << timer.count() << "\n";
224     std::cout << "Humidity and temperature runtime = " << timer.count() << "\n";
225     return rc;
226 }
```

C++ Code – PIR

```
1   extern "C" {
2   #include <wiringPi.h>
3   #include <stdio.h>
4   #include <stdlib.h>
5   #include <string.h>
6   #include "MQTTClient.h"
7   }
8   #include <csignal>
9   #include <iostream>
10  #include "include/rapidjson/document.h"
11  #include "include/rapidjson/stringbuffer.h"
12  #include "include/rapidjson/prettywriter.h"
13  #include "include/rapidjson/writer.h"
14  #include <chrono>
15  #include <fstream>
16  #include <typeinfo>
17
18  // Pi variables
19  #define PIN 17
20
21  // MQTT variables
22  #define CLIENTID    "pir_client"
23  #define TOPIC       "PIR"
24  #define QOS         0
25  #define TIMEOUT     10000L
26
27  char* ADDRESS;
28
29  // RapidJson variables
30  using namespace rapidjson;
31  using namespace std::chrono;
32
```

```
33  int publish_message(std::string str_message, const char *topic, MQTTClient client){
34      // Initializing components for MQTT publisher
35      MQTTClient_message pubmsg = MQTTClient_message_initializer;
36      MQTTClient_deliveryToken token;
37
38      // Updating values of pubmsg object
39      char *message = new char[str_message.length() + 1];
40      strcpy(message, str_message.c_str());
41      pubmsg.payload = message;
42      pubmsg.payloadlen = (int)std::strlen(message);
43      pubmsg.qos = QOS;
44      pubmsg.retained = 0;
45
46      MQTTClient_publishMessage(client, topic, &pubmsg, &token); // Publish the message
47      int rc = MQTTClient_waitForCompletion(client, token, TIMEOUT);
48      return rc;
49  }
50
51  std::string json_to_string(const rapidjson::Document& doc){
52      //Serialize JSON to string for the message
53      rapidjson::StringBuffer string_buffer;
54      string_buffer.Clear();
55      rapidjson::PrettyWriter<rapidjson::StringBuffer> writer(string_buffer);
56      doc.Accept(writer);
57      return std::string(string_buffer.GetString());
58  }
59
60  int main(int argc, char* argv[])
61  {
62      auto start = high_resolution_clock::now(); // Starting timer
63
64      std::string input = argv[1]; // IP address as command line argument to avoid hard coding
65      input.append(":1883"); // Append MQTT port
66      char char_input[input.length() + 1];
67      strcpy(char_input, input.c_str());
68      ADDRESS = char_input;
69
```

```
70      MQTTClient client;
71      MQTTClient_connectOptions conn_opts = MQTTClient_connectOptions_initializer;
72      int rc;
73
74      MQTTClient_create(&client, ADDRESS, CLIENTID, MQTTCLIENT_PERSISTENCE_NONE, NULL);
75      conn_opts.keepAliveInterval = 20;
76      conn_opts.cleansession = 1;
77
78      if ((rc = MQTTClient_connect(client, &conn_opts)) != MQTTCLIENT_SUCCESS)
79      {
80          printf("Failed to connect, return code %d\n", rc);
81          exit(EXIT_FAILURE);
82      } else{
83          printf("Connected. Result code %d\n", rc);
84      }
85
86      wiringPiSetup(); // Required for wiringPi
87      pinMode(PIN, INPUT);
88      bool motion = false;
89      int count = 0;
90      int numIterations = 100000;
91      while(count <= numIterations) {
92          if(count == numIterations){
93              rapidjson::Document document_done;
94              document_done.SetObject();
95              rapidjson::Document::AllocatorType& allocator1 = document_done.GetAllocator();
96              document_done.AddMember("Done", true, allocator1);
97              std::string pub_message_done = json_to_string(document_done);
98              rc = publish_message(pub_message_done, TOPIC, client);
99          }
100         else {
101             motion = digitalRead(PIN);
102             //Create JSON DOM document object for humidity
103             rapidjson::Document document_pir;
104             document_pir.SetObject();
105             rapidjson::Document::AllocatorType &allocator2 = document_pir.GetAllocator();
106             document_pir.AddMember("PIR", motion, allocator2);

107             try {
108                 std::string pub_message_pir = json_to_string(document_pir);
109                 rc = publish_message(pub_message_pir, TOPIC, client);
110             } catch (const std::exception &exc) {
111                 // catch anything thrown within try block that derives from std::exception
112                 std::cerr << exc.what();
113             }
114         }
115         count = count + 1;
116     }
117
118     // End of loop. Stop MQTT and calculate runtime
119     MQTTClient_disconnect(client, 10000);
120     MQTTClient_destroy(&client);
121     auto end = high_resolution_clock::now();
122     std::chrono::duration<double> timer = end-start;
123     std::ofstream outfile;
124     outfile.open("piResultsCppLong.txt", std::ios_base::app); // append to the results text file
125     outfile << "PIR publisher runtime = " << timer.count() << "\n";
126     std::cout << "PIR runtime = " << timer.count() << "\n";
127     return rc;
128 }
```

C++ Code – LED

```
1    extern "C" {
2        #include <wiringPi.h>
3        #include <stdio.h>
4        #include <stdlib.h>
5        #include <string.h>
6        #include "MQTTClient.h"
7    }
8    #include <csignal>
9    #include <iostream>
10   #include "include/rapidjson/document.h"
11   #include <chrono>
12   #include <fstream>
13
14   //using namespace std;
15   using namespace rapidjson;
16   using namespace std::chrono;
17
18   #define CLIENTID    "ledSubscriber"
19   #define TOPIC       "LED"
20   #define QOS         0
21
22   int pin;
23   volatile MQTTClient_deliveryToken deliveredtoken;
24   bool led_status;
25   std::string session_status;
26   char* ADDRESS;
27   int num_messages = 0;
28   auto start = high_resolution_clock::now(); // initialize start
29
30   // Required callback
31   void delivered(void *context, MQTTClient_deliveryToken dt)
32   {
33       printf("Message with token value %d delivery confirmed\n", dt);
34       deliveredtoken = dt;
35   }
36
37   // Callback function for when an MQTT message arrives from the broker
38   int msgarrvd(void *context, char *topicName, int topicLen, MQTTClient_message *message)
39   {
40       num_messages = num_messages + 1;
```

