Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Petter Njerve Rostrup

A Distributed-to-Centralized Architectural Model for Smart City Applications and Services through Container Orchestration

Master's thesis in Master of Science in Informatics
Supervisor: Sobah Abbas Petersen
June 2019

NTNU
Norwegian University of Science and Technology

Petter Njerve Rostrup

A Distributed-to-Centralized Architectural Model for Smart City Applications and Services through Container Orchestration

Master's thesis in Master of Science in Informatics
Supervisor: Sobah Abbas Petersen
June 2019

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

NTNU
Norwegian University of
Science and Technology

Abstract

The usage of IoT devices is rising exponentially and this leads to a massive increase in data generated from such devices. Currently, most of the generated data is sent directly to cloud platforms for processing. This creates physically long distances between the devices and their platform, causing an increased strain on network infrastructures and setting higher baseline requirements for them. New smart city ICT-systems should therefore be designed with this in mind to lessen the total unnecessary usage of long distance network communications. This can be done by utilizing computation between the devices and the cloud platforms through a distributed-to-centralized form of computing known as fog-to-cloud computing. This thesis investigates how this computational paradigm can be adopted into future ICT-systems for a smart city context while facilitating data management from distributed sources and up to cloud computing solutions. It proposes an architectural model based on research done into the state of fog-, cloud- and fog-to-cloud computing. This research is then made into requirements for the design of the model to act as a scientific basis. The model utilizes concepts like containerization and container orchestration and argues for the suitability of those for a smart city scenario. A useful feature of this is to enable usage of a wider array of IoT-devices as resources for the ICT-system operation and runtime. Through implementing and evaluating the architectural model, the thesis then aims to start a discussion on standardization of architectures and best practices for use in smart cities, neighbourhoods and buildings.

Keywords: Cloud Computing, Fog Computing, Edge Computing, Smart City, Smart Building, Containerization, Container Orchestration, Swarm Computing, Cluster Computing, IoT

Preface

This thesis concludes my studies of informatics at the Norwegian University of Science and Technology (NTNU). The project was conducted under the Department of Computer Science (IDI) and in collaboration with the FME ZEN research centre.

I would like to thank my supervisors on the thesis, Amir Sinaeepourfard and Sobah Abbas Petersen, for their advise and support throughout the thesis. Special thanks to Amir for being with me throughout the process providing advice, discussion and valuable feedback. I would also like to thank Siren Grasmo, my girlfriend, for the support and motivation she has provided throughout the thesis.

Sammendrag

Bruken av IoT-enheter øker eksponentielt, og dette fører til en enorm økning i data generert fra slike enheter. For øyeblikket sendes det meste av de genererte dataene direkte til skyplattformer for behandling. Dette skaper fysisk lange avstander mellom enhetene og plattformene deres, noe som medfører økt belastning på nettverksinfrastrukturen og fastsetter høyere basiskrav for dem. Nye "smart city" IKT-systemer bør derfor utformes med tanke på dette for å redusere den totale unødvendige bruken av langdistanse nettverkskommunikasjon. Dette kan gjøres via beregninger og behandligner mellom enhetene og skyplattformene gjennom en "distributed-to-centralized" form for databehandling, kjent som "fog-to-cloud"-databehandling. Denne oppgaven undersøker hvordan dette beregningsparadigmet (computational paradigm) kan bli vedtatt i fremtidige IKT-systemer for en "smart city"-kontekst, samtidig som dataforvaltning tilrettelegges fra distribuerte kilder og opp til sky-løsninger. Den foreslår deretter en arkitektonisk modell (architectural model) basert på forskning gjort på standarder og status i "fog"-, "cloud"- og "fog-to-cloud"-databehandling. Denne forskningen er videre utformet som krav til design av modellen. Modellen utnytter konsepter som "containerization" og "container orchestration" for deretter å argumentere hvor egnede de er for et "smart city"-scenario. En nyttig funksjon av dette er å muliggjøre bruk av et bredere utvalg av IoT-enheter som ressurser for driften av IKT-systemet. Gjennom implementering og evaluering av den arkitektoniske modellen, forsøker oppgaven å starte en diskusjon om standardisering av arkitekturer og beste praksis for bruk i smarte byer, nabolag og bygninger.

Abbreviations

FME = Centres for Environment-friendly Energy Research
ZEN = Zero Emission Neighbourhood
ZEB = Zero Emission Building
IoT = Internet of Things
NTNU = Norwegian University of Science and Technology
ICT = Information and Communication Technology
F2C = Fog to Cloud
D2C = Distributed-to-Centralized
QA = Quality Attribute
GDPR = General Data Protection Regulation
TPWM = Tiered-Priority Workload Management
PaaS = Platform-as-a-Service (Cloud Computing)
SaaS = Service-as-a-Service (Cloud Computing)
IaaS = Infrastructure-as-a-Service (Cloud Computing)
API = Application Programming Interface
API+ component = API, Request Handler & Data Aggregator component

List of Figures

Contents

1 Introduction

1.1 Introduction

The usage of IoT devices are increasing rapidly and are projected to reach 10 Billion connected IoT devices by 2020 [1]. Many to most of these devices generate data such as sensor data, usage statistics etc. This massive and growing increase in data generation has a proportional demand for aggregation and processing of this data in order for it to be useful in analytics, services and applications. Computational paradigms such as cloud computing is a popular approach to data aggregation and analytics [2] and this poses new problems and challenges to be faced. Studies done on the data generation of a smart city points to large amounts of data being generated ,without accounting for the data being generated from other sources (like mobile devices), with the need rising in the future [3]. The growing strains on networking infrastructure needed for a total reliance on a centralized computational paradigm need to be addressed. Some proposed strategies to mitigate and relieve these strains are pre-processing data, distributed data organization and fog computing [3]. Some cities like Barcelona have already implemented some smarter approaches to city management [4] while broader projects like +CityXchange [5] are becoming more popular. In the future this will only increase (as shown in figure 2.2) and there will be a corresponding need for network infrastructure and possibly to handle the increasing flow of data through new applications and services as well.

In an IoT setting, the fog is the layer that stretches from the outer edges of the network up to where it is eventually stored [6]. It helps to visualize fog computing as a form of low-lying cloud closer to the ground, thereby its namesake, when considering its relation to cloud computing. Fog computing is, generally speaking, the concept of performing computational tasks out towards this outermost layer (edge) of the network. This is usually in the form of data pre-processing, aggregation, real-time data storage, providing interfaces to edge IoT devices and services. The purpose of fog computing is generally not to replace the cloud but to extend it by acting as an extension of the cloud itself by providing computational capacity closer to the edges of the network. By performing tasks like data analytics closer to the edges of the network, it will reduce communications overhead [7] while also reducing overall strain on the network. Furthermore, functionality like real-time data access and pre-processed/aggregated data will lessen the need for these tasks in the cloud-layer and make it easier to keep up with rising demands and development. Another useful benefit will be the possibility of having applications and services physically closer to the devices and data sources. This in turn will greatly reduce latency and would be very useful in time-sensitive applications like crisis prevention and

other time-sensitive operations.

In a smart city context, there is a need to properly manage the flow of the increasing amounts of data being generated that normally flows directly to cloud computing systems. One promising solution is to utilize both fog- and cloud-computing combined as fog-to-cloud-computing to manage the flow of the data from where it is generated, up through fog intermediaries and finally up to the cloud. This solution is presented through a paper published by the ZEN research centre named "Data Preservation through Fog-to-Cloud (F2C) Data Management in Smart Cities" [8]. It focuses primarily on the management of data and persistence through the management of the data flow, and not necessarily on usage in applications and services. This thesis will therefore focus on how the fundamental principles of Fog-to-Cloud Data Management can be applied to new applications and services in a smart city context to help alleviate the aforementioned problems and challenges.

1.2 Thesis Structure

The structure of the thesis will be as follows:

1. Introduction - Introducing the problem the thesis will tackle and the motivation behind it.
2. Background - Background research done to better understand the problem and to provide a scientific basis for the main part of the thesis.
3. Methodology - Research method applied for the various parts of the thesis. Describes the general process of the thesis and provides insight into methodology applied for the different parts of the research and other work done.
4. Results - The main part of the thesis where the proposed solution and all related parts are outlined. Divided into four parts which follows the sequence of development: 1. Requirements to the design of the model. 2. The architectural model (proposed solution). 3. The implementation of the architectural model. 4. The evaluation metrics to evaluate the architectural model and the implementation.
5. Discussion - Evaluation of the architectural model and implementation using the requirements and quality metrics. After that, there is a discussion on the architectural design choices, scientific contributions and reflections on the thesis as a whole.
6. Conclusion - Conclusion of the thesis, answering the research questions and proposals for future work.

2 Background

The thesis proposes an architectural model that aims to start discussions of standardization and best practices for applications and services for use in smart cities, neighbourhoods and buildings. It follows principles of distributed-to-centralized data management as a core part of the model and designs the rest based on research done on the state of fog-, cloud- and fog-to-cloud-computing. This chapter is meant to provide insight into the context of the thesis as well as showing the background research done. It has an overview of the different network layers of centralization, architectures assessed through the literature review, other essential papers and an overview of the technologies, platforms and frameworks from the model and implementation.

2.1 Smart Cities

There are much discussion and many different definitions on what exactly constitutes a "smart city". This discussion varies from programmers and technologists definition of applying tech to city government and services [9], to a wider definition from other disciplines defining it as innovating and imagining new better ways of providing these services [10]. The thing most of the definitions have in common is collecting data from various sources and utilizing that to provide better services and improve general governance in cities. This is a growing field of studies with several large government-funded initiatives, like EU smart cities and community lighthouse projects [10], are progressing steadily in pioneering smarter cities. There are also several projects focusing on smart cities on a lower scale, like the ZEN centre (more info on this later on in the thesis) focusing on smart neighbourhoods.

The smart city impacts many facets of the governance of the city and the daily lives of its citizens [11]. These areas are as shown in figure 2.1: smart environment, smart mobility, smart economy, smart governance, smart people and smart living. All of these areas together form the entire context of what a smart city is and can involve. By applying gathered data into decision making, service handling and application development, all of these areas can be affected in a positive manner. This is usually realized through smarter city services for the citizens, better utilization of available resources such as electricity through grid computing, e-democracies and other applications to involve the citizens in decision-making.

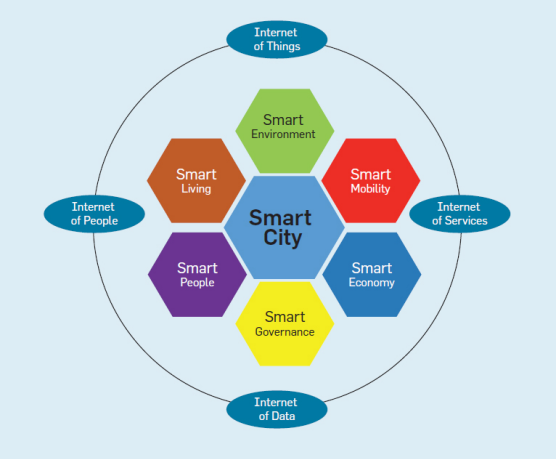


Figure 2.1: A model of the different areas of a smart city

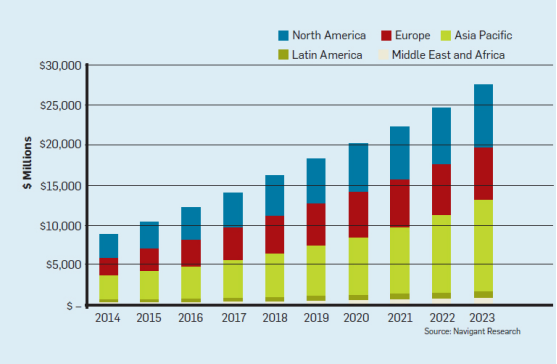


Figure 2.2: An overview of the monetary investments into smart city project by region

2.2 Network Centralization

Different computational paradigms often act on different levels of centralization with regards to networking. Some computational paradigms are designed around heavily centralized networking (cloud computing), while others are designed to function at the utter edges of the network (edge computing) with others being somewhere in between (fog computing). This section elaborates on the different layers of this centralization of networking and aims to provide insight into this field of research in the context of smart cities.

2.2.1 Identifying Network Layers

To identify the different possible network layers when considering degrees of centralization, the relevant parts of the network in a smart city context must be examined. This will involve looking at the highest and lowest forms of centralization. The most centralized form is then a heavily centralized form of computation, and the least centralized will be the smaller devices at the edges of the network. The most centralized layer would then be the cloud layer and the least centralized would then be the edge layer at the utter edges of the network. Between those two, there is a layer called the "fog layer" that bridges the gap of computation on the edges and centre. In addition, there is a "fog-to-cloud" layer that is a combined term for computation extending from the fog layer to the cloud layer or vice versa. Much of the literature use the term fog and edge computation interchangeably, but there are differences to the layers in addition to the overlapping [12]. To simplify the situation somewhat, not all forms of computational work at the edge of the network is necessarily considered a part of the fog while some forms of fog computing can extend or interact with the edges of the network. The layout of this Fog-to-Cloud environment (distributed-to-centralized) could therefore be considered to have “2.5 layers” where in some cases the utter edges of the network can be argued as a distinct layer on its own. Furthermore there are several definitions and description of what constitutes the fog computing layer and they all share the common though that the fog extends from the end of the cloud layer and to the edge of the network itself. The fact that the terms edge- and fog computing are often used interchangeably does complicate the manner.The description given by Aazam and Huh [13] encompasses this thinking well; “Fog computing refers to bringing networking resources near the underlying networks. It is a network between the underlying network(s) and the cloud(s). Fog computing extends the traditional cloud computing paradigm to the edge of the network, enabling the creation of refined and better applications or services. Fog is an edge computing and micro data center (MDC) paradigm for IoTs and wireless sensor networks (WSNs).” The thesis will then focus on the environment of the cloud layer, the fog layer and their combined definition of the fog-to-cloud layer. The thesis will therefore take basis in fog-to-cloud being the combined layer of both the cloud- and fog layer. Any form of centralization on the edges then becomes a part of the fog, while computation strictly confined to the edge without centralization is then a part of the edge layer (as shown in figure 2.3. This makes the fog layer and cloud layer the relevant network layers along with their combined fog-to-cloud layer.

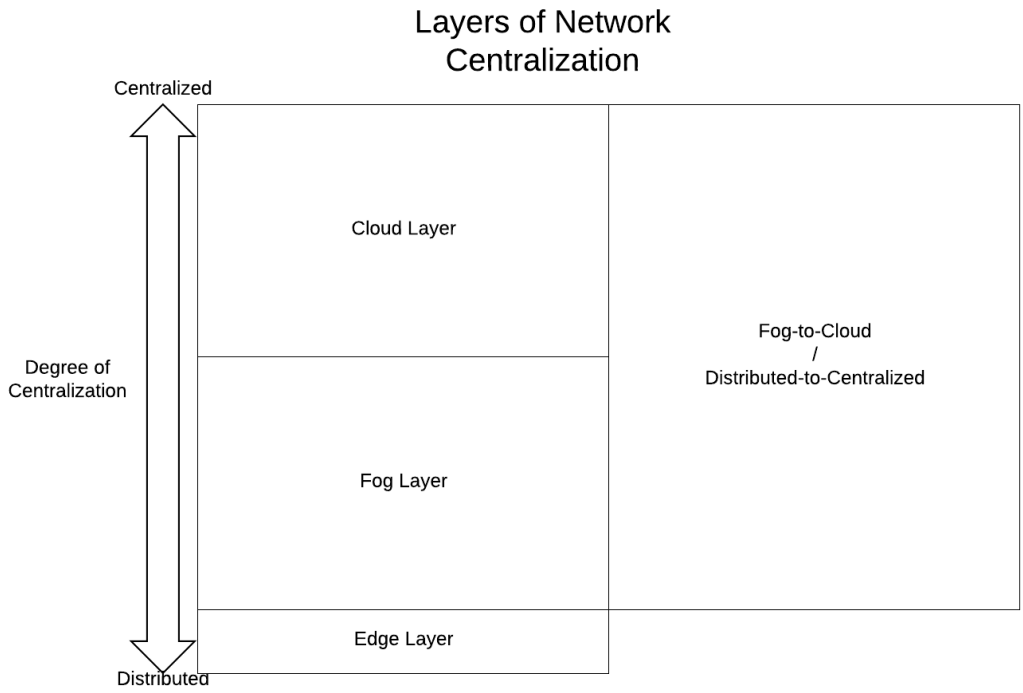


Figure 2.3: Illustration of the different network layers with regards to network centralization

2.2.2 Centralized Architecture

Network Layer

The Cloud Layer The "Cloud Layer" is a collection of everything relating to the act of providing or utilizing applications and/or services in the cloud. The cloud itself is a heavily centralized form of computation or data-management consisting of connected data-centres of varying sizes that together forms a centralized platform. Due to the complex and costly nature of building cloud platforms, the norm is to utilize services provided by bigger companies for cloud computing tasks instead of building one. As of 2019, cloud service providers like Amazon Web Services (AWS), Google Cloud Platform (GCP) and Microsoft Azure dominates most of this trade [14]. These providers sell access to their cloud computing services in the form of Platform as a Service (PaaS), Infrastructure as a Service (IaaS) and/or Services as a Service (SaaS). Cloud computing has become exceedingly popular and is presently heavily embedded both in commercial and private usage. In the future this paradigm will undoubtedly increase in popularity and usage, further increasing the need for expansion of the existing services and possibly new actors in the market as well. As an example, AWS (amazons cloud service) makes

up a small portion of the e-commerce giants business, but is responsible for most of the delivered operating income [15].

Computational Layer

Cloud Computing Michael Armbrust et. al defines cloud computing like this: "Cloud computing refers to both the applications delivered as services over the Internet and the hardware and systems software in the data centers that provide those services." [16]. It is a broad definition that covers the entirety of the scope for cloud services and applications available. This broad definition is fitting as the field of cloud computing is a massive enterprise and is seen as a massively growing market with large potentials for profit. Cloud computing is mainly used in either transaction processing (i.e., read and update workloads) or OLAP (OnLine Analytical Processing) workloads. These forms of computation are usually made available through scalable pay-as-you-go virtual machines. This makes virtual machine hosting systems one of the most popular forms of establishing cloud computing platforms.

Smart city services and applications, like many other current services and applications, utilize the cloud to both host their services and perform most of computational tasks there as well. This makes cloud computing a relevant computational paradigm to consider when either designing an independent smart city service or when interfacing with one is necessary. It is a considerable undertaking to create new cloud computing platforms, and it would therefore generally be preferable to use an existing platform instead of establishing your own. This again has lead to little publicly available research and material on how one would go about establishing their own cloud platform. The largest providers available tend to keep to secrecy on the inner workings of their platform due to them spending large sums of both money and work-hours in their research and development.

Examples of cloud computing include [17]: SaaS examples: BigCommerce, Google Apps, Salesforce, Dropbox, MailChimp, ZenDesk, DocuSign, Slack, Hubspot.

PaaS examples: AWS Elastic Beanstalk, Heroku, Windows Azure (mostly used as PaaS), Force.com, OpenShift, Apache Stratos, Magento Commerce Cloud.

IaaS examples: AWS EC2, Rackspace, Google Compute Engine (GCE), Digital Ocean, Magento 1 Enterprise Edition*.

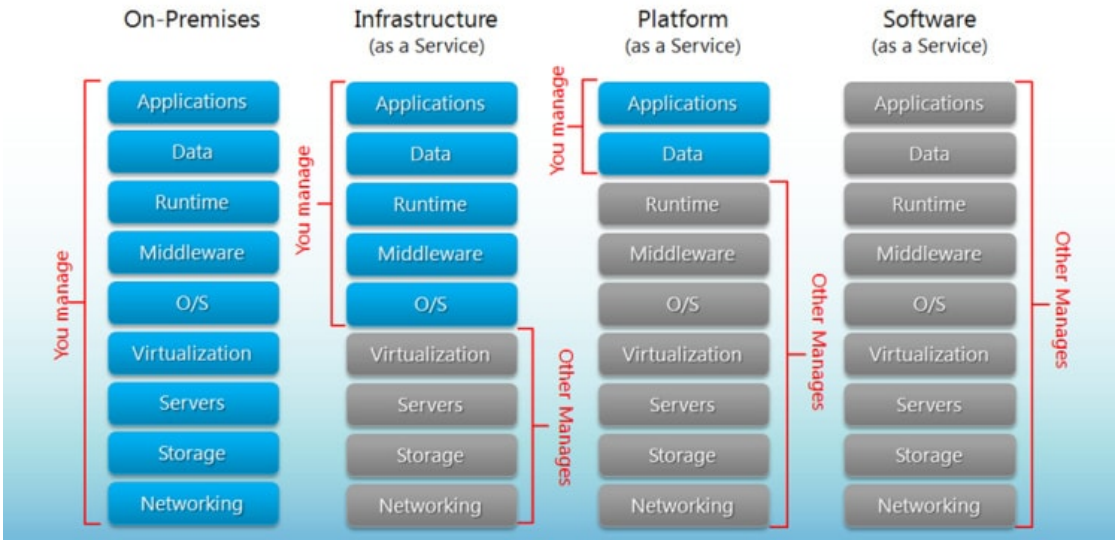


Figure 2.4: Illustration of the differences betwen IaaS, PaaS and SaaS in Cloud Computing. Provided by HostingAdvice

2.2.3 Distributed Architecture

Network Layer

The Fog Layer The "fog layer" is the network layer that is more distributed and physically closer to the edges of the network compared to the cloud layer [6]. A fog is considered a cloud that is less dense and physically closer to the ground, so this visualization gives a general idea of what the fog layer is in comparison to the cloud. There are some disagreements on where the fog begins and where it ends. The similarities in the definitions lie in the fact that it starts at the edges of the network and extends up to the cloud layer. The disagreement often lies in what extent computing at the extreme edges of the network constitutes fog computing or if it is then edge computing. Some other concepts like "cloudlet" [18] (smaller data-centres/servers providing cloud-like functionality to edge mobile devices) would also fit comfortably within the parameters of the fog.

Edge Layer As mentioned above, the terminology of edge computing and fog computing is often used interchangeably, making it harder to properly distinguish the two. This thesis uses the definition of the edge being more or less exclusive to the logical edge of the network [19] and as soon as it is a bit more centralized, it is in the context of the fog. The edge layer will then be the network layer of the extreme edges of the network exclusively and the possible computations made on the devices in that context.

