NTNU
Norwegian University of
Science and Technology

IoT-Based pervasive game framework

A proof of concept case study

Petter Bakkan Johansen

Master of Science in Computer Science
Submission date: May 2018
Supervisor: Dag Svanæs, IDI

Norwegian University of Science and Technology
Department of Computer Science

Executive Summary

With the Internet of Things (IoT) playing a key role in the Fourth Industrial Revolution it is clear that it will become a larger part of our day to day life. IoT offers unique capabilities for having tiny sensors and embedded systems take part in the Internet and expand the way we use the Internet today. Researchers are currently researching the benefits of utilizing playful digital technology to encourage health promoting activities for both children, adolescents, and elderly, and pervasive games have shown promising signs both for reducing sedentary behavior and as tool for rehabilitation for elderly stroke victims. The combination of pervasive games and IoT is at the time of writing a fairly unexplored field of research and the research done in this paper seeks to help establish the state of the art and provide a framework that could help drive the research forward.

In this master thesis, a framework for prototyping and developing pervasive games that utilize Internet of Things was developed and evaluated in a proof-of-concept manner. The evaluation was done with a proposed technology stack that included a Raspberry Pi Zero W, several RFduinos, the Bluetooth Low Energy and Wi-Fi protocols, the MQTT protocol, and the Unity game engine. The chosen technologies were based on a list of requirements that were proposed as suited features for supporting prototyping and development of IoT-based pervasive games. The list of requirements were established based on a state of the art presentation on IoT, Cloud technology, pervasive games, and exergames.

The proof-of-concept evaluation was based around a case study with the game ”Follow the Red Dot”, a simple pervasive game where a ”dot” is transfered between different devices when players interact with the device that is currently in control of the dot. ”Follow the Red Dot” was chosen as the case as its structure allows for implementations in both local and distributed settings, in addition to supporting a virtual mirroring of the active game devices. The case study was split into three different cases to provide insights into different attributes of the technologies that were evaluated.

The results showed that the technology stack satisfied all the requirements that were initially set for technologies. The framework developed also provided the intended functionality when integrated with the suggested stack. There were however some issues experienced with the RFduinos and the discussion suggests that there likely exists better alternatives that would provide the same functionality, but in more reliable fashion. The results also brings forth that developing a stack that is suited for every type of IoT-based pervasive games isn't really feasible as the pervasive game genre includes so many different sub-genres that vary in architecture, interaction, and potential IoT usage. The research finally proposes that wider and more specific research should be performed on the different technologies that exists within each of the layers, as the limitations of this thesis restricted the amount of research done on the different technologies.

Sammendrag

Med tingenes internetts (IoT) sentrale rolle i "Den fjerde industrielle revolusjonen" er det klart at det vil utgjøre en større del av vårt daglige liv. Tingenes internett tilbyr unike evner for å knytte små sensorer og integrerte systemer opp mot internettet og utvide måten vi benytter internettet i dag. Forskere undersøker for tiden fordelene med å benytte lekne digitale teknologier for å stimulere til helsefremmende aktiviteter for både barn, ungdom, og eldre, og pervasive spill har vist lovende tegn både for å redusere stillesittende aktivitet og som verktøy for rehabilitering av eldre slagrammede. Kombinasjonen av pervasive spill og tingenes internett er i skrivende stund et relativt uutforsket forskningsfelt, og forskningen som er gjort i denne avhandlingen skal bidra til å skape et bilde av eksisterende forskning på temaet og utvikle et rammeverk som skal bidra til å drive forskningen fremover.

I denne masteroppgaven ble et rammeverk for prototyping og utvikling av pervasive spill som benytter seg av tingenes internett utviklet og evaluert som et proof of concept. Evalueringen ble gjort basert på en foreslått teknologi-stack som inkluderte en Raspberry Pi Zero W, flere RFduinoer, Bluetooth Low Energy og Wi-Fi protokollene, MQTT protokollen, og spillmotoren Unity. Teknologiene ble valgt ut basert på en liste med kriterier som var foreslått som passende egenskaper for å støtte prototyping og utvikling av pervasive spill som benytter seg tingenes internett. Kravlisten ble etablert på bakgrunn av eksisterende forskning på tingenes internett, sky-teknologi, pervasive spill, og exergames.

Evalueringen av proof of conceptet var basert på en casestudie med spillet ”Follow the Red Dot”, et enkelt pervasivt spill der en ”dott” blir sendt mellom ulike enheter når spillere samhandler med den enheten som er i besittelse av dotten. ”Follow the Red Dot” ble valgt som case på grunn av at strukturen dens støtter både lokale og distribuerte implementasjoner i tillegg til at den støtter virtuell speiling av de aktive spillenhetene. Casestudiet var delt inn i tre forskjellige caser for å kunne gi innsikt i de forskjellige egenskapene til teknologiene som ble evaluert.

Resultatene viste at teknologi-stacken tilfredsstiller alle kravene som i utgangspunktet ble satt for teknologiene. Rammeverket som ble utviklet støttet også den tiltenkte funksjonaliteten når det ble integrert sammen med den foreslåtte stacken. Det ble også opplevd noen problemer med RFduinoene og diskusjonen rundt resultatene foreslår at det høyst sannsynlig eksisterer bedre alternativer som vil gi samme funksjonalitet, men på et mer pålitelig vis. Resultatene viser også til at det å velge en stack som passer for alle typer IoT-baserte pervasive spill ikke er gjennomførbart på grunn av at den pervasive spillsjangeren inneholder så mange forskjellige undersjangerer som varierer i arkitektur, interaksjon og potensiell bruk av tingenes internett. Til slutt så foreslår forskningen at bredere og mer spesifikt forskning på de ulike teknologiene som eksisterer i de ulike lagene bør utføres, siden begrensningene på denne avhandlingen begrenset mengden forskning som ble gjort på de ulike teknologiene.

Acknowledgment

I would like to thank my supervisor professor Dag Svanæs for great assistance and feedback throughout the entire project. This project would not have been realized without your help. It has been truly amazing to get to work with someone as exited and affectionate about the research as you.

I would also like to thank everyone who offered to babysit and helped take care of my daughter throughout the project, giving me the opportunity to commit the needed amount of time and resources towards the research.

Petter Bakkan Johansen

Trondheim, 10. mai 2018

Contents

Chapter 1

Introduction

Bluetooth LE (Bluetooth Smart) and ANT+ System on Chip (SoC), e.g. from Nordic Semiconductor in Trondheim, enables wireless communication between a high amount of cheap, small programmable devices. This is often refereed to as the Internet of Things (IoT). IoT opens up for a wide array of exiting applications, e.g. within sports and games. This research project aims to develop a proof-of-concept of a software framework used for IoT in pervasive games. The general idea is to create a framework that is able to support digital mirroring of physical devices, often known as Digital Twins, and digital manipulation of the device state through a game engine, e.g. Unity Engine. The framework should be usable for prototyping and developing pervasive games that make use of IoT technology.

1.1 Motivation

Being included as key part of what is deemed the Fourth Industrial Revolution, the Internet of Things offers exciting possibilities for applications that target anything from watering your plant while you are gone on holiday to major sensor networks that are used to predict when the tail of airplane needs maintenance. IoT has in recent years moved being a buzzword used by tech enthusiasts to offering smart house applications for anyone interesting in creating a smart home.

Figure 1.1: The pervasive piano stairs.

At the same time there has been an increasing amount of research done on how we can utilize technology as health promoting tool, e.g. to mitigate the increasing sedentary behavior of children, adolescents, and adults, or as a rehabilitation tool for elderly stroke or fall victims. A part of that research has focused on how playful digital technology help encourage the use of technology as a health promoting tool and pervasive games have been proven to have positive implications both for children and elderly users. A relevant example is the piano stairs, shown in figure 1.1, that appeared in cities like Stockholm, Milan, and Auckland, that encouraged people to take the stairs instead of the escalators by playing notes when walking up the stairs.

The motivation behind this research is to provide insights into how pervasive games can utilize IoT technology in order to take advantage of the unique features and the increased availability that IoT offers. The research project aims to develop and evaluate a proof of concept of a framework, built on suitable technologies, that eases the prototyping and development of IoT-based pervasive games. This framework will offer other developers the ability to speed up prototyping and development of pervasive games that make use of IoT technology, allowing for more focus on the potential positive effects of pervasive games.

1.2 Related work

As there, to the best of my knowledge, exists very little research on IoT-based pervasive games, a lot of the relevant work relates to IoT and pervasive games as separate themes. Atzori et al. (2010) gives a thorough description of IoT and presents useful insights into how IoT is perceived by presenting the different orientations of the definitions of IoT. Grieves and Vickers (2017) presents the Digital Twin concept, and its history and current applications. With pervasive games being a big collection of different game genres with diverse and inconsistent definitions, Jegers (2009) offers a useful introduction into the topic of pervasive games and presents background information

that relates to why it is so difficult to define pervasive games as a genre. MacDowell and Endler (2015) presents the article that relates the most to the topic of this research as it presents a suggestion for interaction models that would support the development of IoT-based pervasive games. Finally Brauner et al. (2013) and Gao and Chen (2014) presents insights into exergames ability to be used as a health promoting tool. Chapter 2 presents a detailed state of the art on the relevant topics.

1.3 Objectives

The main objective of this paper is to develop a proof of concept of a framework for prototyping and developing pervasive games that utilize IoT technology. As a part of the research the paper presents the state of the art of IoT-based pervasive games and relevant technologies that are suited to support the developed framework. The technologies and framework is evaluated in a proof-of-concept evaluation. The research is guided by the four research questions presented in the following section.

Research Questions

RQ1 *What existing pervasive games, documented in the scientific literature, utilize IoT technology?*

RQ2 *What are the requirements for IoT technology that makes it easy to prototype pervasive games?*

RQ3 *What are the available tech stacks to realize such a prototyping environment and what is a promising stack for doing proof-of-concept evaluation?*

RQ4 *How suited is the selected stack for creating such a prototyping environment?*

1.4 Contribution

This paper seeks to contribute to the research done on pervasive games that utilize IoT technology by presenting the state of the art of the topic and researching technologies that are available to support the development of IoT-based pervasive games. The paper also develops a proof of concept of a framework that is meant to ease prototyping and development of IoT-based pervasive games for other researchers in order to assist the research on the topic. The paper also brings forth some examples of how pervasive games is and could be utilized as a health promoting tool to mitigate increased sedentary behavior and help with rehabilitation of the elderly.

1.5 Limitations

This research project performs a proof of concept evaluation on the framework that was developed and only seeks to display the frameworks features and ability to support the requirements set in the evaluation criteria. The research does not include a fully implemented framework. The scope of the paper limits the research done on technologies that could be suitable for the proposed technology stack that is used to support the developed framework. A more thorough research into these technologies could bring forth aspects and features of the individual technologies that could affect the outcome of the research. The limitations of the research also affects the supported features of the framework as with more time the framework could have been expanded with additional functionality and a better standard for the API offered by the framework.

1.6 Outline

Chapter 2 establishes a theoretical background on the topics of Internet of Things (IoT), Cloud technology, pervasive games, and exergames. The background information is used to present existing IoT-based pervasive games and help establish evaluation criteria for IoT technology and the framework.

Chapter 3 presents the research methodology used to perform this research and answer the research questions established.

Chapter 4 gives a description of the three cases used in the evaluation of the different technologies and the prototyping framework, in addition to presenting "Follow the Red Dot", which is the pervasive game that each of the three cases are built around.

Chapter 5 establishes the evaluation criteria that are based on the background information and are used for evaluating the proposed stack, associated technology, and the prototyping framework related to research questions 2-4.

Chapter 6 presents the proposed stack for supporting a proof-of-concept evaluation of the prototyping framework. It also presents background information on different technologies that exists within the different layers of the stack, and finally presents the set of technologies chosen to perform the proof-of-concept evaluation.

Chapter 7 presents the design and creation process, including initial concept designs and the iterative implementation of each of the three cases.

Chapter 8 and 9 presents and discusses the results of the research. The proposed technology stack is also evaluated in a before and after implementation relation in order to display the experienced features of the stack compared to the theoretical features.

Chapter 10 and 11 concludes the research and presents recommendations for further work based on the research done.

Chapter 2

Background

2.1 Internet of Things

Claiming to have coined the phrase "Internet of Things" back at a presentation for Procter & Gamble in 1999, Kevin Ashton (Ashton (2009)) introduced the world to a phrase that almost 20 years later has gained a lot of momentum and attention. Even though the Internet of Things Global Standards Initiative (IoT-GSI) in 2012 presented a recommended definition of IoT (Union (2012)), research literature still presents a wide array of different definitions (Union (2012); Atzori et al. (2010)). The wide range of definitions may be an indicator of the strong interest that IoT has received and the lively debates around the topic. Another reason is the syntactic structure of the phrase itself. Being built up of the terms 'Internet' and 'Things' leads to definitions that either has an "Internet oriented" or a "Things oriented" focus, which in turn creates substantial differences in the definitions (Atzori et al. (2010)). Together the terms produces the semantical meaning of "a world-wide network of interconnected objects uniquely addressable, based on standard communication protocols" (epo (2008)).

The 'Things oriented' perspective of IoT ranges from a focus on very simple Things, i.e. Radio-Frequency IDentification (RFID) tags, to the augmentation of the Things' intelligence to make it sustainable, enhanceable, and uniquely identifiable (Sterling (2005)). A common vision within the Things oriented perspective is the ability to provide object visibility, that is the ability to trace an object and be aware of its status, current location, etc. There are different visions of how this visibility should be achieved. Sakamura (Sakamura (2006)) presents the middleware based Ubiquitous ID (uID) which assigns an unique identifier, called ucode, and locations to objects in order to differentiate unique objects. Meanwhile RFID is being widely adopted in commercial environments mainly due to its maturity and low cost, but it will remain as one part of the big IoT puzzle that will also contain Near Field Communications (NFC), Wireless Sensor and Actuator Networks (WSAN), as well as many other communication protocols that incorporate more advanced Things into the IoT.

