Master's thesis

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

Eivind Vold Aunebakk

Internet-of-Things Frameworks For Pervasive Games: A Comparative Study

Master's thesis in Informatics
Supervisor: Dag Svanæs
December 2022

NTNU
Norwegian University of
Science and Technology

Eivind Vold Aunebakk

Internet-of-Things Frameworks For Pervasive Games: A Comparative Study

Master's thesis in Informatics
Supervisor: Dag Svanæs
December 2022

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

Exergames have proven health benefits and may play a bigger role in the prevention of negative health effects and rehabilitation in the future. This thesis is part of the EXACT research project. The project is directed at exploring the use of IoT technologies in exergames in physical rehabilitation. IoT offers new opportunities for building pervasive exergames with the inclusion of a variety of sensors and actuators and this thesis aims to help fill the gap in the limited literature about the development of Internet of things (IoT) based pervasive games. The thesis continues the work of Bärnholt and Lyngby [13] by improving upon their framework and comparing it to a cloud-based solution. Along with the previous findings a literature review was performed to discover additional architectural challenges concerning pervasive IoT games, and a new set of criteria were derived based on the findings. Two different solutions were chosen based on the criteria and implemented. One solution re-implementing the framework of Bärnholt and Lyngby [13] and a minimal solution built on AWS IoT Services. An example game, *Follow the Red Dot*, was implemented using both solutions. The two solutions were evaluated based on the criteria and compared to one another. The evaluation shows that both solutions were able to support the game in a satisfactory manner, however, the AWS solution failed to meet some of the requirements. Though some of the shortcomings are believed to be caused by the minimal implementation and not a fault of the architecture.

Sammendrag

Treningsspill har beviste helseeffekter og kan i framtiden spille en større rolle i forebyggingen av negative helseeffekter, samt rehabilitering. Denne oppgaven er en del av forskningsprosjektet EXACT som går ut på å undersøke bruken av IoT teknologier i treningsspill for rehabilitering. Ved å ta i bruk sensorer og aktuatorer Tingenes Internett (IoT) åpner nye muligheter innen virkelighetsknyttede treningsspill og denne oppgaven sikter på å bidra til den manglende kunnskapen om bruk av IoT i virkelighetsknyttede spill. Oppgaven bygger på arbeidet til Bärnholt and Lyngby [13] ved å utbedre rammeverket deres etter deres anbefalinger og så sammenlikne det med en skybasert løsning. I tillegg til deres tidligere funn ble det gjort et litteratursøk for å kartlegge utfordringer forbundet med virkelighetsknyttede IoT spill, for så å utarbeide et nytt sett med kriterier til en mulig løsning. Det ble valgt to løsninger basert på disse kriteriene: en re-implementasjon av deres rammeverk og en minimal skyløsning ved bruk av AWS IoT Services. Et eksempel spill, *Follow the Red Dot*, ble så implementert ved bruk av hvert av rammeverkene. De to løsningene be så evaluert basert på de tidligere kriteriene og sammenliknet med hverandre. Det viste seg at begge løsningene løste implementasjonen av spillet på en tilfredstillende måte, men at noen av kriterier ikke ble møtt av den skybaserte løsningen. Noen av disse svakhetene er tenkt at stammer fra at løsningen er en minimal implementasjon uten nødvendige abstraksjonsnivåer, ikke svakheter i arkitekturen.

Preface

This thesis is a continuation of the semester project produced in the autumn of 2021, and completes my studies of the Master's degree programme in Informatics at the Norwegian University of Science and Technology.

I would like to thank my supervisor, Dag Svanæs, for his assistance in completing the thesis. His continued feedback and guidance throughout the project have been invaluable.

Table of contents

1 Introduction

1.1 Motivations

The Internet has made the world a lot smaller, allowing one to communicate across all continents. Lately, there's also been a big focus on shorter-range communication, and instead of connecting people, it has become a goal to connect all possible things to the Internet, aptly called the Internet of Things (IoT) [76]. IoT helps us turn on the coffee maker from the comfort of our bed, and turn up the heating on the way home from work. This immense goal of practicality has the consequence of helping us move less than we had to before. Parallel to this, researchers are exploring ways of utilising technology to improve people's health through exercise and rehabilitation.

EXACT is an interdisciplinary research project between the Department of Computer Science (IDI) and the Department of Neuromedicine and Movement Science (INB) at NTNU. The project is directed at exploring the use of IoT technologies in exergames combined with social media in physical rehabilitation. This thesis will focus on solving technical challenges.

1.2 Related work

This thesis builds on the thesis of Bärnholt and Lyngby [13], and one of the suggested implementations builds on their work. The health benefits of exercise are well established and, moreover, there's a lot of research on the benefits of exergames. Exergames have been explored as a tool in exercise and rehabilitation of the elderly, as well as younger patients [62].

1.3 Objectives

The primary goal of this thesis is to explore the possibilities and challenges surrounding the development of pervasive games, and to create an analysis by comparing two possible architectures for IoT-enabled games. The two architectures will be implemented independently and then compared by a set of requirements, followed by a comparative evaluation by implementing the same simple game with both architectures. While many of the existing products, such as the Moto Tiles [50] are not built on the principles of free and open source software (FOOS), this thesis aims to use FOOS components where this is feasible.

The following research questions were derived to help achieve these goals:
RQ1: What architectural challenges exist in relation to pervasive IoT games?
RQ2: What are the requirements for an IoT-based pervasive game architecture and framework?
RQ3: Which architectures and technologies may be suited to implement an IoT game framework?
RQ4: How well do the proof of concept implementations match the requirements found in RQ2?

1.4 Contributions

The thesis contributes to the research of IoT-based pervasive games. There's limited literature about the development of IoT-based games and the resulting analysis of this project aims to fill this gap. The analysis is based on the two example implementations as well as related papers.

1.5 Limitations

The thesis evaluates two possible technology stacks for IoT exergames by comparing proof of concept implementations. Due to the sheer number of possible technology stacks, two were selected due to the natural time constraints of a thesis. The two implementations are different

in many aspects and still prove for an interesting comparison. The AWS implementation was kept simple and does not utilise all of the possibly useful AWS services. The other framework is integrated into Unity, though it has no hard Unity dependencies and can be extracted without too much effort. Part of the comparison is based on the example game, which may not be representative of all exergames. Other games may have exposed different properties of the technologies.

1.6 Outline

Chapter two introduces relevant topics and provides the necessary background information for the rest of the thesis. It covers exergames and pervasive games, IoT with relevant technologies and communication patterns, cloud computing and digital twins, as well as free and open source software. Chapter three describes the research methods utilised to answer the presented research questions. This is followed by chapter four describing the architectural requirements of the two technology stacks. Chapter five describes the Unity implementation and the custom MQTT protocol. The AWS implementation is covered in chapter six. Chapter seven contains the final analysis of the two technology stacks and their implementations. Chapter eight and nine provides answers to the proposed research questions and concludes the findings.

2 Background

The sections 2.1, 2.3, 2.6, and 2.7 are carried over from the semester project. In section 2.3 the subsections 2.3.4 and 2.3.5 are new additions.

2.1 Exergames

The modern human is much less active than our predecessors. Both children, adults and the elderly spend a considerable amount of their time sitting still which is leading to obesity and deteriorating health, which is only worsened by age. A lack of activity doesn't only affect physical health, but also mental health. This has led to an increase in research on these issues and offers an intuitive use case for serious games: the use of games and gaming technology for purposes other than pure entertainment. Such purposes include education, training, health, etc [64]. Serious games that promote physical activity has been shown to improve physical fitness, mood, confidence and the overall quality of life of the player [53]. Exergames is a subset of serious games defined by Oh and Yang [59] as:

> A video game that promotes (either via using or requiring) players' physical movements (exertion) that is generally more than sedentary and include strength, balance, and flexibility activities

While it may not have been intentional, the worldwide phenomenon Pokémon Go has proven to be a salient example of an exergame. It has over 600 million installs to date [16]. The game uses the mobile device's GPS to locate, capture, train, and battle Pokémon, which through AR appear as if they are in the player's real-world location. With game objectives being linked to real-world locations it encourages the player to physically move around. Screenshots of the game can be seen in figure 1. Certain activities has sett requirements such as walking anywhere from 2 to 12km to hatch an egg to obtain a new Pokémon. Althoff et al. [1] found that Pokémon Go leads to a significant increase in physical activity, with particularly engaged users increasing their average number of steps per day by more than 25%, and conclude that:

> Mobile apps combining gameplay with physical activity lead to substantial short-term activity increases and, in contrast to many existing interventions and mobile health apps, have the potential to reach activity-poor populations

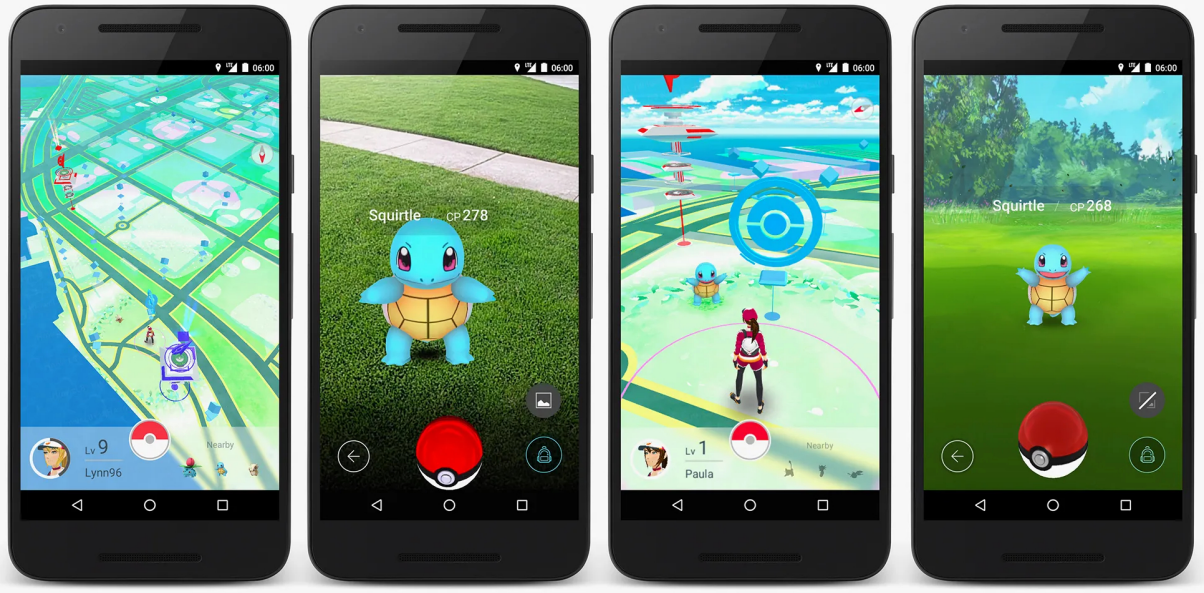


Figure 1: Pokémon Go requires the player to physically walk around to move their in-game character.

Such activities can also lead to unforeseen social impacts such as animal shelters offering dog walks to players who want to hatch Pokémon eggs, which in some instances had great success [56].

Pokémon Go is an example of great success but was mainly targeted at children and young adults. While studies have shown that serious games have much to offer in prevention and rehabilitation, they also show that it's difficult to archive similar results among the elderly [74]. The studies show trends where older people have specific playing preferences and difficulties handling digital games. Naturally this creates requirements for the games including appropriate content, interface design and game demands. The games may have to be adapted to individuals or a smaller audience.

2.1.1 Pervasive games

A pervasive game is a game which brings digital gaming experiences into the real world through the use of non-invasive computational devices.[15] Two examples of such games are Moto Tiles and Johansen's Follow the Red Dot [51]. Moto Tiles is a finished product consisting of modular tiles with embedded pressure sensors and LED's. By connecting multiple tiles and placing them on the floor the user can play a variety of games such as "Simon says". During clinical effect studies, Moto Tiles has shown a positive effect on the physical abilities of the elderly [50]. While Moto Tiles has proved effective and can act as inspiration as to what exergames can achieve, it's developed as a self-contained and finished product. Johansen explores different technologies that can form the basis for a framework for prototyping and developing pervasive games utilising the Internet of Things (IoT). The concept was then evaluated by creating a prototype game called Follow the Red Dot. The game consists of multiple modules where each module has a light and a button; a light appears on one of the modules and the player has to press the button on the lit module to turn it off; the light will then appear on a different module and so on. While the game is not revolutionary in itself, the proposed technologies and the proof of concept set the starting point for Bärnholt and Lyngby [13]. The tiles used in their project, as well as this thesis, are displayed in figure 2.

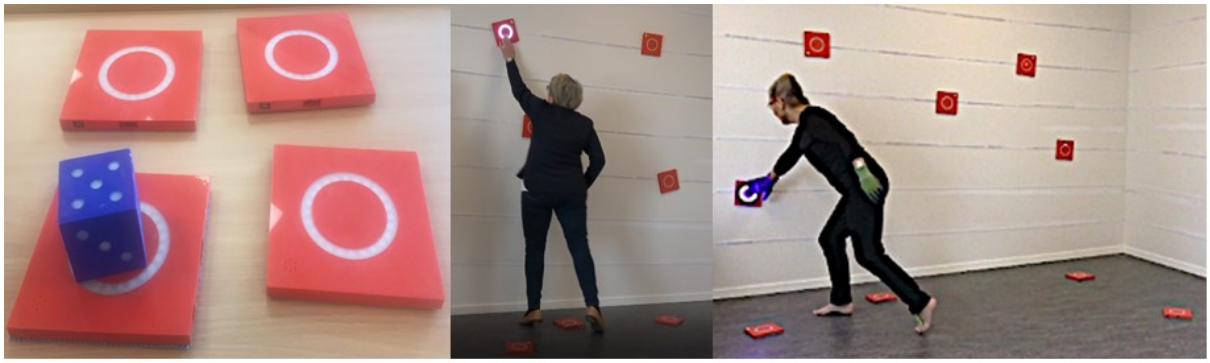


Figure 2: (a) Tiles and cube. (b)(c) Older adult participant training with the tiles [63].

2.2 Architecture and design patterns

An architectural pattern defines the larger architecture of a system [66]. A design pattern is a known method of solving a technical problem on the code level.