Computational Layer

Fog Computing Fog computing is computational tasks, applications, services etc. that is done in the fog layer. It has its beginning as a complimentary extension of cloud computing where fog computing can provide low-latency functionality by being closer to the utilized devices [20]. This is its main functionality and area of use, but it could also be used as a standalone alternative to the cloud when necessary. The core features of fog computing is that it acts as a middle-layer between the devices at the edge and the cloud. This enables data pre-processing, aggregation, analytics etc. close to the data sources and mitigates some of the struggles associated with cloud computing such as high latency due to being physically further from the sources. An example of fog computing can be a small server that processes data from local sensors and sends it to a cloud storage system.

Edge Computing Edge computing, as fog computing, is a computational paradigm centred around devices at the logical extremes of the network [19]. Its main focus is to utilize computation at the site of the distributed devices to the extent of the resources available. This could either be for light applications that does not require a lot of resources, or to combine it with other more centralized systems that have these resources available. If edge computing is to be used on its own at the edges, this sets higher requirements to the devices at the edges or to expand the computational capacity where these devices are located. An example of edge computing can be that a device that generates data processes it locally on the device before sending it to a more centralized system. Or that it stores its own locally generated data in a local database that can be queried from other systems.

2.3 ZEN

The ZEN (Zero Emission Neighbourhoods) research centre is a research centre for environmentally friendly energy established in 2017 by the Research Council of Norway [21]. They have a multi-disciplinary approach to reaching their goals where information-technology is one of those disciplines. The research of this thesis is done in collaboration with this research centre and will therefore take basis in some of their previous research. The most influential paper being "Data Preservation through Fog-to-Cloud (F2C) Data Management in Smart Cities" [8].

2.3.1 Distributed-to-Centralized Architecture

Data Preservation through Fog-to-Cloud (F2C) Data Management in Smart Cities

This paper focuses on how data preservation can be done in smart cities with a distributed-to-centralized (Fog-to-Cloud) approach [8]. The paper argues that the amount of data

that will be generated by sensor devices etc. will increase exponentially and impose great strains on the network infrastructure as this will most likely be handled by cloud systems. Further it argues that time-sensitive applications and needs for real-time data makes a purely cloud-based solution unsuitable in some smart city scenarios. Their proposal to mitigate this is to provide a fog computing solution as an extension of the cloud solutions and provide data management for the entire flow of data from the devices at the edge and up to a more centralized solution. By providing this between the edge and the cloud, you can benefit from the strengths of the cloud while reducing the strain on the network infrastructure by reducing the amount of data sent, and also providing low-latency access to real-time data.

2.4 Architectures

This section covers a brief overview of the architectures discovered and assessed in the literature review. It is included to better understand the theoretical basis for the choice of requirements later on in the thesis.

2.4.1 Cloud Computing Architectures

Dynamic user-integrated cloud computing

This paper proposes a dynamic user-integrated approach to development of cloud solutions [22] in its architecture. It is specifically designed to tackle challenges set by lacking network infrastructure for locations with underdeveloped network infrastructure, using China as their problem scenario. This model proposes that user clients are dynamically added to the datacenter when utilizing services in the cloud so that the datacenter is able to better scale to rising demands without needing a physical expansion.

A new cloud computing approach based SVM for relevant data extraction

This paper proposes a new SVM-based (Support Vector Machine) cloud architecture for storage and retrieval of data in the cloud [23]. It aims to solve the growing concerns connected to the amount of data that is stored and processed in cloud systems. By utilizing a new SVM technique, it develops learning models that can be used to information filtering and extracting the most useful data. With this model, the total amount of stored data can be reduced and thereby freeing resources and ensures more cost-effective cloud storage solutions.

HTC Scientific Computing in a Distributed Cloud Environment

This paper describes a distributed cloud computing system for use in high-throughput computing (HTC) scientific applications [24]. This distribute cloud computing system would be comprised of several separate Infrastructure-as-a-Service (IaaS) clouds that are set into a unified infrastructure.

NoHype

This paper focuses on a central part of many cloud computing architectures: virtualization [25]. It raises several concerns about the way cloud computing infrastructures utilize a "virtualization layer" to handle the different virtualized operations running in the cloud, and the security concerns this raises. One virtualized instance in the cloud can attack the virtualization layer and possibly access other instances handled in this layer, and therefore raising concerns to the confidentiality and integrity of systems running in that cloud. It proposes an altered architecture that serves the same features as the virtualization layer without actually utilizing it, thereby mitigating these concerns.

C-Cloud

This paper presents an alternative to the common datacenter model by utilizing a shared network of more distributed resources over multiple machines [26]. This platform is then comprised of many connected devices such as PCs, laptops, enterprise servers and clusters, instead of larger data centers like most other cloud infrastructures.

2.4.2 Fog Computing Architectures

Vision: mClouds - computing on clouds of mobile devices

This paper presents an alternate computational paradigm for future infrastructures based on mobile devices [27]. The though is that with the growing capacity and resources and increased ubiquity of mobile devices makes a strong case for the use of mobile clouds. This architecture is then comprised of a network of mobile devices that provides cloud-like features from available computational capacity in local mobile devices.

A Fog Operating System for User-Oriented IoT Services

This paper aims to take advantage of technological trends of increased IoT presence and capability [28]. It introduces FogOS, a fog computing architecture for IoT services to capitalize on these developments and increase the capabilities of IoT services. The operating system aims to tackle challenges of varying diversity and heterogeneity of the

IoT devices it will utilize. They then argue how the FogOS is designed to provide and manage IoT services in an effective and efficient manner.

IFCIoT: Integrated Fog Cloud IoT

This paper identifies a need for an extension to the existing data management system through cloud computing [12]. Based on several concerns like network latency and bandwidth constraints they clarify the weaknesses by only utilizing the cloud computing paradigm and states that a new solution will be needed to face growing demands. They then propose an Integrated Fog Cloud IoT solution that extends the existing solutions by moving relevant computational parts to the logical extremes of the network and thereby closer to the utilized data sources. Their solution is then set in an Intelligent Transport System context to provide a better understanding of possible usage. This context gives a general overview of the strengths of the proposed architectural paradigm and more specifics of the solution itself is highlighted. A reconfigurable and adaptive fog-node/edge-server architecture is presented as a viable solution to the given problem. Afterwards they elaborate on other possible usages such as smart cities, localized weather maps, environmental monitoring and real-time agricultural data analytics and control. They argue that fog computing provide several advantages in certain settings and that cloud-computing is lacking as a natural consequence of its centralized nature. The decentralized strategy of fog computing is superior with concerns to future IoT challenges such as network latency and bandwidth constraints. In addition it is also better at real-time responsiveness, mobility support and location-based customization. Fog computing is thought of not as a replacement for cloud computing, but to augment its current capabilities with the strengths of fog computing when feasible.

FOG-Engine

The weaknesses of the current state of cloud computing, such as latency and inefficiency in dealing with big data applications, is underlined at the start of the paper [29]. A Fog approach is proposed to alleviate some of the difficulties by augmenting the already existing cloud architecture with local clusters of computational units physically closer to the sources and sensors. The concept of a "FOG-engine" is proposed in the article to fill the need of devices closer to the source. In the proposed solution IoT devices are equipped with the FOG-engine and they make up a Peer-to-Peer smart system. This smart system coordinates between each other while also providing a gateway to cloud services for other devices. To reduce data communications overhead on data analytics, it is proposed to move the computations closer to the data source. FOG-engine is proposed as a way to enable on-premise processing at the device-level of the IoT. This has several benefits such as lower latency, higher throughput, and less usage of network bandwidth. Proper evaluations are needed however to demonstrate the benefits of the solution.

2.5 Containerization

"Containerization is a lightweight alternative to a virtual machine that involves encapsulating an application in a container with its own operating system. A container takes its meaning from the logistics term, packaging container. When we refer to an application container, we mean packaging software." [30]. In short then, a container is then the entirety of the code that is to be executed, the environment it is to run in and all dependencies packaged neatly together. These containers can then be run on a container platform of the same type and framework. Docker is a popular open-source containerization platform that can be used for this purpose (more on docker below in its own section). So as an example, a container image is built according to a specification provided from a file (Dockerfile in this example). This image can be used to make containers, that are packaged code and dependencies made according to a specific blueprint in the form of the image. These containers can then be built through images on any device that have the docker platform installed.

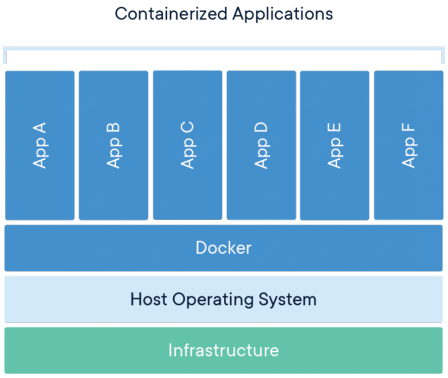


Figure 2.5: Illustration of containerization through the docker platform

2.6 Container Orchestration

"Containers support VM-like separation of concerns but with far less overhead and far greater flexibility. As a result, containers have reshaped the way people think about developing, deploying, and maintaining software. In a containerized architecture, the different services that constitute an application are packaged into separate containers and deployed across a cluster of physical or virtual machines. But this gives rise to the need for container orchestration—a tool that automates the deployment, management, scaling, networking, and availability of container-based applications." [31]. Container orchestration is then the act of managing and organising containers to a specified pattern of

behaviour that is explicitly defined. This definition includes several factors that is needed to orchestrate the different containers and is different depending on the specification for each container orchestration framework. The things all have in common, however, is the specifications of the different containers to be orchestrated (image, number of replications etc.). More information on specific technologies are outlined further on.

2.7 Technologies and Frameworks

This section covers the different programming languages, technologies, frameworks and platforms utilized in the thesis. It is mainly meant as a quick introduction to the purpose and function of each of the entries and not comprehensive explanations.

2.7.1 Programming Languages

Python

"Python is an interpreted, interactive, object-oriented programming language. It incorporates modules, exceptions, dynamic typing, very high level dynamic data types, and classes. Python combines remarkable power with very clear syntax. It has interfaces to many system calls and libraries, as well as to various window systems, and is extensible in C or C++. It is also usable as an extension language for applications that need a programmable interface. Finally, Python is portable: it runs on many Unix variants, on the Mac, and on Windows 2000 and later."[32]. Python is quick to learn and provides an environment that makes it quick to go from an idea to an executable script. Due to the native support on most linux-based operating system, python has become a powerful tool for native-like scripting on these operating systems. This leads to python being dominant in IoT due to the high percentage of those devices running linux.

JavaScript

"JavaScript (JS) is a lightweight interpreted or just-in-time compiled programming language with first-class functions. While it is most well-known as the scripting language for Web pages, many non-browser environments also use it, such as Node.js, Apache CouchDB and Adobe Acrobat. JavaScript is a prototype-based, multi-paradigm, dynamic language, supporting object-oriented, imperative, and declarative (e.g. functional programming) styles."[33]. JavaScript is a well-known language that sees dominant use on the web and increased usage in other parts like back-end usage through Node.js. If web technologies are to be involved in the project at any point, JavaScript is also involved in some way. This is either by native JavaScript files, a super-type of JavaScript like TypeScript, or some other language that is transpiled or compiled to JavaScript.

Typescript

TypeScript is a superset of JavaScript that adds typing to the language and compiles down to native JavaScript[34]. The inclusion of optional typing functionality in the language alleviates some of the most prominent concerns to the language that lies in its on-the-fly variable type assignment in runtime. It is gaining popularity and has become a standard in web application frameworks like angular. In addition to this, it also supports the newer ECMAScript standards and adds this functionality to JavaScript through the compilation.

2.7.2 Standards and Technologies

JSON

"JSON (JavaScript Object Notation) is a lightweight data-interchange format. It is easy for humans to read and write. It is easy for machines to parse and generate. It is based on a subset of the JavaScript Programming Language, Standard ECMA-262 3rd Edition - December 1999. JSON is a text format that is completely language independent but uses conventions that are familiar to programmers of the C-family of languages, including C, C++, C#, Java, JavaScript, Perl, Python, and many others. These properties make JSON an ideal data-interchange language."[35]. JSON is adopted as a standard in machine-to-machine communications on the web and has wide usage in lots of different applications, services and API's. The versatility of the standard makes it easy to send data of many types and structures easily from one client or server to another.

MongoDB

MongoDB is a popular cross-platform document-oriented database that uses documents on a JSON-like format as a base database-entry format [36]. It is classified as a NoSQL (Not only SQL) database that is different from regular relational databases (like MySQL) in that it is not based on how the entries in the database relate to the others. Due to the independent nature of each entry in the database, it is highly scalable and easy to implement simple databases.

2.7.3 Frameworks and Platforms

Docker

Docker is a containerization platform that provides both the platform to run containers on and the tools to create and build containers to run on the platform[37]. It is a

popular platform for containerization and is adopted as a standard of containerization in the field. More information on the process of containerization is provided in the section earlier. The specifics of Docker containerization lies in how the container image is built through the Dockerfile. This file contains explicit instructions on the program environment, dependencies and specific build instructions on what commands to run and build sequence.

Docker-Compose "Compose is a tool for defining and running multi-container Docker applications. With Compose, you use a YAML file to configure your application's services. Then, with a single command, you create and start all the services from your configuration."[38]. Docker-compose is a useful tool for defining an environment of multiple containers and building that environment through it. This YAML file can also be used further for container orchestration in a swarm with small configurations.

Docker Swarm Docker swarm is a configuration of the docker engine that allows it to configure and utilize a network of other docker engine to run containers that are balanced throughout this network as a swarm[39]. It builds this swarm network out of nodes where each node is a docker engine running either on different devices or through VMs. The nodes in the swarm are either of the type "manager" or "worker", where "manager" nodes perform workload balancing and request forwarding while "worker" nodes perform tasks and commands given by manager nodes. Each manager node is also a worker node in addition to its manager role.

Node.js

"As an asynchronous event driven JavaScript runtime, Node is designed to build scalable network applications. In the following "hello world" example, many connections can be handled concurrently. Upon each connection the callback is fired, but if there is no work to be done, Node will sleep."[40]. Node is a popular JavaScript framework that provides back-end and middleware functionality in JavaScript with the ability to build and serve JavaScript applications through Node.js. It is commonly used to serve both API's and web applications of multiple types. Web application frameworks like React, Angular and Vue use node.js as a base for their frameworks.

Node Package Manager (npm) "npm is the world's largest software registry. Open source developers from every continent use npm to share and borrow packages, and many organizations use npm to manage private development as well."[41]. npm is widely used and adopted by many different frameworks and technologies. It is essential in developing and building different web applications and is used as a standard in frameworks such as react, angular and vue.

Express

"Express is a minimal and flexible Node.js web application framework that provides a robust set of features for web and mobile applications."[42]. It is a common lightweight framework used in building web application that are based on node.js. Through the simple server hosting configuration, it is easy to build the web application through node and serve it through Express.

Nginx

"NGINX is open source software for web serving, reverse proxying, caching, load balancing, media streaming, and more. It started out as a web server designed for maximum performance and stability. In addition to its HTTP server capabilities, NGINX can also function as a proxy server for email (IMAP, POP3, and SMTP) and a reverse proxy and load balancer for HTTP, TCP, and UDP servers." [43]. Nginx is popular to use as a serving web solutions with a wider array of functionality than minimalist frameworks like Express.

Angular

"Angular is a platform that makes it easy to build applications with the web. Angular combines declarative templates, dependency injection, end to end tooling, and integrated best practices to solve development challenges. Angular empowers developers to build applications that live on the web, mobile, or the desktop".[44]. Angular is a node.js-based web application framework maintained by Google and is one of the go-to choices when creating web applications alongside frameworks like React and Vue. It is not minimalistic like React and have opinionated selections of standard packages used to serve the application. This makes is able to provide a simple servable web application out of the box without the need for much configuration.

Angular-CLI "The Angular CLI is a command-line interface tool that you use to initialize, develop, scaffold, and maintain Angular applications. You can use the tool directly in a command shell, or indirectly through an interactive UI such as Angular Console."[45]. It is a common tool used in the development and building process of the application. It provides easy-to-use convenient tools for generating projects, components etc. while also providing functionality to serve and build the application for both testing and production environments.

2.7.4 Packages, Libraries and Modules

Mongoose

"Mongoose provides a straight-forward, schema-based solution to model your application data. It includes built-in type casting, validation, query building, business logic hooks and more, out of the box."[46]. It is a package for use in managing and communicating with a MongoDB database through JavaScript. It provides convenient abstractions of typing, modeling and communications to the database that makes it easier to both develop and maintain applications using MongoDB as a database.

Chart.js

Chart.js is a simple module for creating and managing charts in the web application through JavaScript[47]. It provides a simple interface with type libraries for making charts in HTML.

Bootstrap

"Bootstrap is an open source toolkit for developing with HTML, CSS, and JS. Quickly prototype your ideas or build your entire app with our Sass variables and mixins, responsive grid system, extensive pre-built components, and powerful plugins built on jQuery."[48]. Bootstrap provides easy functionality through its library to develop responsive web designs without having to go through the process of creating it yourself. It has a wide array of functionalities that are easily customizable and provide a good foundation for the design of web applications of any type.

3 Research Methodology

3.1 Research Method

This thesis is conducting research in collaboration with the research centre on Zero Emission Neighbourhoods in smart cities (hereby referred to as ZEN Centre) as a part of a larger research initiative with several other master theses. The research method applied in this thesis is a simple method that emphasises the informed selection of a solution to solve a specified problem or reach determined goals. The method goes through several phases during the research process from 1. Assessing and identifying a problem or research opportunity 2. Gathering enough information to make an informed decision on how to possibly solve the issue or reach the goals. 3. Proposing the solution based on the information and insight gathered and 4. Evaluating the solution based on its ability to solve the identified issue or reach the goals set. This process is expanded into more steps for a more detailed process as shown here:

1. Assess a problem or an interesting opportunity and the area of study.
2. Identify the exact problem and explicitly define what is to be solved.
3. Collect sources and build understanding of the problem and its underlying themes.
4. Define goals to achieve and requirements set for a solution to the problem.
5. Propose a solution to the problem that should fulfill goals and requirements.
6. Develop the solution to the problem.
7. Evaluate the solution based on knowledge gained through the development process.
8. Conclude research based on the evaluation and to what extent the problem is solved or mitigated.

These steps are followed throughout the thesis and provide the scientific basis for the proposed architectural model outlined later on in the thesis. This method is realized in several steps as outlined below:

1. Identified the problem through the problem description given for the master thesis, extended discussion and introductory reading session on the field of study. (Introduction Chapter of the thesis)
2. Conducted a structured literature review on the state-of-the-art of the field of study to provide basis for the solution. (Background Chapter of the thesis)

3. Collected supplementary articles and an additional relevant literature review to further increase the theoretical basis. (Background Chapter of the thesis)
4. Made requirements for the solution based on the collected sources and studies to tie the theoretical basis into the design of the solution. (Part 1 of the Results Chapter in the thesis)
5. Designed an architectural model based on the requirements to solve the problem identified earlier. (Part 2 of the Results Chapter in the thesis)
6. Implemented a solution based on the architectural model as a blueprint to test the efficacy of the proposed solution. (Part 3 of the Results Chapter in the thesis)
7. Defined evaluation metrics to use in reviewing the designed model and the implementation of it. (Part 4 of the Results Chapter in the thesis)
8. Evaluated the architectural model using the metrics and to what extent they fulfill the requirements set. (Discussion Chapter of the thesis)
9. Conclude the research with insight gained in the evaluation and propose further work. (Conclusion Chapter of the thesis)

3.2 Research Questions

- RQ1: How can applications/services be made in a distributed-to-centralized context for smart cities?
- RQ1.1: What are the prevailing methodologies and technological trends in the layers of the distributed-to-centralized context?
- RQ1.2: What would be fitting requirements for the development and operation of a system within this context?
- RQ1.3: What would be a fitting architectural model for applications and services in the given context that satisfies the requirements set in RQ1.2?
- RQ1.4: What quality metrics could be used to evaluate the model presented in RQ1.3?

3.3 Requirements

The theoretical background of the conducted literature review and the sum of the collected sources should provide an understanding of the different kinds of architectures and methodologies applied in the relevant layers of network centralization (fog and cloud). By looking at these architectures and their definition, it becomes possible to outline common priorities, focus-areas and challenges. They can then be used as a basis to outline proper

requirements for new ICT-systems to operate within this given context. By documenting these requirements along with possibly some additional requirements added based on personal experience it should then provide a solid basis for the design of an architectural model.

3.4 Evaluation Metrics

To provide a more objective way of evaluating the model itself, there should be defined metrics to evaluate it afterwards. This will be done by selecting fitting QA's (Quality Attributes) that suits the selected requirements and goals of the architectural model. These quality attributes will then serve as a proper method of evaluation for both the architectural model and the implemented system. How well they perform according to the selected quality attributes will provide the basis for discussing the efficacy and suitability of the model and implementation for the discussion chapter.

3.5 Literature Review Research Method

The research method applied for the literature review is based on the structured literature review model (SLR) applied for computer science [49] with three phases; planning, conducting and reporting. The structure for the review itself is outlined below. This review is meant as background research and to create a theoretical scientific basis for the following masters thesis. The final method used is altered somewhat to better fit the context and goal of the review. The purpose of the review itself is to identify and categorize different specified architectures at the different layers outlined (fog and cloud). Therefore it becomes less important to guarantee full coverage of the field (as that would prove to be too large a workload for the review alone) , than it is to focus more in-depth on the more viable candidates to be reviewed. Taking the time constraints into account, this review took a more quality-based approach, rather than quantity-based, to ensure results that can be of further use for the master thesis itself.

3.5.1 Planning

The planning phase consists of five steps: Identification of the need for a review, Commissioning a review, Specifying the research question(s), Developing a review protocol, Evaluating the review protocol. Step 1 and 2 was done when it was identified a need for a literature review as a basis for further research on the masters. The research questions were identified as such:

RQ1: What are the different network layers with regards to network centralization in a smart city context?