In the 'Internet Oriented' perspective the focus lies on promoting the use of the

Internet Protocol (IP) as the network technology that should help move from the Internet of Devices to the Internet of Things. One proposed approach is Internet-0 (Zero) which gives everyday objects the ability to connect to a data network without having to develop new and elaborate protocols (Gershenfeld et al. (2004)). Another approach is presented from the IP for Smart Objects (IPSO) Alliance, it is an simplification of the Internet Protocol that makes it usable on micro-controllers and low-power wireless links without being too resource-intensive, as was the initial assumption of many of the vendors that embraced proprietary protocols (Culler and Chakrabarti (2009)).

Atzori et al. (2010) further introduces the "Semantic orientation", which relates to representing, storing, interconnecting, searching, and organizing information generated by IoT devices. This orientation looks at how different semantic technologies can be utilized to address the issues that might occur when the number of IoT devices skyrockets within the near future[1]. Some of the key challenges that the semantic orientation address' is how to interconnect and organize the information that is generated by the IoT, and how the data best should be stored (Toma et al. (2009)).

The Digital Twin concept

One of the key emerging concepts within the Internet of Things over the last decade is the concept of the Digital Twin. The Digital Twin model, previously referred to as the Mirrored Space Model or the Information Mirroring Model (Grieves and Vickers (2017)), consists of two systems, one physical and one virtual, where the virtual system contains information about the physical system. The two systems exists in a way that allows the two to share information with each other and any alterations on either of the two should be mirrored on the counter part. While the virtual system in many cases serves as a detailed description of the physical system, the virtual system could also be used to predict future behavior and actions of the physical product, meaning that events or actions that could be threating to the lifecycle of the physical system could be predicted and avoided by the virtual system.

For large industrial companies that produce components where product lifecycle management is imperative, like NASA (Glaessgen and Stargel (2012)) and aircraft producers (Tuegel et al. (2011)), the Digital Twin is already a key part of their development. The Digital Twin concept allows companies gather data that could be used to predict component failures, detect anomalies, and evaluate development processes by having the data be processed and simulated by high-performance computers.

[1] According to a Juniper Research study released in March 2017, the amount of IoT devices in the retail environment, including RFID and BLE beacons, will number 12.5 billions within 2021 (RFID Journal).

2.2 Cloud

A key aspect of the IoT vision is making it ubiquitous, reliable, efficient, and scalable, all of which are attributes that Cloud computing posses (Biswas and Giaffreda (2014)). As the IoT vision is being realized more and more of the larger companies, like Microsoft, Google, Amazon, etc., and even local national companies like Telenor here in Norway, are providing Cloud services for IoT. Cloud Computing can be characterized as a "business model to empower omnipresent, on-demand network access to a collection of configurable computing resources such as storage, services, networks, servers, and applications that can be quickly provided and released with minimal management effort or service provider interaction" (Babu et al. (2015)). There are two approaches to the merging of IoT and Cloud computing, one being a Cloud-based IoT and the other being IoT-Centric Cloud.

2.2.1 Cloud-based IoT

The Cloud-based IoT approach seeks to bring IoT functionality into the Cloud, rather than bringing the Cloud functionality to IoT. In this approach the sensed data collected from IoT devices gets stored and processed in the Cloud and makes use of the Clouds boundless processing and storage capabilities. This in turn leads to the possibility to enhance the functionality of the Cloud through the use of IoT data with services like SaaS (Sensing as a Service) (Rao et al. (2012); Zaslavsky et al. (2013); Dash et al. (2010)), SAaaS (Sensing and Actuation as a Service) (Rao et al. (2012)), SEaaS (Sensor Event as a Service) (Rao et al. (2012); Dash et al. (2010)), DaaS (Data as a Service) (Zaslavsky et al. (2013)), and many more (Babu et al. (2015)).

2.2.2 IoT-Centric Cloud

The IoT-Centric Cloud paradigm extends Cloud computing and services to the edge of the network with the objective of reducing latency and high traffic, and to support mobile computing and data streaming (Biswas and Giaffreda (2014)). By moving the Cloud functionality closer to the user and/or sensors it creates a dense geographical distribution allowing for data processing and service execution locally, which also increases the security. A similar model to the IoT-Centric Cloud is the Fog. Fog computing is "a highly virtualized platform that provides compute, storage, and networking services between end devices and traditional Cloud Computing Data centers" (Bonomi et al. (2012)). Instead of moving and distributing the Cloud itself closer to the edge of the network, the fog extends the cloud with fog nodes that can be deployed anywhere with a network connection (Cisco (2015)). These nodes have the capability to perform real-time computations and store data generated at the edge, and send aggregated data to the Cloud, minimizing latency for computational tasks, offloading network traffic, and

providing another level of security for sensitive data before it is sent into the Cloud.

2.3 Pervasive Games

Pervasive games has been a widely researched topic over the last decade and is still a hot topic as the utilization of pervasive or ubiquitous computing becomes more widespread. Unlike the ubiquitous computing vision, which the idea of pervasive games sprung from, the definitions of pervasive games are diverse and inconsistent. One of the main reasons for this is the fact that the term 'pervasive games' is used to denote a wide collective of different sub-genres of computer games, some of them being alternate reality games, augmented reality games, cross media games, and location based games (Jegers (2009)).

Although diverse, the definitions and descriptions of pervasive games have some commonalities which together represents an understandable description of what pervasive games are. The use of ubiquitous computing makes up a large part of what can be used to describe pervasive games. Through the use of non-invasive computational devices pervasive games seeks to bring digital gaming experiences into the real physical world (Benford et al. (2005)) and by doing so they expand in the social, temporal and spatial dimensions of the Magic Circle (Montola (2005)). To clarify, the concept of the Magic Circle was introduced by Johan Huizinga in Homo Ludens (Huitzinga (1944)), a book where he studied the play-element in culture. The Magic Circle sets the social, temporal and spatial dimensions of a game by establishing rules for who are allowed to join, when it is played, and where it is played. Even though traditional games also have the ability the extend its set dimensions, i.e. by creating custom rules, pervasive games have the ability to extend these dimensions to a much larger degree, and in some cases even removing the restrictions on one or more dimensions. This is what distinguishes the pervasive game genre from other game genres, and this is why pervasive games so often are quoted as "games that can be played anywhere at anytime"(Kasapakis and Gavalas (2015); Jegers (2009)).

2.3.1 The Development of Pervasive Games

The technology used in the genre of pervasive games has been evolving continuously since the early days when custom equipment, wearable computers, PDAs, and feature phones made up the most common player equipment (Kasapakis and Gavalas (2015)). With the introduction of smartphones developers quickly started exploiting its key features and in the period between 2009 and 2014 it was the most commonly used equipment for pervasive games. In recent years developers have identified the advantages of integrating wearable technology together with the use of smartphones as it provides the ability to gather additional sensor data that can be utilized within the pervasive

games. Also the use of IoT in pervasive games is currently being researched as these tiny sensing devices coupled with modern wireless communication technology can be used to create a whole new generation of pervasive games that aren't restricted to the technology within smartphones and wearables.

The approach to developing pervasive games has followed one of two different strategies, the first one being to take an traditional, real-world game and augment it with computing functionality, while the other is taking a computer game and mapping it onto real-world settings (Guo et al. (2012)). BotFighters (Olli (2002)), Treasure (Guo et al. (2012)) and Capture the Flag (Sreekumar et al. (2006)) are examples where traditional outdoor games have been augmented in a way that expands the dimensions of the original magic circle of the game. In Capture the Flag the traditional outdoor game has been augmented with smartphones and simple Linux-based Bluetooth device that allows for the creation of a virtual representation of the playing field which remote players can interact with while physical players interact within the real-world playing field. In this case the spatial and social dimensions of the magic circle have been expanded from the traditional game, allowing for players the join the game without having to be within the physical playing field.

Human Pacman (Cheok et al. (2004)) and Pokèmon GO are examples of computer games that have been augmented and blended into a real-world setting. Much like many of the traditional augmented outdoor games these focus on utilizing the physical world as a playground or a map where the game itself is played out. The use of location-aware technology brings out the possibility for players to move around in1 the virtual worlds by moving around in the physical world, mixing the two worlds into one big augmented playground.

2.3.2 IoT in Pervasive Games

With the increased interest and accessibility of IoT and the benefits it brings there is no doubt that making use of IoT in pervasive games would open new and interesting ways of designing and constructing pervasive games. Since the increased accessibility of IoT has happened fairly recently and is still progressing the use of IoT in pervasive game is still in its early research days without having any larger scaled products being produced as of yet. The research focuses on how IoT best can be utilized to create innovative and enhancing experiences for the players. So far the approach of utilizing IoT in pervasive games has been done in one of two ways, using environmental sensor data from Things to affect and enhance different aspects of a game, or using Things as interactive game objects, which is the approach taken by the framework thats presented in this paper.

An example of utilizing environmental sensor data is from the game UKKO, a novel persuasive game to encourage walking to school and engagement with local data (Dickinson et al. (2015)). In this game air quality data is captured by a sensor and is used to encourage kinds walking to school to chose routes that are less exposed to poor air

quality. Treasure (Guo et al. (2012)) on the other hand, utilizes IoT technology in form of ultrasonic 3D tags and MOTE sensors to create real-world interactive game objects.

In MacDowell and Endler (2015) a set of base interaction models for designing pervasive games that use Things as interactive game objects is introduced. Table 2.1 and 2.2 displays the information given in MacDowell and Endler (2015) about the different interaction models, including examples of games they could be used in, whether the Thing is fixed or movable, and the general idea behind the interaction model. All of these interaction models are general enough in order for developers to create different game concepts by utilizing the same interaction models. These interaction models are used as a basis for some of the requirements set for the framework in Chapter 5.

Table 2.1: Base interaction models for IoT Pervasive Games (MacDowell and Endler (2015))

Interaction	Example Games	Tokens	Main Idea
Find the Thing	Mobile Geocaching Game	Fixed or movable	Users search for smart tokens. When a certain token is found, this action is registered on the user's smart device and/or a cloud service (if the game is multiplayer). A user can also pick up a token and bring it somewhere else or do a certain action with it. If the token is being moved, its coordinates could be updated to the cloud service by the user's smart device's own GPS. The user's interaction with the token directly influences the type of gameplay a game with this base interaction might have.
Guess where Things will be	Radar Tag Game	Movable	Tokens are always on the move. Tokens could be owned by certain users or be entirely independent. Players might need to guess a pattern of movement or discern the tokens next position/destination. Players could interact with the tokens when around them in a way to facilitate this movement prediction.
Bring lonely Things together	Smart Tag Game	Movable	Tokens are owned by certain users. In certain moments of the game, some users might be required to match their tokens with another token, which might be in possession of another player. This other player might be required to match his token with this same user, or the contrary, he could have to avoid this user for a certain period of time. These play mechanics will of course vary depending of the real proposition of the game.

Table 2.2: Base interaction models for IoT Pervasive Games (MacDowell and Endler (2015))

Interaction	Example Games	Tokens	Main Idea
Things that color other Things	Zombie Infection or Area Control Game	Fixed or Movable	Tokens are emitters of a certain type of frequency/data. When a player with a smart device and/or a movable token enters the actuation diameter of said emitter, the player is then "colored" or "infected" and is now visible to the game as a player of a certain type/color. The player could now act as a movable emitter to other players or fixed tokens. This type of interaction could lead to interesting tag/area Internet of Things Based Multiplayer Pervasive Games 127 control games, dealing with other gameplay mechanics such as map/area strategy on an urban area and teamwork.
Change the Thing	-	Fixed or Movable	When the player interacts with the token, it writes certain data on it. From then on, the token may behave differently than before, or start broadcasting a different type of data.

2.3.3 Known issues

Much like any genre of video games there exists issues that affect the design, development and gameplay of pervasive games and some of them are more common than others. There exists both technical and non-technical issues that exists partially because of lack of research (Wiberg and Jegers (2006)), but also because the technology hasn't developed enough yet (Broll and Benford (2005)). The most prominent issue is the extensive use of GPS and other positioning technology in location-based pervasive games. The reliance on technology that comes with built-in deviations (Broll and Benford (2005)) produces issues that create frustrations for both players and developers. The most common issue with positioning technology is the fact that there exists no system that preforms well both indoors and outdoors, forcing developers to work around the issues that these technological restrictions set on pervasive game design.

As for the non-technical issues Wiberg and Jegers (2006) identifies two issues re-

lated to the context where pervasive games are played. Surprisingly the "anywhere, anytime" feature of pervasive games proved to create issues that might be hard to foresee for developers. Some players decided to play during work hours, sometimes disturbing the social context set by the workplace. This also separates pervasive games from traditional games, where its usually played during recreational time and not during work. Another issue of the "anywhere, anytime" feature is that being able to play anywhere, at anytime, often resulted in "sofa gaming", meaning that instead of being outdoors playing the game wherever they want, players instead stay at home on their couch. This in turn reduces both the social and the spacial aspects of pervasive games that were built with the intention of having players go outside and explore the game world and socialize with other players.

2.4 Exergaming

A sub-genre within serious and pervasive games that has received a lot of attention from the research community is exergames or Exercise-games, also referenced as interactive video game (DiRico et al. (2009); Epstein et al. (2007); Warburton et al. (2007)), activity promoting game (Lanningham-Foster et al. (2006); Graves et al. (2007); Baranowski et al. (2008)), motion-based games (Alankus et al. (2011)), and active video game (Paw et al. (2008); Brown et al. (2008); de Vries et al. (2008)). Both health related and non-health related researchers have over the past decade devoted a lot of attention[2] to the topic which seeks to combine exercise and playing video games in order to help solve multiple health related issues that are arising in the modern western society. Two of the most prominent issues that researchers are addressing is the concerns related to the ongoing demographical change with the aging population requiring increasing amounts of health care, as well as the increase in obesity in the youngest demographic of the population. These concerns will be further addressed under the general use of exergames later in this section.