The current implementation of this project uses MQTT which is based on a message broker and the publish and subscribe pattern.

2.2.1 Broker

The broker pattern is an architectural pattern used to decouple components by utilising an intermediate message broker. This way multiple components can communicate through a central point without being explicitly aware of each other. A minimal implementation of the pattern is illustrated in figure 3. As seen in figure 4, the multicast pattern is an extension of the broker pattern allowing a sender to broadcast a message to all receivers connected to the broker [49].

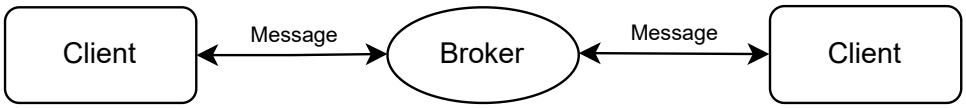


Figure 3: The broker pattern [49].

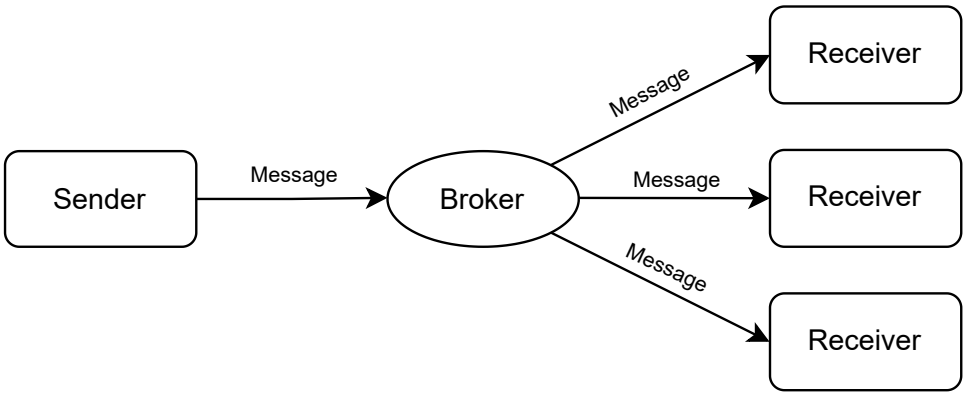


Figure 4: The multicast pattern [49].

2.2.2 Publish and Subscribe

The publish and subscribe patterns works by allowing a client to subscribe to one or more topics, and similarly, when a client wishes to send a message it has to specify the topic. When the broker receives a message it'll broadcast the message to all clients subscribed to the specific topic [49]. The pattern is illustrated in figure 5.

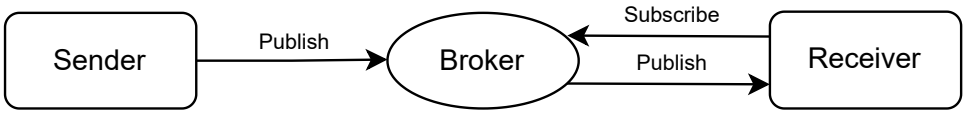


Figure 5: The publish and subscribe pattern [49].

2.2.3 Events / Observer

The event or observer pattern allows observing the state of an object and get notified when the state changes. This is often done by triggering an event/callback function [49].

2.3 Internet of Things

In recent years we have seen a surge of interest in the Internet of Things (IoT) and numerous standards, services and devices have been introduced. The term IoT, dating back over 20 years [4], has no common definition and sees a broad use today. Some of the proposed definitions focus on the things themselves, while others focus on Internet-related aspects such as protocols and network technology, or even the storage, search and organisation of the large volumes of information created by the [76]. One can argue that these propositions are too concerned about the details and current technologies, and as a result, misses what IoT is all about. The International Telecommunication Union has defined it as:

> A global infrastructure for the Information Society, enabling advanced services by interconnecting(physical and virtual) things based on, existing and evolving, interoperable information and communication technologies [69].

This definition puts emphasis on the idea and covers the bigger picture so that there's likely no need to redefine it in the future. Cisco has defined it as "simply the point in time when more 'things or objects' were connected to the Internet than people" [22]. IoT allows monitoring, controlling and collecting data from countless devices such as lights, air conditioning, security systems, washing machines, medical devices, etc.

2.3.1 The IoT technology stack

To describe an IoT solution, Wortmann and Flüchter [76] proposes a three-layer technology stack consisting of the device layer, the connectivity layer and the cloud layer. A simplified version is shown in figure 6.

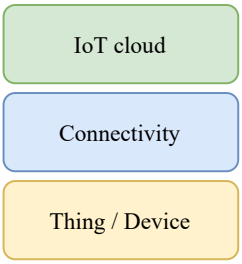


Figure 6: The IoT technology stack [76]

The thing/device layer is the device itself, including the hardware, software and additional components such as sensors and actuators; anything regarding the physical device. The connectivity layer is how the device communicates with the cloud layer and determines the choice of communication protocols, such as MQTT or Zigbee. The Cloud layer encompasses all the aspects of the application of the device. It may handle device communication and management, analytics and data management, and process management in the case to execute and monitor processes across people, systems and devices. The Cloud layer also contains the software providing an interface for interaction between people and the system and devices.

2.3.2 Wi-Fi

Wi-Fi is a collection of wireless network protocols based on the IEEE 802.11 standards [45], allowing wireless transfer of data over radio waves on the 2.4 GHz and 5 GHz bands. Wi-Fi is commonly used to connect devices in a Local Area Network (LAN), and usually as an intermediate step to connect the device to the internet, but many IoT devices also use it for local data exchange. Its wide spread use makes Wi-Fi a noteworthy candidate for most IoT applications; as most areas are already covered by Wi-Fi there's little additional infrastructure cost. Some modern access points claim up to a 150-metre range and on the correct hardware, WiFi6 can achieve speeds up to 9.6 Gb/s; and while these numbers are impressive, the power requirements are higher than desirable on smaller battery powered devices.

2.3.3 MQTT

MQTT [57] is a publish-subscribe network protocol for machine-to-machine communication. It resides in the application layer and will usually run on top of TCP/IP. The protocol defines two types of entities: the message broker and any number of clients. The broker acts as a server which receives all the messages from the clients and relays the information to the appropriate recipients. A client cannot transmit messages to other clients directly; a client must subscribe to a topic from the broker, and the broker will then relay any messages matching the specified topic to that client.

MQTT and Wi-Fi is commonly combined to create a complete protocol for communication between IoT devices.

2.3.4 ESP8266

The ESP8266 is a small form factor and low cost microcontroller [24]. In addition to the official SDK, there are several different alternative SDKs available including Arduino [75]. The numerous available Arduino libraries in combination with its small size and built-in WiFi and Bluetooth capabilities make it a good choice for many IoT applications. The ESP8266 is pictured in figure 7.

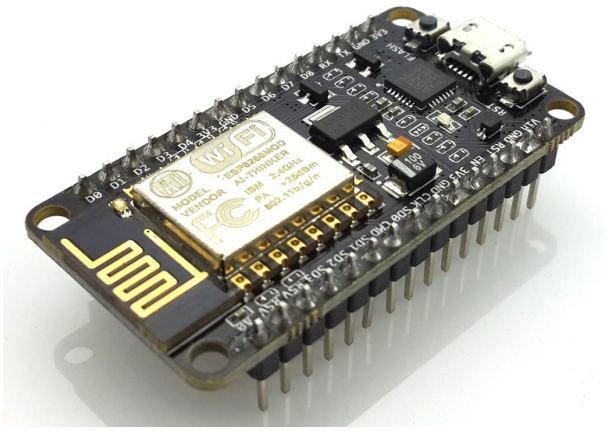


Figure 7: An ESP8266 module [43].

2.3.5 Raspberry Pi

Raspberry Pi [61] is a collection of small and affordable single-board computers. Contrary to the ESP8266, a Raspberry Pi is a fully fledged, general purpose computer capable of running a number of operating systems including, but not limited to, Raspberry Pi OS and other Unix-like operating systems as well as Windows 10 IoT Core. Raspberry Pi's include USB and HDMI ports as well as built-in WiFi. The Raspberry Pi 4 is pictured in figure 8.

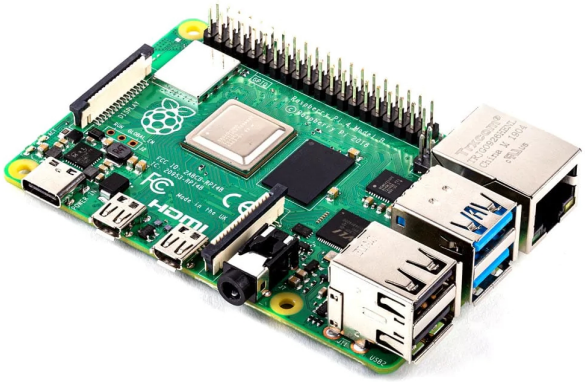


Figure 8: The Raspberry Pi 4 [68].

2.4 Digital Twins

Digital twins consist of two systems, one physical and one virtual [65]. The systems are able to communicate with each other in a way where the virtual system contains all information related the physical system, and any change in either system should be reflected in the other. Digital twins is a concept commonly used in IoT where being able to control and monitor a physical device is a widespread use case in everything from home thermostats to industrial robots. The use of a digital twin aims to aid prototyping, gather data, and make predictions of failures and the physical device's lifetime. Working on a digital version of the product allows for quicker iteration in both design and behaviour. Instead of creating a physical prototype and extensively testing it to make sure it adheres to the product requirements, with a sufficiently advanced twin one can simply run a simulation with automatic testing.

The concept of digital twins is well established in areas with safety-critical systems such as the aerospace industry. The potential for loss of life, or the cost of a mission failure may far out

weight the cost of developing a digital twin which may prevent such failures. An example of this is NASA and the US Air Force where digital twins are an integral part of the development and operation of different vehicles [39].

According to Tao et al. [65] there hasn't been a big focus on exploring the use of digital twins in the development of consumer products. In products with simpler and more predictable systems, the ease of prototyping and testing may not warrant the use of a digital twin as the cost of developing and maintaining said twin might still be substantial.

2.5 Cloud computing

Cloud computing is the on-demand availability of computer system resources, especially data storage (cloud storage) and computing power, without direct active management by the user [55]. Most cloud service providers, such as AWS, Microsoft Azure and Google Cloud, offer services at different levels of abstraction, the most common are: Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS).

Infrastructure as a Service offers abstractions of infrastructure. Common services may include virtual machines, physical servers, storage, firewalls, load balancers, etc. IaaS provides the customer with the ability to run arbitrary software, including applications and operating systems [2].

Platform as a service provides a platform where the customer has little to no control of the under laying infrastructure, but onto which the customer may deploy applications of their choice. This includes databases, web servers, and custom applications.

Software as a service provides the customer with the ability to use the provider's software, running in the cloud. The applications may allow some configuration but the customer has no control of the platform it runs on. An obvious example is a web-based email service, but SaaS also covers services such as online databases and numerous other ready-to-use components.

2.5.1 AWS IoT

Amazon Web Services is a cloud service provider owned by Amazon [5]. It provides cloud computing platforms and APIs on a pay-as-you-go basis, including databases, storage, compute services, and virtual machines. It also provides many free tires of the basic services lowering the barrier of testing the platform for smaller projects.

On October 8 2015 AWS announced a managed cloud platform for IoT [11]. It aims to bring IoT devices into the AWS allowing them to be seamlessly integrated into a larger system [8]. An overview of the services is illustrated in figure 9. AWS IoT provides open source AWS IoT Device SDKs for a number of different devices and languages. The SDKs help connect the devices to AWS IoT and handle most aspects of the communication. The entry point to the IoT cloud services is the device gateway which enables the devices to efficiently communicate with AWS IoT through end to end encryption using X.509 certificates.

AWS IoT includes a MQTT message broker enabling the devices, and connected applications and services to publish and receive messages from each other. One can communicate with MQTT directly or over WebSocket. There's also a REST API allowing devices to publish data via HTTP requests.

The rules engine allows one to apply rules on how to handle incoming messages [9]. This makes it easier to transform and relay data to other AWS IoT services such as storage or for further processing.

A device shadow is a JSON document describing the state of a device and allows devices to store persistent values [7]. The device shadow service also enables other applications to change or retrieve the state regardless of whether the device is online. Any requested changes made

when the device is offline are automatically synced when the device retrieves the state the next time the device goes online. When a change to the state is requested by another application or device the request is stored in the device shadow. The device is notified of the change and should act to reflect the new state. When the device has updated it's internal state the device reports the change to the shadow service. This allows one to see both the desired state, as well as the reported, or actual, state of the device. Due to the bidirectional nature of the communication, the shadow document may act as a digital twin to the connected device.

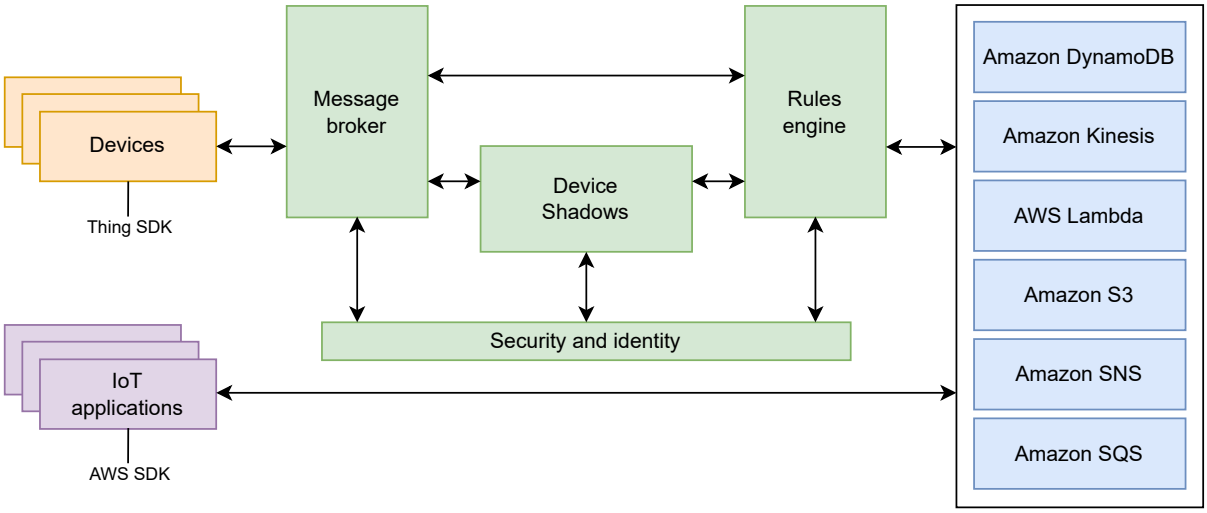


Figure 9: Overview of the AWS IoT Core services [8].