```
41        if(num_messages == 1){
42            start = high_resolution_clock::now(); // Starting timer
43        }
44        int i;
45        char* payloadptr; //payload
46
47        payloadptr = (char*)message->payload; //payload converted to char*
48        int len = strlen(payloadptr);
49        if(payloadptr[len-2] == '}'){ // Fix for a bug in RapidJson
50            payloadptr[len-1] = '\0';
51        }
52
53        rapidjson::Document document;
54        document.Parse(payloadptr); // Parse string to JSON
55        if(document.HasMember("Done")){ // Done message is received from publisher when communication ends. This triggers the
56            MQTTClient_freeMessage(&message);
57            MQTTClient_free(topicName);
58            session_status = "Done";
59            auto end = high_resolution_clock::now();
60            std::chrono::duration<double> timer = end-start;
61            std::cout << "LED subscriber runtime = " << timer.count() << "\n";
62            std::ofstream outfile;
63            outfile.open("piResultsCppLong.txt", std::ios_base::app); // append to the results text file
64            outfile << "LED subscriber runtime = " << timer.count() << "\n";
65            return 0;
66        } else{
67            if(document.HasMember("LED_1")) { // If the message is about the LED status, the LED is switch accordingly
68                led_status = (bool) document["LED_1"].GetBool();
69                pin = document["GPIO"].GetInt();
70                pinMode(pin, OUTPUT);
71                digitalWrite(pin, led_status);
72            }
73            MQTTClient_freeMessage(&message);
74            MQTTClient_free(topicName);
75            return 1;
76        }
77    }
78
```

```
79  // Required callback for lost connection
80  void connlost(void *context, char *cause)
81  {
82      printf("\nConnection lost\n");
83      printf("     cause: %s\n", cause);
84  }
85
86  int main(int argc, char *argv[]){
87      std::string input = argv[1]; // IP address as command line argument to avoid hard coding
88      input.append(":1883"); // Append MQTT port
89      char char_input[input.length() + 1];
90      strcpy(char_input, input.c_str());
91      ADDRESS = char_input;
92
93      wiringPiSetupGpio();
94
95      MQTTClient client;
96      MQTTClient_connectOptions conn_opts = MQTTClient_connectOptions_initializer;
97      int rc;
98      int ch;
99
100     MQTTClient_create(&client, ADDRESS, CLIENTID, MQTTCLIENT_PERSISTENCE_NONE, NULL);
101     conn_opts.keepAliveInterval = 20;
102     conn_opts.cleansession = 1;
103
104     MQTTClient_setCallbacks(client, NULL, connlost, msgarrvd, delivered);
105
106     if ((rc = MQTTClient_connect(client, &conn_opts)) != MQTTCLIENT_SUCCESS) //Unsuccessful connection
107     {
108         printf("Failed to connect, return code %d\n", rc);
109         exit(EXIT_FAILURE);
110     }
111     else{ // Successful connection
112         printf("Connected. Result code %d\n", rc);
113     }
114     MQTTClient_subscribe(client, TOPIC, QOS);
115
116     while(session_status != "Done"){ // Continue listening for messages until end of session
117     //Do nothing
118     }
```

```
119
120     MQTTClient_disconnect(client, 10000);
121     MQTTClient_destroy(&client);
122     digitalWrite(pin, 0);
123     return rc;
124 }
```

A.3 – Monolithic Code

Go Code

```
1   package main
2
3   // #cgo LDFLAGS: -lwiringPi
4   // #include <wiringPi.h>
5   // #include <stdio.h>
6   // #include <stdlib.h>
7   // #include <stdint.h>
8   // #include <string.h>
9   // #include <time.h>
10  // #include <unistd.h>
11  // #define MAX_TIMINGS  85
12  // #define DHT_PIN      7   /* GPIO-4 */
13  // int data[5] = { 0, 0, 0, 0, 0 };
14  // double read_dht_data()
15  // {
16  //  clock_t begin = clock();
17  //  wiringPiSetup();
18  //  uint8_t laststate   = HIGH;
19  //  uint8_t counter     = 0;
20  //  uint8_t j           = 0, i;
21  //  data[0] = data[1] = data[2] = data[3] = data[4] = 0;
22  //  /* pull pin down for 18 milliseconds */
23  //  pinMode( DHT_PIN, OUTPUT );
24  //  digitalWrite( DHT_PIN, LOW );
25  //  delay( 18 );
26  //  /* prepare to read the pin */
27  //  digitalWrite( DHT_PIN, HIGH);
28  //  delayMicroseconds( 40 );
29  //  pinMode( DHT_PIN, INPUT );
30  //  /* detect change and read data */
31  //  for ( i = 0; i < MAX_TIMINGS; i++ )
32  //  {
33  //      counter = 0;
34  //      while ( digitalRead( DHT_PIN ) == laststate )
35  //      {
36  //          counter++;
37  //          delayMicroseconds( 2 );
38  //          if ( counter == 255 )
39  //              break;
40  //      }
41  //      laststate = digitalRead( DHT_PIN );
42  //      if ( counter == 255 ){
43  //          break;
44  //      }
```

```
45  //      /* ignore first 3 transitions */
46  //      if ( (i >= 4) && (i % 2 == 0) )
47  //      {
48  //          /* shove each bit into the storage bytes */
49  //          data[j / 8] <<= 1;
50  //          if ( counter > 16 )
51  //              data[j / 8] |= 1;
52  //          j++;
53  //      }
54  //  }
55  //  /*
56  //   * check we read 40 bits (8bit x 5 ) + verify checksum in the last byte
57  //   * print it out if data is good
58  //   */
59  //  if ( (j >= 40) &&
60  //       (data[4] == ( (data[0] + data[1] + data[2] + data[3]) & 0xFF) ) )
61  //  {
62  //      FILE *f = fopen("reading.txt", "w");
63  //      if (f == NULL)
64  //      {
65  //          printf("Error opening file!\n");
66  //          exit(1);
67  //      }
68  //      fprintf(f, "%d,%d,%d,%d,%d", data[0], data[1], data[2], data[3], data[4]);
69  //      fclose(f);
70  //      clock_t end = clock();
71  //      double time_spent = (double)(end - begin) / CLOCKS_PER_SEC;
72  //      return time_spent;
73  //  } else  {
74  //      FILE *f = fopen("reading.txt", "w");
75  //      if (f == NULL)
76  //      {
77  //          printf("Error opening file!\n");
78  //          exit(1);
79  //      }
80  //      fprintf(f, "%d,%d,%d,%d,%d", data[0], data[1], data[2], data[3], data[4]);
81  //      fclose(f);
82  //      return data[0];
83  //  }
84  // }
85  import "C"
```