The terms exergames and exergaming has been described in various ways and with various definitions due to its interest spanning across multiple research collectives. Oh and Yang preforms a review on existing literature at the time and proposes a definition of exergames focusing on the combination of exertion and video games including strength training, balance, and flexibility activities (Oh and Yang (2010)). As most of the pre-existing definitions used the word ”exercise” in different settings, often without defining what they consider exercise to be, comparing the different studies proves difficult, especially from a health perspective. While some researchers follow standardized measures for monitoring physical activity levels, others simply accept any activity that is more than sedentary activity as a form of exercise. This in turn brings Oh and Yang to the following definitions of exergames and exergaming, which will be the defi-

[2] > 6000 results on the phrase ”exergames” on Google Scholar at the time of writing

nitions this paper will adhere to:

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

Exergaming *An experiential activity where playing exergames, video-games, or computer-based games is used to promote physical activity that is more than sedentary activities and also includes strength, balance, and flexibility activities.*

Exergaming as a health promoting tool

As previously mentioned, Exergame research have found several positive ways that exergames can be used to address current and future health related issues such as the increased pressure on hospitals and other public health services due to the aging population (Brauner et al. (2013); Uzor and Baillie (2014)), and curbing physical inactivity and childhood obesity (Gao and Chen (2014)). In addition to this research also indicated the possibility that physical activity may have positive effects on children's mental abilities, school behavior, and academic achievement (Tomporowski et al. (2011); Lee et al. (2017)) provides promising prospects for the use of Exergame interventions in schools. This range of aspects that the use of exergames is able to address is the major reason for the amount of attention the topic has received from the research community over the last decade, with the interest, both in research and commercial settings, still rising.

As an example of the potential benefits of using exergames for rehabilitation, Uzor and Baillie (2014) performs a long-term study on the use of exergames as a tool in falls rehabilitation. They find that participants who used exergames followed their recommended exercise better than those who used standard care, and that the potential use of exergames as a tool for rehabilitation is great. Brauner et al. (2013) takes a closer look at the usefulness of exergames as a tool to potentially increase physical activity and decrease the level of physical strain for elderly users. The motivation behind the study is the predicted decreasing ratio of jobholders supporting one elderly in the future and the need for preventive actions in order to keep the health and welfare systems from overflowing. Brauner et al. (2013) argue that by increasing physical fitness and awareness of health issues will help elderly to live independently for longer and thus reduce the demands and costs of the health care system. Their research, along with others (Lee et al. (2017)), find that exergames show promising signs as a tool for promoting physical fitness and activity.

The rise in childhood obesity and kids general attractiveness towards video games has led to research focusing on the potential use of exergames to combat the increasing health related issues for children and adolescents that stem from increasing amounts

of sedentary behavior. Research suggests exergaming to be a promising addition to promote physical activity and health among children, but due to the light-to-moderate physical activity that exergaming generates it should not be considered as an replacement for traditional physical activity and sports but rather as a replacement for sedentary activities like video games, Internet browsing, and watching television (Gao and Chen (2014)). Lwin and Malik (2012) backs up these indications and finds that incorporating exergaming into physical education classes leads to positive beliefs and behaviors towards physical activity, subjective norm, intention, and strenuous exercise behavior, particularly among younger children.

Finally, research show that exergaming may provide psychosocial and cognitive impacts, including increased self-esteem, social interaction, motivation, attention, and visual-spatial skills (Staiano and Calvert (2011)). Utilizing exergames in classroom contexts with third- and fourth-graders have proved to reduce absenteeism, improve social skills, leadership skills, self-esteem, and academic performance of highly "at-risk" pupils (Chamberlin and Gallagher (2008)). An important factor to the positive psychological effects is the social interaction between players in multi-player modes that enables players to collaborate or compete with other players (Lee et al. (2017)). Lee et al. (2017) also finds that in comparison with other physical activities or sedentary video games exergames showed significantly greater perceived enjoyment.

Exergames development

The research on using exergames as a tool for promoting physical activity and health awareness has not been strictly positive as the previous section might suggest. Several studies found that the interest in exergames seemed to decline rapidly over time, only providing an initial positive effect () and that several considerations have to be made when designing, developing and implementing exergames.

As mentioned in the previous subsection, Lee et al. (2017) found that exergames supporting multi-player gameplay, either in the form of collaboration or competition, increased the perceived enjoyment of exergames. By supporting multi-player gameplay, either in a local or remote setting, it enhances the social interaction component of exergames that has been found to have a significant impact on positive psychological effects. For exergame developers a key consideration when incorporating multi-player modes is the spatial requirements of the game and the context that it is meant to be used. When designing a remote multi-player exergame developers also have to address latency as high network latency often results in a poor game experience, particularly when cellular networks are being used (Park et al. (2012)).

Sinclair et al. (2007) introduces the dual flow model, an extension of the standard flow model introduced by Csikszentmihalyi (1975), designed to model the attractiveness and effectiveness of an exergame. With the dual model Sinclair et al. (2007) incorporates the dimension of effectiveness which addressed the balance between intensity

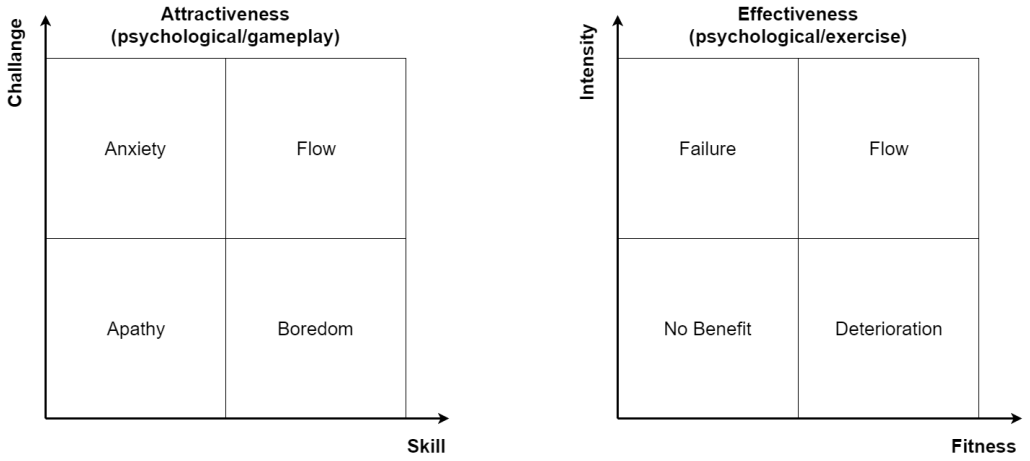

Figure 2.1: Dual Flow Model (Sinclair et al. (2007))

and fitness, and how having an imbalance between the two or lack of both leads to a less enjoyable game experience. The effectiveness dimension is represented with the same four quadrant layout as the standard model, as shown in figure 2.1, where each of the quadrants represents the different states that users can end up in, depending on the balance between intensity and fitness. Both of these dimensions are important to consider when developing exergames, as well as the need for customizations in order to meet the different users fitness and skill levels, allowing for different user groups to have an enjoyable experience when exergaming.

DiRico et al. (2009) argues that in order to assess the real effectiveness of exergames in training rehabilitation programs exergames needs to be implemented according to scientific paradigms on motor control and motor learning. So far most researchers have utilized pre-existing commercial exergames (i.e. Dance Dance Revolution, Wii Sports, etc) to research the effects of exergaming as an tool for rehabilitation and training. By considering these paradigms when designing exergames and the natural user interfaces (NUIs) used for exergames the scientific results of the research could be easier comparable with other research and provide a higher scientific accuracy

Chapter 3

Research Methods

This chapter looks at the research methods chosen for this research project and the overall process of the project. Building upon a preliminary study performed on the topic of prototyping pervasive games with IoT, this research project seeks to further that research and follows much of the same approach as that of the preliminary study. Figure 3.1 is an adaptation from (Oates, 2005, Ch. 3) and shows the chosen model of the research process.

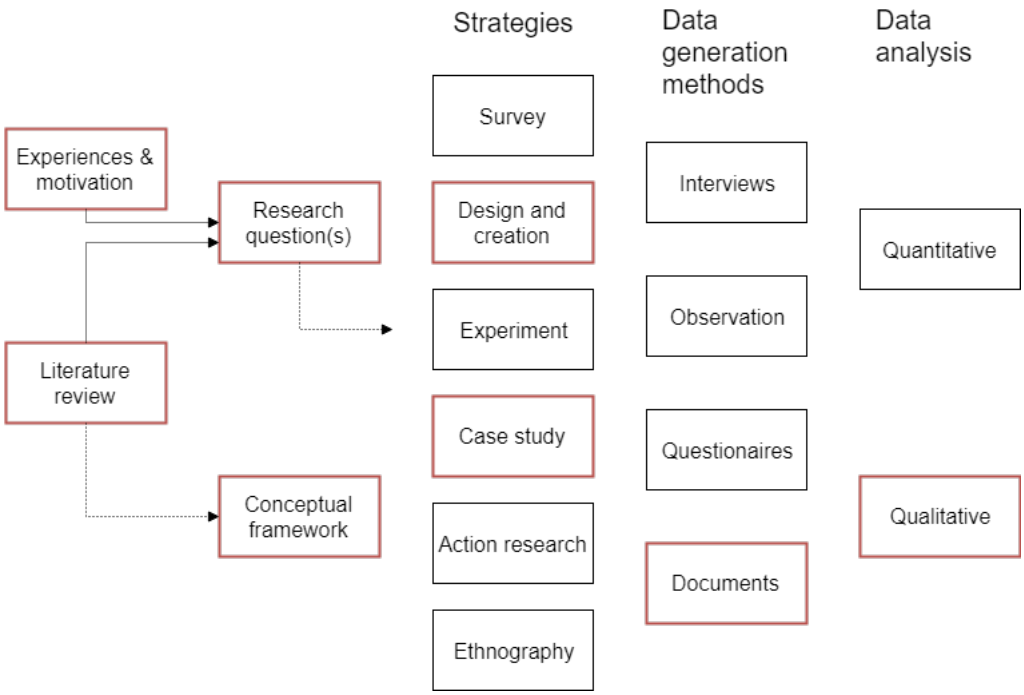

Figure 3.1: Adapted figure from (Oates, 2005, Ch. 3) displaying the possible and chosen elements of the research process.

A conceptual framework was initially developed based on the findings in the literature review and the experiences from the preliminary study. In order to explore and evaluate the framework a set of research questions, presented in section 1.3, were created and these will provide guidance and set the limitations of the research project.

Design and creation has been chosen as the strategy for the research project as the main focus of this research is to develop and evaluate software and the design and creation research strategy provides a suitable process for this kind of research. The design and creation strategy is backed up by a case study that establishes the different cases used in the evaluation. Finally the data collection and analysis will be based on a document-based research focusing on exploring the state of the art in a qualitative manner in addition to gathering data from product documents related to relevant technology used in the research. Each of the following sections within this chapter will go into greater detail of the research methods chosen.

3.1 Strategy

In this thesis the overall end goal is to evaluate a proof of concept for the proposed framework that will be used to prototype and develop IoT-based pervasive games. In addition the suitability of the selected technologies in the proposed technology stack will be evaluated. At the core of the thesis are the three cases that are used to perform the evaluation. For each of these cases an instantiation will be developed to display how the proposed framework could be implemented and to aid the evaluation of the selected technologies. Based on the preliminary study and the literature review done in this study there were, to our knowledge, no existing development methodologies for IoT-based pervasive games, so in order to fulfill this approach a design and creation research strategy has been selected. This strategy provides a suitable procedure that helps present the academic qualities as well as the the technical qualities of the research by following a ”leaning via making” strategy.

The design and creation process is an iterative process made up of the five steps (Vaishnavi and Kuechler (2004)): awareness, suggestion, development, evaluation, and conclusion, as shown in figure 3.2. In the two first steps a proposed solution with a tentative design is produced by trying to recognize the problem and suggesting a possible solution to the problem. In the development step the artefact is developed based on the proposed solution and in some cases the researchers may become aware of restraints which causes deviations from the proposed solution. These kinds of discoveries in themselves provide a knowledge contribution to the research and bring about new awareness of the problem which restarts the cycle as alterations are made to the initially proposed solution. In the case where the artefact has been created in accordance with the suggested solution it is evaluated in order to discover possible deviations and if the artefact solves the problem being researched. In the final step the findings from the cycle is consolidated and a final assessment of the knowledge gained is presented. This knowledge may in turn be used to propose further research in order to address issues that were identified throughout the solution or serve as a contribution

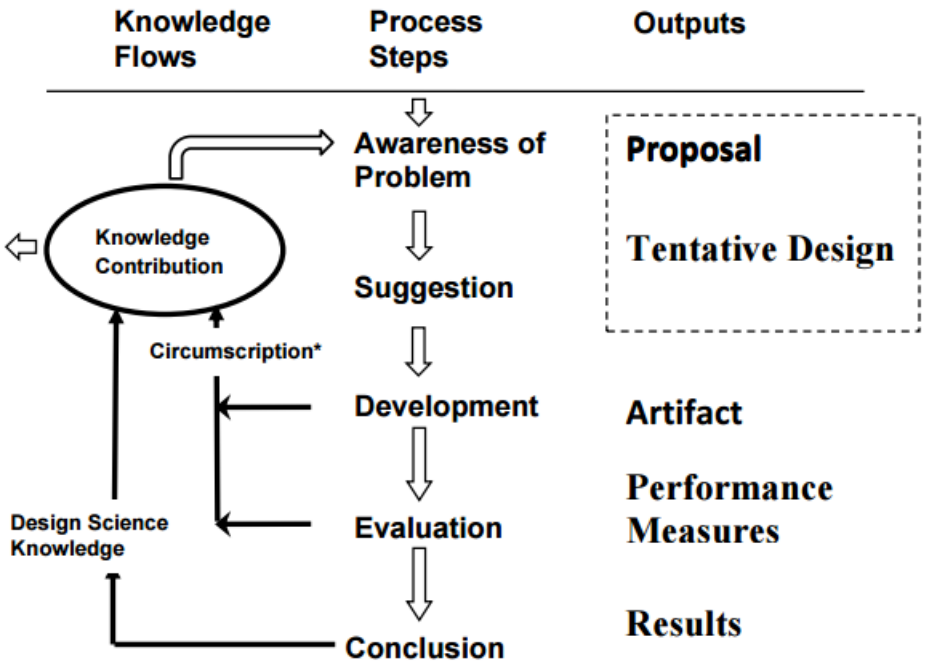

Figure 3.2: Design Science Process Model (DSR Cycle), Vaishnavi and Kuechler (2004)

to solving the overall research problem.