2.6 UNITY-Things

2.6.1 The current technology stack

Bärnholt and Lyngby [13] decided on a modified version of the previously described technology stack (see figure 6). The current framework describes a four-layer technology stack, which differs from the one proposed by Wortmann and Flüchter [76] by renaming the "IoT Cloud" layer to "Application" and adding an additional layer called the "Object abstraction". This is illustrated in figure 10. Application is better suited as it is more generalised than IoT Cloud, while still describing the same thing: a management software, which covers a range of different applications including the Unity Game Engine. The Object abstraction layer holds the different connection types between the devices such as WiFi, Zigbee, Bluetooth etc.

The specific instance of the technology stack utilised in this project can be seen in figure 11. The framework is implemented in the Unity game engine which controls the game logic and manages the data from and to the devices. MQTT is utilised as the messaging protocol, supported by a MQTT-broker running on a Raspberry Pi which will relay the messages between the Unity application and the devices. The devices themselves are Arduino's and Raspberry Pi's fitted with actuators and sensors.

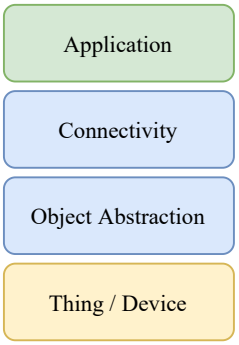


Figure 10: A modified IoT technology stack [13].

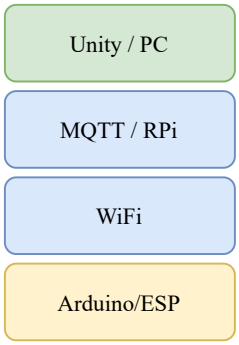


Figure 11: The current technology stack for this project [13].

This results in a framework where the custom Arduino devices can be programmed as dumb MQTT client and their behaviour and device-to-device interaction can be programmed through their digital twins in Unity as if they were ordinary Unity game objects. Prototyping behaviours in Unity is effortless when compared to updating the software on all the Arduino's, especially regarding device-to-device interaction and coordination. This also creates a division between the device developers and the behaviour/game developers, making it possible for them to work independently of each other.

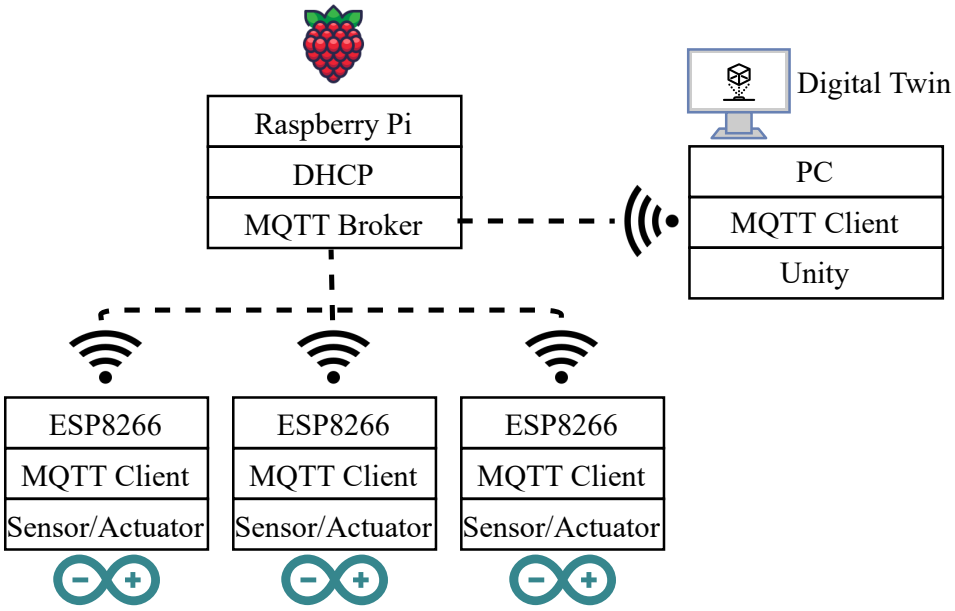


Figure 12: The UNITY-Things solution [13].

2.7 Free and open source software

While there's no official definition[1] of free/libre and open source software (FOSS), it's generally regarded as software where anyone has the right to use, copy, distribute, study, and change the software [34][47].

[1]The Open Source Initiative's definition is recognised by some governments.

The idea of FOSS is as old as software itself. Openness and cooperation is long established in academia, and as a result almost all of the software developed by academics and researchers in the 1950s and 60s was shared openly. However, by the late 1960, as software started to become more complex and the development cost increased there was a growing amount of software that was for sale only under restrictive licenses, and to further increase revenue, a trend began to no longer distribute source code.

The idea of free software was formalised in 1985 when Richard Stallman established the Free Software Foundation (FSF) [32] after founding the GNU Project in 1983. He was distressed by the decline of free software and aimed to create complete operating system so that people could use computers using only free software. The Free Software Definition [34] was published by the FSF in February 1986, where it's noted that the word "free" does not refer to price, it refers to freedom; the freedom to copy a program and redistribute it, and the freedom to change a program, so that you can control it instead of it controlling you [34].
The modern definition describes the four essential freedoms [34]:

- The freedom to run the program as you wish, for any purpose (freedom 0).
- The freedom to study how the program works, and change it so it does your computing as you wish (freedom 1). Access to the source code is a precondition for this.
- The freedom to redistribute copies so you can help your neighbour (freedom 2).
- The freedom to distribute copies of your modified versions to others (freedom 3). By doing this you can give the whole community a chance to benefit from your changes. Access to the source code is a precondition for this.

To be defined as free, a piece of software must grant it's users all of these freedoms. The FSF defines which software is considered free at its own discretion, but as a general rule, software is free if it's licensed under one of the licences approved by the FSF [33].

Another other big organisation is the Open Source Initiative (OSI) [48]. In 1997, Eric S. Raymond published "The Cathedral and the Bazaar", which later became a motivating factor for Netscape to release Netscape Communicator and its source code as free software and start the Mozilla project. This made Raymond and others explore how to make free software more appealing to the commercial-software industry as the word "free" is ambiguous[2] and does not resonate well with the industry. The term "open source" was adopted by some in the free software movement during a strategy meeting and shortly after Bruce Perens and Raymond launched www.opensource.org. At the Freeware Summit (later named the "Open Source Summit"), it was put to a vote and "open source" was the winner and The Open Source Initiative was formed shortly after [46]. The OSI has its own definition of open source software, aptly named The Open Source Definition [47], which defines the following criteria:

1. Free redistribution.
2. The program must include source code or there must be a well-publicised means of obtaining the source code.
3. The license must allow modifications and derived works, and must allow them to be distributed under the same terms as the license of the original software.

[2]The free software community often uses the french adjective "libre" (free, at liberty) to avoid the ambiguity of the word "free".

4. The license may restrict source-code from being distributed in modified form only if the license allows the distribution of "patch files", as to protect the integrity of the original author's source code.
5. No discrimination against persons or groups.
6. No discrimination against fields of endeavour.
7. Distribution of license to all whom the program is distributed to.
8. The license must not be specific to a product.
9. The license must not restrict other software.
10. The license must be technology-neutral.

While both The Free Software Definition and The Open Source Definition refer to almost the same class of software, the main difference is, according to Richard Stallman, that the free software movement campaigns for freedom for the users of computing while the open source idea mainly values the practical advantages [35].

2.7.1 FOSS in practice today

Today's software development is dominated by Git and GitHub. In the Stack Overflow 2018 developer survey almost 90% of the responding developers reported using Git [60], and GitHub has 81M users [37] and 238M repositories [38]. While many larger project has long established internal guidelines, starting a new project can be a daunting task. In 2017 GitHub announced the Open Source Guides [21] which contains resources for how to run and contribute to open source projects [23]. The goal is to aggregate the best practices in the community to help people get started with creating open source projects. The guidelines cover many different aspects of open source for both contributors and maintainers.

2.7.2 Readme

A README file is a simple way of documenting a project. It should never replace proper documentation, but rather act as an introduction to the project. While every project has different needs, some common sections should be included [40]. The **name** and a short **description** of the project to let others know what it's all about; what its goals are and why it is useful. How to **get started** using the project; including how to install the project and its prerequisites, running tests and examples. If contributions are welcome, and if so, **how to contribute** to the project. It can also be beneficial to include how to get support, roadmaps and a changelog.

2.7.3 Contributing guidelines

A CONTRIBUTING file may be included to explain how to best contribute to the project [40]. It should include information such as how to file a bug report or suggest a new feature. How to set up the environment to be able to run the tests and examples. What types of contributions the project is looking for and to what extent tests are required.

2.7.4 Code of conduct

Especially for larger projects and communities having a code of conduct helps set rules for your project's participants and how they're expected to behave (and how not to behave) [41]. Additionally, it should include information such as where the code of conduct applies; if it's only on GitHub issues and pull requests, if it extends to community chats or events, etc, and who it applies to. It should also include how one can report violations, who receives the reports and how the rules are enforced.

2.7.5 Coding conventions

A general rule when contributing to a project is to follow the coding conventions in the file(s) you're contributing to, but a project may benefit from formalising these rules in a dedicated coding conventions document. This should include the conventions for naming, capitalisation, white space, etc, as well as more complex topics such as best practices relevant to the project [40].

2.7.6 License

The most important distinction between a project with source available and an open source project is the inclusion of a licence permitting the use of the source code. Any source code is copyrighted by default, even without a license or otherwise stating copyright [36], and as such can not be legally utilised by a third party.

Choosing the correct license for a project can be a difficult task. In 2013 GitHub launched the website choosealicense.com [20] and an accompanying license picker [42] (see figure 13) which resulted in an immediate doubling of the number of licensed repositories on github.com [12]. There are theoretically an infinite amount of different licenses out there as anyone is free to write their own. In order to make things easier, the different licences are often split into 2 categories: permissive and copyleft/restrictive [36]. Permissive licenses impose few restrictions and will in general only require the inclusion of the license and an attribution notice, but some licenses such as the Unlicense [14] pose no restrictions at all and dedicate the work to the public domain. Copyleft includes a viral clause requiring all modified and extended versions of the program to be open source as well, usually by requiring it to be released under the same licence [31].

The choice of licence may affect the success of a project. Wang [73] found license restrictiveness to be marginally associated with survival at the initial stage of a project, and similarly, Colazo and Fang [18] advocate that volunteer developers are more attracted to projects under a copyleft license. Contrary, copylefted projects are associated with lower developer permanence, meaning that the initial attractions do not necessarily translate into long term commitment. A restrictive licence will also prevent the software from being used in a closed source project, which may negatively affect the adoption rate [36].

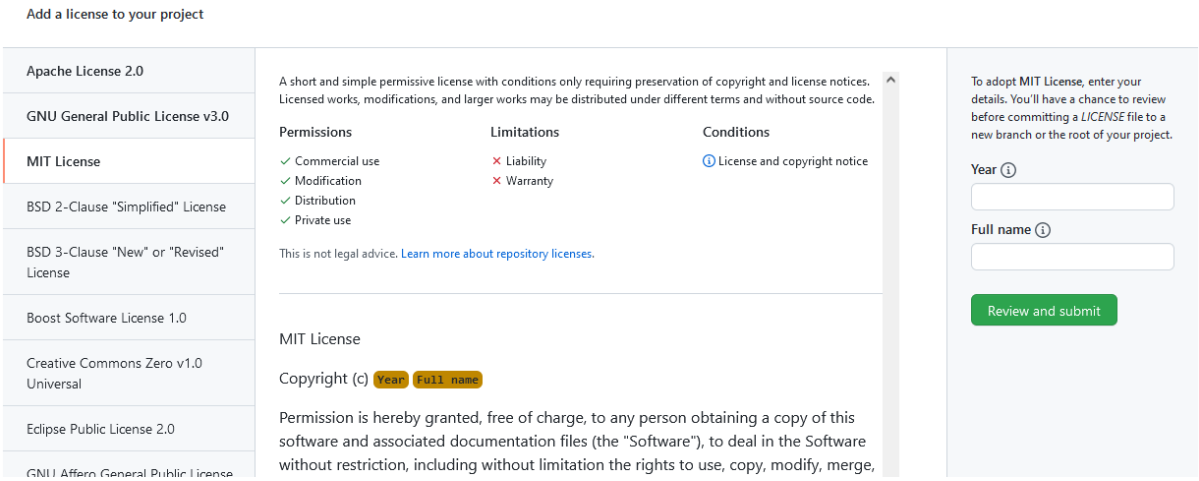


Figure 13: Screenshot of the integrated license picker on GitHub (2021)

2.7.6.1 License alternatives

A selection of some common licences, all of which are listed on choosealicense.com, are described below. All the listed licences permit modification and distribution of the software, as well as commercial use. Private use is permitted with no restrictions, i.e. a modified GPLv3 program can be used in private without the need to disclose the source code.

The Unlicense

The Unlicense [14] is a license which dedicates the work to the public domain. It aims to be short and concise and as such consists of a copyright waiver which imposes no restrictions on the use of the software. As with all of the licences listed here, it also contains a no-warranty statement and liability waiver. While the Unlicense aims to be short and focuses on an anti-copyright message, the FSF recommends CC0[19] over the Unlicense as it's "more thorough and mature" [30].

The MIT License

The MIT License [67] is a short permissive licence which only requires that the licence is included in all copies or substantial portions of the software. It imposes no restrictions on the use of the software.

The Apache License 2.0

The Apache License 2.0 [26] requires the preservation of copyright and license notices as well as any "NOTICE" text file, and any modified files must include a prominent notice stating that it has been changed. A notable difference from the MIT licence is that the Apache License also includes an express grant of patent rights from the contributors to the software, meaning that contributors are prevented from demanding patent royalties in the future for any patents covered by their contributions to the project [52]. All of the following licences also contain some clause on patent use.