```
86    import (
87        "encoding/json"
88        "fmt"
89        "io/ioutil"
90        "log"
91        "os"
92        "os/signal"
93        "strconv"
94        "strings"
95        "syscall"
96        "time"
97
98        mqtt "github.com/eclipse/paho.mqtt.golang"
99        rpio "github.com/stianeikeland/go-rpio"
100   )
101
102   var sessionStatusHT bool = true
103   var sessionStatusPir bool = true
104   var sessionStatusLed bool = true
105   var counter int = 0
106   var start = time.Now()
107   var startPIR = time.Now()
108   var startHT = time.Now()
109   var startLED = time.Now()
110   var dhtStart = time.Now()
111   var dhtEnd = time.Now()
112   var dhtDuration float64
113   var TOPIC_H string = "Humidity"
114   var TOPIC_T string = "Temperature"
115   var TOPIC_P string = "PIR"
116   var TOPIC_L string = "LED"
117   var ADDRESS string
118   var PORT = 1883
119   var temperatureReading float32 = 0
120   var humidityReading float32 = 0
121
122   type tempStruct struct {
123       Temp float32
124       Unit string
125   }
126
```

```go
127    type humStruct struct {
128        Humidity float32
129        Unit     string
130    }
131
132    type ledStruct struct {
133        LED_1 bool
134        GPIO  int
135    }
136
137    type pirStruct struct {
138        PIR bool
139    }
140
141    type reading interface {
142        structToJSON() []byte
143    }
144
145    func saveResultToFile(filename string, result string) {
146        file, errOpen := os.OpenFile(filename, os.O_APPEND|os.O_CREATE|os.O_WRONLY, 0644)
147        if errOpen != nil {
148            log.Fatal(errOpen)
149        }
150        byteSlice := []byte(result)
151        _, errWrite := file.Write(byteSlice)
152        if errWrite != nil {
153            log.Fatal(errWrite)
154        }
155    }
156
157    // Function to handle receipt of message
158    var messagePubHandler mqtt.MessageHandler = func(client mqtt.Client, msg mqtt.Message) {
159        counter++
160        if counter == 1 {
161            startLED = time.Now()
162        }
163        var led ledStruct
164        ledPin := rpio.Pin(12)
165        if strings.Contains(string(msg.Payload()), "Done") {
166            sessionStatusLed = false
167            ledPin.Output()
168            ledPin.Low()
```

```
169             endLED := time.Now()
170             durationLED := endLED.Sub(startLED).Seconds()
171             resultString := fmt.Sprint("LED subsriber runtime = ", durationLED, "\n")
172             saveResultToFile("piResultsGoMonoLong.txt", resultString)
173             fmt.Println("LED subsriber runtime = ", durationLED)
174         } else {
175             json.Unmarshal([]byte(msg.Payload()), &led)
176             ledPin = rpio.Pin(led.GPIO)
177             ledPin.Output()
178             if led.LED_1 {
179                 ledPin.High()
180             } else {
181                 ledPin.Low()
182             }
183         }
184     }
185 
186     //Function to publish a message
187     func publish(client mqtt.Client, sensor string) {
188         if !sessionStatusHT && (sensor == "dht") {
189             doneString := "{\"Done\": \"True\"}"
190             client.Publish(TOPIC_T, 0, false, doneString)
191             client.Publish(TOPIC_H, 0, false, doneString)
192             return
193         } else if !sessionStatusPir && (sensor == "pir") {
194             doneString := "{\"Done\": \"True\"}"
195             client.Publish(TOPIC_P, 0, false, doneString)
196             return
197         } else if sensor == "dht" {
198             dhtEnd = time.Now()
199             dhtDuration = dhtEnd.Sub(dhtStart).Seconds()
200             if (temperatureReading == 0 && humidityReading == 0) || dhtDuration > 1 {
201                 C.read_dht_data()
202                 dhtStart = time.Now()
203                 byteSlice, readErr := ioutil.ReadFile("reading.txt")
204                 if readErr != nil {
205                     log.Fatal(readErr)
206                 }
207                 mySlice := byteSliceToIntSlice(byteSlice)
208                 if mySlice[0] != 0 && mySlice[2] != 0 {
209                     temperatureReading = float32(mySlice[2] + (mySlice[3] / 10))
210                     humidityReading = float32(mySlice[0] + (mySlice[1] / 10))
211                 }
```

```
212            }
213            currentTemperature := tempStruct{
214                Temp: temperatureReading,
215                Unit: "C",
216            }
217            currentHumidity := humStruct{
218                Humidity: humidityReading,
219                Unit:     "%",
220            }
221            jsonTemperature := currentTemperature.structToJSON()
222            jsonHumidity := currentHumidity.structToJSON()
223            client.Publish(TOPIC_T, 0, false, string(jsonTemperature))
224            client.Publish(TOPIC_H, 0, false, string(jsonHumidity))
225            return
226        } else if sensor == "pir" {
227            pirPin := rpio.Pin(17)
228            pirPin.Input()
229            readValue := pirPin.Read()
230            var pirReading bool
231            if int(readValue) == 1 {
232                pirReading = true
233            } else {
234                pirReading = false
235            }
236            currentPIR := pirStruct{
237                PIR: pirReading,
238            }
239            jsonPIR := getJSON(currentPIR)
240            client.Publish(TOPIC_P, 0, false, string(jsonPIR))
241            return
242        }
243    }
244
245    func getJSON(r reading) []byte {
246        return r.structToJSON()
247    }
248
249    func (ts tempStruct) structToJSON() []byte {
250        jsonReading, jsonErr := json.Marshal(ts)
251        if jsonErr != nil {
252            log.Fatal(jsonErr)
253        }
254        return jsonReading
```