3.2 Data collection

With the research being of an exploratory nature it is natural to look into existing research and previously documented experiences within the topic. A document-based research done within the topics of IoT, Cloud technology, pervasive games, and exergames will provide data which can be used to set the initial requirements for the proposed framework and discover proposed directions for the research. The document-based research will mainly follow the approach of a literature review in addition to gathering data from product documents for IoT-devices and other technology used in the research.

Since most of the documents will be made up of academic literature the search terms and the digital libraries containing the literature will be listed in Appendix B.

3.3 Data Analysis

The gathered documents will be treated as vessels that hold content which will undergo a theme analysis in order to provide qualitative data on the topics. Based on the research done in the preliminary study it is clear that there exists little research on the

* Circumscription is discovery of constraint knowledge about theories gained through detection and analysis of contradictions when things do not work according to theory (McCarthy (1980))

topic of IoT-based pervasive exergames and the data needed to perform a quantitative analysis does not exist at the time of writing this paper, and therefore the research will mainly focus on qualitative data.

Chapter 4

Case description

As presented in the previous chapter, this research follows a design and creation research strategy. One of the steps in the strategy is the evaluation of the developed artefact against the proposed evaluation criteria, and in order to be able to perform the evaluation three cases have been designed. These cases will be used in a proof of concept where each of the three cases seek to provide the ability to evaluate the artefact against the different requirements introduced in chapter 7. This chapter presents a high level description of the cases used in the evaluation and provides useful insights into the differences between each of the cases and why they were designed in the first place.

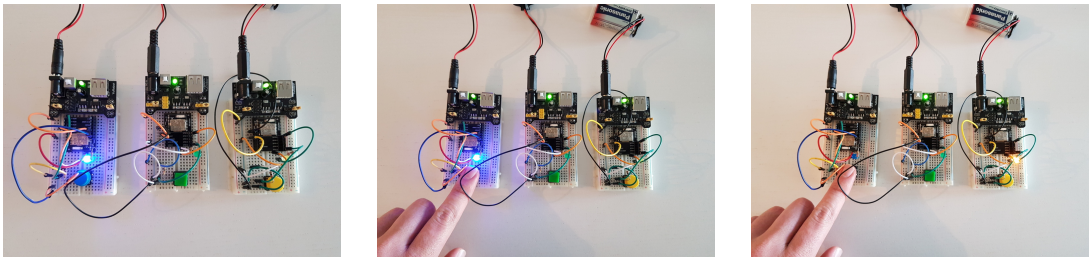

Figure 4.1: Illustration of how the Red Dot moves on to another device after interaction.

Case 1: Local gameplay with the devices placed within the area of a small apartment.

Case 2: Distributed gameplay with the devices placed anywhere in the world where the exists a Wi-Fi Internet connection.

Case 3: Distributed gameplay with the devices placed anywhere in the world where the exists a Wi-Fi Internet connection, and the game is managed from a central devices, i.e. a smartphone.

Each of the cases is an implementation of the game "Follow the Red Dot", a simplistic interactive game that provides loads of opportunities for customization while

still being easy to play and implement. The game is played by having two or more interactive objects that have the opportunity to both display that it is in possession of the Red Dot and a way to recognize human interaction. The main game mechanic is the interaction between the player and the object that is in possession of the Red Dot. Figure 4.1 displays how once the player interacts with the object it looses possession of the Red Dot and the dot is passed on to another object.

All of the three cases are developed so that the devices that the player interacts with behave in the same way and should provide the same experience from a gameplay perspective. The distinctive features of the cases lies in the implementation and should not affect the players ability to play the game. Locality and game management are the attributes that are altered between the three cases.

Figure 4.2: Illustration of the use of a playground within an apartment.

In Case 1 "Follow the Red Dot" is implemented on simple embedded devices in a local playground equal to the size an apartment. Figure 4.2 illustrates an example playground where different devices are spread out within an apartment, while another example could be a tabletop playground similar to the setup shown in Figure 4.1.

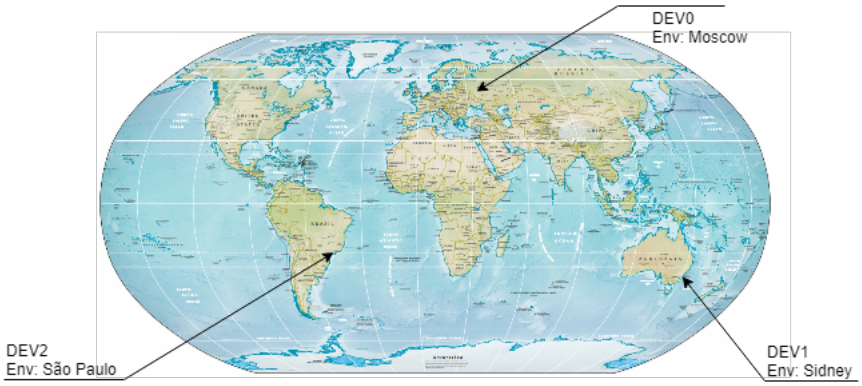

Figure 4.3: With the pervasive version of Follow the Red Dot game objects can be placed anywhere in the world.

Case 2 takes our local version of "Follow the Red Dot" and connects it onto the Internet, allowing for a global, distributed playground. Figure 4.3 illustrates how game moves from a local setting with limitations on the playground area to a global setting where game objects can be placed virtually anywhere in the world, or even in space, as long as there is an Internet connection available. This could prompt an interesting event where developers around the world could connect their own IoT devices to the game and create a global community built around this instance of the game. However, the purpose of this case is not to create the biggest playground possible, but to illustrate how IoT technology and pervasive computing is able to provide limitless possibilities for pervasive game developers when it comes to creating interactive pervasive games and giving players the "anywhere, anytime" aspect that is the unique feature of pervasive games.

The third and final case remain in the realm of the Internet and from a gameplay perspective there is no change from Case 2, but from a developers perspective there are some alterations. Case 3 seeks to introduce a version of Digital Twins, introduced in chapter 2, which will allow developers to display and interact with a virtual representations of the physical game objects. The virtual part of the Digital Twins should mirror the behavior of the physical device, and vice versa, meaning that developers could manipulate the state of the virtual device and have the state changes be mirrored on the digital device. An illustration of how the Digital Twin could be presented within the game engine is displayed in figure 4.4.

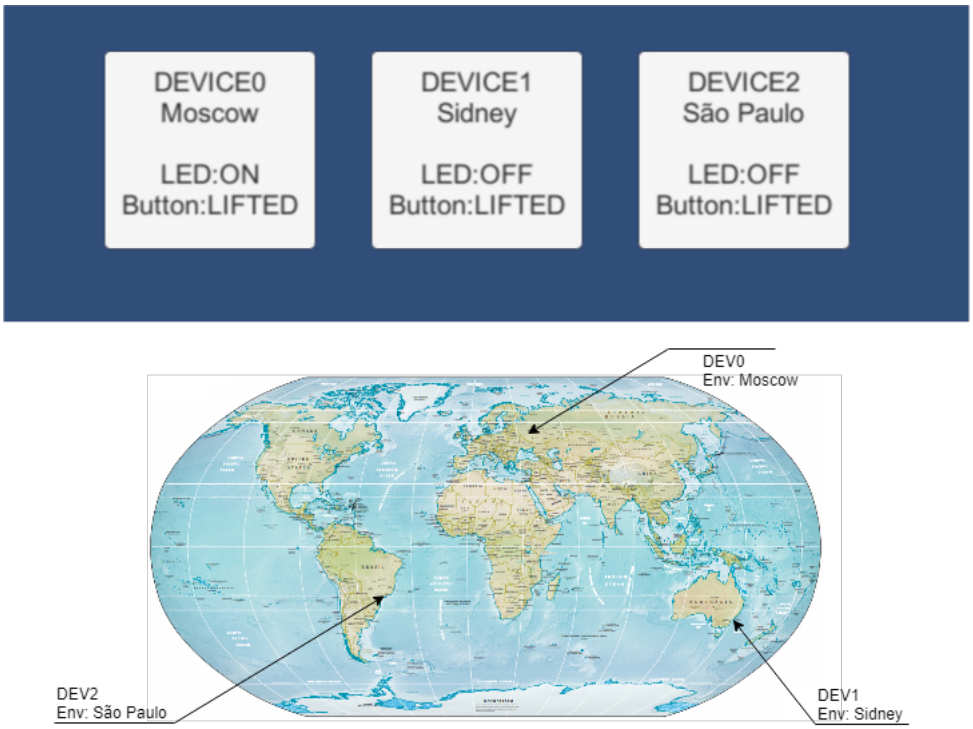

Figure 4.4: Illustration of the Digital Twin setup with the virtual twin in Unity.

In addition to creating the support for Digital Twins the game logic is also moved out of the IoT devices giving the IoT devices a clearer role as an game object that developers can use to build their pervasive games with. Moving the game logic to a game engine like Unity also means that developers are able to build game applications that could be run on computers, smart phones, consoles, and more, with custom visual representations and interpretations of the Digital Twins.

Chapter 5

Evaluation Criteria

This chapter presents the evaluation criteria that are used to evaluate Internet of Things technology, the proposed technology stack, and the framework developed for prototyping IoT-based pervasive games. The criteria are based on findings in existing literature as well as experiences from the preliminary study. The research questions introduced in Chapter 1 are used to group the different evaluation criteria as they relate to the different research questions. Within each group the criteria are ranged from most to least important, with the most important one being listed first. Some of the criteria overlap between the different research questions as these are requirements that are key evaluation features for both the technology used and the suggested framework.

The overall goal of this research is to reduce the cost of prototyping pervasive exergames by developing an intuitive framework built on a technology stack that offers time saving functionality for developers and supports IoT technology that is easy to grasp for both novice and experienced developers. This overall goal is represented in table 5.1 and serves as a guiding principle for the rest of the evaluation criteria.

Table 5.1: Overall criteria for the framework and IoT technology used for prototyping pervasive exergames.

Criteria	**Description**
C1.0 - Reduce implementation cost	The framework and the technology used should reduce the overall cost of prototyping pervasive exergames.

Table 5.2 presents the requirements tied to RQ2. The requirements are based on basic principles of ubiquitous computing, the IoT vision, and the interaction models presented in chapter 2. In table 5.3 the evaluation criteria for RQ3 and RQ4 are presented. The criteria overlap between these two research questions as the same requirements are relevant when looking at different technology stacks and for evaluating the selected stack used in this research. The following section ties the requirements to the three different implementation cases and presents a more detailed description of each relevant requirement and why they are relevant for the case.

Table 5.2: Evaluation criteria related to RQ2.

Criteria	Description
C2.1 - IoT Flexibility	IoT technology should support for different I/O modules to allow developers to customize the game objects to their game design.
C2.2 - Addressability	IoT technology should provide the ability to uniquely identify and address the device.
C2.3 - Device-to-Device communication	IoT technology should support the ability to transfer and receive data from different IoT devices.
C2.4 - Power management	IoT technology should support the ability for developers to handle power consumption.
C2.5 - Distributed vs Local use	IoT technology should support both local and distributed applications.
C2.6 - Scalability	The IoT technology should support the ability to be used in pervasive games with different amounts of devices.

Table 5.3: Evaluation criteria related to RQ3 and RQ4.

Criteria	Description
C3.1 - Addressability	The framework should support communication protocols that allows for unique addressability for each device connected to the framework.
C3.2 - Interoperability	The framework and the technology used should support communication between various IoT devices without knowing the specifications of each device.
C3.3 - Connecting new devices	The framework should provide intuitive handling of connecting new devices both during development and runtime.
C3.4 - Distributed vs Local use	The technology used should support going from a local to a distributed implementation
C3.5 - IoT Flexibility	The framework should allow developers to create custom interaction and visualization of different virtual I/O components that mirror traditional physical I/O modules.
C3.6 - Game logic centralized	The framework should support running the game logic outside of the IoT devices.
C3.7 - Scalability	The framework should support scalability, allowing for increased amounts of devices to be utilized in the prototyped games.

Chapter 6

IoT Technology

This chapter will present information about different technologies that could make up the proposed technology stack for prototyping IoT-based pervasive games. The stack that is proposed for building a prototyping framework is based on four different layers and this chapter will present some of the suitable technologies that exists within each layer. The four layers chosen are based of the four first layers of the five layer model presented in Al-Fuqaha et al. (2015). These four layers are shown in figure 6.1 and include the Objects, Object Abstraction, Service Management, and Application layer. Each of these layers and associated technologies will be presented in the following sections and the information about the technologies will be used to provide a comparison that will derive a proposed stack of technologies that will be used to develop the prototyping framework for IoT-based pervasive games. It is important to note that there exists a lot more technologies within each layer than what is presented in this paper and there could be technologies that are better suited than the ones presented here, but due to the limitations of the paper the ones shown here are the most common technologies used in IoT development.

Figure 6.1: Four layered stack.