RQ2: What are the different architectural solutions available/presented in the respective layers/areas identified in RQ1?

The next step is develop and evaluate a review protocol. This protocol encompasses how the research was done and how it is to be done to produce reproducible results. As the method applied on this review is somewhat altered to include input and suggested articles from others, it has been altered to reflect this change in procedure.

3.5.2 Conducting

Conducting the review is done in five steps: Identification of research, Selection of primary studies, Study quality assessment, Data extraction and monitoring, Data synthesis

Identification of Research

Identification of research consists of specifying what sources to be searched and how to search them. This includes listing sources of research, studies and articles that are to be reviewed as well as determining the terms and their combination to search for. For this search it will be fitting to divide the sources into two categories; primary and secondary sources. This is mainly because of fitting secondary sources that are available, but they are not necessarily held to the same standards as the primary sources. Nevertheless they are included as secondary sources as excluding them would be detrimental to the search.

Primary Sources ACM digital library, IEEE Xplore, ISI web of knowledge, ScienceDirect, SpringerLink, Wiley Online Library.

Secondary Sources OpenFog Consortium, Supervisor Recommendations

	Group 1	Group 2 (Cloud/Centralized)	Group 3 (Fog/Decentralized)
Term 1	Architecture	Cloud Computing	Fog Computing
Term 2	Framework		Edge Computing
Term 3	Infrastructure		

Search Terms The search strategy will then be either term from group 1 combined with either term from group 2 or group 3 $([G1, T1] \vee [G1, T2] \vee [G1, T3]) \wedge ([G2, T1] \vee [G3, T1...T2])$

All search queries will then be used to search for complete matches on all search terms in the query (e.g. Cloud Computing Architecture) published after 2010 (given date to reduce overloading amount of articles). These given queries will then be searched systematically across the given primary sources. Each query will be searched three times

on each source where the sorting is different (most relevant, most citations and most downloads where possible) for variety and to ensure better coverage of the field.

Selection of primary studies

Studies here are selected based on the criteria above as well as some general points that exclude some studies. These are: 1. Duplicates (keep the highest ranking source), 2. The same study published in different sources (keep the highest ranking source), 3. Studies published before 2010 (as a general reference to exclude older studies). These studies are then gathered, assessed by the method described below and the architectures in question is extracted and added to the review.

Study quality assessment

With a selection of primary studies to include, they should be assessed based on some specific criteria that makes it eligible for inclusion in the study, or excludes it as it is not deemed necessary to include for the research. These are made with the research questions, research goal and research strategy in mind to best fit what the research aims to achieve.

Criteria Identification	Criteria
IC1	The study concerns a Fog-To-Cloud environment
IC2	The study is either an architectural proposal or a review/critique of an existing architecture
IC3	The study describes a valid architecture
IC4	That architecture must be usable or reasonably adaptable to a smart-city environment
IC5	The study most contain some reasoning behind the selection of the architecture

(IC = Inclusion Criteria, QC = Quality Criteria)

3.5.3 Reporting

Reporting is done in three steps: Specifying dissemination strategy, Formatting the main report, Evaluating the report. These steps conclude with the end report and the final review itself. To better fit with the overall context of the master thesis, these steps involves how the review is merged with the thesis. It is meant to provide a scientific background for the rest of the thesis, so it is then added into the background chapter of the thesis. The discovered layers and list of architectures are then outlined in seperate sections of the background chapter.

4 Results

4.1 Requirements

These requirements are to act as a baseline for the design of the architectural model. They are based on the previously mentioned literature review and will provide the scientific grounding for the design. This basis will help to ensure that the model adheres to the state of the art and improve its relevance to the field of research.

Each requirement outlined below contain the requirement itself, additional comments to what the requirement is and a specified reason for the selection of the requirement. The requirements are then split into two different categories based on design/implementation priority. These are "Primary Requirements" that are the main focus of the design and "Secondary Requirements" that will affect the design, but not necessarily be central to the design. Both consists of requirements that are significant for the design of the architectural model and the implementation of it.

Primary Requirements

Requirement 1: The system must be able to scale its computations based on resources available. What: By this it is meant that the model must enable and facilitate functionality to scale computations/operations depending on the resources available.

Why: One of the main strengths of using a more distributed form of computing is being able to utilize locally available resources. By using smaller devices with different resources available you can drastically increase the scope of the system without necessarily needed to buy more hardware or renting more server space. This was a common theme throughout most of the architectures of the literature review.

Requirement 2: The system must be able to continue its operation as usual within normal operating parameters (within reasonable limits) when adding or removing resources. What: The model must provide functionality to handle changes to available resources and hardware without inferring significant downtime. This would be either if a device is added, removed, disconnects or malfunctions.

Why: With the inherent volatility that comes with distributed computing, it is important that this risk is mitigated by the model. This is largely due to more points of

failure with the system comprising of several smaller devices instead of fewer centralized servers. This volatility can (and most likely will) affect the system and should therefore be handled in a proper manner.

Requirement 3: The system must be able to utilize devices with a lower amount of resources/computational capacity What: Being able to support the integration of smaller devices (both in size and resources) into the system and making them a part of the operations of the system.

Why: As mentioned before some of the strengths of distributed computing lies in utilizing multiple smaller devices. Due to this nature, there must be functionality to support the usage of smaller devices for either parts of the system, or to support it in its entirety when feasible. These devices would be e.g: raspberry pi's, embedded devices, mobile phones, IoT Devices etc.

Requirement 4: The system must be able to receive and manage data provided from both internal and external sources What: The model must provide functionality to properly store and manage data retrieved from internal and external sources.

Why: In order to provide proper data management in a distributed-to-centralized manner, there must be functionality in place to interface with the system. The focus on this requirement is the storage and management part of that environment. This interfacing is central to the core functionality of the system as it enables data aggregation and pre-processing.

Requirement 5: The system must be able to support cloud functionality in the form of resources for storage and/or data management What: The system must be able to provide functionality that supports the usage of cloud resources (storage, computation etc.). This can be either supported natively in the system itself or providing interfacing options with cloud platforms.

Why: In order to fully facilitate the distributed-to-centralized data management environment, there must be functionality available that bridges the gap between a mainly distributed system to either use or provide services for a cloud solution. If this was not the case then the model would be restricted to the "fog" network environment as it would not be able to interact with the cloud network environment. An important point from the background research is that the fog should extend the capabilities of the cloud and not seek to replace it. By then enabling the bridging of fog and cloud through a distributed-to-centralized data flow, this priority is maintained in the model.

Requirement 6: The system must provide a platform for data aggregation and/or data pre-processing functionality What: The model must be able to provide function-

ality for the processing of data that it manages. Preferably this support is not dependant on a single language or framework, but rather that it is able to use these interchangeably.

Why: In order to provide proper data management throughout the data life-cycle of the data managed by the system, there must be functionality available to process and aggregate this data. Unprocessed data decreases the utility of the data and should therefore be processed in order to be of better use. It is important that this can be handled when needed by the system itself. Often data is also combined with other data sources, data types etc. to further increase its area of use, so it is considered a vital function to support this. This also relates to the background research through both the focus on providing distributed-to-centralized data management but also that many of the assessed architectures provided similar functions or at least strives to.

Requirement 7: Must provide functionality to forward data "upwards" in the network. Either to another "fog" system or to the cloud What: Must provide functionality that enables the "forwarding" of data upwards in the network either to some other distributed system higher up in the hierarchy or straight to a centralized solution. This can be achieved either by providing functions for sending the data at set sizes and intervals to a specified entry-point, or by providing an interface for the extraction of the data for the other system to utilize.

Why: This relates to the sentiment specified earlier about distributed-to-centralized data management. If the model does not provide functionality to "funnel" the data upwards, then it does not properly facilitate the proper data flow that is desired.

Requirement 8: The system must provide support for data life-cycle management What: The system must provide functionality to keep track of, manage and process data throughout the life-cycle of the data that is managed by the system. This will include keeping track of the age of the data and make sure that it is handled according to the needs of the specifications of the data life-cycle. In addition it is required that the system keeps track of the location of the data throughout the phases of the data life-cycle that it manages and forwards requests properly.

Why: One of the strengths of having the system physically closer to the data sources is being able to provide real-time (or at least reasonably close to real-time) data of the sources where such is possible with reduced latency. Furthermore, the data may transform over time to be used in more use-cases than the more recent data. Being able to handle the data at these different phases of its life-cycle will extend the usefulness of both the system and the data it manages. Example life-cycle: Real-time data –> Recent data –> Historical data

Requirement 9: The system must provide an interface to retrieve the data managed by the system This includes routing to the data storage according to the data life-cycle

management of the system.

What: The data that is managed by the system must be retrievable by an authorized/authenticated client according to the area of use for the system. This could include either a report-like function or simply a status "ping" of the selected data.

Why: There must be functionality to access the data managed by the system when needed if that is included in the purpose of the system. Usually this will be central to the core functions of the system as accessing the managed data will be important in smart city/neighborhood/building scenarios.

Secondary Requirements

Requirement 1: Security What: The model must be designed with improved security in mind and mitigate risks of potential security breaches within reasonable limits. Specifically keep in mind the entry-points of the system. All "intractable" parts of the model must either provide a secure entry gateway if needed, or limit outside interactions to it. Internal gateways are to be kept free of outside interaction and all purposefully designed interactions must adhere to common security principles to protect both the integrity of the system, its data and the privacy of potential data sources and users.

Why: Security is important and should always be kept in mind when designing or implementing solutions. It is noted here as a secondary requirements as it is inherently difficult to improve the security of an abstract model as most of the security vulnerabilities adhere to the implemented solution itself.

Requirement 2: The system should be able to utilize an array of commonly available hardware/devices What: The system should be kept as independent as possible from different platforms, operating systems, programming languages and frameworks. If any of these are specified particularly as a dependency, then it should be as interchangeable with similar options as is feasible.

Why: Keeping the model entirely dependent on a specific platform, operating system, programming language and/or framework will severely limit the potential of the model itself. By keeping the model design as independent as possible it has increased usability and adaptability both in its area of use and its ease of use for potential developers. Some of the architectures assessed in the review are dependent on new frameworks introduced in their papers. This tendency to introduce new frameworks and/or languages increases the complexity of the solution and decreases the adaptability of the system as well as creating a steeper learning curve.

Requirement 3: The system should be able act accordingly to the different energy-efficiency needs on embedded devices What: The system should be able to identify and keep track of the different energy-efficiency needs of the devices in use. It should

then be able to act and change its behaviour/runtime according to the possible different needs of these devices.

Why: Different devices have different specifications with regards to resources available, energy-efficiency focus and potential battery capacity. In order to properly manage IoT-devices and other smaller embedded devices it is important to keep this in mind e.g: a battery-powered device should be treated differently than a device that is connected to the power-grid.

Requirement 4: Privacy What: The model must be designed with privacy as a priority. If feasible, the model should take steps to mitigate privacy concerns and take actions to protect people's privacy through the data managed by the system. The data should be anonymized when possible without lessening the function of the system. In addition to this it is important to limit access to data that can be used to identify individuals or groups of people to people that have a proper certification and no intention to exploit the data of the system.

Why: With the rapidly increasing ubiquity of connected devices, privacy has become a large concern both for end-users and involved governments. After the General Data Protection Regulation [50] (GDPR) passed European Union legislation, privacy has gone from a concern to a requirement. Therefore it is important to keep privacy in mind when designing new systems that has even a remote possibility of handling personal information. Big corporations like Microsoft, Google, Amazon etc. has not lowered this concern for people as more and more information leaks on the extent of what these large companies know about your personal information and preferences.

4.2 Architectural Model - Tiered-Priority Swarm Computing (TPSC)

4.2.1 Overview

The goal of this section is to provide the design of an abstract architectural model to be used as a schematic and guide when designing ICT-solutions (applications, services etc.) in a smart city environment. The model is purposefully made to utilize concepts rather than specific technologies and frameworks when feasible to fit this purpose. In addition to this, it is a major focus that the design itself adheres to the concept of distributed-to-centralized data management. This intended compliance is achieved through designing the entirety of the model to act as a "component" itself. It will be able to be used anywhere in this data-flow by facilitating the flow of data from any number of sections of the network structure and providing interfaces for the receival and retrieval of data. In essence this allows an implemented system to be used at any desired level of network centralization from smart buildings up to smart cities and beyond. It could facilitate the entirety of the data flow from the distributed sources to centralized storage, or it could provide this for any smaller section of the data flow. This is up to the implementation of the model as the model itself is able to scale to fit the size of the intended area of use.

The key priorities set for the design of this model revolve around several important concepts. These are: The concept of a scalable system utilizing a wider range of devices, being able to make use of devices on an IoT scale rather than larger dedicated hardware, the support of data at its different phases in its life-cycle and lastly to provide more of a platform with wide areas of use. This is to better fit the different possible use-cases of smart systems up from smart buildings to smart cities. This makes it clear that the design of this architectural model must have a "component" mentality which means that the system is divided into smaller components that together provide the full range of functionality the system is to offer. These components will be small enough in scope to be properly support devices with less resources. In order to make full use of this component-based design, something is needed to coordinate and otherwise manage the components running on different devices. In addition this coordinator/manager will need ways of handling data management in accordance with D2C-DM and the requirements and needs of the data life-cycle.

4.2.2 Swarm Computing (Clusterized Computing)

"Swarm computing" is a term coined from the recently adopted "Docker Swarm" platform as it is that concept that is applied on the platform. The term swarm computing could then be viewed as cluster computing at a large scale with a larger part of the nodes in the network being represented by a multitude of smaller devices. The core concept of this clusterized computing is to build, manage and maintain a form of "runtime environment" that is specified using "containers", their interaction and a desired state. The Docker

Swarm platform is based on docker containers and the specified runtime is mostly in the form of managing parallels of the given containers when they run over capacity and are unavailable for further requests. The term can then be loosely defined as a form of parallel computing based on specified components and the management of those. (More information on containerization and container orchestration are available in the background chapter)

The core strength of this form of computational paradigm lies in the scalable nature along with the impartiality to specific languages and frameworks. This strength makes it a viable candidate to be used in a distributed network setting such as the setting for this thesis. The core concepts that make up swarm computing will be elaborated further on for each important aspect of the paradigm. There are more problems, however, that are not solved solely by swarm computing. That is why the swarm computing paradigm is extended with further functionality in addition to changes to the workload management priority assignment elaborated below in the section on "Tiered-Priority Workload Management".

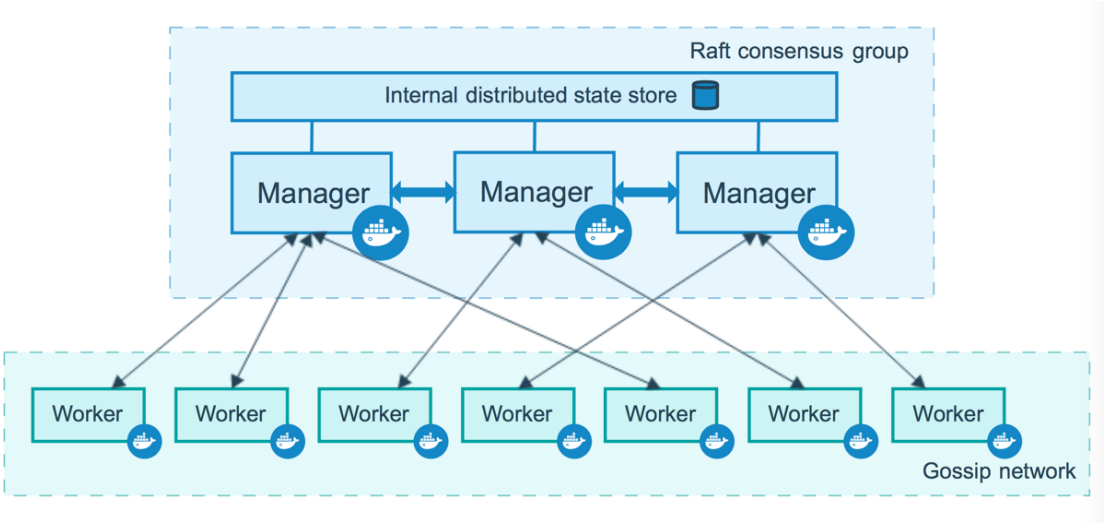


Figure 4.1: Swarm Architecture Example - Case from Docker Swarm implementation

Containerization

Concept Containerization is the concept of dividing parts of (or all) the system into components that are packaged individually with its own run-time environment. A container is defined as "Package Software into Standardized Units for Development, Shipment and Deployment". This in turn is run on a container platform that is able to handle both the packaging and shipping when developing and also the deployment when the system enters the production environment. This is fundamentally different from the current standard of using virtual machines as the container is only packaged with what it

needs to run and therefore requires less overhead as multiple operating systems running are no longer needed. This in turn makes it highly portable and able to run in parallel when required.

Specification The different container architectures specify the structure and syntax for the environment to run inside each container, but all have a standardized form that guarantees the same environment when run on the same platform. The current most popular architecture specify the structure of the container through a file following a specific syntax called "DOCKERFILE" where all packages, languages etc. are outlined along with the install/initializing sequence that is run when the container is "mounted".

Independency from specific Languages and Frameworks Due to the container comprising the entire runtime environment needed for that component, it has a high degree of independency from specific programming languages and frameworks. As long as the chosen architecture is suitable to that type of bundling there is no inherent constraints introduced bu the container itself. Some languages and frameworks has a higher rate of compatibility due to the degrees of dependencies they impose on the container. Bundle size and installation complexity are important factors in this due to the container specification file increasing in proportion to this. Size generally matters less than complexity in most cases, but where storage space is limited this can lead to some frameworks and programming languages being less viable than others.

Container Orchestration

Concept Container orchestration is the practice of managing a multitude of containers (container definition described earlier) according to set specifications to achieve an "ideal state" of the system. The base for this concept is through the management of the right type of containers, the desired amount of those and what behaviour they shall have. The usage for container orchestration varies and can scale up from multitudes of a singular type of container running in parallel to a complex system with multiple types and varied amounts of containers to be run. In order for this to work, you need the containers that are to be used, a somewhat detailed description of the desired "ideal state" of the system, a management system to coordinate these containers to achieve this desired state and a way to forward and delegate workload to the appropriate container type. These concepts will be outlined more in detail below.

Ideal State When designing the system, the "ideal State" is then specified by these different types of containers and the desired number of the different containers that are to be mounted. The orchestration system will then through its managers organize the running containers and mount new containers as needed. This ideal state is meant to be used as a blueprint and a guideline for how the system is meant to be handled for an

ideal scenario. The specified number of containers for a certain type is by no means a min or max, but rather what the system should default to when it is operating within expected parameters.

Container Management In order for the system to maintain and organize its own status during its operation, it needs functionality to both monitor and alter its runtime parameters. This is achieved through container management in some form or another. A common approach to this is to assign different roles to the "nodes" in the network. These nodes are containers of any type that can usually either act as a "worker" or a "manager" node. The manager nodes act as a worker node itself, but will also contribute to organization and management for the cluster. These tasks include checking the status of the different nodes, delegating incoming requests to an appropriate worker that has resources available, stopping or starting containers as needed in accordance with the ideal state etc. The number of manager nodes will also scale as needed by the system.

Forwarding and Workload Delegation For the cluster to work properly there must be a way to manage the workload balance of the nodes and also to ensure that requests are forwarded to a node that is capable of handling it. This task is usually handled by the managers of the cluster as a part of the operating tasks they perform to achieve the ideal state. The core of this concept is mainly to achieve a balance between the resources being used, available resources in the active containers and the total amount of available resources from the hardware. When a task is being forwarded to an appropriate container, but none has the capacity to handle the request, the managers then uses more of the available hardware resources (if any is available) to start a new container to run in parallel with the others.

4.2.3 Tiered-Priority Workload Management

Purpose

As explained above, workload management is central to proper management of a cluster and therefore central to container orchestration. At the time of writing, this allocation of resources is done randomly where resources are available without the functionality to specify otherwise. Usually this has little consequences, but that carries the assumption that the resources available can be used interchangeably without much concern of what kind of device it runs on. This will matter with an increased focus on devices with lower amounts of resources and different priorities when it comes to energy-efficiency and power management. These devices such as IoT-, mobile- and embedded devices have vast differences in storage, computational capacity, battery capacity and possibly bandwidth. In order to properly utilize these kinds of devices alongside larger devices, these differences must be accounted for. The random selection of device resources to utilize as a node in the cluster can lead to devices running on a limited battery capacity

being used while dedicated hardware remains poorly utilized. There will likely be a lot of use-case scenarios where the number of smaller devices heavily outweighs the number of larger devices in the network cluster. Statistically, the randomized selection will lead to the smaller devices being utilized more often than more capable hardware due to the difference in multitude. Tiered-Priority Workload Management (TPWM) would change this by systematically assigning priority to the different devices depending on their capacity and energy-efficiency priorities.

Concept

As mentioned above, the model must be able to handle a vast array of different devices in order to achieve its full potential. These differences in resources and capacity need to be factored into the workload management done by the system. One way to do this would be to utilize a smarter resource allocating method that factors in the inherent differences in the devices of the swarm network. There are several factors that would affect the workload management; storage, GPU- and CPU-capacity, memory, internet connectivity (bandwidth + speed), battery capacity (if applicable) and energy-efficiency priorities. By assessing these and evaluating a total "capacity" of the devices, you can assign them a priority by comparing it to the other devices in the swarm. This can be taken further to assign different "tiers" of priority where the devices will be placed in a tier according to their total assessed capacity. The tiers themselves are dynamic and serves as a guideline for the orchestration management when viewing available resources for workload delegation. The "lookup" of available resources would start at the highest tier, T_0, and if not enough resources are available it will proceed further down the tiers (e.g T_1-T_3).