```go
255    }
256
257    func (ts humStruct) structToJSON() []byte {
258        jsonReading, jsonErr := json.Marshal(ts)
259        if jsonErr != nil {
260            log.Fatal(jsonErr)
261        }
262        return jsonReading
263    }
264
265    func (ps pirStruct) structToJSON() []byte {
266        jsonReading, jsonErr := json.Marshal(ps)
267        if jsonErr != nil {
268            log.Fatal(jsonErr)
269        }
270        return jsonReading
271    }
272
273    func byteSliceToIntSlice(bs []byte) []int {
274        strings := strings.Split(string(bs), ",")
275        result := make([]int, len(strings))
276        for i, s := range strings {
277            if len(s) == 0 {
278                continue
279            }
280            n, convErr := strconv.Atoi(s)
281            if convErr != nil {
282                log.Fatal(convErr)
283            }
284            result[i] = n
285        }
286        return result
287    }
288
289    var connectHandler mqtt.OnConnectHandler = func(client mqtt.Client) {
290        fmt.Println("Connected")
291    }
292
```

```
293 var connectLostHandler mqtt.ConnectionLostHandler = func(client mqtt.Client, err error) {
294     fmt.Printf("Connection lost: %v", err)
295 }
296
297 func main() {
298     // Retrieve IP address
299     if len(os.Args) <= 1 {
300         fmt.Println("IP address must be provided as a command line argument")
301         os.Exit(1)
302     }
303     ADDRESS = os.Args[1]
304     fmt.Println(ADDRESS)
305
306     // Check that RPIO opened correctly
307     if err := rpio.Open(); err != nil {
308         fmt.Println(err)
309         os.Exit(1)
310     }
311
312     // End program with ctrl-C
313     c := make(chan os.Signal, 1)
314     signal.Notify(c, os.Interrupt, syscall.SIGTERM)
315     go func() {
316         <-c
317         os.Exit(0)
318     }()
319
320     // Creat MQTT client
321     opts := mqtt.NewClientOptions()
322     opts.AddBroker(fmt.Sprintf("tcp://%s:%d", ADDRESS, PORT))
323     opts.SetClientID("go_mqtt_client_pir")
324     opts.SetDefaultPublishHandler(messagePubHandler)
325     opts.OnConnect = connectHandler
326     opts.OnConnectionLost = connectLostHandler
327     client := mqtt.NewClient(opts)
328     if token := client.Connect(); token.Wait() && token.Error() != nil {
329         panic(token.Error())
330     }
331
332     // Subscribe to topic
333     token := client.Subscribe(TOPIC_L, 1, nil)
334     token.Wait()
```

```
335
336      // Publish to PIR topic
337      numIterations := 100000
338      for i := 0; i < numIterations; i++ {
339          if i == numIterations-1 {
340              sessionStatusPir = false
341          }
342          publish(client, "pir")
343      }
344
345      endPIR := time.Now()
346      durationPIR := endPIR.Sub(startPIR).Seconds()
347      resultString := fmt.Sprint("PIR runtime = ", durationPIR, "\n")
348      saveResultToFile("piResultsGoMonoLong.txt", resultString)
349      fmt.Println("PIR runtime = ", durationPIR)
350
351      // Publish to dht topic
352      for i := 0; i < numIterations; i++ {
353          if i == numIterations-1 {
354              sessionStatusHT = false
355          }
356          publish(client, "dht")
357      }
358
359      endHT := time.Now()
360      durationHT := endHT.Sub(startHT).Seconds()
361      resultString = fmt.Sprint("Humidity and temperature runtime = ", durationHT, "\n")
362      saveResultToFile("piResultsGoMonoLong.txt", resultString)
363      fmt.Println("Humidity and temperature runtime = ", durationHT)
364
365      // Stay in loop to receive message
366      for sessionStatusLed { //Do nothing
367      }
368
369      // Disconnect
370      client.Disconnect(100)
371
372      end := time.Now()
373      duration := end.Sub(start).Seconds()
374      resultString = fmt.Sprint("Overall runtime = ", duration, "\n")
375      saveResultToFile("piResultsGoMonoLong.txt", resultString)
376      fmt.Println("Overall runtime = ", duration)
377  }
```

Python Code

```python
import paho.mqtt.client as mqtt
import paho.mqtt.publish as publish
import json
import RPi.GPIO as GPIO
import time
import sys
import adafruit_dht
import board

start_total = time.time()

# ------ Variables for all ------ #
MQTT_SERVER = sys.argv[1]
GPIO.setmode(GPIO.BCM)
GPIO.setwarnings(False)

# ------ LED code------ #
start_led = time.time()

MQTT_PATH_LED = "LED"
num_led_messages = 0
PIN_LED = 0  # Will be received in MQTT message

def on_connect(client, userdata, flags, rc):
    print("Connected. Result code: "+ str(rc))
    client.subscribe(MQTT_PATH_LED)

# The on_message function runs once a message is received from the broker
def on_message(client, userdata, msg):
    global num_led_messages
    global start_led
    global PIN_LED
    num_led_messages += 1
    if num_led_messages == 1:
        start_led = time.time()
    received_json = json.loads(msg.payload) #convert the string to json object
    if "Done" in received_json:
        client.loop_stop()
        client.disconnect()
        end_LED = time.time()
        timer = end_LED - start_led
        with open("piResultsPythonMonoLong.txt", "a") as myfile:
            myfile.write("LED subscriber runtime = " + str(timer) + "\n")
```

```
44          print("LED subscriber runtime = " + str(timer) + "\n");
45          GPIO.output(PIN_LED, GPIO.LOW)
46
47      else:
48          led_1_status = received_json["LED_1"]
49          PIN_LED = received_json["GPIO"]
50          GPIO.setup(PIN_LED, GPIO.OUT)
51          if led_1_status:
52              GPIO.output(PIN_LED, GPIO.HIGH)
53          else:
54              GPIO.output(PIN_LED, GPIO.LOW)
55
56  client = mqtt.Client()
57  client.on_connect = on_connect
58  client.on_message = on_message
59  client.connect(MQTT_SERVER, 1883, 60)
60
61  # ------ PIR code ------#
62  start_PIR = time.time()
63  MQTT_PATH_PIR = "PIR"
64
65  # Initial the pir device, with data pin connected to 17:
66  GPIO.setup(17, GPIO.IN)  # Change setup
67  presence = False
68  count = 0
69  while count < 100000:
70      try:
71          presence = GPIO.input(17)
72          temp_json = {"PIR": presence}
73          publish.single(MQTT_PATH_PIR, json.dumps(temp_json), port=1883, hostname=MQTT_SERVER)
74      except RuntimeError as error:  # Errors happen fairly often, DHT's are hard to read, just keep going
75          print(error.args[0])
76      count += 1
77
78  publish.single(MQTT_PATH_PIR, json.dumps({"Done": True}), port=1883, hostname=MQTT_SERVER)
79  end_PIR = time.time()
80  print("PIR publisher runtime = " + str(end_PIR-start_PIR))
81  with open("piResultsPythonMonoLong.txt", "a") as myfile:
82      myfile.write("PIR publisher runtime = " + str(end_PIR-start_PIR) + "\n")
83
```