6.1 Objects layer

Arduino

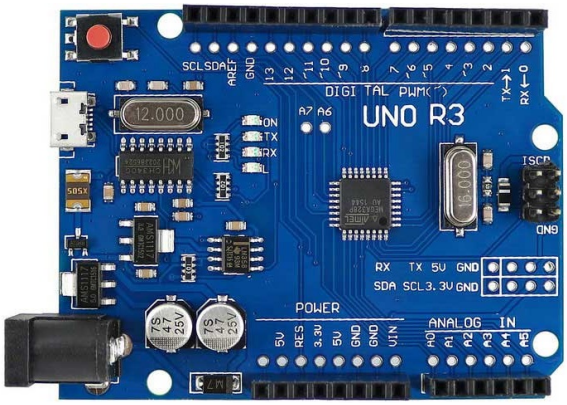

Figure 6.2: Arduino Uno R3

Arduino is an open-source electronics platform that offers easy-to-use Arduino boards and the Arduino programming language. Originally developed as a prototyping tool for students without experiences in electronics and programming, the platform is today one of the most used prototyping platforms for IoT as it offers a simple and accessible user experience for new developers in addition to the entire platform being open source, meaning experienced developers gain a high degree of customizability when developing devices for particular needs. The Arduino boards comes in wide array of different specifications with different pricing (from $10 to $40) and with the option to attach different interfacing shields that are equally user friendly to the platform itself. The common feature of the boards is the standard form factor that allows the Arduino to expose the functionality of the microcontroller in a intuitive and accessible package.

Raspberry Pi

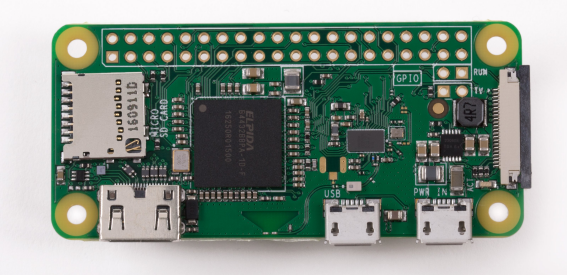

Figure 6.3: Raspberry Pi Zero W

The Raspberry Pi is a series of small sized computers that was originally developed as an educational tool for improving programming and hardware knowledge of pre-

university children. The small computers are able to interface with standard I/O products similar to a traditional computer in the form of screens, keyboards, and mice, in addition to running a fully fledged Linux distribution that supports programming in languages like Python and JavaScript. The Raspberry Pi comes at a cheap price (from $10 to $35) which makes it suitable for developers seeking a more complex device than the basic microcontrollers commonly used for prototyping. Most of the newer Raspberry Pi boards comes with built in Wi-Fi (802.11b/g/n) and Bluetooth support making them suitable to be used as gateways for the IoT for devices that doesn't have the ability to connect directly to the IoT.

ESP8266

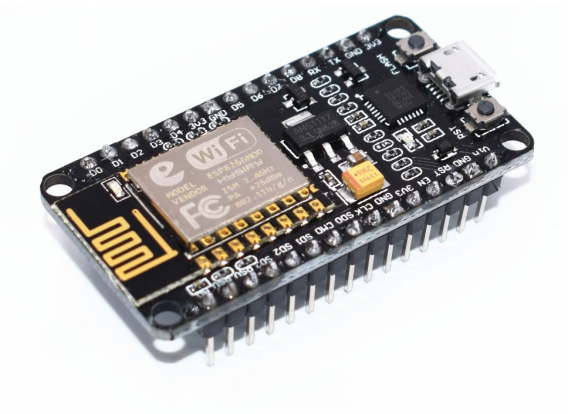

Figure 6.4: ESP8266

ESP8266 is a Wi-Fi microchip with full TCP/IP stack that has become very popular due to its low price (less than $7) and its ability to be programmed from the Arduino IDE. The ESP comes in a series of configurations from different manufacturers with different hardware configurations in regards of active pins, pitch, antenna, and physical dimensions. In addition the ESP8266 supports a wide range of different SDKs, i.e. NodeMCU, Arduino, Espruino, and MicroPython, meaning that developers are free to chose the SDK of their liking to make use of the Wi-Fi capabilities of the ESP8266. By offering built-in Wi-Fi capabilities at a low price the ESP8266 chip has gained a lot of market attention as the most commonly used boards for IoT either required external Wi-Fi modules or a Wi-Fi gateway in order to be connected to the IoT, which would bring the price of the device even higher.

6.2 Object Abstraction layer

ANT

ANT is a proprietary ultra-low power wireless protocol commonly used in monitoring applications where low power consumption over longer periods are vital. The protocol operates on the 2.4GHz spectrum and supports multiple virtual channels existing on the same frequency. The network structure of the ANT protocol supports peer-to-peer, star, broadcast, shared, cluster, and mesh topologies. In order to distinguish each of the ANT networks the networks needs to possess a unique identifier that is used by the nodes to connect to the same networks. In addition to the unique network identifier, nodes that wants to communicate with each other needs to have the same frequency, message period, device type and transmission type (Mehmood and Culmone, 2015). For each channel there exists three different ways to use the channel: independent, where the channel only has one master and one slave; shared, where a single master node receives data from many slave nodes; continuous scan, where the master node receives full-time, allowing it to receive from multiple transmitting masters at the same time (Dynastream Innovations Inc).

ANT provides a management of physical, data link, network, and transport layers of the Open Systems Interconnection (OSI) stack. ANT+ is an extension to ANT that adds functionality for interoperability and defines the data structure, channel parameters, and the network keys that enable ANT+ products to communicate with each other (Khssibi et al., 2013). These configurations are usually made by vendors and define the session, presentation, and application layers.

BLE

Bluetooth Low Energy (BLE), or Bluetooth Smart, is part of the Bluetooth v4.0 specification and developed as a low-powered wireless PAN protocol for control and monitoring applications. Similar to other Bluetooth technology BLE is based on teh IEEE 802.15.1 standard for short-range wireless communication. The BLE protocol stack is made up of two main parts, the controller and the host. The controller is composed of the physical and the link layer, while the host is made up of upper layer functionality like the Attribute Protocol (ATT), the Generic Attribute Profile (GATT), and the Security Manager Protocol (SMP). BLE operates on the 2.4 GHz band, similar to ANT, and defines 40 Radio Frequency (RF) channels with two different types of channels: advertising, which is used for device discovery, connection establishment and broadcast transmission; data, which is used for bidirectional communication between connected devices (Gomez et al., 2012).

Within the link layer the protocol defines the master and the slave role, where the master is able to manage multiple simultaneous connections with different slaves,

forming what is called a piconet. The piconet is structured in a star topology with the master being the center node. In order to provide energy saving functionality the slaves support a sleep mode where they wake up periodically to listen for possible packet receptions from the master. By using a Time Division Multiple Access (TDMA) schema, the master determines when the slaves should listen in addition to providing information about the frequency hopping algorithm.

Wi-Fi

Wi-Fi technology is based on the IEEE 802.11 standards for wireless local area networks (WLAN). Wi-Fi is commonly used to connect users to the Internet, but it also has it use for IoT as it can be used for monitoring and managing lights, power outlets, door locks, etc.. Building on the 802.11 standard Wi-Fi support operation on both the 2.4GHz and 5GHz bands, with the 802.11b/g/n operating on 2.4GHz bands and the 802.11a/n/ac operating on the 5GHz band (Reiter, 2014). In comparison with the other protocols presented in this section, Wi-Fi offers an enormous bandwidth with speeds over 100 Mbps and a range of up to 100 meters. This however implies that the protocol requires a higher level of power usage, but advanced sleep techniques are being developed to add better support for Wi-Fi integration in battery driver microcontrollers. Wi-Fi is also the most commonly used wireless communication protocol and has existing infrastructure deployed in most homes, offices, and public areas within cities, offering developers the ability to deploy their Wi-Fi supported IoT applications almost anywhere.

ZigBee

ZigBee is an open source standard for low-powered, low-rate wireless communication commonly used in residential control and monitoring applications. The protocol is based on the IEEE 802.15.4 link layer standard and operates in the 2.4GHz, 868MHz, and 915MHz bands. ZigBee is as mentioned a low-rate protocol, but supports speeds up to 250KBps. Some of the functionalities that are key for ZigBee to support low-powered devices is that it has the capability to maintain very long sleep intervals and low operation duty cycles to be powered by coin cell batteries for years (Reiter, 2014).

The protocol supports two different device types that can participate in the communication:; the full-function device (FFD), that has three different modes of operation; and the reduced-function device (RFD), that is intended for simple applications, like light switches and passive sensors (Lee et al., 2007). The RFD is only able to communicate with one FFD, while the FFDs are able to communicate with other FFDs and multiple RFDs. The FFDs act as coordinators and are used to set up wireless networks that follow a star topology. Other ZigBee devices are then able to connect to the network by using a specified network identifier that is unique within the deployed network

space.

There exists three different stacks within ZigBee: ZigBee, ZigBee PRO, and ZigBee IP. ZigBee IP supports IP-connectivity for low-powered devices and offers a IPv6-based wireless connection that is built as extension to the IEEE 802.15.4 standard by adding network and security layers. ZigBee PRO is quite similar to regular ZigBee, but is optimized for larger networks of up to thousands of devices compared the the hundreds of devices supported by regular ZigBee.

Z-Wave

Z-wave is a proprietary, interoperable wireless communication protocol used for control and monitoring applications, similar to ZigBee. The protocol provides operates on sub-1GHz bands designed for low-bandwidth data communications, with the protocol running on 868.42MHz in Europe and 908.42MHz in the United States (Fouladi and Ghanoun, 2013). The Z-Wave protocol uses a four layered architecture including the physical, transport, routing, and application layers. The physical layer is made up of an RF transceiver used to transfer data at a rate of either 9.6Kb/s, 40Kb/s, and 100Kb/s. The transport layer is responsible for handling package retransmissions, acknowledgements, waking up low power network nodes, and packet origin authentication. In the network layer the Z-wave protocol uses the controller and up to 232 nodes to form a mesh network with routing capabilities between nodes even between nodes that doesn't have a direct radio connection. The final layer is the application layer where the Z-wave commands are used to forward and decode payloads in order to allow them to be used by the host application.

6.3 Service Management layer

DDS

Data Distribution Service (DDS) is a Real-Time Publish/Subscribe (RTPS) protocol developed by Object Management Group (OMG) for Machine-to-Machine (M2M) communications (Al-Fuqaha et al., 2015). DDS doesn't use the common broker architecture, found in other IoT publish/subscribe protocols like AMQP and MQTT, in order to realize its publish/subscribe functionality, in stead it realizes a high level of reliability and Quality of Service (QoS) by using multicasting. By removing the broker the protocols strengthens its ability to address the real-time constraints of IoT and M2M communications.

The protocol is made up two different layers, the Data-centric Publish-Subscribe (DCPS) layer and the Data-Local Reconstruction Layer (DLRL), with DCPS being in charge of transferring data to the correct recipients and DLRL being an optional layer

that serves as a object-model interface that allows distributed data to be shared bu distributed objects as if the data were local (Esposito et al., 2008). In the DCPS layer there are five entities that make up the data flow of the protocol: Publisher, DataWriter, Subscriber, DataReader, and Topic. The Publisher disseminates the data received from a sending application and the DataWriter communicates to the Publisher the existence and value of data of a set type. The Subscriber is the receiver of published data and forwards it to the receiving application and the DataReader is used by the Subscriber to access the received data. Finally the Topic that connects a DataWriter with a DataReader is the combination of a data type and a name.

DDS supports 23 different QoS policies that allows developers to tailor the protocol to their application requirements related to reliability, security, robustness, etc (Esposito et al., 2008).

CoAP

The Constrained Application Protocol (CoAP) is an application layer protocol developed by the IETF Constrained RESTful Environments (CoRE) working group for IoT applications (Al-Fuqaha et al., 2015). CoAP is based of REpresentional State Transfer (REST) that runs on top of HTTP functionality. CoAP is designed to enable small devices with low power, computation, and communication capabilities to utilize the powerful RESTful interactions. Instead of using TCP CoAP uses UDP in combination with a message layer that handles retransmission in order to reduce the complexity of traditional HTTP. The four message types defined in the message layer are: confirmable, non-confirmable, reset, and acknowledgement. CoAP also supports the HTTP get, post, put, and delete methods and is able to work together with HTTP by utilizing proxies that are able to act as intermediaries that speak CoAP on one side and HTTP on the other, meaning that CoAP applications can make use of existing architecture without making major modifications to the intermediaries.

CoAP is able to run on top of Datagram Transport Layer Security (DTLS) or in combination with TLS in order to provide certain levels of security (Bormann et al., 2012).

AMQP

The open standard protocol Advanced Message Queuing Protocol (AMQP) is developed for message-oriented IoT environments and is employed in the application layer. The protocol supports reliable communication via message delivery guarantee primitives including at-most-one, at-least-once, and exactly once delivery (Al-Fuqaha et al., 2015). AMQP is made up of both a network protocol and a protocol model, where the network protocol specifies the entities that interoperate with each other and the protocol model specifies the messages and commands used by the communicating entities (Luzuriaga et al., 2015).

Additional key features are that the messages data is opaque and immutable, and there is no size limitations to the message. These features allows AMQP support security, reliability, and performance.

MQTT

MQTT is an open Client Server publish/subscribe messaging protocol built to be used for Machine-to-Machine (M2M) communication and the Internet of Things. Its main feature is the lightweight and simple implementation that is designed for devices with tight constraints related to network bandwidth, latency, and connection reliability. The protocol provides Quality of Service (QoS) data delivery with three different levels in order to provide reliability for developers even when their software is deployed in scenarios where the underlying transport is unreliable. The different levels also allows the developers to choose a level that is suited for the application and the reliability of the network the application will be used on.