The Mozilla Public License 2.0

The Mozilla Public License 2.0 [25] is a weak copyleft licence as it requires the source code of licensed files and modifications of those files to be available under the same license. However additional files constituting a larger work may be distributed under a different licence and without source code.

The GNU General Public License v3.0

The GNU General Public License v3.0 [28] is a strong copyleft license requiring all modifications, derivatives and larger works using a licensed work to have the complete source available under the same license.

The GNU Lesser General Public License v3.0

The GNU Lesser General Public License v3.0 [29] is similar to the GPLv3.0, but makes a distinction between works based on the library and works that use the library. That is, it contains a clause allowing a larger work using the licensed work through interfaces provided by the licensed work to be distributed under a different licence. This is applicable is cases such as using a shared library.

The GNU Affero General Public License v3.0

The GNU Affero General Public License v3.0 [27] is a very strong copyleft license similar to the GPLv3.0. The distinction is that AGPLv3.0 considers providing a service over a network as distribution, and as such public use of a modified version of licensed works, on a publicly accessible server, requires granting the public access to the source code of the modified version. This licence is designed to ensure cooperation with the community in the case of network server software, e.g. by preventing a huge cloud service provider from selling a service based on a modified work without contributing to the public version of the software. One can also argue that by utilising a service, even over a network, one should be considered a user of the program providing the service, and as such it's necessary to provide the source code of such programs to fulfil the user's freedom to study and modify the programs they use.

3 Research Methodology

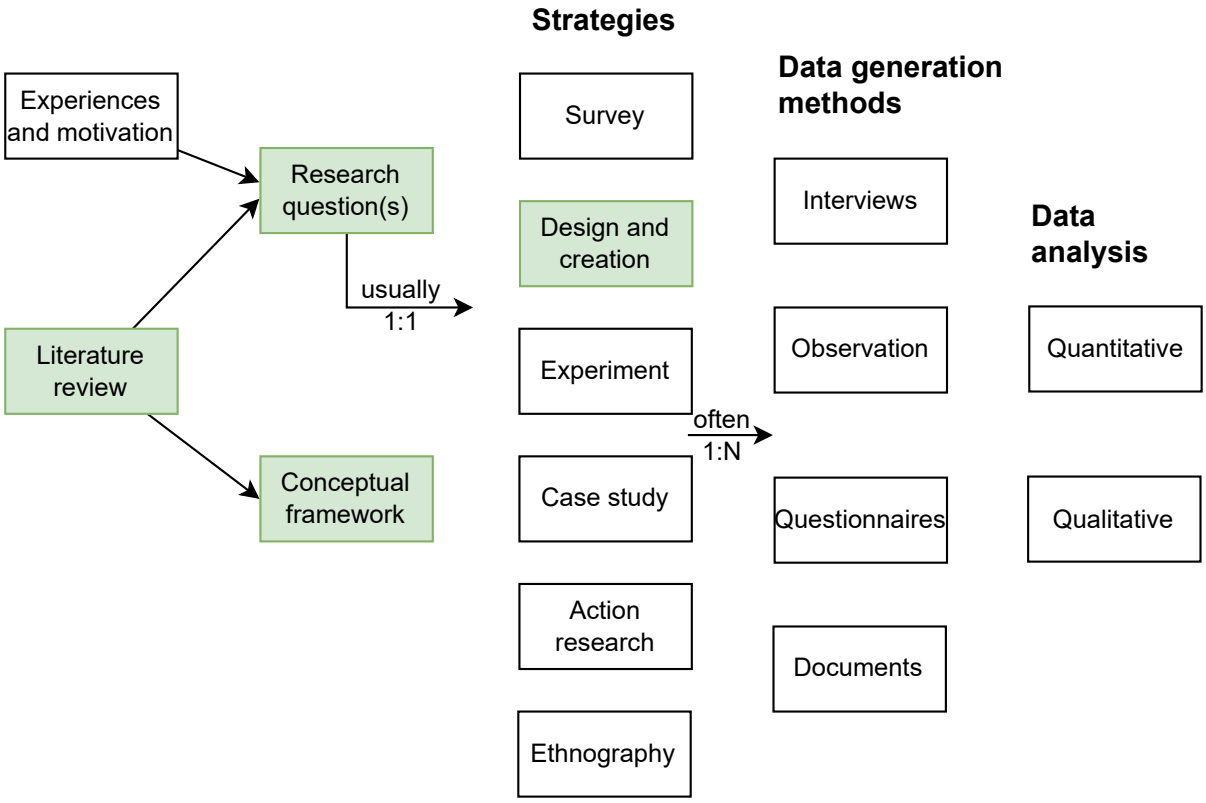


Figure 14: The selected elements from Oates research process model [58].

The research design of the thesis is based on Oates [58] research process model, as shown in figure 14. The methods applied are a literature review, conceptual framework, and design and creation.

3.1 Literature Review

A literature review is the process of reviewing existing literature both before and during a research project. The aim of a literature review may vary a lot based on the current stage of the project. It's commonly used when selecting a topic, and often continues until the work is published. When conducting research it's necessary to conduct a literature review to place the work in the context of the field of research, making sure the work contributes new knowledge or is in another way useful. Literature reviews are also needed to gather knowledge to form the basis on which one can build new knowledge. The literature review helps provide the conceptual framework for the research [58].

3.2 Conceptual Framework

A conceptual framework is derived from the literature review and states how one thinks about the topic and the research process. It should cover the factors that comprise the topic, how one thinks of the topic, how one approaches the research questions, the strategy for analysing the data, how to design and create any artefacts, and how one evaluates the research [58].

3.3 Design and Creation

The design and creation strategy is an iterative process consisting of five steps: awareness, suggestion, development, evaluation, and conclusion [58][72].

1. Awareness: What exactly is the problem? Which criteria are imposed on a solution?
2. Suggestion: How can the problem be solved?
3. Development: Implement the solution.
4. Evaluation: How does the solution compare to the expectations?
5. Conclusion: What did we learn from this? Which aspects need further research?

The first two steps focus on articulating the problem and the solution criteria and creating a tentative design for solving said problem. The next step is creating an artefact based on the tentative design, during the development one may realise unexpected issues with the design or encounter other problems which necessitate changes to the initially proposed solution. These encounters provide new knowledge and raise the awareness of the problem, this is part of the iterative nature of the process. If the artefact is successfully developed it must be evaluated according to the proposed criteria. One must carefully evaluate whether the solution solves the stated problem without deviating from the criteria. The final step is compiling the knowledge gained during the process and presenting it in a proper manner. The knowledge gained is often categorised as either *firm*, repeatable behaviour or facts, or *loose ends*, abnormalities which may form the basis for future research [72]. The process is illustrated in figure 15.

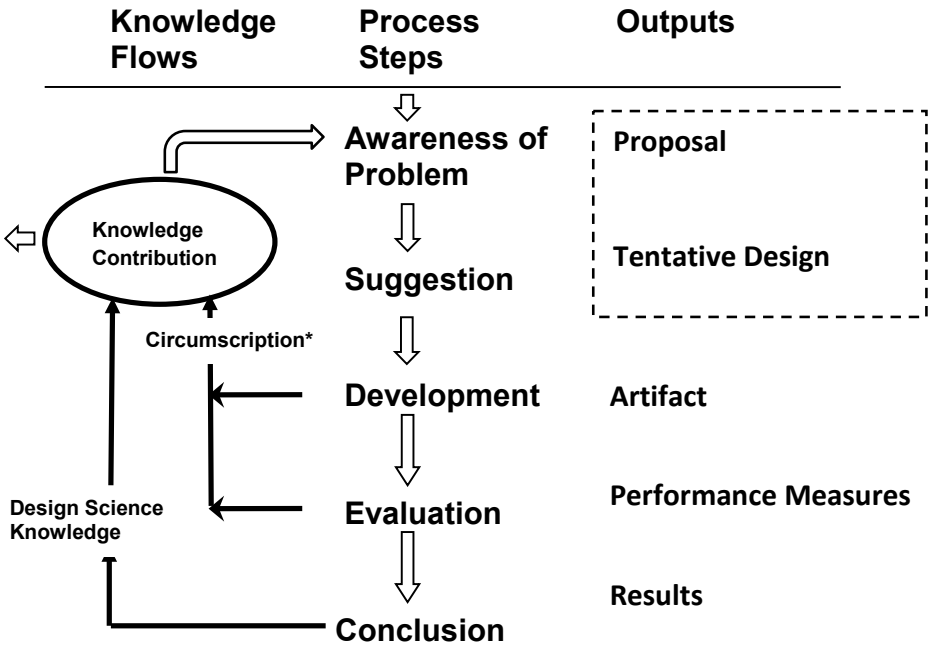


Figure 15: The Design Science Research process model (DSR Cycle) [72].

3.4 Research Questions

The thesis aims to answer the following research questions:

3.4.1 RQ1: What architectural challenges exist in relation to pervasive IoT games?

Before we can start to answer any of the following questions it's necessary to identify the current challenges related to pervasive IoT games. The answer to this question is also a direct dependency to RQ2 as one needs a basis to build the requirements upon. To answer this question a literature review is performed, compiling information from relevant literature as well as the papers this thesis builds upon.

3.4.2 RQ2: What are the requirements for an IoT-based pervasive game architecture and framework?

To create a fair analysis of the two selected architectures it is necessary to form a set of requirements to evaluate them. This research question does not necessitate any information gathering per se, as it builds on the criteria from Bärnholt and Lyngby [13] and the findings from RQ1.

3.4.3 RQ3: Which architectures and technologies may be suited to implement an IoT game framework?

Based on the criteria found in RQ2, we can evaluate different architectures and technologies. As it's not feasible to make numerous prototypes, we can use the criteria to select two promising architectures for the final proof of concept evaluation.

3.4.4 RQ4: How well do the proof of concept implementations match the requirements found in RQ2?

This question forms the basis for the final analysis. Before we can make a proper comparison of the implementations we must first compare each implementation with the common set of requirements. To answer this question two different prototypes, solving the same problem, were created utilising the process of design and creation.

4 Architectural Requirements

4.1 Stakeholders

According to Ian Sommerville [44], a stakeholder is anyone who is affected by the system in some way, i.e. anyone who has a legitimate interest in it. This covers everyone from the end user to the engineers who built the system in the first place.

4.1.1 End users

The end users of the systems are the game developers utilising the frameworks in their game development process. Their main concerns are the ease of use and the capabilities of the frameworks. They depend on any framework to be able to solve the problems they are faced with during the development process. To facilitate this the frameworks need to be flexible and thoroughly documented, including an overview of each feature as well as ready-made examples.

4.1.2 Framework developers

The developers of the frameworks are concerned with good development processes, including cooperation and communication, documentation, and task distribution. To create a successful open source project it's essential to maintain best practices to incentivise community contributions.

4.2 Architecturally Significant Requirements

Chen et al. [17] defines Architecturally Significant Requirements (ASR) as:

> ...those requirements that have a measurable impact on a software system's architecture.

ASRs are the subset of the total system requirements that require extra attention when designing the system's architecture as they are associated with a high cost of change.

4.2.1 Functional and non-functional requirements

Functional requirements are requirements that explicitly state what the system should and should not do. They describe how the system reacts to specific inputs and situations. The requirements can be expressed as general use cases or as very specific details of a process [44]. Non-functional requirements define more general constraints that apply to the system as a whole, rather than specific features [44].

4.2.2 Functionality

4.2.2.1 Transparent linking of digital and physical objects

The main feature of the frameworks is the transparent use of digital twins in pervasive games. The digital objects should match their physical counterpart as closely as possible in the context of game development. Interacting with the physical object through the digital twin should be as easy as interacting with a purely digital object.

4.2.2.2 Centralised game logic

By running the game logic in a central location such as a server or a game running on a PC, the game logic is decoupled from the functionality of each device. This allows independent development of devices and games using said devices. Being able to keep the logic in one place also allows more rapid prototyping and easier deployment as the devices don't need to change.

4.2.2.3 Robust communication

To facilitate a game, the communication between the game and the connected devices must be reliable, and with acceptable latency in the context of the specific game. As the frameworks target a general audience the latency should be minimised to enable use in a broad spectre of games, while still keeping reliability as the main concern.

4.2.3 Quality Attributes

Quality attributes are non-functional requirements used to evaluate the system. Quality attributes are usually architecturally significant [17].

4.2.3.1 Usability

The adoption of the frameworks is dependent on their usability. To be able to reach a wide audience the systems should be familiar to the users of target platforms, in addition to being properly documented.

4.2.3.2 Modularity

One of the main attributes of the frameworks should be modularity. A modular approach is essential to enable developers to develop and integrate their own devices, as well as devices from third parties.

4.2.3.3 Interoperability

To facilitate the use of a large set of devices the systems should be flexible enough to be able to support multiple communication protocols. The due of standardised protocols makes it easier to integrate new devices without designing specialised components for each device.

4.2.3.4 Performance

Performance is an important attribute of many games, and as such it must be considered when designing a framework targeting games. Latency must be kept to a minimum to avoid introducing perceived discontinuity between the digital and physical devices. The framework is unlikely to involve heavy processing as its main occupation is to forward messages to synchronise states. To be able to support a large number of simultaneous devices communication should be kept to a minimum. Using lightweight protocols enables the use of less powerful devices.

4.2.4 Evaluation criteria

The following criteria are derived from the requirements. The criteria are based on the criteria from Bärnholt and Lyngby [13].

C1.1 - Reduce implementation cost	The framework and the technology used should reduce the overall time and cost of prototyping pervasive exergames.
C1.2 - Integration	Seamless integration with the development environment.
C1.3 - Customisation	The framework should allow developers to create custom interactions and visualisations of different devices that mirror the physical device.
C2.1 - Centralised game logic	The framework should support running the game logic in a central location outside of the IoT devices.
C2.2 - Addressability	The technology should provide the ability to uniquely identify and address each device.
C2.3 - Distributed & local use	The technology should support both local and distributed applications.
C3.1 - Interoperability	The framework and the technologies used should support communication with various IoT devices.
C3.2 - Scalability	The framework and the technologies used should support varying amounts of devices.
C3.3 - Connecting new devices	The framework should provide handling of connecting new devices both during development and at run time.
C4.1 - Free and open source	The framework should be open source, as well as being built on open source components.

Table 1: Modified evaluation criteria based on the criteria from Bärnholt and Lyngby [13].