```
84  # ------ Humidity and Temperature code ------#
85  # Initial the dht device, with data pin connected to:
86  start_HT = time.time()
87  dhtDevice = adafruit_dht.DHT11(board.D4)
88  count = 0
89  while count < 100000:
90      try:
91          humidity = dhtDevice.humidity # Get current humidity from dht11
92          temperature = dhtDevice.temperature # Get current temperature from dht11
93          hum_json = {"Humidity": humidity, "Unit": "%"}
94          publish.single("Humidity", json.dumps(hum_json), hostname=MQTT_SERVER)
95          temp_json = {"Temp": temperature, "Unit": "C"}
96          publish.single("Temperature", json.dumps(temp_json), hostname=MQTT_SERVER)
97      except RuntimeError as error:  # Errors happen fairly often, DHT's are hard to read, just keep going
98          error.args[0]
99      count += 1
100
101 publish.single("Humidity", json.dumps({"Done": True}), port=1883, hostname=MQTT_SERVER)
102 publish.single("Temperature", json.dumps({"Done": True}), port=1883, hostname=MQTT_SERVER)
103 end_HT = time.time()
104 print("Humidity and temperature runtime = " + str(end_HT-start_HT))
105 with open("piResultsPythonMonoLong.txt", "a") as myfile:
106     myfile.write("Humidity and temperature publisher runtime = " + str(end_HT-start_HT) + "\n")
107
108 end_total = time.time()
109 with open("piResultsPythonMonoLong.txt", "a") as myfile:
110     myfile.write("Overall runtime = " + str(end_total-start_total) + "\n")
111
112 client.loop_forever()
```

C++ Code:

```
1   extern "C" {
2       #include <wiringPi.h>
3       #include <stdio.h>
4       #include <stdlib.h>
5       #include <string.h>
6       #include "MQTTClient.h"
7   }
8   #include <csignal>
9   #include <iostream>
10  #include "include/rapidjson/document.h"
11  #include "include/rapidjson/stringbuffer.h"
12  #include "include/rapidjson/prettywriter.h"
13  #include "include/rapidjson/writer.h"
14  #include <chrono>
15  #include <fstream>
16
17  //using namespace std;
18  using namespace rapidjson;
19  using namespace std::chrono;
20
21  // variables for all
22  volatile MQTTClient_deliveryToken deliveredtoken;
23  char* ADDRESS;
24
25  // LED variables
26  #define CLIENTID_LED    "ledSubscriber"
27  #define TOPIC_LED       "LED"
28  #define QOS              0
29
30  int pin_LED;
31  bool led_status;
32  std::string session_status;
33  int num_messages = 0;
34  auto start_led = high_resolution_clock::now(); // initialize start
35
36  // PIR variables
37  #define CLIENTID_PIR    "pir_client"
38  #define TOPIC_PIR       "PIR"
```

```
39  #define TIMEOUT     10000L
40  #define PIN_PIR 17
41
42  // Humidity and Temperature variables
43  #define CLIENTID_HT   "hum_temp_client"
44  #define TOPIC_T       "Temperature"
45  #define TOPIC_H       "Humidity"
46
47  #define MAXTIMINGS  85
48  #define DHTPIN      7
49
50
51  int dht11_dat[5] = { 0, 0, 0, 0, 0 }; //first 8bits is for humidity integral value, second 8bits for humidity decimal, third for tem
52
53  // ------ LED code ------ //
54  void delivered(void *context, MQTTClient_deliveryToken dt) // Required callback
55  {
56      printf("Message with token value %d delivery confirmed\n", dt);
57      deliveredtoken = dt;
58  }
59
60  // Callback function for when an MQTT message arrives from the broker
61  int msgarrvd(void *context, char *topicName, int topicLen, MQTTClient_message *message)
62  {
63      num_messages = num_messages + 1;
64      if(num_messages == 1){
65          start_led = high_resolution_clock::now(); // Starting timer
66      }
67      int i;
68      char* payloadptr; //payload
69      payloadptr = (char*)message->payload; //payload converted to char*
70      int len = strlen(payloadptr);
71      if(payloadptr[len-2] == '}'){ //Fix for Paho MQTT bug
72          payloadptr[len-1] = '\0';
73      } else if (len > 28){
74          payloadptr[len-8] = '\0';
75      }
76      rapidjson::Document document;
77      document.Parse(payloadptr); // Parse string to JSON
```

```
78      if(document.HasMember("Done")){ // Done message is received from publisher when communication ends. This triggers the end of the
79          MQTTClient_freeMessage(&message);
80          MQTTClient_free(topicName);
81          session_status = "Done";
82          auto end_led = high_resolution_clock::now();
83          std::chrono::duration<double> timer = end_led-start_led;
84          std::cout << "LED subscriber runtime = " << timer.count() << "\n";
85          std::ofstream outfile;
86          outfile.open("piResultsCppMonoLong.txt", std::ios_base::app); // append to the results text file
87          outfile << "LED subscriber runtime = " << timer.count() << "\n";
88          return 0;
89      } else{
90          if(document.HasMember("LED_1")) { // If the message is about the LED status, the LED is switch accordingly
91              led_status = (bool) document["LED_1"].GetBool();
92              pin_LED = document["GPIO"].GetInt();
93              pinMode(pin_LED, OUTPUT);
94              digitalWrite(pin_LED, led_status);
95          }
96          MQTTClient_freeMessage(&message);
97          MQTTClient_free(topicName);
98          return 1;
99      }
100 }
101
102 // Required callback for lost connection
103 void connlost(void *context, char *cause){
104     printf("\nConnection lost\n");
105     printf("     cause: %s\n", cause);
106 }
107
108 int publish_message(std::string str_message, const char *topic, MQTTClient client){
109     // Initializing components for MQTT publisher
110     MQTTClient_message pubmsg = MQTTClient_message_initializer;
111     MQTTClient_deliveryToken token;
112
113     // Updating values of pubmsg object
114     char *message = new char[str_message.length() + 1];
115     strcpy(message, str_message.c_str());
116     pubmsg.payload = message;
117     pubmsg.payloadlen = (int)std::strlen(message);
```