The MQTT protocol has two actor types, the broker and the client, where the broker is the center of the communication protocol. The broker is responsible for handling all messages transfered by receiving, filtering, and forwarding messages, as well as keeping track of all subscribed clients and persistent messages. The client is a subscriber, a publisher, or in some cases both, which is connected to a broker. The clients could be implemented on any device that is able to support the MQTT library, meaning that anything from small IoT devices to massive supercomputers could be clients that publish and subscribe to different topics on a broker.

The topics are used to provide subject-based filtering, which is the chosen filtering option used in MQTT. By specifying which topics a client would like to publish or subscribe to developers are able to filter different messages to the correct receivers and have easily implement state machines that act based on messages received on different topics. The structure of the topics are level based, meaning that a topic consists of one or more levels, where each level is separated by a forward slash. The levels are made up of strings, but MQTT also supports two different wildcards that are used when subscribing to topics. An example to illustrate this could be a set of sensors setup within a house with which gives the following topics:

```
House/Livingroom/Temperature

House/Livingroom/Humidity

House/Kitchen/Temperature

House/Kitchen/Humidity

House/Garage/Temperature
```

If the client wants to receive all messages related to temperature the client could subscribe to the following topic by using the single level wildcard instead of having to subscribe to three different topics:

```
House/+/Temperature
```

Similarly if the client wanted to gather all sensor data within the house the client could use the multi level wildcard in the following way:

```
House/#
```

XMPP

Extensible Messaging and Presence Protocol (XMPP) is an instant messaging standard used for multi-party chatting, voice, and video calling and telepresence (Al-Fuqaha et al., 2015). The protocol is developed as an open source project by the Jabber open source community. XMPP offers the ability for developers to have authentication, access control, privacy measurement, hop-by-hop and end-to-end encryption, as well as compatibility with other protocols. In comparison with CoAP, XMPP doesn't follow the REST architectural style, but rather the Availability for Concurrent Transactions (ACT) architectural style. This style is built on persistent XML streams that send XML stanzas, which are relatively small pieces of structured data. The architecture involves ubiquitous knowledge of network availability and a conceptually unlimited number of concurrent information transactions in the context of a given client-to-server or server-to-server session. The key features of the ACT architectural style are: the use of unique global addresses based on Domain Name System (DNS); the ability for XMPP entities to advertise their availability (or presence) through the presence stanza which facilitates real-time interactions between entities; persistent XML streams over TCP connections that allow for immediate routing or delivery of XML stanzas in client-to-server and server-to-server streams; the XML stanza which is a structured set of data containing routing information and a payload; a distributed ubiquitous network of clients and servers that allows for peer-to-peer communication through a series of client-to-server and server-to-server communications (Saint-Andre, 2011).

6.4 Application layer

Unity Engine

Unity Engine is a game engine developed by Unity Technologies that supports both 2D and 3D development for up to 27 different platforms, including mobile devices, consoles, and traditional computers. Games developed in Unity is commonly developed in C#, but the engine also supports JavaScript even though its deprecation process has

been started with the release of Unity 2017. Unity is being used by developers and game studios both in the indie community and in professional AAA companies due to its intuitive interface and cross-platform integration capabilities. The engine also has great support for bringing in third party assets from applications like Maya, 3ds Max, Photoshop, and Blender, as the engine itself lacks support for detailed modeling. Additionally the engine has an enormous asset library where developers share and their custom assets that can be used to boost the developing process and reduce the time to market.

Unreal Engine 4

Unreal Engine 4 was developed by Epic Games and is one of the most popular game engines that are publicly available. Unreal Engine 4 is the fourth installment of the Unreal Engine and supports development for 15 different platforms with the major focus being on the latest consoles, virtual reality, and mobile platforms. The engine offers an industry leading visual performance in addition to providing a user friendly and intuitive interface for developers. Games developed in the engine are built on C++ and are able to use Blueprint visual scripting, which offers the ability for developers to rapidly prototype and design gameplay mechanics without having to write code, making it more accessible for developers without previous programming experience.

CryEngine

CryEngine is developed by the development company Crytek and is an powerful engine designed to create games for the available high-end platforms, including PC platforms and newer consoles like the Xbox One and PS4. Compared to the two previously described engines CryEngine is designed more for experienced game developers with previous experience from game development in game engines. However, its power grants amazing graphical and performance capabilities that can hardly be matched by any other publicly available game engine. CryEngine supports several different scripting languages, including C++, C# and Lua, and offers a built in Sandbox programming experience that is well suited for rapid testing of new game mechanics.

6.5 Proposed technology stack

Based on the three cases presented in Chapter 4 and the different technologies presented in this chapter a proposed stack for doing a proof-of-concept evaluation of the framework developed for prototyping IoT-based pervasive games is shown in figure 6.5. The stack a set of proposed technologies with suitable features that make up the entire four layered stack, with some layers using multiple technologies to create a stack that fulfills the requirements presented in Chapter 5 in the best possible way. It is important

to note that with all of the available technologies within each layer it is possible to build several different stacks that could provide functionality that could be more suitable for specific implementations.

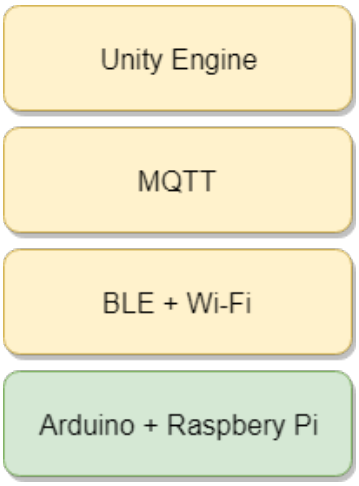

Figure 6.5: Proposed technology stack.

In the objects layer it is proposed to utilize both Arduino and Raspberry Pi technology. The reason for combining these two technologies is that the simplicity of developing prototyping code on the Arduino makes it suitable for deployment in local area networks (LAN) when paired with suitable technologies from the Object Abstraction and Service Management layers. The Raspberry Pi is selected to offer a gateway for the Arduino LAN in order to connect it to the IoT for Case 2 and 3. More specifically for this implementation a Raspberry Pi Zero W is used as the gateway and the RFduino is used to make the local area network. The RFduinos are tiny Arduino boards that comes with built in BLE and low-power configuration support. Another approach to realizing this layer could be to go with the ESP8266 board and remove the need to use a gateway for the IoT, but the RFduino and Raspberry Pi setup was chosen because of the lower-powered capabilities that the RFduino board offers making them highly suitable as game objects in pervasive games.

For the Object Abstraction layer Bluetooth Low Energy and Wi-Fi was chosen to make up the technologies used in the layer. The Wi-Fi is used to connect the Raspberry Pi to the Internet as it offers the ability to create a more movable gateway that isn't restricted to a wired ethernet connection. Choosing BLE over any of the other WLAN protocols comes down to availability and cost of products supporting the different protocols in addition to the range support of the different protocols. Ant is restricted to a range of about 30 meters while ZigBee, BLE, and Z-wave offer up to 100 meter ranges. Z-Wave boards or shields are relatively expensive (from $50) compared to boards supporting the other protocols. ZigBee is equally usable compared to BLE and in the long term with ZigBee increasing its market share it could be the way to go, however, for this research BLE will be used and evaluated.

MQTT has been chosen as the protocol to fulfill the Service Management layer. Its lightweight and simple implementation makes it highly suitable for a prototyping environment as it easy to use for new developers and offers a high level of device abstraction through its publish/subscribe architecture. The freedom to have messages bodies of any type relieve developers of the need to follow intricate message structures, and the low latency and QoS support makes for reliable message delivery which is key for real time game interactions.

The final technology chosen to fulfill the stack is Unity Engine. Both Unity and Unreal Engine 4 offers developers an intuitive user experience and has a low entry bar for new developers that wishes to get into game development. The reason for choosing Unity over Unreal is its excellent support for cross-platform deployment and mobile support. Having pervasive games supported by and managed on mobile devices allows developers to deploy their games easier in any setting and helps maintain the pervasiveness of these types of games.

Chapter 7

Design and creation

Within this chapter the design and creation process will be presented. Every stage of the process, from the determination of requirements for the framework to the final iterative cycle of the implementation, will be addressed in detail. The first section introduces the requirements set for the framework to be deemed successful. These requirements are based on information found in existing literature and experiences from the preliminary study. The second section presents the high level logic of the three cases used to evaluate different aspects of the framework and technology stack. Each of the cases explore different attributes of the framework in order to display the usability of the framework in different settings that should be supported in an IoT-based pervasive game framework. The final section presents the implementation process and how keeping with the iterative approach provided useful insights to the framework throughout the entire development phase. The final evaluation is outside the scope of this chapter and will be presented in chapter 8.

7.1 Initial concept designs

In order to evaluate the proposed conceptual framework against the requirements established in the previous section a proof by demonstration approach was chosen. For the proof by demonstration three cases has been developed in order to address the different requirements. Each of the cases is an implementation of the game ”Follow the red dot”, introduced in chapter 4. The reason for choosing this game is that the game mechanics are easy to implement and the interaction with and between the different devices requires features that are mirrored in the evaluation criteria. The game itself is also quite customizable allowing for custom game types with local and/or distributed play, single- or multi-player, and the ability to monitor different statistics like reaction time, which in turn offers multiple ways to evaluate the framework against the requirements.

Each of the three cases evolve in complexity in regards to the implementation and overall game logic. The first case is the most basic and is made up of an local imple-

Table 7.1: Properties of the three cases.

Case	IoT Device(s)	Playground	Possible # devices	Game logic
1	RFduino	Local, BLE range	<=9	One RFduino Host
2	RFduino + RPI ZW	Distributed	>2	All RFduino Hosts
3	RFduino + RPI ZW	Distributed	>2	Unity Engine

mentation running only on the RFduinos supporting up to eight devices to be used as interactive game objects. The second case expands the first case to also use the Raspberry PI Zero W which offers the ability to create a distributed system communicating through the use of the MQTT messaging protocol. This offers the ability to support more than eight devices in addition to widening the play area from a local BLE range to anywhere with an Internet connection. The game logic on both of these cases is handled on the RFduino Hosts. In the third case the global playground from the second case is maintained, but the game logic is now moved out of the RFduino Hosts and onto a game made in the Unity Engine. The communication still happens through the MQTT messaging protocol and the Unity game acts as another MQTT Client along with the Raspberry PIs. The subsequent sub-sections presents in detail the conceptual idea behind each case along with the requirements that are suitable for evaluation within each of the cases.

7.1.1 Case 1: Local - RFduino

The first case seeks to present the ability to take a closer look at the functionality of the RFduino and provide the ability to evaluate the RFduino against some of the proposed requirements for an IoT device that would be used in IoT-based pervasive exergames. At the core of this implementation is the Gazell Link Layer (GZLL) which is used for communication between the RFduino devices. The GZLL is a wireless link between one Host and up to eight Devices where the communication is based on a star topology as shown in figure 7.1. Since the GZLL utilizes the same radio antenna as the one used for Bluetooth Low Energy communication the range is restricted to a maximum of 100 meters in an outdoor setting without any obstacles (Nordic Semiconductor), and considerably less in an indoor multi-room setting.

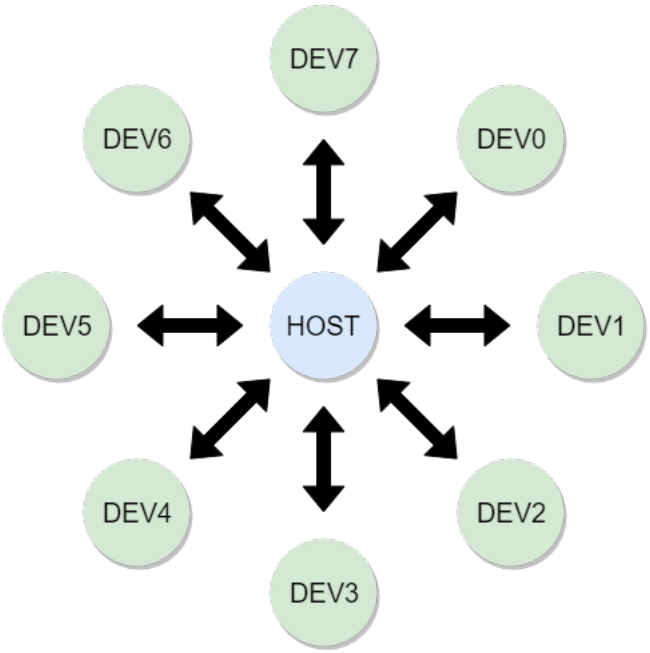

Figure 7.1: Star Topology of the GZLL

Due to the nature of the GZLL it is natural to implement the game logic on the Host as all communication has to pass through this device. In addition to being responsible for running the game logic, the Host can also be customized to act as an interactive device in the same way as the RFduinos running as Devices, which provides a potential total of nine interactive devices to be used in play. An option to running with a single GZLL network with nine RFduinos is to expand the network into multiple GZLL networks by connecting several Hosts together either through wired serial communication or the Inter-Integrated Circuit (I2C or I^2C) bus. Although this provides for the option to have more devices, it does not provide much expansion in the playground area as the wired connections limits the maximum possible range between a pair of Hosts.

With the game logic running on the Host there is only need for two types of messages when running with a single GZLL network as done in this case. One message going from the RFduino Host to the RFduino Device indicating that the device is now in possession of the red dot, which should trigger the RFduino Device to display some sort of indication that it is in possession of the red dot and is now listening to some sort of input from the player. The second message is the one going from the Device to the Host indicating that player has interacted with the device when in possession of the red dot and that the Host now should pass the red dot onto another device. The determination of the next device that should get the red dot could either implemented in a way that follows an exact pattern or a random selection based on random number generation (RNG). Since neither of the solutions involves any advanced complexity in regards to implementation the RNG version was used for all the cases in this project.