5 Unity Implementation

The solution is an implementation of digital twins aiming to provide Unity game objects nearly identical to the physical device. The physical devices contain no game logic and merely serve as a physical manifestation of the logical object. The communication is done using MQTT, with a custom protocol. A seen in figure 12, the example game is built using ESP8266's where a Raspberry Pi serves as the MQTT broker.

5.1 Unity

Unity [70] is a popular cross-platform game engine which utilises C# as its scripting language. The integrated 3D environment and extendable editor make it a great candidate for this kind of project.

The main building blocks in Unity are the game objects. A game object is often used to represent a single entity in a game, such as a player, a bullet, or an obstacle. The game objects can also be nested, allowing each object to serve as a part of the larger entity or simply to group entities for easier maintainability.

A game object is composed of components. A component can be a custom script acting on the object or its environment, or one of the built-in types such as a *Sprite Render* or a *Rigidbody*. Unity requires all game objects to have a *Transform* component representing the object's position, scale and rotation in 3d-space.

Unity games are usually organised into a number of scenes, where each scene represents a part of the game or a level. A scene is a collection of game objects, lighting information and other environment data.

5.1.1 Manager

The implementation requires the existence of an *ExactManager* in the scene. The manager is responsible for keeping track of all the connected devices and connecting them to their digital twin. It handles incoming messages as well as transmitting the outgoing messages.

5.1.2 Device

A *Device* represents a single device. It handles the connection the the attached components and serves as the base for the digital representation of a physical object.

The link to the physical device can be achieved in a number of ways. It's first linked by the device-id/MAC-address, if no id is provided or the provided it is not a match. The physical device is then linked using the device name and if there's no name match the device is linked by the device type. Failing all three the physical device is not linked to any *Device*.

5.1.3 Device Component

While a physical device could be represented as a single class derived from *Device*, it's often composed of multiple components such as a LED-ring and a RFID-reader. A *DeviceComponent* is a class meant to represent a single a physical component connected to the device. This Device-DeviceComponent composition matches the GameObject-Component philosophy Unity is built on, which should provide an intuitive and recognisable interface for Unity developers.

5.1.4 Inspector

A Unity component's representation in the editor interface is called an inspector. As the name suggests it allows one to inspect the current state of the component, and each component has a default inspector providing an interface for viewing and editing the component's public variables. Like most parts of the Unity editor, the inspector can be customised on a per component basis. One can affect how a field is presented using attributes [54]. The IMU inspector is pictured in figure 16, where we can see that the sensitivity value is displayed as a slider limited to a value between 0 and 1. In addition there's added a button labelled *Tap* at the bottom of the inspector which can be pressed to issue a tap event as if it was issued by the physical device. *On Tap()* is a *UnityEvent* where one can add listeners to the list via drag and drop in the editor, in this case the sole listener is the *OnTapped* function of the *Follow the Red Dot* script on the *GameManager* game object. The function takes one parameter, the device that was tapped, which in the case is the device the IMU is attached to.

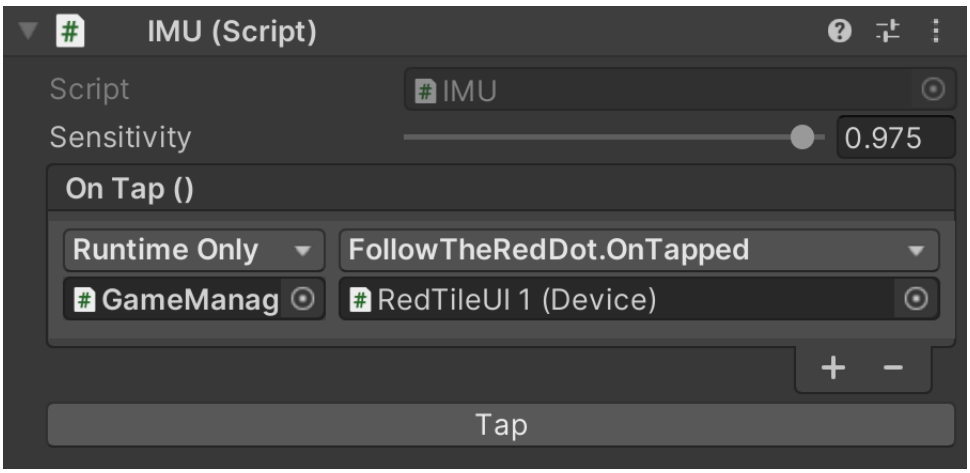


Figure 16: Screenshot of the IMU inspector.

5.1.5 Events

A *UnityEvent* is a callback that can be invoked from code. It can't take any arguments but has the nice property that it's persistent and listeners can be assigned via the editor. This allows connecting otherwise completely decoupled code with a simple drag and drop interface. Though the callback can't be invoked with arguments one can still bind a function that requires them by assigning a value to the arguments in the editor. This is demonstrated in figure 16.

Using C# attributes one can assign a callback function to listen for changes done to a variable through the editor. In the example implementation, this is utilised to immediately synchronise changes of a value on a twin to the physical device.

5.2 Message Protocol

Connection

The protocol follows a discovery pattern. As there's no direct coupling between Unity and the devices the protocol relies on both parts broadcasting their existence by publishing to specific topics when they want to connect. A device publishes to *exact/connected* when powered on to try to connect to Unity. If Unity is running, it'll attempt to link the device to a digital twin. On startup Unity will publish to *exact/all_ devices/are_ you_ connected* to prompt any already

powered devices to reconnect. Figure 17 demonstrates a case where Unity is started after the device.

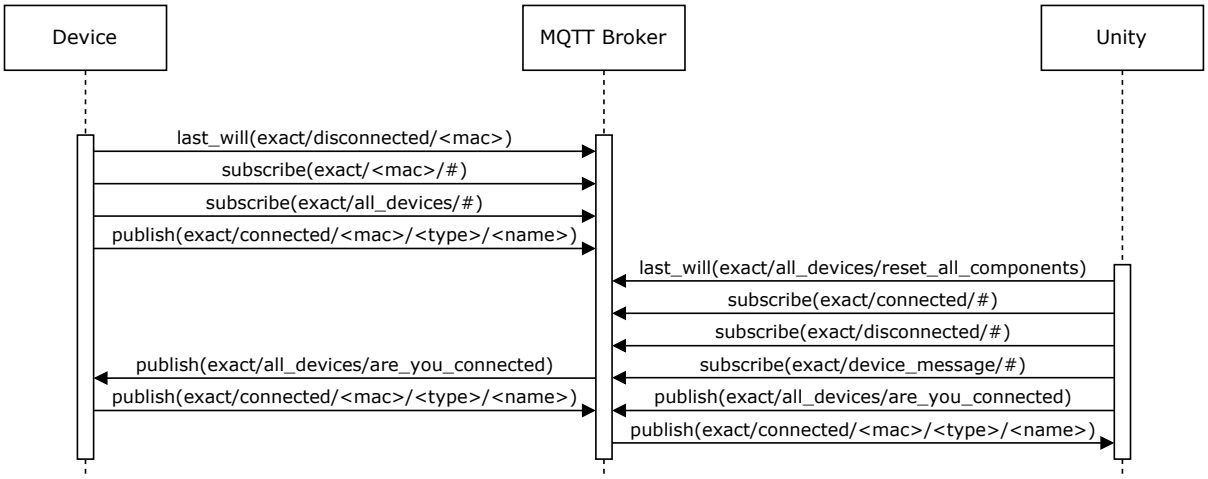


Figure 17: Diagram of the connect and disconnect protocol.

Last will

Last Will and Testament (LWT) is a built-in feature of MQTT. A client may specify a last-will message when connecting to the MQTT broker. The broker with publish this message if the event that the client ungracefully disconnects. As seen in figure 18, the protocol utilises LWT to notify Unity when a device disconnects. Similarly, Unity has a last will to have the devices reset into a ready state in the event of a game crash or network loss. Unity's last will can be seen in figure 17.

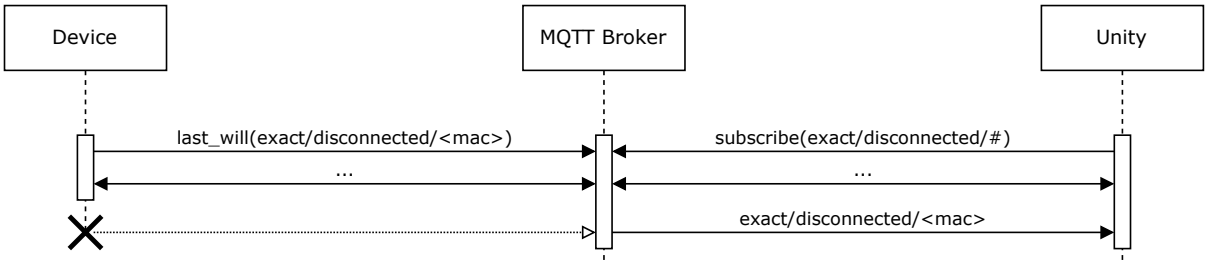


Figure 18: Diagram of last will.

Action

An action message is a message sent by Unity whenever it wants to perform an action on a physical device. This can be anything from turning on a LED to setting the angle of a servo motor.

The topic of the message is *exact/<mac>/action/<component>/<action_name>* where *<mac>* is the unique identifier of the device, *<component>* is the component that should perform the action and *<action_name>* is a string identifying the action to be performed. Any supplementary data is stored in the payload. Any message starting with *exact/<mac>* indicates that it's a message sent from Unity to a specific device.

An example request to set the colour of an RGB-led to red could have the topic *exact/00-11-22-33-44-55/action/rgb_led/set_color* and payload *255,255,0*.

Event

An event message is a message from a device to notify Unity of an event triggered on said device. An event can be a button being pressed, motion detected, or any other change we might be interested in.

The topic of the message is *exact/device_message/<mac>/event/<component>/<event_name>* where *device_message* indicates this is a message sent from a device. As with the action message *<mac>* is the unique identifier of the device, *<component>* is the component where the event occurred and *<event_name>* is a string identifying the type of event. An event can also include extra data in the payload.

An example event triggered by a user tapping an IMU may be sent with the topic *exact/device_message/00-11-22-33-44-55/event/imu/tap*. This example is shown in figure 16.

Get

The get message is for Unity to request a value from a component in a physical device. The device then responds with a value message. The message has a topic on the form *exact/<mac>/get/<component>/<variable_name>*. The structure is similar to an action message except that is cannot include a payload.

A request such as wanting to know the value of a temperature sensor may look like this: *exact/00-11-22-33-44-55/get/temp_sensor/current_temp*.

Value

A value message is the device's response to a get message from Unity. The message may also be used for a value that is continuously updated without Unity having to repeatedly send get messages. The message has the same structure as an event message with the topic *exact/device_message/<mac>/value/<component>/<variable_name>* and where the value of the requested variable is stored in the payload.

5.3 Messages in Unity

A message sent from a device to Unity is propagated through several layers. The MQTTHandler maintains the connection to the MQTT broker and is the component that receives the raw message. Connect and disconnect messages are processed immediately, while events and value messages are matched and forwarded to a connected device. The device simply sends the message to the correct device component where it's processed. The three-layer division helps map the logical objects to the physical ones, as well as exposing APIs at varying abstraction levels, creating more choices for the user.

As seen in figure 19, an outgoing message is gradually composed by the different layers. Similarly, 20 show how an incoming message is decomposed as it's propagated down and through the layers.

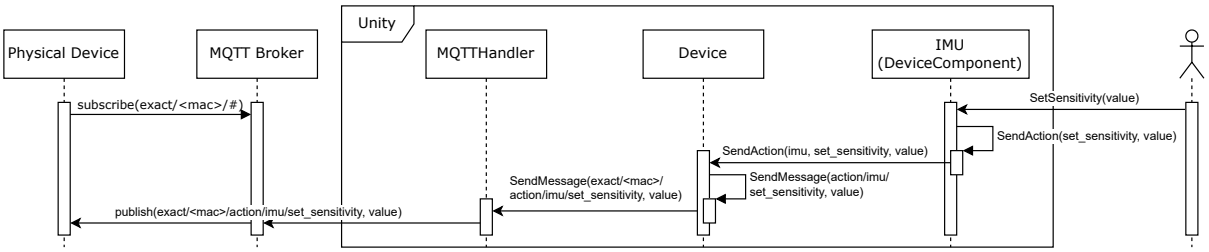


Figure 19: Diagram of the message propagation in Unity when the user changes the sensitivity of the IMU.

The top layer, the device components, are built to make the MQTT connection transparent to the gameplay programmer. Figure 20 demonstrates the twin behaviour of the physical device and the in-game object; an event caused by tapping the IMU is identical to an event from pressing the corresponding in-game button.

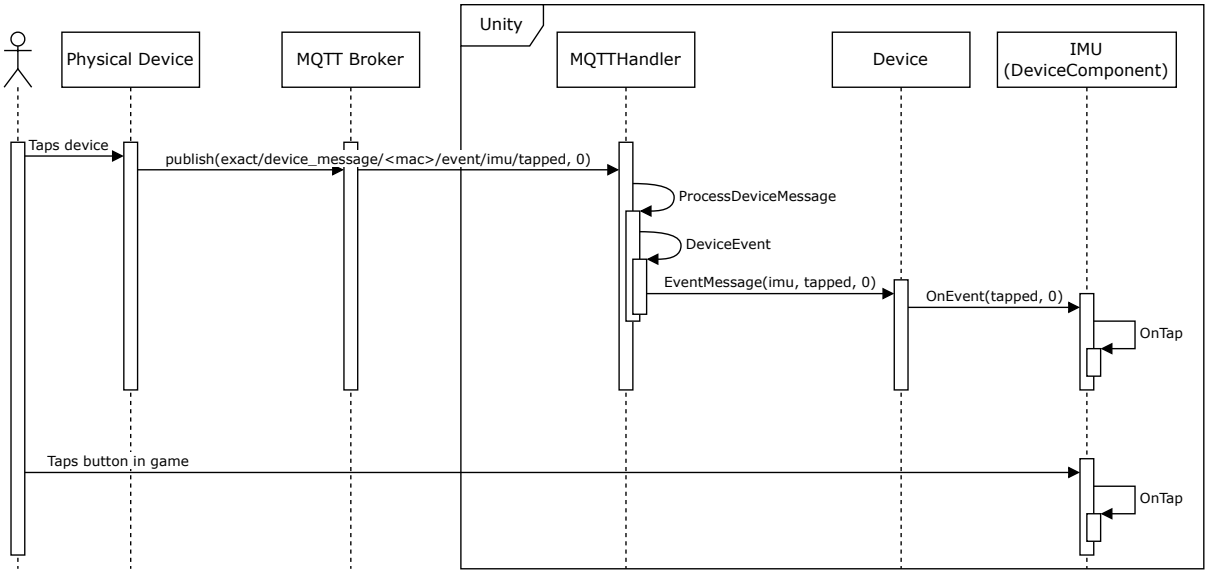


Figure 20: Diagram of the message propagation in Unity when the user taps on a physical IMU, compared to pressing the twin-button in the GUI.