```
118     pubmsg.qos = QOS;
119     pubmsg.retained = 0;
120 
121     MQTTClient_publishMessage(client, topic, &pubmsg, &token); // Publish the message
122     int rc = MQTTClient_waitForCompletion(client, token, TIMEOUT);
123     return rc;
124 }
125 
126 std::string json_to_string(const rapidjson::Document& doc){
127     //Serialize JSON to string for the message
128     rapidjson::StringBuffer string_buffer;
129     string_buffer.Clear();
130     rapidjson::PrettyWriter<rapidjson::StringBuffer> writer(string_buffer);
131     doc.Accept(writer);
132     return std::string(string_buffer.GetString());
133 }
134 
135 // Reading of the dht11 is rather complex in C/C++. See this site that explains how readings are made: http://www.uugear.com/port
136 int* read_dht11_dat()
137 {
138     uint8_t laststate   = HIGH;
139     uint8_t counter     = 0;
140     uint8_t j       = 0, i;
141 
142     dht11_dat[0] = dht11_dat[1] = dht11_dat[2] = dht11_dat[3] = dht11_dat[4] = 0;
143 
144     // pull pin down for 18 milliseconds. This is called "Start Signal" and it is to ensure DHT11 has detected the signal from MCU
145     pinMode( DHTPIN, OUTPUT );
146     digitalWrite( DHTPIN, LOW );
147     delay( 18 );
148     // Then MCU will pull up DATA pin for 40us to wait for DHT11's response.
149     digitalWrite( DHTPIN, HIGH );
150     delayMicroseconds( 40 );
151     // Prepare to read the pin
152     pinMode( DHTPIN, INPUT );
153 
154     // Detect change and read data
155     for ( i = 0; i < MAXTIMINGS; i++ )
156     {
157         counter = 0;
158         while ( digitalRead( DHTPIN ) == laststate )
159         {
160             counter++;
161             delayMicroseconds( 1 );
162             if ( counter == 255 )
163             {
164                 break;
165             }
166         }
167         laststate = digitalRead( DHTPIN );
168 
169         if ( counter == 255 )
170             break;
171 
172         // Ignore first 3 transitions
173         if ( (i >= 4) && (i % 2 == 0) )
174         {
175             // Add each bit into the storage bytes
176             dht11_dat[j / 8] <<= 1;
177             if ( counter > 16 )
178                 dht11_dat[j / 8] |= 1;
179             j++;
180         }
181     }
182 
183     // Check that 40 bits (8bit x 5 ) were read + verify checksum in the last byte
184     if ( (j >= 40) && (dht11_dat[4] == ( (dht11_dat[0] + dht11_dat[1] + dht11_dat[2] + dht11_dat[3]) & 0xFF) ) )
185     {
186         return dht11_dat; // If all ok, return pointer to the data array
187     } else  {
188         dht11_dat[0] = -1;
189         return dht11_dat; //If there was an error, set first array element to -1 as flag to main function
190     }
191 }
192 
193 int main(int argc, char* argv[])
194 {
195     auto start = high_resolution_clock::now();
196 
197     std::string input = argv[1]; // IP address as command line argument to avoid hard coding
198     input.append(":1883"); // Append MQTT port
```

```
199     char char_input[input.length() + 1];
200     strcpy(char_input, input.c_str());
201     ADDRESS = char_input;
202     int rc;
203
204     // ------ LED code ------ //
205     wiringPiSetup();
206
207     MQTTClient_connectOptions conn_opts = MQTTClient_connectOptions_initializer;
208     MQTTClient client_led;
209     MQTTClient_create(&client_led, ADDRESS, CLIENTID_LED, MQTTCLIENT_PERSISTENCE_NONE, NULL);
210
211     MQTTClient_setCallbacks(client_led, NULL, connlost, msgarrvd, delivered);
212
213     if ((rc = MQTTClient_connect(client_led, &conn_opts)) != MQTTCLIENT_SUCCESS) //Unsuccessful connection
214     {
215         printf("Failed to connect, return code %d\n", rc);
216         exit(EXIT_FAILURE);
217     }
218     else{ // Successful connection
219         printf("Connected to led. Result code %d\n", rc);
220     }
221
222     MQTTClient_subscribe(client_led, TOPIC_LED, QOS);
223
224     // ------ PIR code ----- //
225     auto start_pir = high_resolution_clock::now(); // Starting timer
226
227     MQTTClient client;
228     MQTTClient_create(&client, ADDRESS, CLIENTID_PIR, MQTTCLIENT_PERSISTENCE_NONE, NULL);
229     conn_opts.keepAliveInterval = 20;
230     conn_opts.cleansession = 1;
231
232     if ((rc = MQTTClient_connect(client, &conn_opts)) != MQTTCLIENT_SUCCESS){
233         printf("Failed to connect, return code %d\n", rc);
234         exit(EXIT_FAILURE);
235     } else{
236         printf("Connected to PIR. Result code %d\n", rc);
237     }
238
```

```
240     bool motion = false;
241     int count = 0;
242     int num_iterations = 100000;
243     while(count <= num_iterations) {
244         if(count == num_iterations){
245             rapidjson::Document document_done;
246             document_done.SetObject();
247             rapidjson::Document::AllocatorType& allocator1 = document_done.GetAllocator();
248             document_done.AddMember("Done", true, allocator1);
249             std::string pub_message_done = json_to_string(document_done);
250             rc = publish_message(pub_message_done, TOPIC_PIR, client);
251         }
252         else {
253             motion = digitalRead(PIN_PIR);
254             //Create JSON DOM document object for humidity
255             rapidjson::Document document_pir;
256             document_pir.SetObject();
257             rapidjson::Document::AllocatorType &allocator2 = document_pir.GetAllocator();
258             document_pir.AddMember("PIR", motion, allocator2);
259             try {
260                 std::string pub_message_pir = json_to_string(document_pir);
261                 rc = publish_message(pub_message_pir, TOPIC_PIR, client);
262             } catch (const std::exception &exc) {
263                 // catch anything thrown within try block that derives from std::exception
264                 std::cerr << exc.what();
265             }
266         }
267         count = count + 1;
268     }
269 
270     // End of PIR loop. Calculate runtime
271     auto end_pir = high_resolution_clock::now();
272     std::chrono::duration<double> timer_pir = end_pir-start_pir;
273     std::ofstream outfile;
274     outfile.open("piResultsCppMonoLong.txt", std::ios_base::app); // append to the results text file
275     outfile << "PIR publisher runtime = " << timer_pir.count() << "\n";
276     std::cout << "PIR runtime = " << timer_pir.count() << "\n";
277 
278     // ------ Humidity temperature code ------ //
279     auto start_HT = high_resolution_clock::now(); // Starting timer
280 
```