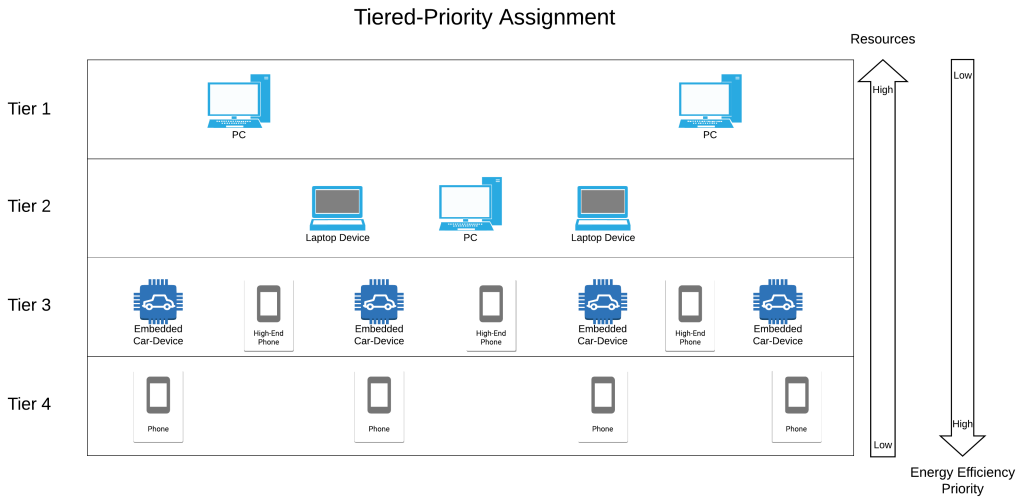


Figure 4.2: Tiered-Priority Example

Priority Assignment

The assignment of priority to a given device can be done in several ways. One is to manually assign a priority-tier to a device when it is added to the swarm. This would only require a simple configuration document for the swarm managers to utilize where it will act as a quick lookup of resources on the lists of units. One other would be to have a script that automatically assigns priority-tier when a device is added to the swarm. This would require a script that is capable of assessing the capabilities of a given device based on the earlier specified factors (storage, GPU- and CPU-capacity, memory, internet connectivity, battery capacity and energy-efficiency priorities). The automatic priority assignment would be preferable as that enables the swarm to potentially add and remove compatible devices dynamically from devices on the same network.

As of writing there are no viable frameworks or technologies supporting a customized workload management algorithm to the extent envisioned for this model. This could be achieved through several sub-networks in the swarm that would be operate as two distinct swarm networks and would therefore result in poorer resource utilization and scalability. Another option would be to develop a container orchestration system that allows priority assignment for workload management or to modify an existing solution. This however, is sadly beyond the scope of this thesis so it will not be featured in the example implementation later on in the thesis.

4.2.4 Data Management

Concept

One of the core design principles of this thesis is to enable and facilitate the flow of data from the edges of the network to a more centralized state (D2C-DM). In order for this model to function in a distributed, centralized or distributed-to-centralized capacity, it needs to be able to handle the appropriate flow of data. For this compatibility it needs to be able to receive data from data sources at the edges of the network like sensors etc. while also providing an interface to access this data to facilitate a more centralized system that will handle the data further. In addition to this interfacing, the model must be able to handle the management of data at different points in its lifecycle. This will be specified according to the needs of the system and the data that it manages. It can be to anything from real-time data and up to historical data depending on the implemented system and its degree of centralization. A more distributed implementation may require the handling of real-time (or close to real-time) while a more centralized implementation may require the handling of historical data. Either way it must be able to handle this range of data and the data transformation specified for the data throughout the part of the lifecycle managed by the system.

Data API

To achieve a usable interface both for the retrieval and receival of data, a middle-layer is needed between the swarm and its sources and clients. This middle-layer shall provide an API that can be utilized by both the implemented system and any other system that are to interact with it. Some likely examples of this could be sensors sending data to the system, some application viewing data handled by the system, another system that has processed data it sends to the implemented system and/or a more centralized system that continues the flow of data and manages the data further. All these possible scenarios are handled by serving this middle-layer for interaction with the implemented system.

Data Lifecycle Management

Data in a smart city setting usually comes from the edges of the network through sensors and the like. It is often combined with other data either from other sensors or other sources. When combined with other sources of data, it can provide insight into a larger context and provide information beyond the scope of a singular source and type of data. Combining, transforming and aggregating data is therefore very common in smart cities. It is then vital to facilitate this process and enable it as an inherent part of the design of this model. The core part of this would be to enable data transformation that is triggered either through an action (when receiving data and/or pre-processing before passing it on etc.) or is time-triggered either through the duration it has been stored in the system or through the age of the data according to its timestamp. By utilizing the container orchestration architecture, the implemented system can create both data transformation containers to process the data and data management containers to manage the process and schedule data transformation tasks if time-triggered events are desired.

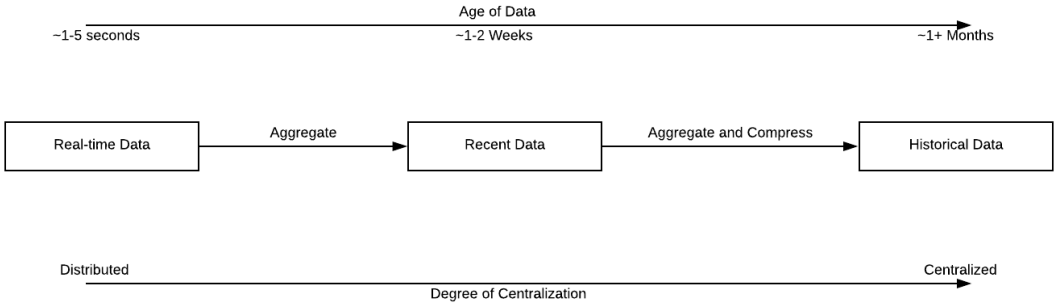


Figure 4.3: Example of typical data life-cycle in a smart city scenario. Illustrates data flow from distributed to centralized based on age of data

Request Handler

The model aims to support any phase of the data lifecycle from a small part to the entire lifecycle. This could possibly entail the management of data from the sources themselves up to and including historical data storage. No matter how this is envisioned to be solved in an implemented solution it should have support from the model. The managed data may not only go through several transformations, but could end up with different storage solutions at different physical locations all depending on the needs of the data management. To handle this properly the model requires a request handler. This handler would be called on by the middle-layer (API) when a request is made to the swarm. It will then process the form of the request (POST/GET), authentication and security validation on the request, and lastly the location(s) of the data required. Depending on the lifecycle management policy of the implemented system, the data could be at different locations and using different query languages and frameworks. The handler will therefore be abstract in its design to allow for as many different possible solutions desired by the implemented system. This would entail that the developers utilizing this architectural model would have to design the specifics of their request handler as needed to fit their architectural priorities and needs.

4.3 Implementation

In order to test out the efficacy of the model and to provide proper evaluation methods, it is useful to develop an example of a possible implementation of the model. This implementation will act as a proposed solution to realize the architectural model using currently available technologies and frameworks. Firstly, there will be an example of an ideal implementation to be used as a template for creating architectures following the model in addition to the actual realized architecture applied for the implementation. This is to show a more general architecture that utilizes the model and how this is adapted into a more specialized solution for the given scenario. The specialized architecture will be the main focus of this chapter and will contain more detailed descriptions of the applied design, utilized technologies and frameworks and how the principles of the model apply to the different parts/components of the system.

The purpose of this implementation is to:

- Provide an example of an implemented solution based on the architectural model
- Test the efficacy of the solution for solving the problem description outlined earlier
- Test the validity of the architectural model as a solution to the outlined requirements from the background research
- Outline possible flaws to the architectural model that needs to be addressed now or in future work
- Provide the basis for a system to properly evaluate both the architectural model and the implementation of it

4.3.1 Scenario

The scalability of the model makes it so that it can applied to different degrees of network centralization. To show this scalability, it can be useful to show it at a smaller scale and outline how it can be scaled up to fit a larger scenario or used in unison with other more centralized systems. One common use-case on a more distributed scale is a Smart Building scenario. This scenario will therefore focus on a small scale smart building system set in a simulated apartment complex that provides monitoring of life-quality factors in the apartments. The "building" has two apartments that have a smaller embedded device (raspberry pi) capable of monitoring the temperature in each apartment in addition to a dedicated machine situated in a server-room in the basement of the building. The needs here is then to provide a system capable of managing the temperature data from the smaller devices and providing an interface to monitor the data that is managed by the system. The scale of the implementation is therefore both to show an example for more distributed use-cases and also to reduce the amount of work needed to implement the system due to time constraints.

The total list of the systems needs/goals will be as follows:

- Provide an interface to receive temperature data from sensors
- Process the data received and compress data to minute-basis rather than seconds.
- Provide interface to request and retrieve the data.
- Build a web application for users to interface with the system to display the data managed by the system.
- Provide interface for other systems to retrieve data for further use.

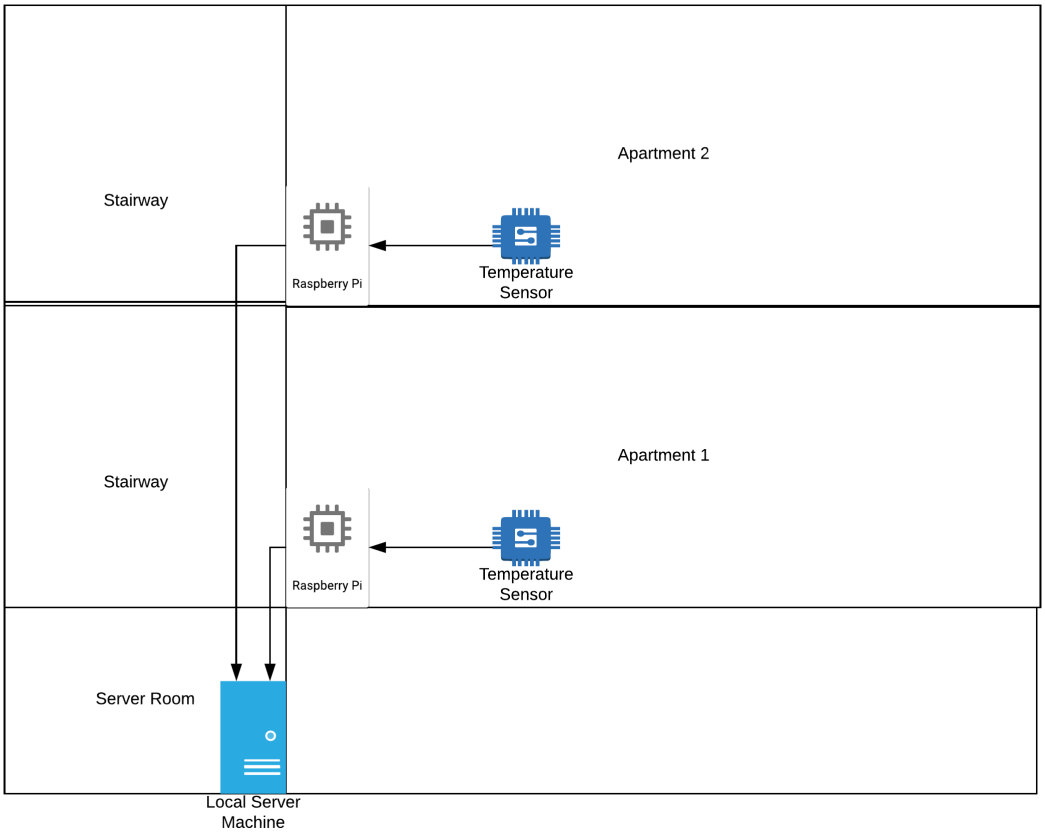


Figure 4.4: Apartment building scenario with displayed hardware available

4.3.2 Implementation of TPSC

One of the core concepts of the architectural model (Swarm Computing) is to divide the system into components, apply containerization to these components and utilize container orchestration to run and manage it. This applied to scale can be envisioned

as swarm computing due to the numerous nature of smaller devices working together to fulfill the computational needs of the system. It is then important that the implemented system has a high degree of componentization to allow for scaled operation through container orchestration. Further it is important to facilitate the flow of data to the system, within the system and from the system to comply with distributed-to-centralized data management priorities. Lastly it is important to manage the container orchestration to allow for prioritized workload management to better comply with energy-efficiency policies of the utilized devices. With these factors in place through the concepts, designs and methods outlined in the architectural model (TPSC), the system can then be viewed as utilizing the model properly.

The implemented system achieves most of these priorities, with some exceptions for tiered priority assignment due to technological constraints in the available technologies and frameworks. This will be outlined further on in the outlined architecture below as well as in the discussion.

Architecture Template

To better understand the basis of the architecture, a proposed architectural template will be outlined here in short. This was set up early in the implementation to function as a general-use template for implemented systems utilizing the TPSC architectural model. It adheres to the principles outlined in the model and includes all the concepts, designs and methods. This template was used as a starting-point in designing the implementation, where changes were made to better suit the scenario and scale the system would operate within. It is included here to give more context and examples on how to utilize the architectural model in addition to functioning as a starting-point for the final architecture outlined below.

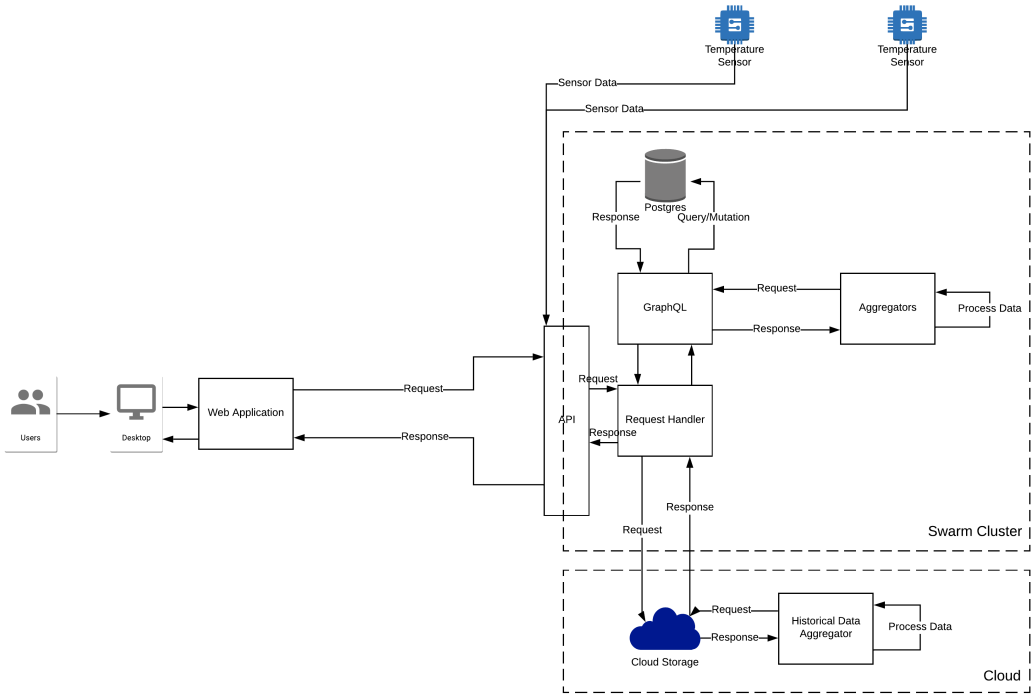


Figure 4.5: Architecture Template designed for a generalized use-case for TPSC implementation

Figure 4.5 shows an illustration of the template with some references to technologies and frameworks that would function well with the intended use-cases of the system. These technologies can freely be exchanged with others with same levels of functionality. This example shows an example of how the different components could be designed to achieve the intended level of functionality as according to the architectural model. The components inside of the Swarm Cluster section are intended to be containerized and can be scaled through replication to suit workload needs of the system. It also adheres to distributed-to-centralized data management through the API to allow receival and retrieval of data, and through the request handler to retrieve the data from its correct stage of the data life-cycle (real-time, recent or historical data). The cloud portion is meant to show how the systems can facilitate data management for the entirety of the data life-cycle. This data life-cycle can be handled through real-time data from sensors, recent data stored in the swarm itself and historical data storage allocated to a cloud storage/management system.

4.3.3 Architecture

Architecture Overview

This section covers the various technologies and frameworks utilized in the implementation of the architectural model. The major components and central features are listed below with sections detailing their major and minor dependencies. A major dependency is a technology or framework that is central to the core functionality and completely necessary for the system to build and operate within its parameters. A minor dependency is mainly packages utilized in the component that serves a purpose for some of the functionality, but is easier to replace with some other package or technology that serves the same function. This section covers a brief description of the purpose of the dependency while more detailed description is found below and in the discussion chapter. Figure 4.6 shows the revised architectural template that has been modified to suit the scale of the scenario.

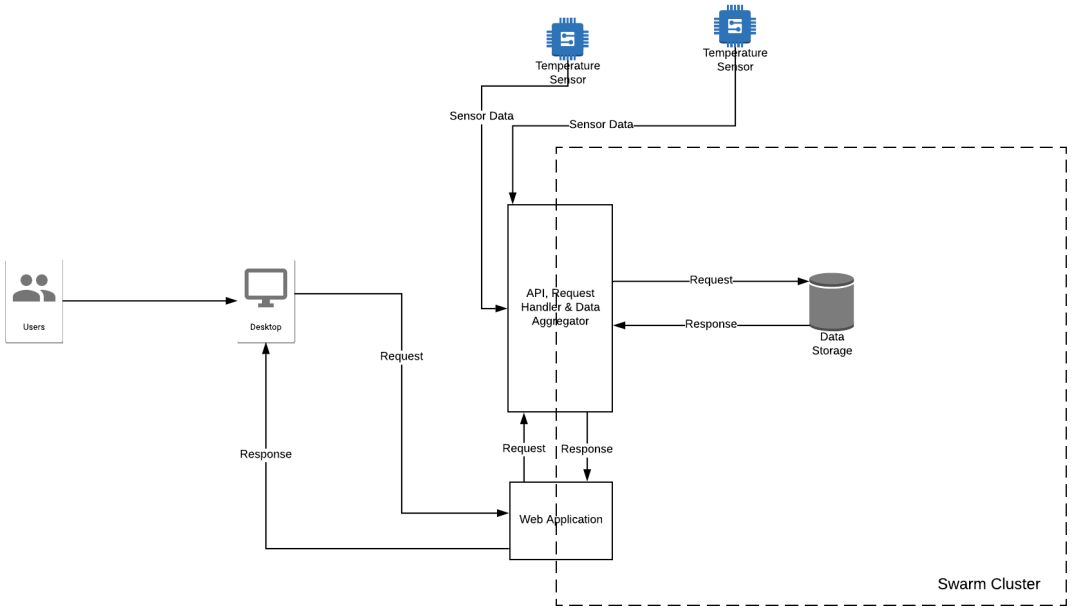


Figure 4.6: Illustration of revised architectural template to better fit scenario

Container Orchestration Architecture Major Dependencies:

- Docker - Containerization of component through official MongoDB image referenced in "docker-compose.yml" file.
- Docker-Compose - Container orchestration and management of services running on the swarm through the service configuration supplied in the "docker-compose.yml"

file.

API-, Request Handler- & Data Aggregator Architecture Major Dependencies:

- Docker - Containerization of component for use in swarm.
- Express - Serve API for data storage and data retrieval both internally and externally.
- Javascript - Express server configuration, API routing and data aggregation.
- Node - Backend JS framework to serve API through Express setup.
- Node Package Manager (npm) - Package management through dependency injections and installation in docker container.

Minor Dependencies:

- Mongoose - Database modelling and management of database communication through connection controllers.

Web Application Architecture Major Dependencies:

- Docker - Containerization of component for use in swarm.
- NginX - Serving the web application on exposed port.
- Angular7 - Developing and building the web application to be served on Nginx. Provides functionality for processing and displaying the data managed by the swarm network.
- Angular-CLI - CLI utility of development and building of application.
- Typescript - Standard language used in Angular. Used for data processing of data from the database and formatting it to display through chart plugin.
- Node - Backend framework for serving web application through Nginx configuration.
- Node Package Manager (npm) - Package management through dependency injections and installation in docker container.

Minor Dependencies:

- Bootstrap - CSS library for use in the design of the web application.
- Chart.js (Ng2-charts) - Versatile chart plugin from npm for use in displaying data from database in web application.

Data Storage Architecture Major Dependencies:

- Docker - Containerization of component for use in swarm.
- MongoDB - NoSQL document database for storage of temperature data from sensors. Interactions to the database is handled through the API.

Temperature Sensor Major Dependencies:

- Python 3 - Scripting of "temperature sensor" functionality to simulate a temperature sensor running on a raspberry pi.

Project Structure

This section explains the file structure of the project and provides some explanations on the central parts of the code. This is supposed to act as a brief overview with the code shown as figures, while a deeper explanation will be provided later on. The finished project for the implementation can be found on Github on this github link (https://github.com/petterrostrup/TPSC).

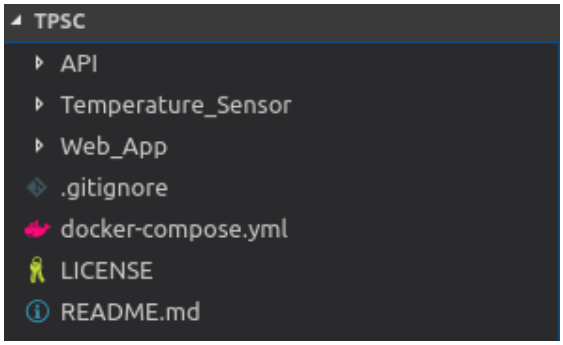


Figure 4.7: Structure of project for TPSC implementation

The structure of the project (as shown in figure 4.5) consists of three directories as well as a "docker-compose" file. The directories; API, Temperature_Sensor and Web_App make up the bulk of the code while the "docker-compose" file is used in container orchestration and management. The "docker-compose" file (see figure 4.8 for part 1 and figure 4.9 for part 2) contains specified information for the building of "images" for the containers and instructions on how they are supposed to run and interact with each other. This is also known as the ideal state of the orchestration management. The three services in the file represents each type of container to be run in the swarm with images (docker image to run as a base for the container), volumes (storage), ports (exposed ports outward and port-forwarding to the container), deploy (swarm behaviour and expected ideal state within that swarm, especially replicas which is the number of this container to run), and network (the services that are expected to communicate lies on the same network). More details on this will be outlined in the "Container Orchestration" section.