This case provides the ability to evaluate five of the requirements proposed in Chap-

ter 5, even though it doesn't utilize the framework in any way, this first case utilizes non-invasive computational devices to bring a digital game into the real physical world, which is the core of pervasive games (Benford et al. (2005)). The requirements being evaluated here are focused towards the IoT-device and the proposed properties it should possess in order to be suitable in IoT-based pervasive games. Some of the requirements will overlap between the different cases as the cases may provide different aspects to the evaluation of the requirement. The five requirements evaluated in this case are:

C1.0 - Reduce implementation cost
C2.1 - IoT Flexibility
C2.2 - Addressability
C2.4 - Power management
C2.6 - Scalability

Reduce implementation cost

The RFduino runs Arduino code with some additional features at its top level. The Arduino code is designed to be featured on prototyping boards like the RFduino and the technical skills required to get started is minimal. These types of micro-controllers are designed to be used for prototyping and to remove the need to build custom circuitry before testing your application, meaning that developers easily can start prototyping their applications without having to invest a lot of time into both building custom circuitry and achieving a high degree of technical skill.

IoT Flexibility

Being able to provide developers with a high degree of customizability in regards to how to handle input and output on the physical side is an important part of realizing the five base interaction models for designing pervasive games presented in section 2.3. With the "Follow the red dot" game there is a need for at least two different input and output modules, one for the player to interact with when the device is in control of the dot and one to indicate that the device is currently in control of the dot. For this research project an LED and a pushbutton has been used as the I/O modules, and altering either of the modules is possible without breaking the game. One example of an alteration would be to use a photocell for input and an audio source for output, neither causing any significant changes to the code or game logic.

Addressability

Having the ability to have uniquely addressable objects is a key property of the IoT vision (epo (2008) and pervasive computing. Even though the RFduinos used in this case aren't connected directly to the Internet, the importance of being able to address the correct device when communicating is still present. The GZLL messaging protocol defines nine different roles with associated identifiers, these being the Host and Device0-8. Each of these roles have a corresponding address within the GZLL network where each of the addresses are made up of an unique byte long prefix address in addition to a 2-4 byte long "base address" (Semiconductor (2017)). Since the RFduinos operate on the same base address by default it is possible to have multiple devices running on the same role with the same address within a GZLL network. This will lead to message loss due to the star topology message handling used in the GZLL protocol, as every message sent from the Host to an Device will only be received by the first Device that polls for a message on the Host. However the GZLL protocol allows for altering of the 2-4 byte "base address" both for the Host and the Devices, allowing for multiple GZLL networks in the same proximity without having message loss occurring.

Power management

Multiple interaction models proposed in section 2.3 are easily fulfilled by having a wireless IoT device. Being able to run of batteries provides the ability for the RFduino to be completely wireless and have a high level of mobility, but it also creates the need to assess the power consumption on the device. The RFduino offers the option to go into sleep mode, a low powered passive mode, when it isn't performing any tasks. This can be controlled by setting the amount of sleep time for the device upon the function call, which in this case the devices that aren't in control of the red dot could be set to sleep mode in order to reduce power consumption. The structure of the GZLL protocol supports this ability well by being designed to have the RFduino Host be unable to send direct messages to the RFduino Devices, but rather have the Devices pull queued messages from the Host. This means that by having the Devices pull messages whenever they leave sleep mode there won't be cases where the Host sends a package to a sleeping Device that could potentially result in a package loss.

On the Host however, finding occasions where it can enter sleep mode is a bit more difficult as it has to be awake to handle message pull requests and other messages from the Devices. Also by having the game logic based on the Host means that it at all times have to be ready to delegate the next host for the red dot. One way of solving this is to have the Host be the only RFduino connected to an infinite power source and have it stay as a stationary unit, reducing the number of mobile units by one, which in this case won't have any game breaking implications.

Scalability

Being able to support scalability is a key feature of pervasive computing and is also an important feature for the IoT technology being used in pervasive games. With the GZLL protocol having a limited support of only nine different devices, one Host and eight Devices, the scale of the pervasive games built with these devices is limited. The GZLL protocol is ,as mentioned in the previous subsection, designed to minimize power consumption by having the RFduino Devices pull messages from the RFduino Host in order to receive data. This in turn means that the Devices have to occasionally have to send pull requests in order to see if there are messages queued up on the Host, as the Host has no way to indicate to the Devices that there currently resides a message in the TX pipe that is ready to be sent. The way this is solved is usually through having the RFduino Device send a NULL message to the Host at a certain frequency, and when the Host receives a NULL message it checks the TX FIFO of the Device for a message and piggybacks the message onto the ACK payload that is sent back to the Device. According to the product documentation (Semiconductor (2017)) for the nRF5 chip used on the RFduino, the RFduino Host is designed to have two FIFOs, one RX and one TX, for each device, giving a total of 16 FIFOs which are able to store three packets each. This capacity should be able to support the limited amount of messages sent between the RFduinos in the implementation of ”Follow the Red Dot” when running with the maximum amount of devices supported by the GZLL protocol.

7.1.2 Case 2: Distributed - IoT

In the second case the goal is to take the local GZLL implementation of ”Follow the Red Dot” from Case 1 and expand it into a distributed, pervasive implementation that could support game play from anywhere with an Internet connection. In order to connect the GZLL networks to the Internet and allow for pervasive communication the network has to be connection to a device that supports Internet connectivity. Due to the diverse connection opportunities supported by the RFduino (Bluetooth LE, wired serial communication, I^2C, etc) this can be done in several ways and for this particular case a Raspberry PI Zero W will be connected to the RFduino Host of an GZLL network through the use of serial communication through the USB ports. The support of Wi-Fi communication and the MQTT messaging protocol, which was introduced in chapter 2, on the Zero W allows it to run as both a MQTT Subscriber Client and a MQTT Publisher Client communicating with other MQTT Clients connected to the same MQTT Broker. The MQTT Broker used for this research project is running on a remote server located at the Norwegian University of Science and Technology.

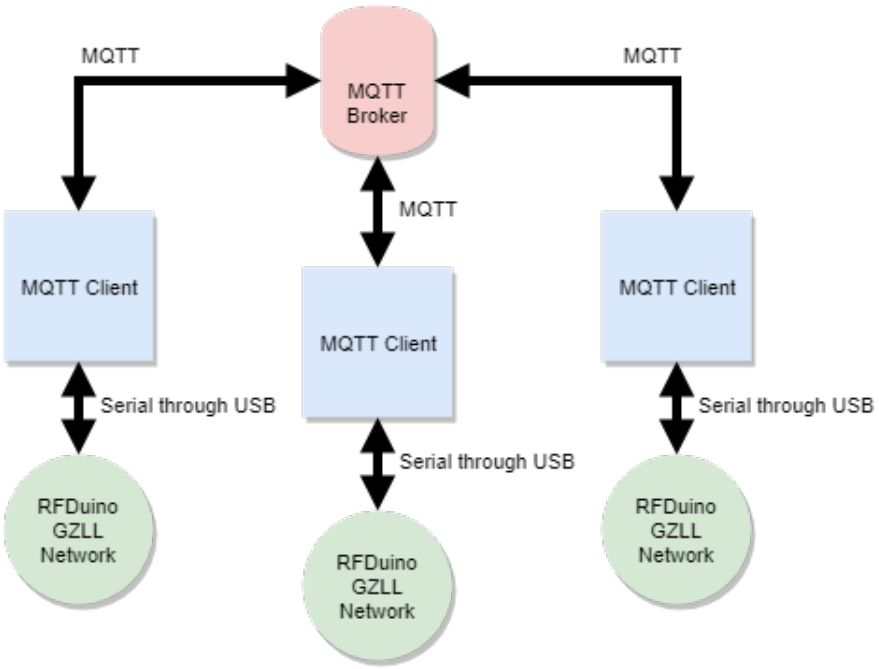

Figure 7.2: General communication setup for Case 2

Figure 7.2 illustrates the general communication setup used for Case 2. The setup utilizes three different communication protocols and is a prime example of the flexibility in communication protocols for IoT devices and how this flexibility can be used to connect virtually any electronic device to the Internet of Things. The pervasiveness is in this case achieved in the MQTT layer, where the MQTT Client could be placed anywhere in the world where there is an Internet connection available. Be it in the same room as another MQTT Client in the network or across the globe, the MQTT protocol allows the ”Follow the Red Dot” game to be played anywhere.

Similarly to Case 1 the idea is to have the RFduino Host to be in charge of delegating the red dot to the RFduino Devices, however in this case there are multiple RFduino Hosts and additional game logic has to be implemented in order to handle the delegation between the multiple hosts. For the particular implementation used in the testing two RFduino Hosts, with an equal amount of RFduino devices in their network, were used and a virtual beacon indicating which of the two was in charge of the red dot delegation was added. For each time the button was pressed on the RFduino Device with the red dot a random number generator on the Host with the beacon was used to determine if the Host currently holding the beacon should keep it or pass it on to the other Host. As long as a Host doesn't have possession of the beacon it remains passive, waiting for the reception of the beacon.

With the addition of serial communication between the RFduino Host and the Raspberry Pi and the MQTT communication between multiple Raspberry Pis comes the need to support additional messages. The messages within the RFduino GZLL network used in Case 1 remains unchanged, while messages used to handle the transfer of the beacon is added. The message used to handle handle the beacon is a simple ID,

i.e. `A` or `B`, indicating the ID of the Host that should receive the beacon. The same type of message has to be passed between the RFduino Host and the Raspberry Pi through the serial communication. More details about how these messages are implemented are presented in section 7.2.

In Case 2 seven of the set requirements from Chapter 5 will be evaluated with a focus on the pervasive aspect introduced in this case. Some of the criteria overlap with some of those tested in Case 1, but as Case 2 expands on the functionality of Case 1 it offers different aspects to the requirements which should be evaluated. The upcoming subsections seeks to present the reasoning behind why these requirements are relevant to the case and how the implementation is used to evaluate each of the different requirements. The evaluation itself and the associated findings are presented in chapter 8 and 9. The five requirements evaluated in Case 2 are:

C2.2 and **C3.1** - Addressability
C2.3 - Device-to-device communication
C2.6 - Scalability
C3.2 - Interoperability
C3.3 - Connecting new devices
C3.4 - Distributed vs Local use

Addressability

The MQTT messaging protocol allows clients to publish to topics that multiple other clients are subscribed to without the knowledge of who or how many are currently subscribed. In the case of the ”Follow the Red Dot” implementation a need to handle the addressability of these MQTT messages arises as the MQTT layer is used to transfer the host beacon between hosts and it is crucial that the correct host receives the beacon when it is sent from one host to another. The addressability issue could easily be handled by utilizing either the message topics of the MQTT protocol or by adding a sort of identification to the message body which is associated with an unique identifier set in the code running on the Raspberry Pis. For this particular implementation, running with only two Raspberry Pis and two RFduino Hosts, either of the two implementations would provide the necessary addressability. Table 7.4 displays examples of how the MQTT messages and topics could be structured for each of the two implementations.

Regarding the messaging between the Raspberry Pi and the RFduino Host there is no need to handle the addressability as the serial communication goes directly through the USB-ports on each of the devices and none of them have any other form of wired serial connections.

Table 7.4: Examples of handling addressability for MQTT messages

Identification location	MQTT Topic	MQTT Message
Topic	Master/FtRD/RPI1	`"Beacon"`
Message	Master/FtRD	`"RPI1:Beacon"`

Device-to-device communication

As mentioned in the previous section, handling the addressing of the data being transfered is easily handled, however the reliability of the messaging needs to be addressed when two additional layers are added to the messaging stack. In Case 1 the concurrency and addressability of the Gazell Link Layer protocol is presented and this remains unchanged for Case 2. The MQTT messaging protocol also offers a high degree of reliability through three different Quality of Service (QoS) levels, as presented in section 6.3, where the developer choses the level based on the constraints of the application.

The final communication layer is the serial communication between the Raspberry Pi and the RFduino Host. The serial communication is a bare boned service that doesn't provide any form of error detection or error correction, but by utilizing the USB ports on each of the devices instead of the pins the implementation is at least provided with error detection and a retry function. By adding a cycle redundancy check (CRC) any data corruption that occurs during the serial transfer should be detected and used to trigger an error handling function, i.e. in the form of a package retransmission.

Scalability

Even in a prototyping setting is important to have support for scalability in an IoT based application as the ability to support more devices gives a higher degree of freedom for the developers when designing pervasive games. By connecting the RFduino local network to a Raspberry Pi gateway that supports MQTT communication the developers gain the opportunity to have the potential of thousands of IoT devices communicating simultaneously (Scalagent), which should enough for most prototyping cases. However, with the increased scale comes the issue of handling the beacon distribution as each of the devices needs to be aware of all the other connected devices.

Interoperability

The use of the MQTT messaging protocol provides the option to utilize other IoT devices in the same role as combination of the Raspberry Pi and the RFduino networks, as it is built on the publish/subscribe paradigm. The protocol allows for any device supporting MQTT to subscribe and publish to the topics used in the application, meaning that the developers could use other types of IoT devices as game objects without having to alter the message structure used for the Raspberry Pis.

Connecting new devices

The level of abstraction in the MQTT messaging protocol makes the task of connecting new devices to a network as easy as can be, especially outside runtime. Since the overall code is quite simple, adding additional devices, either of the same type with the Raspberry Pi and RFduino setup or other types of IoT devices with MQTT support, requires very little customization and the implementation overhead is small. However, if we were to have the ability to add additional devices during runtime the code would quickly become more complex, as the game logic runs on several different units, meaning that all of them need exact knowledge of all the units that are connected during runtime. The overhead here is quite severe, mainly due to the fact that all devices needs to be identified and the ability to provide information about themselves. Take for example a case where an Raspberry Pi with an RFduino network consisting of one Host and four Devices were to connect during runtime. The RFduino Host would need to provide information about how many Devices it has connected and receive information about all of the units that are already running in the game before being able to engage in the gameplay. It is far from an impossible task to complete, but compared to the relatively simple and straightforward code that would be needed if the amount of active devices were predetermined, it is clear that it would provide a lot of extra work, especially in a prototyping setting.