5.4 Follow the Red Dot

Follow the Red Dot is a simple, yet effective game for testing the capabilities of the implementation. The game consists of three or more tiles with a light and a button each. One on the tiles will light up and the player has to tap the tile. When the tile is tapped the light is turned off and another tile lights up, and so on. If the player takes too long to tap a tile the game is over.

While the rules are simple, this is a non-trivial problem to implement on three separate micro-controller-enabled tiles. Moving the game logic to a centralised component in Unity allows easier prototyping and makes it possible to quickly iterate on the specific rules, e.g. one can adjust the time the player has to tap the tile before the game resets, or adjust the method used to select the next tile, without the need to flash all the micro-controllers each time.

5.4.1 The tiles

To accurately model the tiles the following device components were made: the IMU, LED ring, tone player, and RFID reader. In addition, a dice detector component was created, which subscribes to events from the RFID reader to display the value of any RFID-enabled dice placed on the tiles (see figure 2a). Though, the RFID reader was not utilised in the prototype game.

5.4.2 Game logic

The game is governed by a single script, the GameManager, containing the game logic. The GameManager is notified when a device is tapped, be it the physical device or the in-game button, through an event call to the OnTapped function (listing 1), where the device that issued the event is passed as an argument. The event is connected through the Unity editor, as seen in figure 16. This way there's no hard coupling between the GameManager and the IMU, and any other device may be utilised as well.

```
public void OnTapped(Device device)
{
    if (device != active) { return; }
    scoreKeeper.Score++;
    SetNextActive();
}
```

Listing 1: The function called when a tile is tapped.

The process of changing the active device is made trivial by using the ready-made device components. As seen in listing 2, by using the interface provided by the LedRing-component, both the physical LED-ring and the visual representation are turned off.

```
var led = active.GetComponent<LedRing>();
led.StopFading();
led.SetColor(Color.black);
```

Listing 2: Example of turning off the led ring on the active device.

After selecting the newly active device, one can turn on the LED ring of the new device in a similar fashion. Listing 3 demonstrates how one could instruct the led ring to fade from full intensity to completely off in a span of 5 seconds. As the ring turns on a tone is played for a short duration to guide the player in the right direction.

```
active.GetComponent<LedRing>().StartFading(Color.red, 1.0f, 0.0f, 5.0f);
active.GetComponent<TonePlayer>().PlayTone(500.0f, 0.1f);
```

Listing 3: Example of turning on the led ring and playing a tone on the active device.

6 AWS Implementation

The AWS implementation is based on AWS IoT Core. The devices are registered as things in AWS IoT and are granted the required permissions to be able to communicate through the AWS message broker. The game logic for the Follow the Red Dot example game is implemented as a simple Python script running on an EC2 instance with the same permissions.

6.1 Device

The devices are Raspberry Pis equipped with a LED and a button (figure 21). Raspberry Pi was chosen due to its combination of small size and the ease of prototyping.

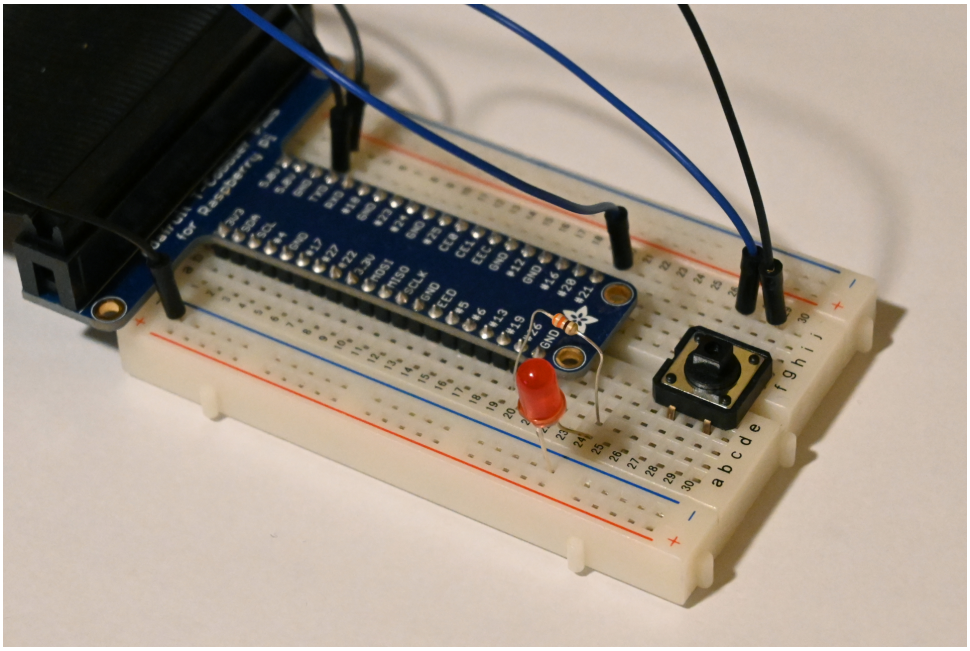


Figure 21: A led and a button connected to the Raspberry Pi.

6.1.1 Circuit

A button should normally be connected in a pull-up or pull-down configuration, but the Raspberry Pi has internal pull-up resistors making this redundant. The LED is connected with a 330Ω resistor. The circuit is shown in figure 22.

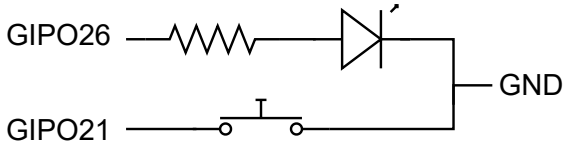


Figure 22: Circuit diagram of the led and button connected to the Raspberry Pi.

6.1.2 Logic

Most of the logic is already supplied by the AWS IoT Device SDK [6]. The code is based on an example of using device shadows. When the device receives a shadow delta message

containing the differing desired values from the reported ones the device can turn on or off the LED according to the desired state. When the button is clicked the device publishes a MQTT message to the topic *things/<device_ name>/click*.

6.1.3 Shadows

The implementation relies on the device shadows to update the state of each device. This way the server isn't required to talk to any device directly. An added bonus it that any updates sent when the device is offline are delivered when it reconnects.

6.1.3.1 Create

The first time a device connects to AWS there's no existing shadow document for that device. When the device tries to request the document it's rejected and the device submits a new document with its current state. This is illustrated in figure 23.

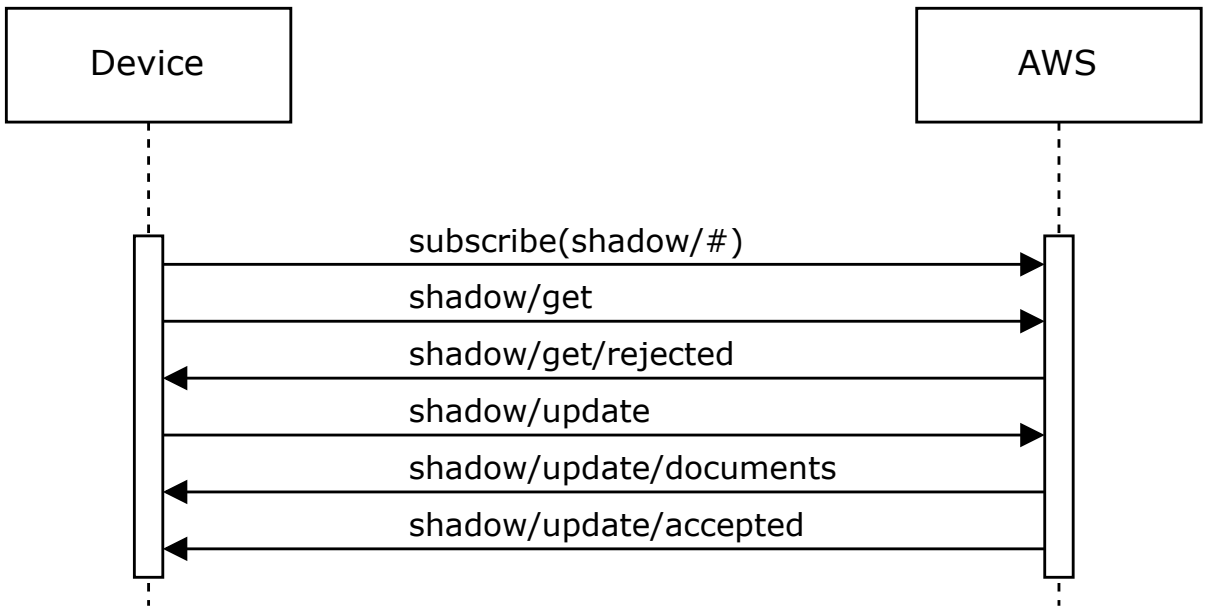


Figure 23: Diagram of the MQTT messages exchanged when a device connects without an existing shadow document.

6.1.3.2 Update

Figure 24 shows an example where the shadow document already exists. When the device connects it's able to retrieve the document and apply any changes to its state. If the state is changed it'll publish an updated document with the reported state as seen in figure 23.

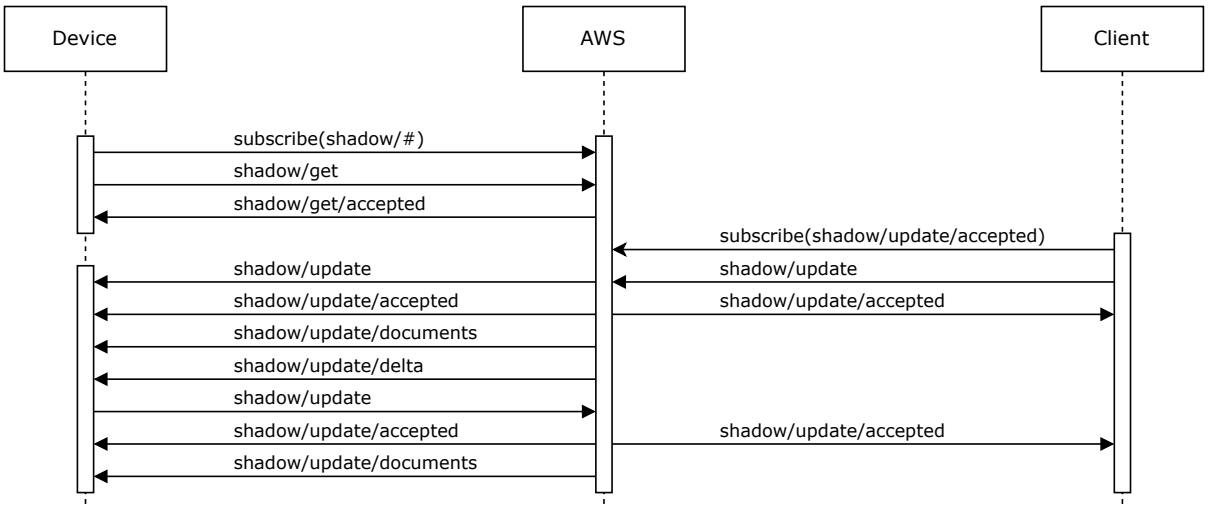


Figure 24: Diagram of the MQTT messages exchanged when a device connects, and later a client updates the device shadow.

6.2 Server

The server is set up as a Python script running on an AWS EC2 instance.

6.2.1 EC2

Elastic Compute Cloud (EC2) is a scalable virtual computing service allowing the user to choose among numerous configurations [10]. It was a good fit for this project as the free tire is more than capable and the integration with the AWS MQTT service makes it convenient to set up communication with the devices.

6.2.2 Game logic

The server maintains a list of the connected devices and the currently active device. When it receives a *click* message from the active device, it uses the shadow service to turn off the LED on said device and turns on the LED on a new device making it the active device. Setting the state of a device is just a matter of publishing a MQTT message with the desired state. The function for setting the state of the LED is shown in listing 4.

```
def set_device_light(device_name, light_state):
    message_topic = f"{aws_prefix}/{device_name}/shadow/update"
    message_json = f'{"state": {"desired": {"light": "{light_state}"}}}'
    mqtt_connection.publish(
        topic=message_topic,
        payload=message_json,
        qos=mqtt.QoS.AT_LEAST_ONCE)
```

Listing 4: The function called when a tile is tapped.

7 Analysis

7.1 Evaluation criteria

Following the implementation of the example game, both solutions were compared to the evaluation criteria found in table 1.

C1.1 Reduce implementation cost

> The framework and the technology used should reduce the overall time and cost of prototyping pervasive exergames.

Both solutions made prototyping both faster and easier at no extra cost. Using the Unity framework most of the time was spent developing the various device components for the red tiles. The relatively small interface of the framework is able to support the development of complex component behaviours. With the use of the ready-made device components, it was easy to develop the Follow the Red Dot game. The AWS implementation does not support device abstraction to the same extent as the Unity implementation, though this is feasible to implement given some time. The AWS device SDK made it very quick to set up synchronisation of the LED connected to the device with the device shadow document. Requesting changes to the document from the game logic was also very straightforward.

C1.2 Integration

> Seamless integration with the development environment.

One of the design goals of the Unity framework was to seamlessly integrate it into the Unity editor. When developing the example game, the use of the device components was familiar to the use of other Unity components. The AWS solution was not built with a particular development environment in mind. Thought, this minimal approach of not using a game engine does remove some distractions.

C1.3 Customisation

> The framework should allow developers to create custom interactions and visualisations of different devices that mirror the physical device.