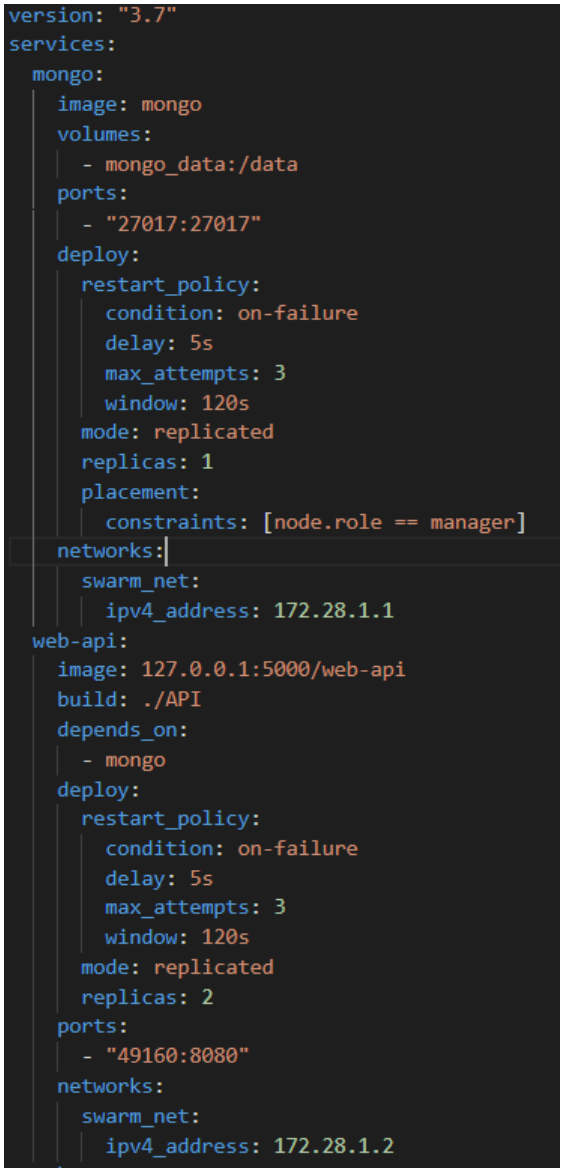


Figure 4.8: Part 1 of the docker-compose file for Container Orchestration

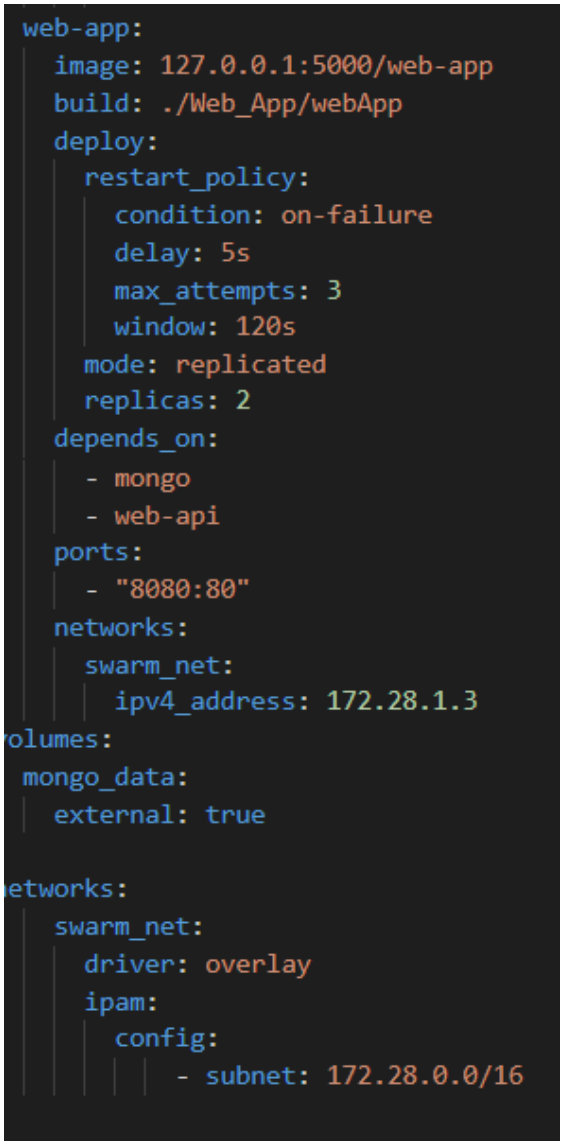


Figure 4.9: Part 2 of the docker-compose file for Container Orchestration

The API directory consists of the code necessary to build and run the API of the system that handles retrieval and receival of data according to the needs of the system. It serves a POST function (see figure 4.10) on /api/temp that takes in a multitude of temperatures from a JSON and saves them to the database. It does some pre-processing on the data it receives through an "average_on_minute" function that takes all data recieved on a second-basis and calculates an average for every data entry on that minute and saves that to the database instead. The API also serves a GET function (see figure 4.11) that gets entries from the database and returns them as a JSON. Both the POST and GET functions use the "Temperature" model (see figure 4.12) as a mongoose schematic

model (described in more detail later) to interact with the database. This component includes the following concepts outlined in the model: API, Request Handler and Data Aggregator. All of these will be described more in detail later in their sections.

```
// process temperatures (accessed at POST http://localhost:8080/api/temp)
.post(function(req, res) {
    var tempsToSend = aggregator.average_on_minute(req.body);
    console.log(tempsToSend);

    // save the temp and check for errors
    Temperature.collection.insert(tempsToSend, function (err, docs) {
        if (err){
            return console.error(err);
        } else {
          console.log("Multiple temperatures inserted to Temperatures collection");
          res.json({ message: 'Multiple temperatures inserted to Temperatures collection' });
        }
    })      You, 3 days ago • temp sensor working, api POST to temperature done, api GET temp
})
```

Figure 4.10: API POST function on /api/temp

```
// get all the temperatures (accessed at GET http://localhost:8080/api/temp)
.get(function(req, res) {
    console.log("Getting Temperatures");
    Temperature.find(function(err, temperatures) {
        if (err)
            res.send(err);

        res.json(temperatures);
    });
```

Figure 4.11: API GET function on /api/temp

```
var TemperatureSchema   = new Schema({
    sensor: String,
    time: Date,
    temp: Number
});
```

Figure 4.12: Model used by API for database modeling

The Web_App directory consists of a web application that sends requests for data to the API, processes the data and displays the information using a chart plugin. It sends a GET request to the API (/api/temp) and transforms the data acquired into a usable format for the chart.js library. This is done through these three functions: "mergeAndProcess", "splitAndSort", and "setLabelsAndData" found in the "app.component.ts" file in the source directory of the web application. "mergeAndProcess" (see figure 4.13)

takes the JSON-response received from the request and processes the data by merging measurements for the same minute and changing some variable types to adhere to the "Temperature" (see figure 4.14) interface. "splitAndSort" (see figure 4.15) takes in the processed list of temperature measurements done in "mergeAndProcess" and splits them into sub-arrays where every array is the measurements from the same temperature sensor. "setLabelsAndData" takes the list of lists created by "splitAndSort" (see figure 4.16 for part 1 and figure 4.17 for part 2) and makes labels and datasets to be used in chart.js.

```
mergeAndProcess(duplicateList: [Temperature]): [Temperature] {
  for (let j = 0; j < duplicateList.length; j++) {
    if ((typeof duplicateList[j].time) == 'string') {
      duplicateList[j].time = new Date(duplicateList[j].time);
      duplicateList[j].time.setSeconds(0);
    }
    for (let k = 0; k < duplicateList.length; k++) {
      if ((typeof duplicateList[k].time) == 'string') {
        duplicateList[k].time = new Date(duplicateList[k].time);
        duplicateList[k].time.setSeconds(0);
      }
      if (duplicateList[j]._id != duplicateList[k]._id) {
        if ((duplicateList[j].sensor == duplicateList[k].sensor) &&
        (duplicateList[j].time.getFullYear() == duplicateList[k].time.getFullYear()) &&
        (duplicateList[j].time.getMonth() == duplicateList[k].time.getMonth()) &&
        (duplicateList[j].time.getDate() == duplicateList[k].time.getDate()) &&
        (duplicateList[j].time.getHours() == duplicateList[k].time.getHours()) &&
        (duplicateList[j].time.getMinutes() == duplicateList[k].time.getMinutes())
        ){
          duplicateList[j].temp = ((duplicateList[j].temp + duplicateList[k].temp)/2);
          duplicateList.splice(k, 1);
        }
      }
    }
  }
  return duplicateList;
}
```

Figure 4.13: mergeAndProcess function used in the WebApp

```
export interface Temperature {
    _id: string;
    sensor: string;
    time: Date;
    temp: number;
}
```

Figure 4.14: Temperature model interface used in the WebApp

```
splitAndSort(dataToSort: [Temperature]): [[Temperature]] {
  let splitList: [[Temperature]];
  let itemFound = false;
  dataToSort.forEach(element => {
    itemFound = false;
    if (typeof splitList != 'undefined') {
      let listLength = splitList.length;
      for (let i = 0; i < listLength; i++) {
        listLength = splitList.length;
        if (splitList[i][0].sensor == element.sensor) {
          splitList[i].push(element);
          itemFound = true;
          break;
        }
      }
      if (!itemFound) {
        splitList.push([element]);
      }
    }
    else {
      splitList = [[element]];
    }
  });
  return splitList;
}
```

Figure 4.15: splitAndSort function used in the WebApp

```
setLabelsAndData(labelListData: [[Temperature]]) {
  let labelList = [];
  let dataList = [];
  let toDate;
  let fromDate;

  labelListData.forEach(e => {
    e.forEach(f => {
      if (typeof fromDate == 'undefined') {
        fromDate = f.time;
      }
      if (typeof toDate == 'undefined') {
        toDate = f.time;
      }
      if (f.time < fromDate) {
        fromDate = f.time;
      }
      if (f.time > toDate) {
        toDate = f.time;
      }
    });
  });

  let minutes = (toDate.getTime() - fromDate.getTime()) / 1000;
  minutes /= 60;
  minutes = Math.abs(Math.round(minutes)) + 1;

  for (let i = 0; i< (minutes); i++) {
    labelList.push(fromDate.toLocaleString());
    fromDate = new Date(fromDate.getTime() + 60000);
  };
```

Figure 4.16: Part 1 of the setLabelsAndData function used in the WebApp

```
    labelListData.forEach((e, i) => {
      if (typeof dataList[i] == 'undefined' ) {
        dataList.push({
          data: new Array(minutes),
          fill: false,
          label: e[0].sensor
        })
      }
      e.forEach(f => {
        let newLabel = f.time.toLocaleString();
        labelList.forEach((g, j) => {
          if (newLabel == g) {
            console.log(dataList[i]);
            dataList[i].data[j] = f.temp;
          }
        });

      });
    } );
    this.lineChartLabels = labelList;
    this.lineChartData = dataList;
  }
}
```

Figure 4.17: Part 2 of the setLabelsAndData function used in the WebApp

The Temperature_Sensor directory contains a python script that acts as the temperature sensors that generates data for the system. This was originally intended to use an actual temperature sensor connected to a raspberry pi, but that was abstracted away

due to issues with the sensor and time constraints (more information on this later). This is technically considered "out of scope" of the system as the system specifications are that it is to provide functionality for the receival of data from sensors. It is included as an "example-data" provider to show how this can be done and to provide data for the system to manage and process. The "sensor.py" and "sensor2.py" (see figure 4.18 for part 1 and figure 4.19 for part 2) files are python scripts that is to simulate the temperature in the apartments mentioned in the scenario and are running on the raspberry pi's mentioned earlier. They have upper and lower temperature limits set and a dynamic function for determining the increase or decrease of temperature "measured" in periods.

```
import random
import time
import datetime
import json
import urllib.request

myurl = "http://127.0.0.1:49160/api/temp"

tempBot = 16
tempTop = 22

percentile = 0.10
direction = 0
timeToFlip = 30
timeToSend = 60

temp = 20
flipTime = 0
sendTime = 0
tempList = []

def myconverter(o):
    if isinstance(o, datetime.datetime):
        return o.__str__()
```

Figure 4.18: Part 1 of the sensor python script

```
while True:
    percentCheck = random.random()
    flipTime += 1
    sendTime += 1

    if percentCheck < percentile:
        newTemp = temp + (0.1*direction)

        if (newTemp < tempTop) and (newTemp > tempBot):
            temp = round(newTemp, 2)

    if flipTime == timeToFlip:
        somePercent = random.random()
        if somePercent < 0.33:
            direction = -1
        elif (somePercent > 0.33) and (somePercent < 0.66):
            direction = 0
        else:
            direction = 1

        flipTime =  0

    newObject = {
        "sensor": "temp-apartment-1",
        "time": datetime.datetime.now(),
        "temp": temp
    }
    tempList.append(newObject)

    if sendTime == timeToSend:
        req = urllib.request.Request(myurl)
        req.add_header('Content-Type', 'application/json; charset=utf-8')
        jsondata = json.dumps(tempList, default = myconverter)
        jsondataasbytes = jsondata.encode('utf-8')   # needs to be bytes
        req.add_header('Content-Length', len(jsondataasbytes))
        print (jsondataasbytes)
        response = urllib.request.urlopen(req, jsondataasbytes)
        sendTime = 0
        tempList = []

    print("This is the temp in Celsius:" + str(temp))
    time.sleep(1)
```

Figure 4.19: Part 2 of the sensor python script

Hardware

This section covers the hardware used in development and testing of the implementation. The devices outlined in this section was chosen to best fit the scenario with a dedicated computer in a server room and a raspberry pi for each "apartment" with temperature sensors attached.

The dedicated computer was set as a swarm manager node for the container orchestration and the two raspberry pi's were added to the swarm network as worker nodes (worker1 and worker2). All devices ran the docker engine and were connected to the same network. In addition to joining the swarm network as worker nodes, the pi's were running one of the temperature sensor scripts each outside of the swarm.

Hardware Specs Dedicated Computer: HP Pavilion 14-bf182no laptop

- Intel® Core™ i5-8250U processor
- Nvidia GeForce 940MX with 2 GB RAM
- 8 GB DDR4 RAM
- 256 GB SSD

Raspberry Pi: 2x Raspberry Pi 3 Model B

- SoC: Broadcom BCM2837.
- CPU: 4x ARM Cortex-A53, 1.2GHz.
- GPU: Broadcom VideoCore IV.
- RAM: 1GB LPDDR2 (900 MHz)
- Networking: 10/100 Ethernet, 2.4GHz 802.11n wireless.
- Bluetooth: Bluetooth 4.1 Classic, Bluetooth Low Energy.
- Storage: microSD.
- GPIO: 40-pin header, populated.

Container Orchestration

Purpose and Functionality The purpose of the container orchestration in the implementation is to provide workload balancing, significant scaling functionality and build/deploy automation. This is one of the core concepts of the architectural model and is important when considering smart city/neighbourhood/building scenarios. In the scenario, there are two devices available that has the primary function of measuring temperature and pass that along. This uses a fraction of the available resources they have and could

therefore be better utilized. By setting up a swarm network and adding these devices as worker nodes, they can provide these unused resources to the local system and help with scaling and load balancing. This makes it so that the swarm network can provide the basis for a much larger system than would be possible to host on either of the individual devices. In this example, a relatively average laptop can be used in unison with these other devices and provide a scaling system for a reduced cost when considering the alternative would be to buy a larger server to host it (or buy space on a cloud system).

Orchestrating the Swarm Network The container orchestration uses Docker as a base for all the containers and Docker Swarm as the implemented swarm network framework. This works by initializing a swarm on the first manager node (the swarm can have multiple managers as well as workers) and generating a token that can be used to join the network. Now all devices on the same network with the docker engine installed can join the swarm network as either a worker- or a manager node (as shown in figure 4.20). When all desired nodes are added to the swarm, this can be viewed by a "docker node ls" command (as shown in figure 4.22). This swarm network is then utilized by deploying services consisting of container configurations (image etc) and service configuration (number of containers to run of the type etc) to the swarm service stack. This service configuration is either provided as parameters to a "docker service create" command, or as a yml file (as shown in figure 4.21) These services are then created and the workload is balanced across the nodes available in the swarm network (as shown in figure 4.23). For the services of this implementation, this is shown through the figures below. All the containers running on the manager node is shown in figure 4.24, and all the containers running on the worker nodes are shown in figure 4.25 and figure 4.26.

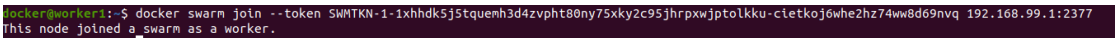


Figure 4.20: Joining the swarm as a worker node on a device

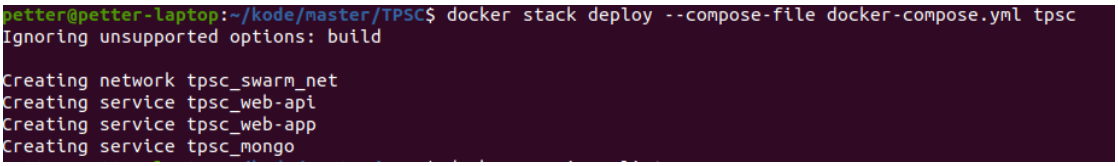


Figure 4.21: Deploying services to the swarm network using a yml file

Figure 4.22: Overview of all nodes running in the implemented swarm network

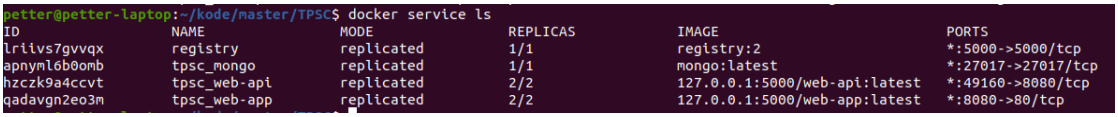


Figure 4.23: Overview of the services running in the swarm network after deployment

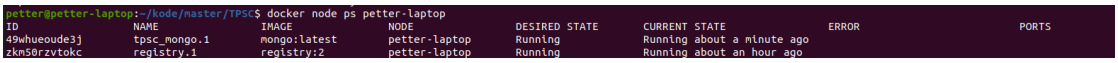


Figure 4.24: Overview of all containers running in the implemented swarm network on the manager node

Figure 4.25: Overview of all containers running in the implemented swarm network on the worker1 node

Figure 4.26: Overview of all containers running in the implemented swarm network on the worker2 node

API+ - API, Request Handler & Data Aggregator

Purpose and Functionality This component is a combination of several central concepts from the architectural model. Its combined functionality encompasses that of the API, Request Handler and Data Aggregator concepts. The component then provides functionality such as an interface to post and get data from the swarm, handling these request in a proper manner that forwards it to the right container, and data aggregation functionality. In essence, this container acts as an all-encompassing middle-layer between the swarm and external users/devices with all functions it needs built in.

Web Application

Purpose and Functionality This components purpose is to show an example of a user interface for the users to interact with the system and access the data managed by it. It is a relatively simple web application that sends a data request to the API for the sensor data managed by the swarm, processes the received data and formats it into a visual representation through a chart module. This component is built into the swarm in this implementation, but this can also be hosted externally with minimal changes as it only interfaces with the swarm through the API which is also served for external interfacing.

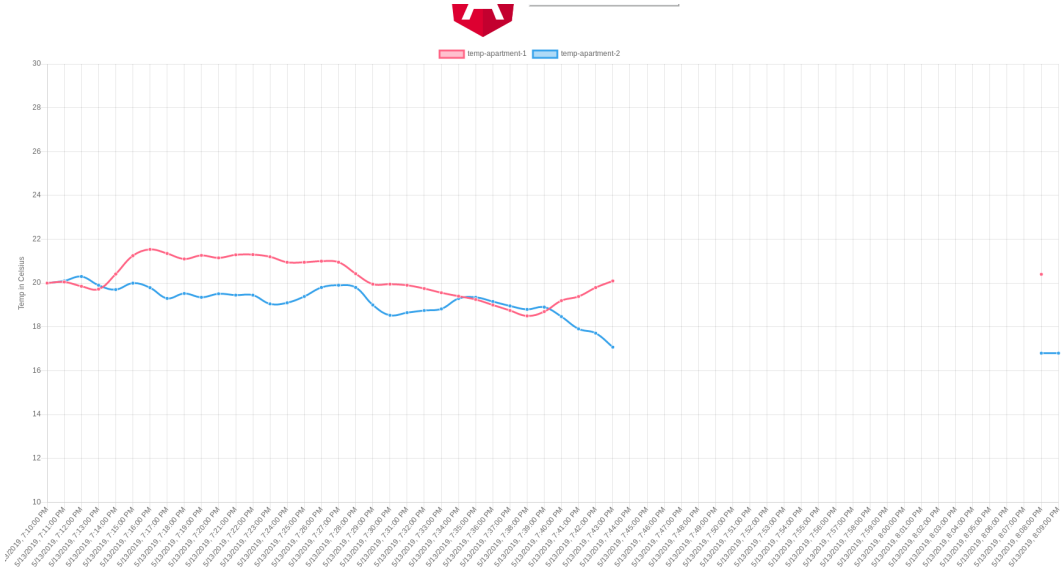


Figure 4.27: Image from Web Application with example data from two different sensors with a time-gap between measurements

Data storage

Purpose and Functionality The data storage components purpose is to provide storage opportunities for the swarm network operations as well as to provide persistence to the services. This component does not manage data aggregation, processing or requests as that is the purpose of other components (the API component in this case).

Temperature Sensors

Purpose and Functionality The temperature sensors is implemented using a python script that simulates room temperature of the apartments in the implementation and is to run on the raspberry pi's to serve this purpose. They are not a part of the swarm implementation as they are acting as external sensor data providers. The main purposes of these scripts are to provide relevant example data and to show how the implemented system can handle input from sources like temperature sensors.

4.4 Evaluation Metrics

In order to properly test the efficacy of the architectural model and the implementation of it, there is a need for objective ways of measuring how well they fulfill their role and solve the issues listed earlier. One way of doing this is to establish important quality

attributes (QA) for the system that is based on the requirements and focus points of the model itself. This section covers the definition of these quality attributes, while their selection, possible measurements and fulfillment by the model and implementation will be covered by the discussion.

4.4.1 Quality Attributes

Scalability

Scalability is a measure on how well the system can scale its own operations to meet both high and low workloads without impacting the performance of the system. This will be both how much they are equipped with functionality to handle these changes and how well it can handle a workload that is designed to stress-test the system or even to break it.

Modifiability

To what degree the system can be altered for the individual parts of the system without having to make changes multiple times. More or less to what degree of effort is needed to make changes to the system without having to alter large parts of the code.