```
281     double temperature = -1;
282     double humidity = -1;
283     count = 0;
284     auto dhtStart = high_resolution_clock::now();
285     auto dhtEnd = high_resolution_clock::now();
286     std::chrono::duration<double> dhtTimer;
287     while(count <= num_iterations) {
288         if(count == num_iterations){
289             rapidjson::Document document_done;
290             document_done.SetObject();
291             rapidjson::Document::AllocatorType& allocator1 = document_done.GetAllocator();
292             document_done.AddMember("Done", true, allocator1);
293             std::string pub_message_done = json_to_string(document_done);
294             rc = publish_message(pub_message_done, TOPIC_T, client);
295             rc = publish_message(pub_message_done, TOPIC_H, client);
296         }
297         else {
298             dhtEnd = high_resolution_clock::now();
299             dhtTimer = dhtEnd - dhtStart;
300             if((temperature == -1 && humidity == -1) || dhtTimer > (std::chrono::seconds(1))) { //need to get values from
301                 int *readings = read_dht11_dat();
302                 dhtStart = high_resolution_clock::now();
303                 int counter = 0;
304                 while (readings[0] == -1 && counter < 5) {
305                     readings = read_dht11_dat(); // Errors frequently occur when reading dht sensor. Keep reading until values are ret
306                     counter = counter + 1;
307                 }
308                 if (readings[0] != -1) {
309                     humidity = readings[0] + (readings[1] / 10);
310                     temperature = readings[2] + (readings[3] / 10);
311                 }
312                 else{
313                     humidity = 0;
314                     temperature = 0;
315                 }
316             }
317             //Create JSON DOM document object for humidity
318             rapidjson::Document document_humidity;
319             document_humidity.SetObject();
320             rapidjson::Document::AllocatorType &allocator2 = document_humidity.GetAllocator();
321             document_humidity.AddMember("Humidity", humidity, allocator2);
322             document_humidity.AddMember("Unit", "%", allocator2);
323
324             //Create JSON DOM document object for temperature
325             rapidjson::Document document_temperature;
326             document_temperature.SetObject();
327             rapidjson::Document::AllocatorType &allocator3 = document_temperature.GetAllocator();
328             document_temperature.AddMember("Temp", temperature, allocator3);
329             document_temperature.AddMember("Unit", "C", allocator3);
330             std::string pub_message_humidity = json_to_string(document_humidity);
331             rc = publish_message(pub_message_humidity, TOPIC_H, client);
332             std::string pub_message_temperature = json_to_string(document_temperature);
333             rc = publish_message(pub_message_temperature, TOPIC_T, client);
334         }
335         count = count + 1;
336     }
337
338     // End of loop. Calculate runtime
339     auto end_HT = high_resolution_clock::now();
340     std::chrono::duration<double> timer_HT = end_HT-start_HT;
341     std::ofstream outfile2;
342     outfile2.open("piResultsCppMonoLong.txt", std::ios_base::app); // append to the results text file
343     outfile2 << "Humidity and temperature publisher runtime = " << timer_HT.count() << "\n";
344     std::cout << "Humidity and temperature runtime = " << timer_HT.count() << "\n";
345
346     while(session_status != "Done"){ // Continue listening for messages until end of session
347         //Do nothing
348     }
```