As the state of the physical device is synced to the device components in Unity, setting up a visualisation of the device becomes trivial. Custom interactions are supported by using the interfaces each component supplies, as was done in the example game. The AWS solution does not provide any means of displaying any type of visualisation, but connecting an additional device for display purposes is feasible. Custom interactions can be implemented in code even though there's currently no explicit support for this.

C2.1 Centralised game logic

> The framework should support running the game logic in a central location outside of the IoT devices.

Both solutions support, and rely, on centralised game logic. Both solutions implemented the devices as pure I/O devices with very little logic contained in them. The Unity solution implemented the game logic in a script running in the game engine, while the AWS solution ran the

logic in an EC2 instance. The AWS solution has the added benefit of having the capability to run the game logic on one of the devices.

C2.2 Addressability

> The technology should provide the ability to uniquely identify and address each device.

Both solutions are able to uniquely address each device. In Unity, this is enforced by the protocol where the message topics contain the MAC-address of the relevant device. There's still the possibility of having poorly configured devices, or devices which purposely misidentify themselves. AWS has strict enforcement of identities through the use of certificates.

C2.3 Distributed & local use

> The technology should support both local and distributed applications.

The Unity solution is primarily designed for local use but also supports distributed deployment by utilising a remote MQTT broker. The game logic can also run remotely. The AWS solution is heavily dependent on AWS and as a result, there's no way to run it locally. The game logic can run on a local device or PC.

C3.1 Interoperability

> The framework and the technologies used should support communication with various IoT devices.

Through the use of MQTT, both solutions are able to support a variety of devices. The solutions each have a custom MQTT protocol that the devices must explicitly support. One can feasibly implement an adaptor to support third-party devices.

C3.2 Scalability

> The framework and the technologies used should support varying amounts of devices.

The Unity implementation is limited by the message throughput in both the MQTT broker and Unity. Depending on the game and the devices, Unity may slow down when there's a large number of devices connected due to Unity's normal performance constraints when handling many game objects. The AWS solution is configurable to handle near any number of devices through the various distribution and load balancing features of cloud providers like AWS.

C3.3 Connecting new devices

> The framework should provide handling of connecting new devices both during development and at run time.

Both solutions are able to handle the connection of new devices both during development and at run time. In Unity, one can link each device to a specific instance of a virtual device during development. One can also use the device type the device reports when connecting to link it to a device in a specific group or to instantiate a new device of the respective type to handle any number of new devices. The AWS solution does not have logically distinct device types, so the devices must be handled in code. To connect a device to AWS the device must first be

registered, this can be done by explicitly issuing a certificate to each device. To support new devices one can load a claim certificate on each device during production. The claim certificate can then be exchanged for a normal certificate the first time the device connects to AWS.

C4.1 Free and open source

> The framework should be open source, as well as being built on open source components.

Neither solution is completely open source, but both utilise some open source components. The Unity solution is largely open course with the exception of Unity itself. The C# source of the Unity Engine and the Unity Editor has been available on GitHub since 2018, but it's licensed with reference only licence [3][71]. The licence only permits reading the code, not modifying or redistribution. The AWS Device SDK is open source under the Apache License 2.0 [6], but not AWS itself. While the Unity framework can be made completely open source by porting it to a different engine, the same would be challenging with the AWS solution as few cloud service providers open source their services.

7.2 Summary

The comparison is summarised in table 2.

	Unity-things	AWS EC2
C1.1	Both solutions makes prototyping both faster and easier.	
C1.2	The framework is integrated into the Unity editor. No framework is provided for the devices.	The solution does not provide any integration for the game logic. The devices utilise the AWS IoT SDK.
C1.3	The Unity editor streamlines the setup of custom interactions and visualisations.	The solution provides no framework for custom interaction and visualisation but allows for the implementation of most features.
C2.1	Both solutions support centralised game logic.	
C2.2	Both solutions support device identification.	
C2.3	The solution supports both local use and distributed use by hosting the MQTT broker remotely	As the solution relies on AWS it does not support local use.
C3.1	Both solutions support a variety of devices through MQTT.	
C3.2	Unity is able to support a great number of devices.	An AWS solution can be configured to support nearly any number of devices.
C3.3	Devices are automatically detected when they try to connect to the game.	Devices need to be registered to AWS. This can be done in multiple ways including bulk registration or using a claim certificate that allows automatic registration the first time the device connects.
C4.1	Open source with the exception of Unity itself.	Open source with the exception of AWS.

Table 2: The criteria from table 1 in relation to the proposed solutions.

8 Discussion

8.1 RQ1: What architectural challenges exist in relation to pervasive IoT games?

One of the challenges regarding the difficulties of the development of pervasive IoT games is the ease of prototyping and rapid iteration. This may stem from multiple separate issues such as using ad hoc solutions for each project which leads to a lack of mature development tools. Dealing with physical devices is usually more cumbersome when compared to other objects in a game, and the lack of proper integration of the devices results in unwanted complexity. To support rapid iteration it may be useful to have the game logic in a central location, this simplifies the deployment process as the devices can be kept as is when only the behaviour of the game is changed. If the logic is handled by one or more devices it's necessary to go through the potentially time-consuming process of updating the devices for each iteration.

As a cost-saving measure, it's desirable to be able to support off-the-shelf devices. While many devices support standardised communication protocols such as MQTT, the structure of the topics and messages still vary. A general solution must be able to provide adaptors to devices with varying message structures.

Most communication methods used in games are not necessarily suitable for use in IoT devices. It's challenging to design a communication method which satisfies the requirements of performance, reliability, and latency required by some games while still supporting a variety of different devices.

A production-ready architecture should be designed with security and privacy in mind to prevent eavesdropping and malicious behaviour in competitive games. Securing the communication to the devices may require more powerful devices, and may complicate the process of connecting new devices.

8.2 RQ2: What are the requirements for an IoT-based pervasive game architecture and framework?

Following the two implementations, the criteria for IoT-based pervasive game architectures found in chapter 4 appear to apply. The requirements are still highly dependent on the application, but for a general-purpose solution, all of the criteria should preferably be met.

Regarding the criteria of scalability, no exact figure for the number of devices was given as the number of devices needed will vary wildly depending on the application. This requirement is regarded as the least important for a general-purpose product as the architecture of a framework for distributed games supporting thousands or millions of devices may look very different to a framework for use in the home of a single player.

While the AWS solution communicates with the AWS shadow documents, the Unity framework is implemented using an ad hoc protocol. An additional requirement may state that the architecture should be built on open and standardised protocols as this will make it easier to integrate new devices and may increase the adoption rates of the framework.

8.3 RQ3: Which architectures and technologies may be suited to implement an IoT game framework?

With the variety of available technologies, it is challenging to create a comprehensive comparison. Following the literature review, this thesis has considered two separate architectures built on different technologies.

The Unity Things technology stack described in section 2.6 utilises a Raspberry Pi MQTT broker, ESP8266 for the devices, and a PC running the game logic in Unity. The ESP8266 is a staple IoT component used in numerous applications, both commercially and DIY alike. When designing a new solution it's often beneficial to use a mature product which is likely to be supported in the future. MQTT is a largely adopted, lightweight, and human-readable protocol, all of which make it easier to work with. MQTT was chosen for both solutions and successfully satisfied its role.

The Raspberry Pi for running a message broker and DHCP server was chosen to simplify the setup of the devices. The devices rely on a hard-coded network SSID and password, as there's no solution implemented for communicating this information to the devices. The Raspberry Pi provides a known network for the game and devices to connect to. This solution is not ideal and would not be feasible for a commercial product. With a more flexible device setup, the Raspberry Pi would be redundant and the MQTT broker could be hosted remotely, or on the PC along with the game. The use of a Raspberry Pi in the devices for the AWS solution made for very easy prototyping. It is well-documented and provides easy-to-use libraries to get started.

Using a game engine provides many features out of the box. With some familiarity, it makes it trivial to set up simple interactions and visualisations. Unity was chosen for this particular framework but any other engine would provide many of the same benefits. The use of a large game engine comes with a cost in terms of resource consumption. If the game requires remote hosting of the logic, using Unity may be more costly compared to a more minimal solution.

Cloud technologies have seen widespread use in IoT applications. Contrary to the first solution, by connecting the devices to a remote broker like AWS the devices appear to be stand-alone, only requiring an internet connection. The integrated broker, which is accessible from other AWS services is a very powerful technology.

8.4 RQ4: How well do the proof of concept implementations match the requirements found in RQ2?

The Unity solution largely meets all of the requirements. C3.2 is difficult to judge without performing a large-scale test, which is outside the possibilities of this thesis. The solution fails on the requirement of standardised protocols discussed in RQ2.

The AWS architecture meets most of the requirements. C1.2 is not met as the implementation does not target any particular environment for the game logic. C1.3 is not met largely for the same reason; the implementation is very bare bones and not representative of all the features that may be included in a complete AWS solution. C2.3 is not met as the architecture largely inherits the cloud architecture of AWS and, to my knowledge, there's no way to host the services locally. The AWS solution is more suited for a large number of devices where managing the devices on your own becomes infeasible. Using a cloud platform has many benefits regarding scalability and performance, in the ease of setting up devices with the SDK.

9 Conclusion

The goal of this thesis was to explore the possibilities and challenges surrounding the development of pervasive games by conducting a comparative study of two selected architectures. Four research questions were derived to achieve this task.

First, a literature review was performed to identify the challenges that exist in relation to pervasive IoT games. Based on the literature review and the previous findings of Bärnholt and Lyngby [13] we were able to identify several challenges which formed the basis for the next question.

The second research question was to derive a set of requirements for an IoT-based pervasive game architecture and framework. Based on the criteria from Johansen [51] a modified set of requirements where formed by considering the challenges identified in research question 1. Said requirements can be found in table 1.

Two potential architectures were selected based on the criteria found in research question two. The Unity-Things framework proposed by Bärnholt and Lyngby [13] and a cloud-based solution built on AWS. The AWS solution was known to violate requirement C2.3 but was selected due to its differences compared to the Unity-based solution as well as the prevalence of cloud applications in the IoT space.

Following this, two potential solutions were implemented. One of the solutions was a re-implementation of the Unity-Things framework, with deeper integration in the Unity game engine and a component-based design. The other solution was based on the AWS shadow services, utilising Raspberry Pi as the devices.

To answer research question four, how well do the proof of concept implementations match the requirement requirements created in research question two, the two solutions were individually evaluated by the requirements. The Unity solution was found to largely meet the criteria. The AWS solution failed on some points, which is believed to be due to the minimal implementation of the solution.

Both solutions were found to be viable alternatives for building pervasive games. A simpler cloud-based solution may perform better for creating e.g. a global Follow the Red Dot game, while a game engine-based solution is more suited for applications that take advantage of the graphical futures the engine has to offer.

References

[1] Tim Althoff, Ryen W White, and Eric Horvitz. "Influence of Pokémon Go on physical activity: study and implications". In: *Journal of medical Internet research* 18.12 (2016), e315.

[2] A. Amies et al. *Developing and Hosting Applications on the Cloud: Develop Hosting Applica Cloud*. IBM Press. Pearson Education, 2012. ISBN: 9780133066852. URL: https://books.google.no/books?id=4gwIYbtTH5MC.

[3] Aras Pranckevičius. *Releasing the Unity C# source code*. 2018. URL: https://blog.unity.com/technology/releasing-the-unity-c-source-code.

[4] Kevin Ashton et al. "That 'internet of things' thing". In: *RFID journal* 22.7 (2009), pp. 97–114.

[5] AWS. *AWS homepage*. 2022. URL: https://aws.amazon.com.

[6] AWS. *AWS IoT Device SDK v2 for Python on GitHub*. 2022. URL: https://github.com/aws/aws-iot-device-sdk-python-v2/.

[7] AWS. *AWS IoT Device Shadow service*. 2022. URL: https://docs.aws.amazon.com/iot/latest/developerguide/iot-device-shadows.html.

[8] AWS. *How AWS IoT works*. 2022. URL: https://docs.aws.amazon.com/iot/latest/developerguide/aws-iot-how-it-works.html.

[9] AWS. *Rules for AWS IoT*. 2022. URL: https://docs.aws.amazon.com/iot/latest/developerguide/iot-rules.html.

[10] AWS. *What is Amazon EC2?* 2022. URL: https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/concepts.html.

[11] AWS. *What is AWS IoT?* 2022. URL: https://docs.aws.amazon.com/iot/latest/developerguide/what-is-aws-iot.html.

[12] Ben Balter. *Open source license usage on GitHub.com*. 2015. URL: https://github.blog/2015-03-09-open-source-license-usage-on-github-com/ (visited on 11/2021).

[13] Magnus Bärnholt and Andreas Schatvet Lyngby. "An Internet-of-Things Software Framework for Exergames". MA thesis. NTNU, 2019.

[14] Arto Bendiken. *The Unlicense*. 2010. URL: https://spdx.org/licenses/Unlicense.html (visited on 11/2021).

[15] Steve Benford, Carsten Magerkurth, and Peter Ljungstrand. "Bridging the physical and digital in pervasive gaming". In: *Communications of the ACM* 48.3 (2005), pp. 54–57.

[16] Craig Chapple. *Pokémon GO Catches $5 Billion in Lifetime Revenue in Five Years*. URL: https://sensortower.com/blog/pokemon-go-five-billion-revenue (visited on 12/2021).

[17] Lianping Chen, Muhammad Ali Babar, and Bashar Nuseibeh. "Characterizing Architecturally Significant Requirements". In: *IEEE Software* 30.2 (2013), pp. 38–45. DOI: 10.1109/MS.2012.174.

[18] Jorge Colazo and Yulin Fang. "Impact of license choice on Open Source Software development activity". In: *Journal of the American Society for Information Science and Technology* 60.5 (2009), pp. 997–1011. DOI: https://doi.org/10.1002/asi.21039. eprint: https://asistdl.onlinelibrary.wiley.com/doi/pdf/10.1002/asi.21039. URL: https://asistdl.onlinelibrary.wiley.com/doi/abs/10.1002/asi.21039.

[19] Creative Commons. *CC0 1.0 Universal*. URL: https://spdx.org/licenses/CC0-1.0.html (visited on 11/2021).

[20] Curated by GitHub, Inc and 100+ contributors (https://github.com/github/choosealicense.com). *Choose an open source license*. URL: https://choosealicense.com/licenses/ (visited on 11/2021).