Compatibility

To what degree the system is able to operate within its normal parameters on different devices without modification or alterations to the system or the devices.

Interoperability

To what degree the system is able to switch out parts of the system and their dependencies with others of similar function while still functioning in a similar manner.

Availability

How available is the system with regards to up-time, planned downtime and time to update the system. To what degree does the system plan for and handle potential factors that could result in downtime.

Maintainability

How well does the system support changes such as new business requirements, bug-fixes and addition of new features. How much effort is required to make changes to the system that normally occur over the life-cycle of the system over time. Also how much time is needed to restore the system after a failure.

Performance

Performance shows how well the system performs while within its normal operating parameters and how much load the system is able to handle before it slows down significantly.

Reliability

How well does the system continue to operate within its normal operating procedures. How reliable the system is itself while not under the stress of irregular factors affecting it.

Energy-Efficiency

In what manner does the system support factors such as energy-efficiency priorities and potentially battery capacity of the devices utilized.

Privacy

To what degree does the system facilitate privacy protection functionality and policies. How well the information flow and control is handled from a design and implementation perspective as to protect the privacy of actors involved in the system.

Security

In what way does the system implement and enforce security measures designed to protect the integrity, confidentiality and privacy of the system with regards to its internal operations and stored data. How well is the system able to defend itself from known and unknown security threats.

5 Discussion

5.1 Evaluating the Architectural Model

This section covers the evaluation of the architectural model according to the requirements set and the evaluation metrics outlined in the results chapter.

5.1.1 Fulfilling Requirements

Primary Requirements

Requirement 1: The system must be able to scale its computations based on resources available. This is fulfilled via the scalable container orchestration functionality that is present in the swarm network. All components of a system using the model are to be containerized and can then be scaled either up or down according to the needs of the system and the workload.

Requirement 2: The system must be able to continue its operation as usual within normal operating parameters (within reasonable limits) when adding or removing resources. The swarm network is automatically load-balanced according to the operating specification of the services that are set to run on the swarm network. When adding new resources to this network, they become available for load balancing and can be used by a manager node in delegating and distributing the workload on the system. If resources are removed from the network, the managers will detect a change in the number of containers running and correct that by running new instances of those missing containers on other available nodes.

Requirement 3: The system must be able to utilize devices with a lower amount of resources/computational capacity This is solved by the model allowing for building a number of smaller components that are small enough to be run on devices with less resources available. There is, however, an explicit need for the device to run on an operating system that is capable of running the containerization platform. As of writing there are no containerization platforms that are able to run on directly on motherboards or with minimalistic operating systems, so a certain level of operating system is needed on the device to be compatible with the model. An example of what is required is shown

in the compatibility matrix of the docker engine [51]. This is then mostly fulfilled with the current technologies available, but will be entirely fulfilled when containerization is able to run with less dependency on operating systems.

Requirement 4: The system must be able to receive and manage data provided from both internal and external sources Through the data management module of the architectural model, this is handled sufficiently. The model provides a Data API outwards from the swarm network where data can be received and the Request Handler then processes the request for storage in the proper data storage solution (most likely a database).

Requirement 5: The system must be able to support cloud functionality in the form of resources for storage and/or data management The architectural model supports cloud functionality in two possible ways. One is to handle requests for data through the API that a cloud system can access and send requests to, the other is to incorporate it into the system by handling the cloud functionality through the Request Handler component. By creating functionality in the Request Handler, requests are sent where the requested data is located, and this could then lead the request to a cloud system through this integration.

Requirement 6: The system must provide a platform for data aggregation and/or data pre-processing functionality This can be solved in several ways through the architectural model. One way, is to provide it as a part of the data lifecycle management components, that can aggregate and process data based on the age of the data if relevant. Another is to build it into the Request Handler that processes and aggregates the data received in a POST request before storing it, or to process the data from a GET request before sending it back.

Requirement 7: Must provide functionality to forward data "upwards" in the network. Either to another "fog" system or to the cloud This can be solved in different ways in the model. It can provide an interface in the API to handle data requests from a cloud or fog system and send the data that it requires. The data lifecycle management can also provide this functionality by monitoring the data stored in the swarm and then send this data further depending on the needs of this data flow.

Requirement 8: The system must provide support for data life-cycle management In the Data Management module, there is a section that covers the data lifecycle management. According to the different needs of the data lifecycle, it can either be processed on requests or triggered by the age of the data by a monitoring data aggregator component. Data processing on requests as an example cloud be that data stored within the

system must be processed in a specified way e.g: sensor data must be processed from average measurement from every minute to average measurement from every hour when the data is requested by a cloud system for historical data storage. This data life-cycle management is then handled by the Request Handler. When data processing is triggered by the age of the data e.g: the system is to process and send sensor data that has an age of over 2 weeks. This is then handled by a data aggregator component that monitors the database.

Requirement 9: The system must provide an interface to retrieve the data managed by the system This is realized through the Data API provided by the model. This API can handle GET requests sent to it and provide the data requested through the Request Handler in the swarm.

Secondary Requirements

Requirement 1: Security To address the security of the model the possible weak points of the model must be addressed. An overview of potential weak points in the model and how it is mitigated would suit this purpose nicely. The nodes and containers in the swarm network normally does not expose any entry-points or ports outwards of the swarm network unless specified. This makes it easier to control the entry-points for systems using the model, as all of them are deliberately exposed through the definition of the services running in the swarm. Normally the only exposed entry-point in the model is the API itself as the rest are strictly only for container-to-container and node-to-node communications. This means that as long as the requests received through the API is sanitized and verified in the Request Handler, then most security concerns should be mitigated.

Requirement 2: The system should be able to utilize an array of commonly available hardware/devices The versatile nature of containerization and container orchestration should mitigate any concerns tied to usage of specific technologies, frameworks or platforms as long as they can be built into a container. The only limitation put on the usage of devices is that any device to be added to the swarm network as a node must be able to run the chosen containerization platform. Any limitations here are then tied to the compatibility of that containerization platform and not the model itself. At the time of writing, there is little support for containerization that is not dependent on an operating system and the support for obscure or highly specialized operating systems are lacking. The docker compatibility matrix [51] shows compatibility with a wide range of different operating systems, but may not offer compatibility with specific operating systems used on some embedded devices. This requirement is therefore partly fulfilled, with some minor limitations to obscure devices or devices without the ability to run a compatible operating system until they are supported.

Requirement 3: The system should be able act accordingly to the different energy-efficiency needs on embedded devices Through the tiered-priority workload management, the workload balancing of the swarm network is prioritized where systems with a higher energy-efficiency focus have a lower priority than devices without. This ensures that the system would utilize devices with energy-efficiency focus on a lower frequency than the other devices. The devices of a higher tier of prioritization would then need to be utilized fully before the lower-tiered devices would be available for selection. This results in a more balanced approach where energy-efficiency focus is maintained.

Requirement 4: Privacy The privacy of the identities connected to the managed data can be maintained through proper authorization of the requests sent to the API. The Request Handler can authenticate requests sent through the API and ensure that the data is only handed to clients that have the proper authorization for the data that is requested. This authentication can be made with the type of data requested in mind and ensure that the data have different levels of authorization connected to them and better ensure privacy and integrity of the system.

5.1.2 Assessing Quality Attributes

Scalability

It is difficult to provide an exact measure on how well the model and system will scale to rising demands, as this is often tied to usable resources and hardware specifications. The best way to "measure" this system is to examine the tools the system has available to mitigate and handle rising workloads, and also how the system handles lower amounts of workloads. There are two ways to improve scalability: Vertical: To increase, we add more resources, such as memory, disks or processors into one system. Horizontal: We increase the number of computing units and divide the load. The key indicators for measuring this attribute are:

The model supports scaling both vertically and horizontally through the nature of swarm computing through container orchestration. It is able to scale vertically seamlessly through the workload balancing of the system. This is done by the manager nodes being able to utilize these new resources as soon as they are available in the swarm network. The process of adding new nodes to the swarm network is easily achieved by installing the container orchestration platform on a device and adding it to the swarm network as shown in figure 4.20. It is a simple method completed in one command when utilizing docker swarm as an example. Horizontal scaling is done through the load balancing function of the swarm itself. The manager nodes will automatically scale to rising demands by allocating new resources available in the swarm network and creating new instances of the containers as needed.

The model can therefore be considered to have a high degree of scalability when factors

such as hardware is excluded from the equation.

Modifiability

The model is highly modifiable through containerization and container orchestration. Containerization supports all programming languages, frameworks and platforms as long as they can be bundled properly into a container. This means that a component can potentially be removed in its entirety and replaced with another component comprised of entirely different programming languages and dependencies without changing the rest of the system as long as it serves the same functionality as the component it is replacing. It is also easy to update and make changes to a container by changing the code and/or dependencies and then building a new version of that image that the container uses. This can then be applied to the service using that type of container by performing a rolling update to that service using docker swarm as an example [52]. The service is then being sequentially replaced by containers of the new image version with no downtime and no loss of work as the containers are not replaced with new ones until they have completed their tasks.

This makes the model highly modifiable with the specifics of how modifiable the system is to become is up to the developers utilizing the model.

Compatibility

Compatibility when referring to programming languages, frameworks and platforms is almost universal when applying containerization. If the combination of code, environment and dependencies can be built and bundled by the containerization platform, then it can be applied to this model. Compatibility when referring to devices it is run on, is a bit different due to some technological limitations on available containerization platforms. This is due to the available platforms like docker is not universally compatible with all operating systems and have some requirements set. The compatibility matrix of docker [51] shows a high degree of compatibility, but with restrictions to obscure and customized operating systems that could potentially be found on some embedded devices and micro-controllers.

The model is then highly compatible on a general level, with some reductions to compatibility with the smallest and older embedded devices.

Interoperability

As mentioned in the evaluation of the modifiability of the model, containerization enables the components to be replaced with other components of similar/same functionality. The degree of similarity required between the components depends on the implementation due to possible dependencies being in place from container to container, and therefore

requiring the same functions. Other than that, there are no inherent limitations set in the model as the containers can act as entirely independent units.

This results in the model having a high degree of interoperability only being potentially hampered by limitations set by the implementations themselves.

Availability

The container orchestration present in the swarm handles all faults and crashes that can happen in the network. If a container is crashing, the manager nodes will try to restart the containers according to the "restart_policy" defined in the service specification of the affected containers as written in the docker compose reference [53]. If a node crashes or is disconnected, the containers running on that node will be moved or restarted on other available nodes if there are enough resources available. This results in a high level of fault tolerance that will only truly result in the system being down if all the manager nodes have crashed or is disconnected from the swarm network. This can also be mitigated by increasing the number of manager nodes and distributing them to different devices and on different network connections as explained here in the docker documentation [54]. Other than this, the container orchestration system is capable of rolling updates as explained in the modifiability evaluation of the model. It is then possible to update the system without having downtime.

All these different factors leads to the model having a high level of fault tolerance and no need for downtime for updates, resulting in a high degree of availability.

Maintainability

The system uniformity of the containerization platform means that if the system can be built on your development devices, it should function the same way in the production environment without much alteration. These changes can then be applied to the system through the rolling updates as explained earlier without introducing any downtime to the system. That means that the model supports the ability to not only make changes that should affect the production environment quickly without much need for testing it in production and also being able to stage these changes without downtime. It is then easy to change the system and introduce bugfixes.

These factors in addition to the modifiability explained earlier, leads to the model supporting a high degree of maintainability.

Performance

Performance is difficult to discuss without factoring in how the model is implemented or what hardware it has to run on, so this will be a short description of what design choices

in the model that can factor into performance. The most relevant part of the model to discuss in this context is the models ability to utilize the resources available to boost the performance of the system. This is achieved through the container orchestration as it can create new instances of containers on nodes with available resources. Assuming that the implemented system has components with containers of varying sizes, this should result in the swarm network utilizing most (if not all) of the available resources. This is due to the resource demands of the new containers will vary due to the differing sizes of the new containers. It is important to note here that the actual performance of the system varies much depending on how it is implemented and what devices the swarm network consists of. If the system is comprised of few large components on a swarm network with many small devices, this can lead to a poor usage of resources as the remaining resources on the devices can't be used for a new instance of the container as each container requires too much.

In total, the model provides adequate functionality to utilize available resources and increase performance, but is otherwise lacking as the resulting performance is mainly decided by the implementation and the developers.

Reliability

The fault tolerance and availability discussed in the availability evaluation of the model affects reliability as well as these other factors. The ability of the system to have high availability and high fault tolerance directly translates to a high level of reliability. The container orchestration strives to maintain what is described as the "ideal state" of the swarm network that is defined through the services running on it. This leads to the swarm network not running unused scaled up containers and will always focus on returning the swarm network to a state that is designed to handle the normal operating parameters of the system. That, in addition to the earlier mentioned factors, results in the model having a high degree of reliability.

Energy-Efficiency

The model is to utilize a tiered-priority workload management that sets a priority to devices depending on the resources of the device and energy-efficiency priorities of the device added to the swarm network. As an example of this priority setting: battery-powered device would have a low priority and dedicated server machines would have a high priority. This means that devices that have available capacity would be differentiated between its total level of resources and energy-efficiency priorities when selecting a node to run a container in. It is then more feasible to utilize devices that are more focused on running efficiently with regards to energy-usage without having the regular randomized selection picking them at a higher frequency than more suitable devices.

This feature of the model maintains the energy-efficiency priorities of the devices it is

to utilize and could therefore be considered energy-efficient.

Privacy

The model has some functions in place that makes it easier to maintain privacy through authentication and verification that can be built into the Request Handler of the model. The only external entry-point in the model is through the Data API that sends requests to the Request Handler for further handling. By authenticating the requests for data and verifying that they should have access to the data they are requesting, it becomes easier to guarantee the confidentiality of the data when it can be identifiable.

Other than this, the model does not have any extra functionality in place for privacy. Therefore it can be considered to have a certain minimum of privacy protection, but anything further than this is up to each implementation.

Security

The model has some advantages when it comes to security such as only one external entry-point to the system through the Data API. The network feature utilized in the swarm network is not externally exposed and as such all ports and communications inside the swarm is not accessible from outside the swarm network unless specified. This makes it easier to build a secure system due to only needing to focus on the singular entry-point when considering authorization, validation and sanitizing incoming requests. All functions of the API is explicitly defined and is handled by the Request Handler afterwards. It is then immune to common security concerns such as SQL injection and buffer overflow due to this abstraction and no direct line of communication being available to the database. The API is capable of utilizing HTTPS connections thereby enabling secure connections to and from the system. When considering DDoS attacks, this can be handled by keeping track of incoming requests in the Request Handler and utilizing packages in the API to block and prevent such attacks [55].

The utilization of an API and a Request Handler for all requests made to the system puts several layers of abstraction between the data storage and the outside requests. This makes it inherently more secure and enables the system to implement security measures in these abstraction layers.

5.2 Evaluating the Implementation

This section covers the evaluation of the implementation of the architectural model. Most of the evaluation of the architectural model is relevant for the implementation as well due to the implementation being a realization of the model that follows it as a blueprint. Therefore, this section focuses more on where it differs from the model and is

more closely tied to the specific architecture in relation to the scenario. Keep in mind that some design choices in this implementation were made to reduce the work required to create the implementation. This is mainly due to the implementation serving as an example, and also being made with one person using some unfamiliar technologies and frameworks.

5.2.1 Fulfilling Requirements

Primary Requirements

Requirement 1: The system must be able to scale its computations based on resources available. The system utilizes docker swarm for the container orchestration, which includes functionality for scaling the implementation. The defined services run across the nodes in the network and scale up as needed by allocating and utilizing available resources on nodes in the network.

Requirement 2: The system must be able to continue its operation as usual within normal operating parameters (within reasonable limits) when adding or removing resources. The implementation used three nodes in the network where two are worker nodes and one is a manager node. The nature of the swarm functionality of the docker engine makes it easy to add or remove nodes to this network with a simple join/leave command (as shown in figure 4.20). As long as the resources removed are not of the manager node or a new manager node is added beforehand, then it should not be a problem to add or remove resources.

Requirement 3: The system must be able to utilize devices with a lower amount of resources/computational capacity The system utilizes raspberry pi's as worker nodes without any issues. They are considered lightweight computational devices and are representative to devices of similar size available as IoT devices. As long as the device is capable of running an operating system compatible with the docker engine [51], then there is no issue to add it to the network. Other than this the system has few components, but they are of varying sizes. This makes it so that even small devices can run the containers with the smallest requirement of resources. As an example, the laptop mentioned in the hardware section of the implementation was capable of running two docker machines [56] (local VMs running the docker engine) and using them as worker nodes in addition to already being a part of the swarm as a manager node. That laptop alone was capable of running the entirety of all the containers divided between itself and its own hosted virtual machines.

Requirement 4: The system must be able to receive and manage data provided from both internal and external sources The Data API served through Express on

Node.js is capable of receiving data from multiple sources. The management of that data is handled through the Request Handler and Data Aggregator situated in the same component as the API. This combined component then handles both the receival and management of data in cooperation with the Data Storage component that stores the data.

Requirement 5: The system must be able to support cloud functionality in the form of resources for storage and/or data management The system in its current state is only capable of supporting cloud functionality externally through the Data API and not as an internal part of the Request Handler. This could be supported, however, through an extension of the Request Handler that connects it to the cloud service. If the Request Handler is extended in this way, it could then do a check on the request and forward it to the cloud when appropriate.

Requirement 6: The system must provide a platform for data aggregation and/or data pre-processing functionality Through the Data Storage and API+ components this is achieved. The data is received through the API, aggregated and processed through the Data Aggregator and stored properly through the Request Handler and the Data Storage component.

Requirement 7: Must provide functionality to forward data "upwards" in the network. Either to another "fog" system or to the cloud This is partly supported through external requests only. The cloud can request the data that it needs through the API and then manage it further as it sees fit. Internally that functionality is not developed as a part of the Request Handler (mostly due to time constraints), and would need to be added later on. The system is fully able to support this functionality but is not added as the system is now.

Requirement 8: The system must provide support for data life-cycle management The data life-cycle of the implementation was selected as a simple life-cycle that is managed through triggered events. The data was to be transformed at receival and pre-processed when requested before sending the data. The simple transformation was to process the data received from measurements every second to aggregated averages of measurements every minute. This is done in the Data Aggregator part of the API component. The pre-processing on requests are done in the same Data Aggregator.

Requirement 9: The system must provide an interface to retrieve the data managed by the system The Express API provides GET methods that retrieves data from the Data Storage. The GET function as it is now is a simple function that retrieves the data managed as a whole. This can (and would have been) be expanded to query based on

measurements from and to a specified time to improve the usability of the data retrieval. It was simplified to query the data as a whole due to time constraints as the design of the web application is not what the implementation intended to showcase.

Secondary Requirements

Requirement 1: Security At the time of completion, the implementation has not utilized much extra security counter-measures other than protections related to injections and overflow. The system was supposed to be run and accessed locally to serve as an example of an implementation and has therefore less need for strong security. It lacks security measures connected to authentication and verification of access as any malicious attacks to it would require to be hosted externally and not locally. This was deemed as a necessary time-saver as the exclusion of these simple mechanics saved time in the implementation and does not lessen what the implementation aimed to show. The fix for this is relatively simple to implement and is a fully supported functionality in the implementation. Authentication can be solved by adding authentication and verification requirements to the Data Storage component and then adding username and password to the query sent to the API. The implementation does however protect against injections and overflow attacks through the extra layers of abstraction between the client and the data (API and Request Handler). This indirect form of communication with the database increases the integrity of the system as a whole and eliminates some of the more common forms of attack such as injection attacks (SQL injections etc.). In addition to this, the Request Handler sanitizes the request by assigning typing and schema-modelling (through Mongoose) before sending it to the database.

Requirement 2: The system should be able to utilize an array of commonly available hardware/devices As discussed earlier, the system is able to use a wide array of hardware/devices as long as they have the ability and capacity to run an operating system that is compatible with docker [51]. Many devices are capable of running a simple lightweight linux distribution and this guarantees compatibility with the system through the docker engine.

Requirement 3: The system should be able act accordingly to the different energy-efficiency needs on embedded devices Tiered-priority workload management was supposed to satisfy this requirement. It was very troublesome to implement due to lack of support in the platforms and current technological limitations, so it was not added to the system. There is no support for customized node selection in the load balancer of the manager nodes and would therefore need to be implemented either from scratch or to build it into the docker platform. This is sadly beyond the scope of the thesis and could rather be done as further work (more on this in the further work section of the conclusion).

Requirement 4: Privacy As mentioned earlier, privacy protection was deprioritized due to the localized nature of the implementation. As no external network communication is expected, there is little need for verification of access to data that is generated as example data for use in the system-context alone. The Data Storage component does however support this feature in the database and could easily be implemented if needed.

5.2.2 Assessing Quality Attributes

Scalability

The scalability of the implementation is affected by the same factors as the architectural model. It is easy to add new nodes to the docker swarm network through the "swarm join" functions and this enables vertical scaling. Horizontal scaling is managed by the manager nodes in the swarm network and will create new instances of the containers of the different services as demands rise.

Modifiability

The implementation consists of three different components in the swarm (Web Application, API+, Data Storage). These components can freely be modified by making changes to the code, its environment or dependencies. Then these changes are made by building new images that includes the changes. These new images can be rolled out the the components by pushing the new images to the image registry service in the swarm (shared collection of images to be used in the swarm that is accessible by all nodes in the swarm) and issuing a "service update" command to the swarm network. This can be a bit tricky the first time, but this process can also be automated into an "update script" that issues these commands for you for increased ease of use.

Compatibility

The implementation is compatible with all devices that have the ability to run the docker engine. If the device either has or is capable of running an operating system that is compatible with docker (see the docker compatibility matrix) [51], it is guaranteed to be completely compatible with the system.

Interoperability

The implementation has three types of components that run in the swarm network. These components could potentially be exchanged for other components of similar functionality without much issue. The web application component can freely be exchanged as it is only reliant on functionality available in the API and no other component is reliant on the

functionality it offers. The Data Storage component can also be exchanged for any type of storage component without much issue. The only change necessary to exchange that component is to make changes to the Request Handler so that it uses the proper form of communication (it uses Mongoose currently which is only for use with MongoDB). The API+ component can also be exchanged with another as long as it serves the same API functionality (GET and POST for /api/temp HTTP requests) and communicates with the Data Storage in the same way.