```
347             //Do nothing
348         }
349 
350         MQTTClient_disconnect(client, 10000);
351         MQTTClient_destroy(&client);
352         MQTTClient_disconnect(client_led, 10000);
353         MQTTClient_destroy(&client_led);
354         digitalWrite(pin_LED, 0);
355 
356         auto end = high_resolution_clock::now();
357         std::chrono::duration<double> timer = end-start;
358 
359         std::ofstream outfile3;
360         outfile3.open("piResultsCppMonoLong.txt", std::ios_base::app); // append to the results text file
361         outfile3 << "Overall runtime = " << timer.count() << "\n";
362         std::cout << "Overall runtime = " << timer.count() << "\n";
363 
364         return rc;
365     }
```

Appendix B – Results

B.1 – Power Results

Table B.1: Table with full maximum instantaneous power results for microservices architecture

Go Power Consumption (Watt)	Python Power Consumption (Watt)	Cpp Power Consumption (Watt)
3,92	3,95	3,95
4,19	3,91	3,91
4,37	3,89	3,89
4,30	3,94	3,94
4,21	3,91	3,91
4,32	3,94	3,94
4,18	3,90	3,90
4,35	3,93	3,93
4,17	3,97	3,97
4,12	3,96	3,96

Table B.2: Table with full maximum instantaneous power results for monolithic architecture

Go Power Consumption (Watt)	Python Power Consumption (Watt)	Cpp Power Consumption (Watt)
3,68	3,19	3,42
3,58	3,18	3,35
3,68	3,19	3,48
3,60	3,21	3,38
3,66	3,21	3,35
3,71	3,24	3,45
3,71	3,19	3,42
3,62	3,19	3,42
3,65	3,20	3,64
3,61	3,20	3,37

Table B.3: Table with full total power results for microservices architecture

Go Power Consumption (Watt)	Python Power Consumption (Watt)	Cpp Power Consumption (Watt)
3,69	3,73	3,28
3,42	3,78	3,58
3,59	3,72	3,53
3,69	3,75	3,49
3,53	3,73	3,58
3,85	3,74	3,49
3,60	3,73	3,57
3,94	3,74	3,57
3,72	3,75	3,57
3,69	3,68	3,29

Table B.4: Table with full total power results for monolithic architecture

Go Power Consumption (Watt)	Python Power Consumption (Watt)	Cpp Power Consumption (Watt)
3,54	3,50	3,40
3,59	3,45	3,66
3,48	3,49	3,57
3,40	3,47	3,57
3,48	3,49	3,66
3,56	3,54	3,48
3,57	3,47	3,44
3,48	3,49	3,53
3,72	3,52	3,75
3,53	3,44	3,44

B.2 – Runtime Results

Table B.5: Table with full DHT runtime results for microservices architecture

Go Runtime	Python Runtime	Cpp Runtime
15,24	419,80	7,46
15,75	383,96	8,01
15,63	376,24	7,88
15,44	374,48	8,19
16,01	369,93	7,93
15,25	376,13	7,86
15,61	372,96	7,66
15,58	374,37	8,23
16,64	372,03	7,62
16,29	370,32	7,94

Table B.6: Table with full LED runtime results for microservices architecture

Go Runtime	Python Runtime	Cpp Runtime
113,27	134,75	98,06
114,44	115,40	87,76
102,26	109,02	83,71
125,65	138,85	85,79
115,17	106,05	86,95
109,76	102,81	86,65
122,57	112,80	83,91
102,36	108,61	82,49
88,95	113,40	87,24
82,22	103,84	77,70

Table B.7: Table with full PIR runtime results for microservices architecture

Go Runtime	Python Runtime	Cpp Runtime
5,94	178,96	3,38
7,76	192,46	3,24
6,96	179,37	3,61
7,80	179,21	3,47
8,02	177,61	4,19
7,74	179,69	3,72
7,39	177,42	3,26
8,02	180,53	3,41
8,68	177,38	3,77
8,05	178,57	3,84

Table B.8: Table with full overall runtime results for microservices architecture – concurrent

Go Runtime	Python Runtime	Cpp Runtime
119,10	382,20	105,80
123,30	377,40	92,10
106,90	382,70	88,30
129,20	379,90	89,10
119,10	382,40	91,90
113,80	380,50	88,80
126,80	378,20	87,40
116,30	380,80	87,30
117,90	380,30	87,30
119,00	383,30	88,70

Table B.9: Table with full overall runtime results for microservices architecture – cumulative

Go Runtime	Python Runtime	Cpp Runtime
134,45	733,51	108,89
137,95	691,83	99,00
124,85	664,64	95,21
148,89	692,53	97,45
139,19	653,60	99,07
132,75	658,63	98,24
145,57	663,18	94,83
125,96	663,52	94,14
114,27	662,81	98,63
106,56	652,73	89,48

Table B.10: Table with full DHT runtime results for monolithic architecture

Go Runtime	Python Runtime	Cpp Runtime
20,01	350,86	8,41
20,31	351,71	8,45
20,42	353,03	8,01
20,21	354,74	8,15
20,02	350,76	8,11
20,18	351,86	8,37
20,58	352,71	8,16
20,11	352,03	8,46
20,25	354,81	8,15
20,40	350,66	8,16

Table B.11: Table with full LED runtime results for monolithic architecture

Go Runtime	Python Runtime	Cpp Runtime
120,10	117,47	80,40
97,45	107,88	77,86
113,67	107,52	84,86
108,05	116,40	83,45
121,06	107,03	80,87
107,44	118,15	80,36
118,87	108,88	81,41
119,03	108,59	80,12
107,05	117,24	80,08
99,51	107,39	84,00

Table B.12: Table with full PIR runtime results for monolithic architecture

Go Runtime	Python Runtime	Cpp Runtime
6,21	154,45	4,36
6,30	153,13	4,44
6,31	152,89	3,99
6,26	154,71	3,74
6,20	152,14	4,42
6,20	153,42	3,96
6,49	154,13	3,78
6,11	152,13	3,78
6,25	154,71	4,35
6,32	152,91	3,95

Table B.13: Table with full overall runtime results for monolithic architecture

Go Runtime	Python Runtime	Cpp Runtime
121,02	623,46	81,56
98,43	629,71	78,88
115,31	614,10	85,75
108,99	628,81	84,13
121,86	610,34	82,54
108,08	623,46	81,18
120,09	628,62	82,14
121,03	616,11	80,83
107,14	627,99	81,07
99,49	612,12	84,91

B.3 – Memory Results

Table B.14: Table with full maximum RSS results for microservices architecture

Go Maximum Resident Set Size (kB)	Python Maximum Resident Set Size (kB)	Cpp Maximum Resident Set Size (kB)
98840	40221	21864
96712	40092	21776
98460	40212	21828
98796	39648	21828
98944	39960	21816
99232	40088	21776
97044	40236	21796
91980	39653	21892
97176	40240	21868
99188	39959	21796

Table B.15: Table with full maximum RSS results for monolithic architecture

Go Maximum Resident Set Size (kB)	Python Maximum Resident Set Size (kB)	Cpp Maximum Resident Set Size (kB)
88528	13588	15176
89544	13696	15136
25784	13668	15140
90176	13648	15148
88356	13656	15212
89952	13588	15184
89068	13699	15152
89848	13666	15160
89542	13646	15112
90177	13659	15084

Table B.16: Table with full total RSS results for microservices architecture

Go Total Resident Set Size (kB)	Python Total Resident Set Size (kB)	Cpp Total Resident Set Size (kB)
190944	584199	185172
190938	584184	185178
190941	584193	185173
190939	584192	185169
190940	584195	185170
190942	584195	185171
190940	584188	185170
190934	584190	185168
190940	584187	185169
190945	584184	185171

Table B.17: Table with full total RSS results for monolithic architecture

Go Total Resident Set Size (kB)	Python Total Resident Set Size (kB)	Cpp Total Resident Set Size (kB)
194658	1325931	180301
194655	1325930	180301
194661	1325934	180305
194663	1325929	180291
194666	1325916	180299
194660	1325929	180331
194657	1325933	180307
194653	1325931	180302
194661	1325931	180300
194662	1325932	180294

Table B.18: Table with full CPU results for microservices architecture

Go CPU (%)	Python CPU (%)	Cpp CPU (%)
99,67	64,00	93,67
100,67	64,33	95,33
98,67	65,67	94,33
98,00	65,67	94,00
97,67	65,33	90,33
97,00	65,33	94,00
100,67	65,33	97,00
96,33	65,67	93,00
95,67	66,33	91,67
96,67	65,33	97,00

Table B.19: Table with full CPU results for monolithic architecture

Go CPU (%)	Python CPU (%)	Cpp CPU (%)
107,00	83,00	99,00
107,00	82,00	98,00
144,00	84,00	99,00
107,00	83,00	99,00
106,00	84,00	98,00
106,00	84,00	99,00
108,00	82,00	99,00
107,00	83,00	99,00
106,00	82,00	98,00
106,00	84,00	99,00