[21] Curated by GitHub, Inc and 200+ contributors (https://github.com/github/opensource.guide). *Open Source Guides*. URL: https://opensource.guide (visited on 11/2021).

[22] Dave Evan, Cisco. *The Internet of Things - How the Next Evolution of the Internet Is Changing Everything*. 2011. URL: https://www.cisco.com/c/dam/en_us/about/ac79/docs/innov/IoT_IBSG_0411FINAL.pdf.

[23] Nadia Eghbal. *Announcing Open Source Guides*. 2017. URL: https://github.blog/2017-02-14-announcing-open-source-guides/ (visited on 12/2021).

[24] Espressif Systems. *ESP8266 product page*. 2022. URL: https://www.espressif.com/en/products/socs/esp8266.

[25] Mozilla Foundation. *Mozilla Public License 2.0*. 2012. URL: https://spdx.org/licenses/MPL-2.0.html (visited on 11/2021).

[26] The Apache Software Foundation. *Apache License, Version 2.0*. 2004. URL: https://spdx.org/licenses/Apache-2.0.html (visited on 11/2021).

[27] Free Software Foundation, Inc. *GNU Affero General Public License v3.0 or later*. 2007. URL: https://spdx.org/licenses/AGPL-3.0-or-later.html (visited on 11/2021).

[28] Free Software Foundation, Inc. *GNU General Public License v3.0 or later*. 2007. URL: https://spdx.org/licenses/GPL-3.0-or-later.html (visited on 11/2021).

[29] Free Software Foundation, Inc. *GNU Lesser General Public License v3.0 or later*. 2007. URL: https://spdx.org/licenses/LGPL-3.0-or-later.html (visited on 11/2021).

[30] Free Software Foundation, Inc. *Various Licenses and Comments about Them*. URL: https://www.gnu.org/licenses/license-list.en.html#Unlicense (visited on 11/2021).

[31] Free Software Foundation, Inc. *What is Copyleft?* URL: https://www.gnu.org/copyleft/ (visited on 11/2021).

[32] Free Software Foundation, Inc. *Free Software Foundation*. URL: https://www.fsf.org/ (visited on 11/2021).

[33] Free Software Foundation, Inc. *Various Licenses and Comments about Them*. URL: https://www.gnu.org/licenses/license-list.html (visited on 11/2021).

[34] Free Software Foundation, Inc. *What is Free Software?* 2019. URL: https://www.gnu.org/philosophy/free-sw.html (visited on 11/2021).

[35] Free Software Foundation, Inc. *Why Open Source Misses the Point of Free Software*. URL: https://www.gnu.org/philosophy/open-source-misses-the-point.html (visited on 11/2021).

[36] Brian M. Gaff and Gregory J. Ploussios. "Open Source Software". In: *Computer* 45.6 (2012), pp. 9–11. DOI: 10.1109/MC.2012.213.

[37] GitHub, Inc. *Search*. 2021. URL: https://web.archive.org/web/20211127002147/https://github.com/search.

[38] GitHub, Inc. *Search*. 2021. URL: https://web.archive.org/web/20211119120759/https://github.com/search.

[39] Edward Glaessgen and David Stargel. "The Digital Twin Paradigm for Future NASA and U.S. Air Force Vehicles". In: *53rd AIAA/ASME/ASCE/AHS/ASC Structures, Structural Dynamics and Materials Conference*. DOI: 10.2514/6.2012-1818. eprint: https://arc.aiaa.org/doi/pdf/10.2514/6.2012-1818. URL: https://arc.aiaa.org/doi/abs/10.2514/6.2012-1818.

[40] Open Source Guides. *Starting an Open Source Project*. URL: https://opensource.guide/starting-a-project/ (visited on 12/2021).

[41] Open Source Guides. *Your Code of Conduct*. URL: https://opensource.guide/code-of-conduct/ (visited on 12/2021).

[42] Phil Haack. *Choosing an Open Source License*. 2013. URL: https://github.blog/2013-07-15-choosing-an-open-source-license/ (visited on 11/2021).

[43] HiLetgo. *ESP8266 product page*. 2022. URL: http://www.hiletgo.com/ProductDetail/1906570.html.

[44] Ian Sommerville. *Software Engineering*. 10th ed. Pearson Education Limited, 2016. ISBN: 9781292096131.

[45] "IEEE Standard for Information technology—Telecommunications and information exchange between systems Local and metropolitan area networks—Specific requirements - Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications". In: *IEEE Std 802.11-2016 (Revision of IEEE Std 802.11-2012)* (2016). DOI: 10.1109/IEEESTD.2016.7786995.

[46] The Open Source Initiative. *History of the OSI*. URL: https://opensource.org/history (visited on 11/2021).

[47] The Open Source Initiative. *The Open Source Definition*. 2007. URL: https://opensource.org/docs/osd (visited on 11/2021).

[48] The Open Source Initiative. *The Open Source Initiative*. URL: https://opensource.org (visited on 11/2021).

[49] Intel. *Choose the Right Communication Pattern for Your IoT Project*. 2016. URL: https://www.intel.com/content/www/us/en/developer/articles/technical/communication-patterns-for-the-internet-of-things.html (visited on 05/2022).

[50] Jari Due Jessen and HH Lund. *Evaluation and Understanding of Playware Technology-Trials with Playful Balance Training*. Technical University of Denmark, 2016.

[51] Petter Bakkan Johansen. "Iot-based pervasive game framework-a proof of concept case study". MA thesis. NTNU, 2018.

[52] Jeffrey Robert Kaufman. *How to make sense of the Apache 2 patent license*. 2018. URL: https://opensource.com/article/18/2/how-make-sense-apache-2-patent-license (visited on 11/2021).

[53] E. I. Konstantinidis et al. "The interplay between IoT and serious games towards personalised healthcare". In: *2017 9th International Conference on Virtual Worlds and Games for Serious Applications (VS-Games)*. 2017, pp. 249–252. DOI: 10.1109/VS-GAMES.2017.8056609.

[54] Microsoft (15 contributors). *Attributes (C#)*. 2022. URL: https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/attributes.

[55] Ahmadreza Montazerolghaem, Mohammad Hossein Yaghmaee, and Alberto Leon-Garcia. "Green Cloud Multimedia Networking: NFV/SDN Based Energy-Efficient Resource Allocation". In: *IEEE Transactions on Green Communications and Networking* 4.3 (2020), pp. 873–889. DOI: 10.1109/TGCN.2020.2982821.

[56] News 10, A Gray Media Group, Inc. Station. *Shelter dogs benefit from Pokemon Go craze; Gamers are helping walk dogs.* URL: https://www.wilx.com/content/news/Pokemon-Go-craze-benefits-shelter-dogs-386795261.html (visited on 12/2021).

[57] OASIS. *MQTT Version 5.0.* 2019. URL: https://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.html.

[58] Briony J Oates. *Researching information systems and computing.* Sage, 2005.

[59] Yoonsin Oh and Stephen Yang. "Defining exergames & exergaming". In: *Proceedings of meaningful play* 2010 (2010), pp. 21–23.

[60] Stack Overflow. *Developer Survey Results 2018.* 2018. URL: https://web.archive.org/web/20190530142357/https://insights.stackoverflow.com/survey/2018/#work-_-version-control (visited on 12/2021).

[61] Raspberry Pi Foundation. *Raspberry Pi homepage.* 2022. URL: https://www.raspberrypi.com/.

[62] Nina Skjæret et al. "Exercise and rehabilitation delivered through exergames in older adults: An integrative review of technologies, safety and efficacy". en. In: *Int. J. Med. Inform.* 85.1 (Jan. 2016), pp. 1–16.

[63] Sruti Subramanian et al. "ExerTiles: A Tangible Interactive Physiotherapy Toolkit for Balance Training with Older Adults". In: *Proceedings of the 32nd Australian Conference on Human-Computer Interaction.* OzCHI '20. Sydney, NSW, Australia: Association for Computing Machinery, 2021, pp. 233–244. ISBN: 9781450389754. DOI: 10.1145/3441000.3441043. URL: https://doi.org/10.1145/3441000.3441043.

[64] Tarja Susi, Mikael Johannesson, and Per Backlund. *Serious Games: An Overview.* IKI Technical Reports. 2007. URL: http://urn.kb.se/resolve?urn%20=%20urn:nbn:se:his:diva-1279.

[65] Fei Tao et al. "Digital twin-driven product design, manufacturing and service with big data". In: *The International Journal of Advanced Manufacturing Technology* 94.9 (Feb. 2018), pp. 3563–3576. ISSN: 1433-3015. DOI: 10.1007/s00170-017-0233-1. URL: https://doi.org/10.1007/s00170-017-0233-1.

[66] R.N. Taylor, N. Medvidovic, and E. Dashofy. *Software Architecture: Foundations, Theory, and Practice.* Wiley, 2009. ISBN: 9780470167748.

[67] Massachusetts Institute of Technology. *MIT License.* 1980. URL: https://spdx.org/licenses/MIT.html (visited on 11/2021).

[68] The Pi Hut. *Raspberry Pi 4 Model B product page.* 2022. URL: https://thepihut.com/products/raspberry-pi-4-model-b.

[69] International Telecommunication Union. *Overview of the Internet of things.* 2012. URL: https://www.itu.int/ITU-T/recommendations/rec.aspx?rec=y.2060.

[70] Unity Technologies. *Unity.* 2022. URL: https://unity.com.

[71] Unity Technologies. *Unity C# reference source code on GitHub.* URL: https://github.com/Unity-Technologies/UnityCsReference.

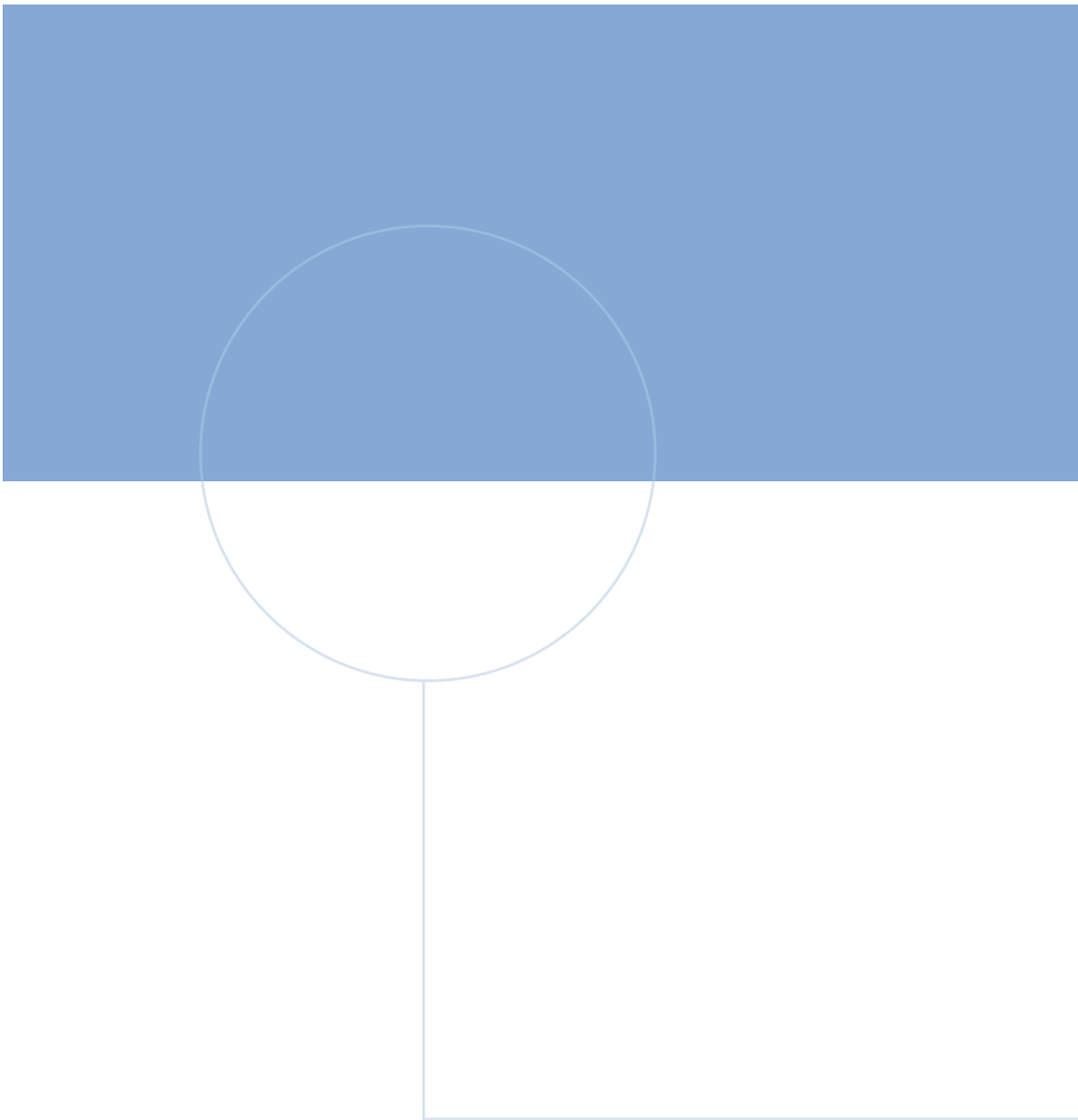
[72] Vijay Vaishnavi and William Kuechler. “Design research in information systems”. In: (2004).

[73] Jing Wang. “Survival factors for Free Open Source Software projects: A multi-stage perspective”. In: *European Management Journal* 30.4 (2012), pp. 352–371. ISSN: 0263-2373. DOI: https://doi.org/10.1016/j.emj.2012.03.001. URL: https://www.sciencedirect.com/science/article/pii/S0263237312000199.

[74] Josef Wiemeyer and Annika Kliem. “Serious games in prevention and rehabilitation—a new panacea for elderly people?” In: *European Review of Aging and Physical Activity* 9.1 (2012), pp. 41–50. DOI: 10.1007/s11556-011-0093-x.

[75] Wikipedia contributors. *ESP8266 — Wikipedia, The Free Encyclopedia*. 2022. URL: https://en.wikipedia.org/wiki/ESP8266#SDKs.

[76] Felix Wortmann and Kristina Flüchter. “Internet of things”. In: *Business & Information Systems Engineering* 57.3 (2015), pp. 221–224.



NTNU

Norwegian University of
Science and Technology