Availability

Due to the size of the swarm network, it has only one manager node. If the machine running the manager node is disconnected or crashes, there will be noticeable downtime as the system must be restarted or reconnected. This makes the implementation have a low amount of fault tolerance if it is the manager node. Other than this, the other nodes can be disconnected or crash without any downtime as the manager node will reschedule the containers on other available nodes instead. In total, this will result in a relatively high availability which can be increased by adding more nodes as managers to the network.

Maintainability

The factors mentioned in the modifiability evaluation of the implementation is the same for maintainability. The same process of updating the containers seamlessly makes it easy to introduce bugfixes or changes to the system without introducing downtime or other issues.

Performance

The implementation has both the Web Application and the API+ components running in parallel in the swarm by default deployment. They are both relatively lightweight devices and are able to run without difficulty on all devices in the swarm network (including the raspberry pi's). This makes it easier to utilize all the resources that are available to their full extent and therefore providing a good basis for the performance of the system. Other than that, some data processing is done by the clients as the web application needs to process the data to a usable state for the chart visualizations. That further decreases the total load on the system itself and therefore makes it able to handle more requests than it normally would.

Reliability

The ideal state of the system is defined through the "docker-compose.yml" file and is to function as the default state of the system that it will revert to whenever feasible. It comprises of the three types of components (Web Application, API+ and Data Storage) where both the API+ and Web Application components have replicas running in parallel. These handle most of the workload set on the system by requests and makes the system very capable of functioning at normal operating procedures.

Energy-Efficiency

The implementation sadly does not have tiered-priority workload management implemented due to difficulties and lacking support in the technologies and frameworks available. This does mean that the system is not able to distinguish between the devices that the swarm comprises of and will use the default workload management of Docker Swarm that selects nodes at random. Some restrictions have been set to the system such as the Data Storage only being able to run on the manager nodes, and does make a bit of a difference as long as manager nodes are only set on devices without much need of energy-efficiency priorities.

Privacy and Security

The implementation in its current state is only meant to be served locally to show the efficacy of the architectural model. It is therefore not needed to defend against the same form of malicious attacks that it could be subject to on a hosted online service that is exposed to the internet. This means that normal security measures such as authentication, verification and encryption/decryption is not included in the implementation. It does have the ability to include these features, but the time to implement them would be significant and therefore deemed excessive for this local implementation. These features can be added to the system to prepare it for internet hosting by enabling them in the database, using authentication in requests, verifying authenticated access to the data, encrypting/decrypting requests and enabling HTTPS in the API. The implementation does, however, have security measures in place with request abstraction between the client and database and the sanitizing of request content. These security measures does protect the system from common attacks such as injection- and overflow attacks.

5.3 Architectural Design Choices

5.3.1 General Choices

The implementation is made with the scenario in mind and its scope is therefore limited to a smart building system using three devices with two temperature sensors sending data to the system. This has resulted in some deliberate deviations from the architectural template as presented in the results chapter under the implementation section. Since the scale of the implementation is on the smaller end of what the architectural model is capable of, the swarm network has been reduced to three components that together provide all the functionality the swarm is to have. These components can be divided into several other components when the system is to function at a larger scale and the implementation would then be a lot more similar to the one presented as the template.

5.3.2 Container Orchestration

Choice of Technologies and/or Frameworks

For the implementation, Docker Swarm was selected to provide the container orchestration functionality and Docker for the containerization. Docker is the standard choice for containerization and Docker Swarm is a solid contender for container orchestration.

Originally, Kubernetes was considered for this functionality, but that framework proved a bit troublesome for a local distributed implementation as it is primarily designed for use on a cloud platform. After some trial and error, the decision was made to switch to Docker Swarm due to networking troubles especially. Docker Swarm is a simpler framework as Kubernetes tries to be a more all-encompassing alternative.

Priority Assignment

Tiered-priority as envisioned in the architectural model proved to be troublesome to implement due to technological limitations in the frameworks available. Container orchestration is currently designed mostly for use in cloud- and large enterprise systems due to technological trends favoring cloud computing utilization. In this context, there is little need for a prioritized selection of nodes as most are equal in terms of resource-availability and energy-efficiency priorities and would therefore be best served with a randomized selection. For use in smart cities/neighbourhoods/buildings on the other hand, this is not the case and important for optimal implementations. These limitations can be mitigated through either developing it into the frameworks, or making a plugin either for Docker Swarm or Kubernetes as they are open-source frameworks. This however, would require significant insight into the inner-workings of the frameworks and a great deal of effort to develop. Therefore priority assignment in node selection is rather

a subject for further work in this field.

5.3.3 API+ - API, Request Handler & Data Aggregator

This component is a combination of the functionality of the Data API, Request Handler and the Data Aggregator modules of the architectural model. This was done mainly because of the size and scope of the implemented system, and also to show that multiple concepts can be combined into a single component and is not required to be its own component. For a larger system, however, there are merits to having each of the concepts as their own component to better provide scaling on the exact components where it is needed, and not having to waste resources. This system is on a more distributed scale and can therefore function with fewer larger components.

Choice of Technologies and/or Frameworks

The API uses express on top of node to serve its interface. Express is a well-known web application framework that is often used to serve applications of varying scale. It is a minimalist framework with simple functions for building services such as an API used in this implementation. The other concepts were built into this express framework implementation by providing the functionality built into the API as integral functionality called in the POST and GET functions displayed earlier in the project structure section. The Request Handler is represented through the API by the requests being forwarded to the correct container of the swarm network, and the Data Aggregator is present through the data processing scripts called in the API (such as "temp_average_minute.js").

The technology and frameworks utilized in this component can freely be exchanged with other of similar functions as long as it is able to serve an API that handles HTTP/HTTPS POST and GET requests. If Node.js is still to be used as a base, alternatives like Nginx [43], Koa [57] and other similar node-based web frameworks could be used instead. If a more homogeneous programming language selections with the raspberry pi's is wanted, Django [58] could be used instead as it is a python-based web framework with similar functionality to Express.

5.3.4 Web Application

Choice of Technologies and/or Frameworks

The web application is built using Angular and served using Nginx with node as a base. These technologies are easy to use in combination due to the common reliance on both node and npm. Angular is a popular web application framework and was selected mainly due to familiarity for development purposes, but can easily be exchanged for an implementation of another node-based web application framework (react, vue.js etc.).

All these frameworks have their own pros and cons and the choice between them usually boils down to personal preferences. The choice of utilizing Nginx to serve the application was mainly due to ease of use. As Angular provides its own routes etc. as part of the build, only a simple server framework is needed to provide the entry-route to the routes managed by Angular. After this, angular takes care of the rest. For this reason, Nginx is a good fit due to the ease of use and short time to implement. Due to Angular's reliance on node.js, it is a good idea to keep the server based on the same framework. If another technology is desired, then the web application framework should be exchanged as well as to not have unnecessarily many dependencies.

5.3.5 Data Storage

Choice of Technologies and/or Frameworks

MongoDB was selected as the database technology and framework for this implementation. It is a NoSQL (Not only SQL) document database that stores each entry as a document with keys and values. It is a widely adapted technology and is suited as a storage solution for this implementation due to its scalable nature. This is different from a SQL database that is inherently relationary and focuses on how each data entry relates to the other data. Temperature measurements from sensors do not require relations to each other and would therefore be suited for a document database implementation. As the implemented system is to store recent data only, then there should be no major differences in utilizing other data storage solutions such as MySQL or PostgreSQL with regards to scale.

Data Storage in Implementation

Managing persistent data storage in a swarm network is difficult due to several factors relating to the nature of swarm computing. The containers that make up the swarm network can be situated on any node in the network and can potentially be replicated multiple times. All the containers running on different nodes will have separate storage spaces and will therefore have different sets of data depending on the workload management of the system. When restarting the data storage containers, they are not guaranteed to be situated on the same node as previously. This needs to be accounted for and can be solved in several different ways:

1. Restrict data storage service placement to manager nodes. This is the solution utilized in the implementation (mainly due to time constraints and little familiarity with persistent data storage in container orchestration). It works with one master manager node as this guarantees that the container will always run on the same node, as node roles are not meant to be changed. This is by far the easiest solution as it circumvents the entire problem by making the container deployment deterministic. It is not the best solution, however as this restricts the number of

manager nodes of the swarm network.

2. Shared volume for the containers. Volumes are a mechanism used in the Docker Engine that persists data generated by the containers [59]. Shared volumes are volumes that are accessible for other containers in the swarm. This can be done either by binding mounted volumes or by installing extensions for Docker that can handle file transfers between the volumes. Docker Swarm has this functionality, but it is tricky to implement properly. Kubernetes has a better implementation of the volumes and makes sharing them in the swarm network a lot easier [60].

By using shared volumes, the Data Storage implementation can also be properly scalable by using replication of the database solutions. Replication is a feature for databases that uses multiple replicas of the database and uses load balancing methods between them to function as one singular larger database. This is a feature available in several databases such as PostgreSQL [61] and MongoDB [62]. By using the replication functionality of the database system, each replica can be set into its own container and load balanced across the swarm network. There are several guides on how to accomplish this both for Docker Swarm [63] and Kubernetes [64].

The Data life-cycle

The life-cycle of the data used in the implementation is a simple form of life-cycle. It is generated as real-time data on the raspberry pi's and then "converted" to recent data when it reaches the system. The temperature sensors measures the temperature every second and sends a batch of these to be processed at given intervals (every 5 minutes etc). The Data Aggregator module of the API+ component then combines and takes an average of the measurements for every measurements in the same minute. The system then keeps this data for up to a few weeks before it is removed removed from the system. This is where it could potentially send it to a cloud storage for storing of historical data, but that was not added due to time constraints. This could be modified in the Request Handler to schedule transfers of data triggered by the age of the data stored by the Data Storage component.

5.3.6 Temperature Sensor

Choice of Technologies and/or Frameworks

Python is a versatile and widely used programming language that is easy to use and quick to implement a script in. Linux-based operating systems provide native support for both python development and running scripts and is therefore a suitable option to use in this implementation. The raspberry pi's commonly runs on the raspbian operating system (a configured version of the linux-based debian operating system) and it is therefore a natural choice to use python when implementing simple scripts to run right on the device.

Various implementation choices

Temperature Sensor Simulator vs. Actual Sensor

Originally the implementation was going to use DS18B20 temperature sensors connected to the GPIO-pins on the raspberry pi's to measure data (as shown in this link: http://www.circuitbasics.com/raspberry-pi-ds18b20-temperature-sensor-tutorial/). It was then planned to use python to read the values of the temperature sensors as temperature measurements and send it to the API at given intervals. This was changed to only using pure python both for measuring the temperature and to send it to the API. It was mainly due to difficulties with the temperature sensor provided with the equipment. The script was ready and followed several guides on how to properly use the DS18B20 temperature sensor and raspberry pi's together, but it did not work as intended. The raspberry pi's were not able to read the value from the sensors and it is unclear whether that was due to faults with the pi's or the sensors. After a few days of trying to get this working, the temperature sensors were abstracted away with a python script that simulates temperature measurements from the sensor instead. Due to the temperature sensors mainly being added to show how the system is able to receive and manage data, this was deemed as an adequate compromise.

The images below are from the attempted connection and reading of the temperature sensor.

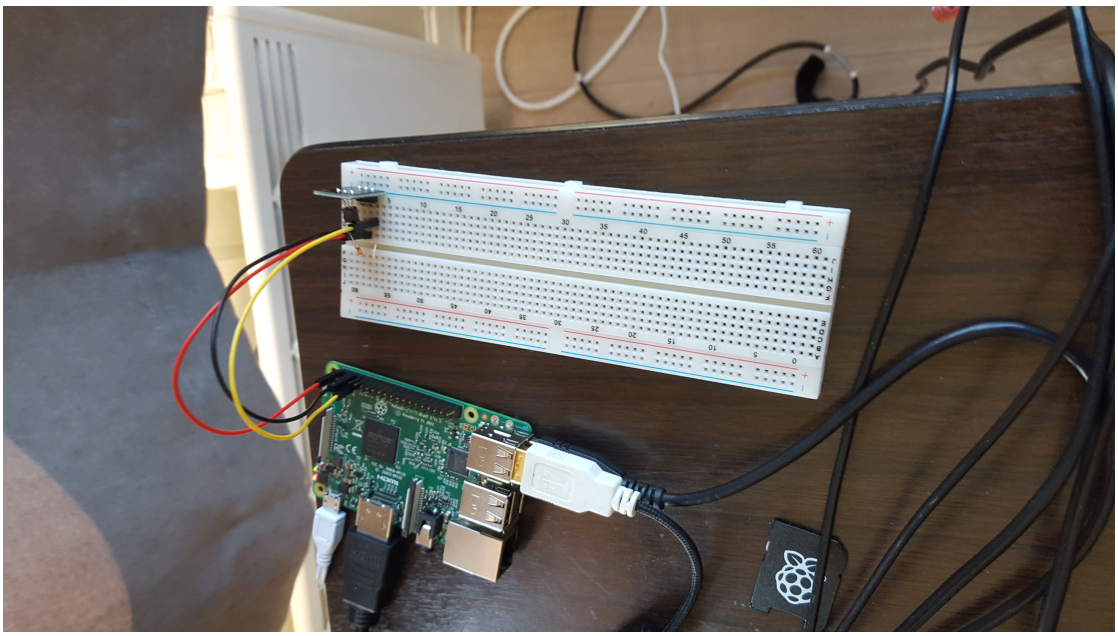


Figure 5.1: The raspberry pi connected to a breadboard with the temperature sensor

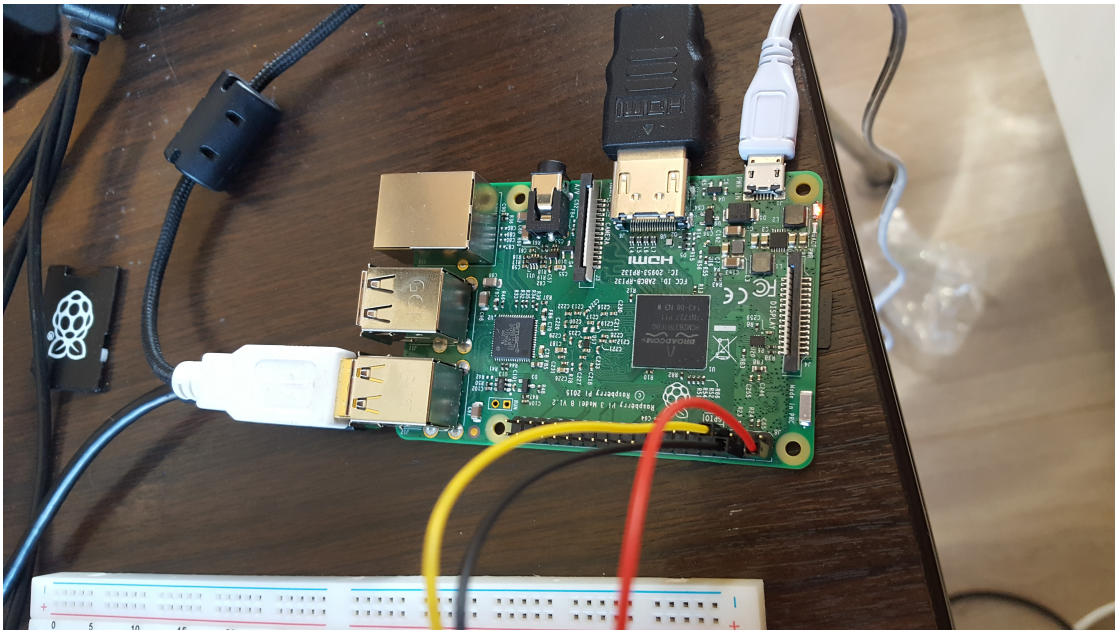


Figure 5.2: The raspberry pi with connected wires to GPIO-pins

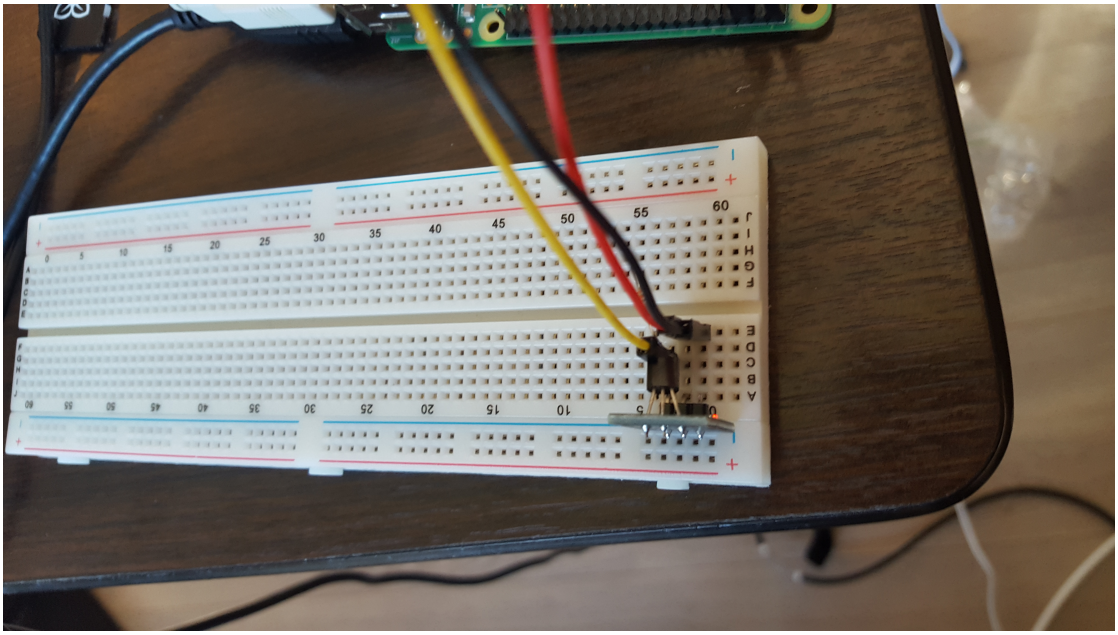


Figure 5.3: The breadboard with the sensor and connected wires

5.4 Contribution

This section covers the scientific contribution of this thesis as a whole for the area of study and the different parts involved. The contribution is split between contributions for the field of smart city ICT-services/applications and the ZEN research centre.

5.4.1 ZEN Centre - F2C Data Management

The ZEN centre has done a wide variety of work into the field of smart zero-emission neighbourhoods. One of the papers submitted by the centre covers the flow of data from distributed sources to centralized data management as outlined earlier in the Background chapter of the thesis [8]. The thesis shows how this form of data management relates to general applications and services in a smart environment at different scales and degrees of centralization (smart buildings-> smart neighbourhoods -> smart cities). It shows how applications and services using the model can either manage the entirety of the data flow (as shown in the architecture template example in figure 4.5) or manage parts of it (as shown in the architecture of the implementation in figure 4.6). This should provide insight into better practices when developing ICT-services/applications that has the ability to facilitate D2C data management either fully or partly.

5.4.2 General Contributions

Other than the contributions for the ZEN centre, the thesis shows the viability of container orchestration and swarm computing for smart city ICT-services/applications. The architectural model provides the blueprint and design insight with the example implementation to properly develop and implement systems that would function well in this context. The architectural model can be applied to systems for smart buildings, smart neighbourhoods and smart cities without compromising the core concepts of the model. This versatility makes it a good starting point when designing systems to function in this context and could help in setting new standards for development of services and applications in smart cities.

5.5 Reflections

This section covers reflections about the work, the process, lessons learned through the thesis and what could have (and perhaps should have) been done differently. This is purely subjective as the writers viewpoint into the thesis as a whole

5.5.1 Reviewing the work process

Originally, the problem and setting for the thesis was vague and needed much clarification in both problem definition and the execution of a solution. The area of research was relatively unfamiliar and demanded a lot of work to gain the proper amount of insight needed. Smart cities, neighbourhoods and buildings is a relatively new area of research where a lot of work is either in progress or completed as a pioneering study. There are a lot of discussion about what it means when discussing smart cities and that did not make it easy to understand at all. With a good amount of patience and taking the time needed the insight needed was finally gained and work could begin in earnest. It took a lot of scientific reading through the literature review and relevant articles supplied by the co-supervisor to get the background research in place. After this, the process became a lot more smooth as more time was spent on doing actual work and not only the background. It was here that the idea cam together of using container orchestration in a smart city scenario using a combination of dedicated- and IoT-devices. The problem was that the deadline for the thesis was not something far off in the distance, but rather something a few months down the line. This unbalanced workload management of the thesis lead to the last 4-5 months being a lot more intense than they had to be.

5.5.2 Reviewing the work as a whole

All in all the quality of the thesis is satisfactory when considering the starting point of the thesis. After the background research was finished, there were about 4-5 months remaining before the deadline. Throughout the background research there were ideas that formed that would be shaped into the core idea of the thesis, so it was not a fresh start at that point. The idea of utilizing container orchestration formed early on in the research process, but there was a need to investigate if that could be a potential solution. The scientific backing from the literature review and supplementary papers lead to it being potentially viable and it ended up as a good solution to the problem. It was satisfying to test of the efficacy of something thought out so far in advance through the rest of the process on the thesis.

5.5.3 Hindsight

The master thesis itself is done over a period of about 9 months excluding summer and other vacations. That is a long period to work on the same project especially when that work involves learning about the ins and outs of a new possibly unfamiliar field of study. The problem and thesis is by no means the same as what it was at the beginning about 9 months ago, and this can be difficult to handle. The regular work process of university studies revolves around the structure of the semesters and means that generally, no course is longer than about 5 months. Being used to this pattern of working, it took time to adjust to a larger project that spans as long as the thesis. Most of the time spent on the

first semester were not at all as structured as it should have been as subconsciously the task of writing the thesis felt like a task for the next semester. By not distributing the work evenly over the time given, the last semester of the master has been harsh with long workdays and a need for higher levels of productivity. On the other hand, this meant that the majority of the master thesis was written later in the project and resulted in less of the work having to be adjusted and rewritten as priorities changed. By managing the allotted time better and setting proper milestones throughout the process, this same thing could have been achieved without having to make up for the lost time through tougher working conditions.

5.5.4 Lessons learned

All in all there has been a lot to learn through the process of researching and writing the thesis. These lessons learned are listed below here as follows:

1. Plan out the entirety of the process from start to finish and set achievable goals early on in the process. By properly planning out the process from the beginning, it is easier to see the bigger picture and avoid the feeling of the problem being something far off in the distance. The deadlines along the way for milestones also helps with feeling a bit more urgency towards the thesis as a whole.
2. Take the time needed to properly investigate and understand the field of research before taking any major decisions that will shape the entirety of the thesis. Priorities change over time as a consequence of the ideas you have maturing and/or based on the amount of insight into the field of research. By taking the time needed to gain this insight beforehand, the decisions taken for possibly solutions etc. will not have to changed as much over the course of the process. This will reduce the amount of work that has to be redone or altered to fit the shift in priorities and vision.
3. Plan the completion of the thesis well before the deadline and at least a week before. This gives time to fix and polish the content of the thesis. It also gives time for others to read and review the content and grammar of the thesis. If there is time, try putting down the thesis for a day or two depending on how near it is the deadline. Refreshing the perspective on the thesis can be needed to better find and improve mistakes and inconsistencies.

6 Conclusion

6.1 Conclusion

The goal of the thesis was to look at how services and applications can be designed and implemented for a smart city context. It would be to help alleviate the growing strains on the network infrastructure imposed by growing amounts of data generated from IoT devices that is being sent to cloud systems. It takes a basis in the fundamental principles of the ZEN proposed "Data Preservation through Fog-to-Cloud (F2C) Data Management" paper and looks at how to utilize this in the design and implementation of these services and applications.

The Tiered-Priority Swarm Computing (TPSC) architectural model is proposed as a guideline, blueprint, foundation and/or best practice in how these services can be designed to best adhere to these principles. It utilizes concepts like containerization and container orchestration to show how these concepts can be applied in a smart city context and how they can improve the scalability and stability of ICT-systems using the architectural model. The model then shows its viability through the implemented system and acts as an example of how architectures using the model can be implemented. Due to some technological limitations of the current container orchestration platforms, the full potential of the model is not fully realized yet. Other than this, however, it should function well for smart city services/applications and even better when the container orchestration platform supports prioritized workload management.

6.2 Research Questions

6.2.1 RQ1: How can applications/services be made in a distributed-to-centralized context for smart cities?

This question is tackled through the proposed architectural model and its definition. It provides a generalized architectural overview of important modules and features that would be important to manage the priorities of a smart city ICT-system. The model is based on research and gathered materials of similar systems (both implemented and some that are not) through the literature review and gathered relevant materials. The efficacy of this model is then tested through the implementation and evaluated through the evaluation metrics presented in the results chapter.

6.2.2 RQ1.1: What are the prevailing methodologies and technological trends in the layers of the distributed-to-centralized context?

This is presented in the background chapter as a combined effort of the literature review and other relevant papers etc. The case for the different layers of network centralization is presented and outlined first. Afterwards the chapter has an overview of all architectures identified and assessed by the review. This is to present what the results of the literature review argues as the prevailing methodologies and technological trends.

6.2.3 RQ1.2: What would be fitting requirements for the development and operation of a system within this context?

Based on the foundation set by the gathered materials of the background chapter, a set of requirements were set that would fit with the priorities and problems identified through the assessed architectures. Each relevant architecture from the literature review was examined to find focus points, priorities and problems outlined in each. This was to provide a good foundation for the requirements and a way to tie the background research into the design of the model.

6.2.4 RQ1.3: What would be a fitting architectural model for applications and services in the given context that satisfies the requirements set in RQ1.2?

This is answered by the design of the architectural model presented in the Results chapter. It was designed based on the requirements to fit the context of D2C smart city services and applications. It is to function as a basis, blueprint and to some extent best practices in designing and implementing ICT-systems in a smart city environment that enables the handling of data according to D2C data management.

6.2.5 RQ1.4: What quality metrics could be used to evaluate the model presented in RQ1.3?

The quality metrics is represented as quality attributes set according to the requirements and insight gained when designing the model and developing its implementation. They are presented in the last section of the Results chapter and are used in the Discussion chapter to evaluate the implementation and the architectural model itself.

6.3 Future work

One flaw of the architectural model lies in the lack of support for prioritized workload management in container orchestration platforms. It is not impossible to do, but as of now it is not supported. Most of the usage of container orchestration lies in cloud computing system and problems relating to implementation in fog computing is therefore not explored to the same extent. The thesis argues that there is a solid foundation to using container orchestration as swarm computing, but this feature needs to be available for the model to realize its full potential. Too many IoT-devices and other small and/or embedded devices has unused resources that can be utilized for it to be ignored as a possibility. Popular container orchestration platforms like Docker Swarm and Kubernetes are open-source, so it is feasible to extend existing platforms rather than making one that supports it natively. One proposal for further work is then to add the feature of prioritized selection in the container orchestration systems that allows manager nodes to select available nodes based on a prioritized selection rather than the standard randomized.

Furthermore, the work in the thesis is meant to serve as a starting point for the discussion of standardization of ICT-systems in the context of smart cities, neighbourhoods and buildings. It does by no means provide this proposed architectural model as an absolute perfect design, but rather one with promising concepts that can be of use in smart cities. It tries to provide the best reasoning for the choices made, but a lot of architectural work is known to be subjective to the person designing it. This is one view of the problem and how it can be solved from the requirements set and problem to solve in the thesis. More work needs to be done, more systems need to be developed and more architectures need to be designed in order to properly set standards and best practices.

Bibliography

[1] *State of the IoT 2018: Number of IoT devices now at 7B – Market accelerating.* https://iot-analytics.com/state-of-the-iot-update-q1-q2-2018-number-of-iot-devices-now-7b/. Accessed: 2018-10-30.

[2] *State Of Enterprise Cloud Computing, 2018.* https://www.forbes.com/sites/louiscolumbus/2018/08/30/state-of-enterprise-cloud-computing-2018/267c0b5a265e. Accessed: 2018-10-30.

[3] A. Sinaeepourfard et al. "Estimating Smart City sensors data generation". In: *2016 Mediterranean Ad Hoc Networking Workshop (Med-Hoc-Net)*. June 2016, pp. 1–8. DOI: 10.1109/MedHocNet.2016.7528424.

[4] Tuba Bakıcı, Esteve Almirall, and Jonathan Wareham. "A Smart City Initiative: the Case of Barcelona". In: *Journal of the Knowledge Economy* 4.2 (June 2013), pp. 135–148. ISSN: 1868-7873. DOI: 10.1007/s13132-012-0084-9. URL: https://doi.org/10.1007/s13132-012-0084-9.

[5] *+CityXhange.* https://cityxchange.eu/. Accessed: 2019-02-12.

[6] *What is fog computing? Connecting the cloud to things.* https://www.networkworld.com/article/3243111/internet-of-things/what-is-fog-computing-connecting-the-cloud-to-things.html. Accessed: 2018-11-13.

[7] M. Satyanarayanan et al. "Edge Analytics in the Internet of Things". In: *IEEE Pervasive Computing* 14.2 (Apr. 2015), pp. 24–31. ISSN: 1536-1268. DOI: 10.1109/MPRV.2015.32.

[8] A. Sinaeepourfard et al. "Data Preservation through Fog-to-Cloud (F2C) Data Management in Smart Cities". In: *2018 IEEE 2nd International Conference on Fog and Edge Computing (ICFEC)*. May 2018, pp. 1–9. DOI: 10.1109/CFEC.2018.8358732.

[9] *What is a smart city - Technopedia.* https://www.techopedia.com/definition/31494/smart-city. Accessed: 2019-05-19.

[10] *Smart Cities - European Commission.* https://ec.europa.eu/info/eu-regional-and-urban-development/topics/cities-and-urban-development/city-initiatives/smart-cities_en. Accessed: 2019-05-19.

[11] Rida Khatoun and Sherali Zeadally. "Smart Cities: Concepts, Architectures, Research Opportunities". In: *Commun. ACM* 59.8 (July 2016), pp. 46–57. ISSN: 0001-0782. DOI: 10.1145/2858789. URL: http://doi.acm.org/10.1145/2858789.

[12] A. Munir, P. Kansakar, and S. U. Khan. "IFCIoT: Integrated Fog Cloud IoT: A novel architectural paradigm for the future Internet of Things." In: *IEEE Consumer Electronics Magazine* 6.3 (July 2017), pp. 74–82. ISSN: 2162-2248. DOI: 10.1109/MCE.2017.2684981.

[13] M. Aazam and E. Huh. "Fog Computing: The Cloud-IoTIoE Middleware Paradigm". In: *IEEE Potentials* 35.3 (May 2016), pp. 40–44. ISSN: 0278-6648. DOI: 10.1109/MPOT.2015.2456213.

[14] *Top cloud providers 2019*. https://www.zdnet.com/article/top-cloud-providers-2019-aws-microsoft-azure-google-cloud-ibm-makes-hybrid-move-salesforce-dominates-saas/. Accessed: 2019-05-20.

[15] *In 2018, AWS delievered most of Amazon's operating income*. https://www.zdnet.com/article/in-2018-aws-delivered-most-of-amazons-operating-income/. Accessed: 2019-05-20.

[16] Michael Armbrust et al. "A View of Cloud Computing". In: *Commun. ACM* 53.4 (Apr. 2010), pp. 50–58. ISSN: 0001-0782. DOI: 10.1145/1721654.1721672. URL: http://doi.acm.org/10.1145/1721654.1721672.

[17] *Iaas vs PaaS vs SaaS: Examples and how to differentiate (2019)*. https://www.bigcommerce.com/blog/saas-vs-paas-vs-iaas/#examples-of-saas-paas-and-iaas. Accessed: 2019-05-21.

[18] M. Satyanarayanan et al. "The Case for VM-Based Cloudlets in Mobile Computing". In: *IEEE Pervasive Computing* 8.4 (Oct. 2009), pp. 14–23. ISSN: 1536-1268. DOI: 10.1109/MPRV.2009.82.

[19] *What is edge computing? GE Digital*. https://www.ge.com/digital/blog/what-edge-computing. Accessed: 2019-05-21.

[20] *What is fog computing? Techradar*. https://www.techradar.com/news/what-is-fog-computing. Accessed: 2019-05-21.

[21] *About us - FME ZEN*. https://fmezen.no/about-us/. Accessed: 2019-05-21.

[22] Guannan Hu et al. "A Dynamic User-integrated Cloud Computing Architecture". In: *Proceedings of the 2011 International Conference on Innovative Computing and Cloud Computing*. ICCC '11. Wuhan, China: ACM, 2011, pp. 36–40. ISBN: 978-1-4503-0567-9. DOI: 10.1145/2071639.2071649. URL: http://doi.acm.org/10.1145/2071639.2071649.

[23] Saouli Hamza et al. "A New Cloud Computing Approach Based SVM for Relevant Data Extraction". In: *Proceedings of the 2Nd International Conference on Big Data, Cloud and Applications*. BDCA'17. Tetouan, Morocco: ACM, 2017, 1:1–1:8. ISBN: 978-1-4503-4852-2. DOI: 10.1145/3090354.3090355. URL: http://doi.acm.org/10.1145/3090354.3090355.

[24] Randall Sobie et al. "HTC Scientific Computing in a Distributed Cloud Environment". In: *Proceedings of the 4th ACM Workshop on Scientific Cloud Computing*. Science Cloud '13. New York, New York, USA: ACM, 2013, pp. 45–52. ISBN: 978-1-4503-1979-9. DOI: 10.1145/2465848.2465850. URL: http://doi.acm.org/10.1145/2465848.2465850.

[25] Eric Keller et al. "NoHype: Virtualized Cloud Infrastructure Without the Virtualization". In: *SIGARCH Comput. Archit. News* 38.3 (June 2010), pp. 350–361. ISSN: 0163-5964. DOI: 10.1145/1816038.1816010. URL: http://doi.acm.org/10.1145/1816038.1816010.

[26] P. Dutta et al. "C-Cloud: A Cost-Efficient Reliable Cloud of Surplus Computing Resources". In: *2014 IEEE 7th International Conference on Cloud Computing*. June 2014, pp. 986–987. DOI: 10.1109/CLOUD.2014.152.

[27] Emiliano Miluzzo, Ramón Cáceres, and Yih-Farn Chen. "Vision: MClouds - Computing on Clouds of Mobile Devices". In: *Proceedings of the Third ACM Workshop on Mobile Cloud Computing and Services*. MCS '12. Low Wood Bay, Lake District, UK: ACM, 2012, pp. 9–14. ISBN: 978-1-4503-1319-3. DOI: 10.1145/2307849.2307854. URL: http://doi.acm.org/10.1145/2307849.2307854.

[28] N. Choi et al. "A Fog Operating System for User-Oriented IoT Services: Challenges and Research Directions". In: *IEEE Communications Magazine* 55.8 (Aug. 2017), pp. 44–51. ISSN: 0163-6804. DOI: 10.1109/MCOM.2017.1600908.

[29] F. Mehdipour, B. Javadi, and A. Mahanti. "FOG-Engine: Towards Big Data Analytics in the Fog". In: *2016 IEEE 14th Intl Conf on Dependable, Autonomic and Secure Computing, 14th Intl Conf on Pervasive Intelligence and Computing, 2nd Intl Conf on Big Data Intelligence and Computing and Cyber Science and Technology Congress(DASC/PiCom/DataCom/CyberSciTech)*. Aug. 2016, pp. 640–646. DOI: 10.1109/DASC-PICom-DataCom-CyberSciTec.2016.116.

[30] *What is Containerization in DevOps?* https://www.linuxnix.com/what-is-containerization-in-devops/. Accessed: 2019-05-22.

[31] *What is Kubernetes? Container orchestration explained*. https://www.infoworld.com/article/3268073/what-is-kubernetes-container-orchestration-explained.html. Accessed: 2019-05-22.

[32] *General Python FAQ - Python 3.7.3 documentation*. https://docs.python.org/3/faq/general.html. Accessed: 2019-05-22.

[33] *JavaScript | MDN*. https://developer.mozilla.org/en-US/docs/Web/JavaScript. Accessed: 2019-05-22.

[34] *TypeScript - JavaScript that scales*. https://www.typescriptlang.org/. Accessed: 2019-05-22.

[35] *JSON*. https://www.json.org/. Accessed: 2019-05-22.

[36] *What is MongoDB | MongoDB*. https://www.mongodb.com/what-is-mongodb. Accessed: 2019-05-22.

[37] *Why Docker? | Docker*. https://www.docker.com/why-docker. Accessed: 2019-05-22.

[38] *Overview of Docker Compose | Docker Documentation*. https://docs.docker.com/compose/overview/. Accessed: 2019-05-22.

[39] *Swarm mode overview | Docker Documentation*. https://docs.docker.com/engine/swarm/. Accessed: 2019-05-22.

[40] *About | Node.js*. https://nodejs.org/en/about/. Accessed: 2019-05-22.

[41] *About npm | npm Documentation*. https://docs.npmjs.com/about-npm/. Accessed: 2019-05-22.

[42] *Express - Node.js web application framework*. https://expressjs.com/. Accessed: 2019-05-22.

[43] *What is NGINX? - NGINX*. https://www.nginx.com/resources/glossary/nginx/. Accessed: 2019-05-22.

[44] *Angular - What is Angular?* https://angular.io/docs. Accessed: 2019-05-22.

[45] *Angular - CLI Command Reference*. https://angular.io/cli. Accessed: 2019-05-22.

[46] *Mongoose ODM v.5.5.10*. https://mongoosejs.com/. Accessed: 2019-05-22.

[47] *Chart.js | Open source HTML5 Charts for your website*. https://www.chartjs.org/. Accessed: 2019-05-22.

[48] *Bootstrap - The most popular HTML, CSS and JS library in the world*. https://getbootstrap.com/. Accessed: 2019-05-22.

[49] Anders Kofod-Petersen. "How to do a Structured Literature Review in computer science". In: (May 2015).

[50] *Key changes with the General Data Protection Regulation - EUGDPR*. https://eugdpr.org/the-regulation/. Accessed: 2019-04-27.

[51] *Docker - Compatibility Matrix*. https://success.docker.com/article/compatibility-matrix. Accessed: 2019-05-22.

[52] *Apply rolling updates to a service | Docker Documentation*. https://docs.docker.com/engine/swarm/swarm-tutorial/rolling-update/. Accessed: 2019-05-23.

[53] *Compose file version 3 reference | Docker Documentation*. https://docs.docker.com/compose/compose-file/. Accessed: 2019-05-23.

[54] *Administer and maintain a swarm of Docker Engines | Docker Documentation*. https://docs.docker.com/engine/swarm/admin_guide/#add-manager-nodes-for-fault-tolerance. Accessed: 2019-05-23.

[55] *express-rate-limit - npm*. https://www.npmjs.com/package/express-rate-limit. Accessed: 2019-05-24.

[56] *Docker Machine Overview | Docker Documentation*. https://docs.docker.com/machine/overview/. Accessed: 2019-05-24.

[57] *Koa - next generation web framework for node.js*. https://koajs.com/. Accessed: 2019-05-25.

[58] *Django overview | Django*. https://www.djangoproject.com/start/overview/. Accessed: 2019-05-25.

[59] *Use volumes | Docker Documentation*. https://docs.docker.com/storage/volumes/. Accessed: 2019-05-26.

[60] *Volumes - Kubernetes*. https://kubernetes.io/docs/concepts/storage/volumes/. Accessed: 2019-05-26.

[61] *Replication, Clustering, and Connection Pooling - PostgreSQL wiki*. https://wiki.postgresql.org/wiki/Replication,_Clustering,_and_Connection_Pooling. Accessed: 2019-05-26.

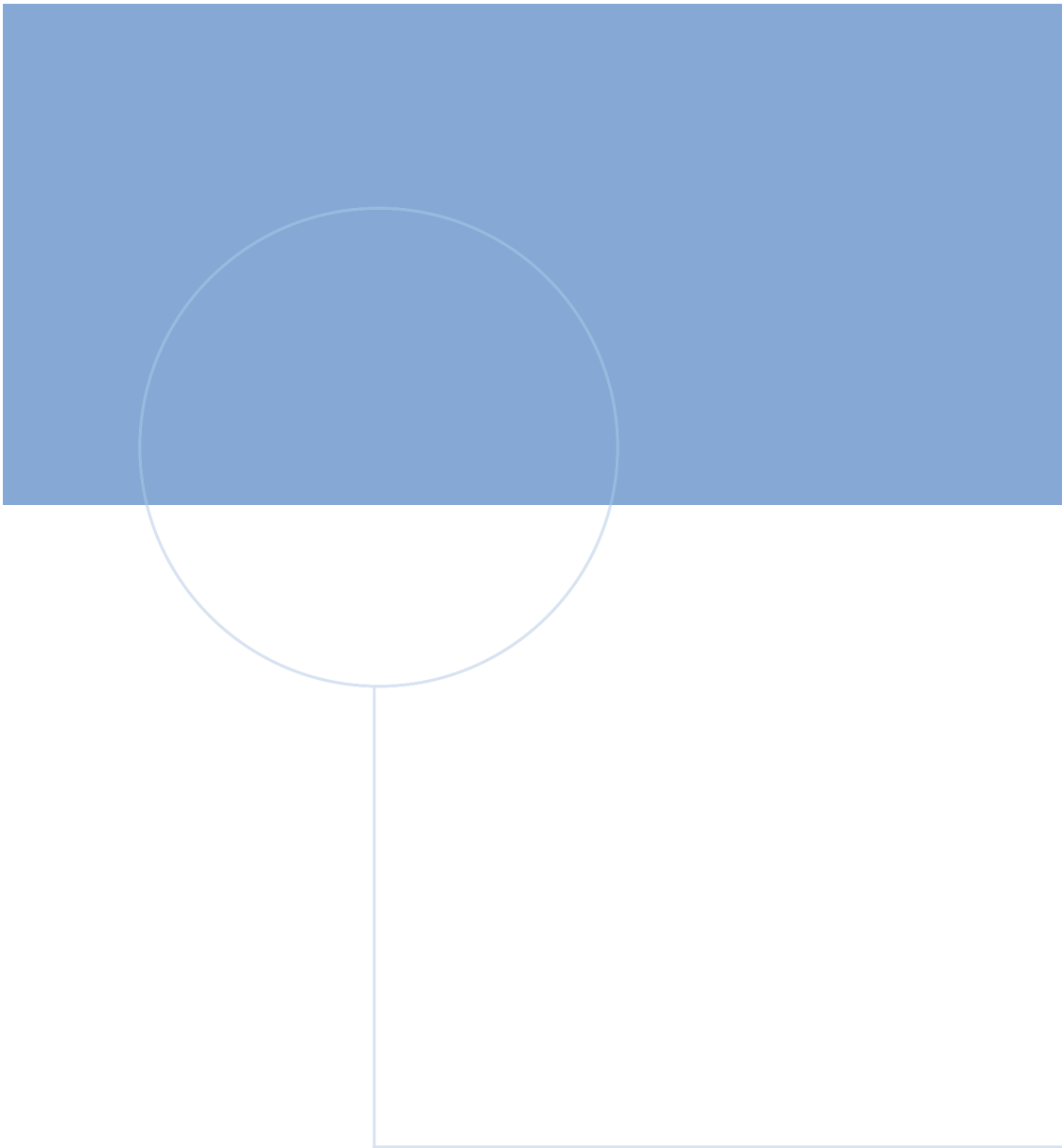
[62] *Replication - MongoDB Manual*. https://docs.mongodb.com/manual/replication/. Accessed: 2019-05-26.

[63] *MongoDB ReplicaSet with Docker Swarm - Alberto Manuel Rojas Mendez - Medium*. https://medium.com/@albertorojasm95/mongodb-replicaset-with-docker-swarm-8461ecd904b0. Accessed: 2019-05-26.

[64] *Running MongoDB on Kubernetes with StatefulSets - Kubernetes*. https://kubernetes.io/blog/2017/01/running-mongodb-on-kubernetes-with-statefulsets/. Accessed: 2019-05-26.

7 Appendix

The code for the implementation is not included here, as it is available on github through this link (https://github.com/petterrostrup/TPSC)



NTNU
Norwegian University of
Science and Technology