Distributed vs Local use

The core feature of this case is the ability for the game to be played in a distributed fashion, meaning that the game no longer is restricted to the range of the radio unit on the RFduinos. In theory the game could be played from anywhere in the world with an available Wi-Fi connection and multiple players could collaborate across a geographically distributed playground. With the added functionality also comes added complexity, compared to the simple implementation in Case 1 the two additional messaging layers adds a higher degree of complexity. While the MQTT messaging between the two Raspberry Pis is reliable, fast, and easy to implement, the serial communication between the Raspberry Pi and the RFduino Host requires quite a bit of error handling and error recovery as most forms of serial communication is prone to errors due to multiple factors.

7.1.3 Case 3: Distributed - Digital Twin

In the third and final case the pervasive implementation created in Case 2 is expanded, as illustrated in figure 7.3, in order to utilize the proposed framework for prototyping pervasive exergames. The overall goal of this case is to move the game logic outside of the IoT devices in order to provide a higher degree of customizability for developers when it comes to code implementation and the use of a wider array of IoT devices.

Moving the game logic to the game engine gives the IoT devices the role of being purely interactive game objects, removing the need to have complex game logic running on these devices. This eliminates the need to develop extensive code for each type of IoT device as they only need to support ways to interact with and record the state information of their components and have them mirrored in the engine. By providing a communication standard for the IoT devices through the use of custom messages and MQTT message topics it should allow for novice developers to connect their device and take part in pervasive games from virtually anywhere in the world.

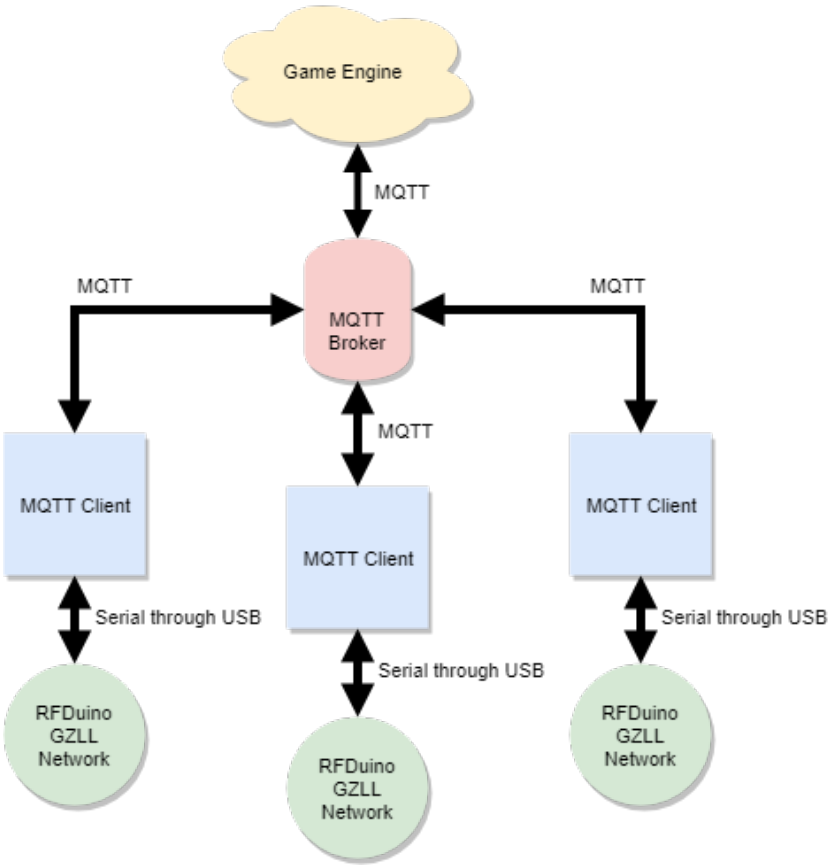

Figure 7.3: General communication setup for Case 3

In order to move the game logic outside of the IoT devices themselves the core of the framework is implemented within the Unity Engine in C# code. By running as an MQTT Client the game engine communicates with the IoT devices in the same way as the IoT devices communicated with each other in Case 2, through publishing and subscribing to a set of predetermined MQTT topics. The communication structure returns to a star topology, similar to the one presented case 1, but with the game engine as the center of the topology. Figure 7.4 illustrates how all the communication now goes back and forth between the IoT devices and the game engine. Even though MQTT offers the option to have direct communication between the devices, similar to the implementation in Case 2, the framework is designed to have the IoT devices not consider the

existence of other devices. The IoT devices should only have to support communication that allows the framework to have real-time mirroring between the physical and virtual twin representations of the IoT devices.

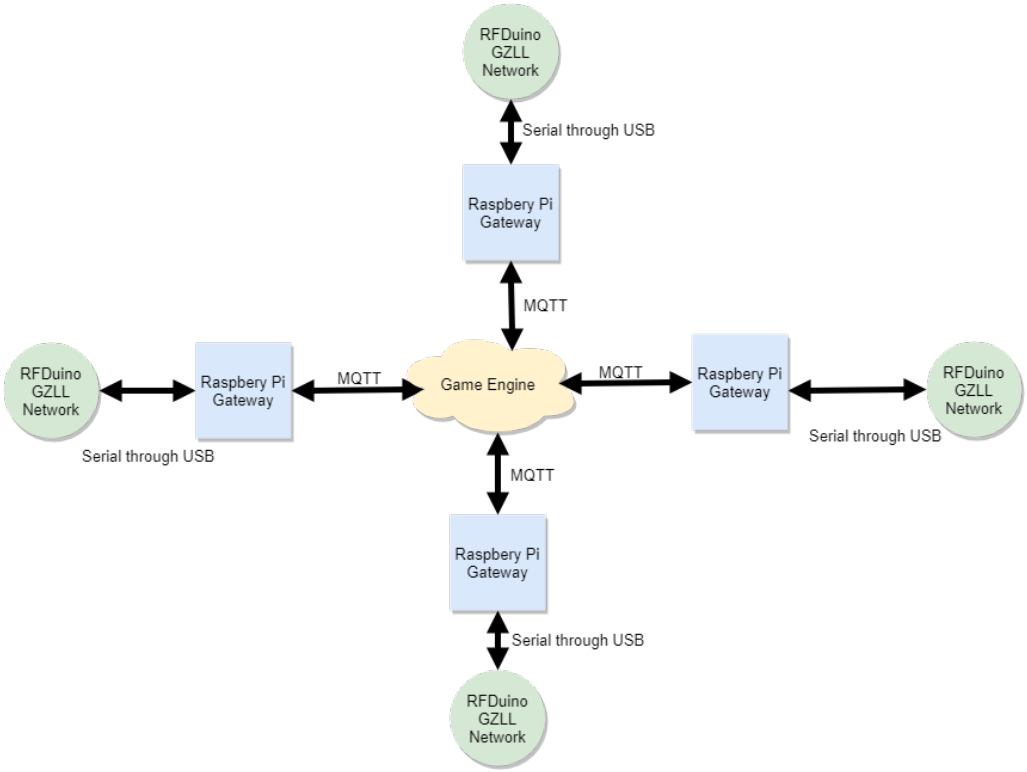

Figure 7.4: Star topology communication setup of Case 3.

Having a framework support numerous different IoT devices with various I/O components means that the framework needs to have a standardized way for developers to identify components and their ways of interacting with the framework. An example is shown in figure 7.5 where there's two RFduinos with different I/O's. The one on the left detects user interaction through the use of an ultrasonic sensor while the one on the right uses a regular push button. Given that the game developer wants to have both of the devices be valid game objects the framework needs to provide a level of abstraction for the different I/O components in order to have the game act in the same way when input is detected from an ultrasonic sensor, a push button, or any other type of input component. The same goes for output. This relieves the developer of the need to have specific handling for each type of input device, reducing the implementation time which is critical to the prototyping process. A more detailed description of how this is handled and implemented through the use of dedicated MQTT topics and component classes is presented in section 7.2.

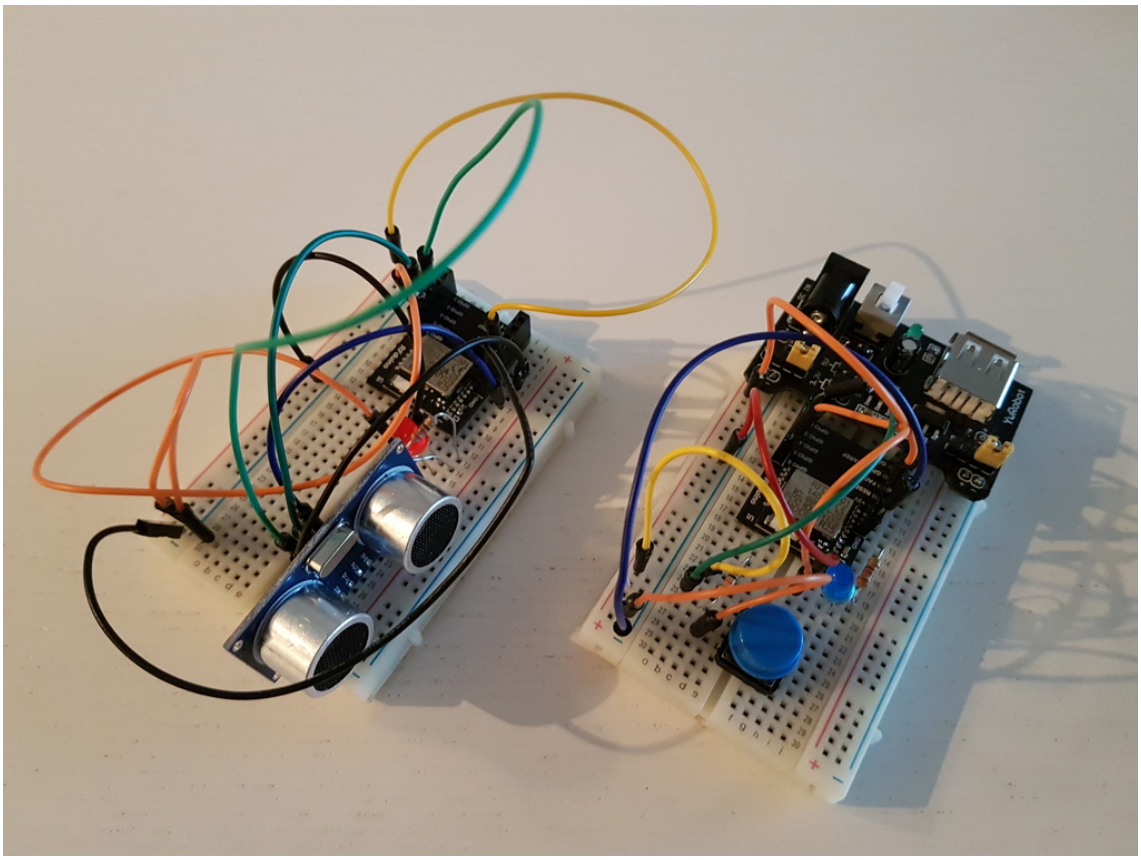

Figure 7.5: Two RFduinos setup with different input components.

Since Case 3 utilizes most of the same setup and functionality of case 2 a lot of the requirements evaluated in Case 2 overlaps with Case 3 and won't be reevaluated for this case. Case 3 will focus on the evaluation of the features that the prototyping framework introduces and how they effect the overall prototyping process. Four of the requirements from Chapter 5 will be evaluated in this case and the upcoming subsections presents why these requirements are relevant to this case and how the implementation is utilized to perform the evaluation. The requirements evaluated for Case 3 are:

C1.0 - Reduce implementation cost
C3.3 - Connecting new devices
C3.5 - IoT Flexibility
C3.6 - Game logic centralized
C3.7 - Scalability

Reduce implementation cost

Being able to reduce the implementation cost of prototyping pervasive games is the overall goal of the entire framework. Every feature implemented is added to reduce

the development time for developers in the prototyping stage, making it easier for developers to have their products come to fruition. The most critical features that are implemented to achieve this is the ability to have abstract I/O components for the IoT devices, the support of any IoT device that is able to support MQTT, and having the game logic moved out of the IoT devices and into the Unity game engine. By abstracting the different I/O components that the IoT devices can support into a component with standardized functionality, the need to have individual handling for each component within the game engine is removed, meaning developers could utilize existing IoT devices which they may have at their disposal at the prototyping stage without having to make alterations on the hardware side. Also, by running all the communication with the game engine through MQTT means that almost any IoT device with the ability to connect to the Internet could be used as a game object, as MQTT is now supported by most major programming languages.

In addition to implementing key features that remove implementation overhead for the developers, it is important that the framework is intuitive and requires little effort for the developers to familiarize themselves with. By providing well a documented open source API developers will be able to integrate and expand on the framework to prototype pervasive games in a more efficient way compared to developing the functionality from the ground up by themselves. Outside of the API the framework code should also be well commented to provide useful insights for the more curious developers that seek to understand the underlying mechanisms of the framework.

Connecting new devices

In Case 2 the issue with connecting new devices in runtime when the game logic run on several different devices was addressed, but by having the game logic run only in the game engine the task of connecting new devices during runtime becomes much easier. Since the code on the game engine is the only code that needs to be aware of how many IoT devices are connected as game objects the overhead of writing code specific to handle new connection in runtime is reduced severely. With the implementation of ”Follow the Red Dot” in this case there hasn't been added any specific code to connect devices during runtime, but with the framework's support for setting up new Digital Twins and including them as active game objects the task that remains for the developers is to add functionality for scaling their game when new objects are added.

IoT Flexibility

As mentioned previously, one of the key features of the framework is the ability for developers to utilize the IoT devices of their choice with associated I/O components. The reason for this is the same as in Case 1, as the more IoT devices and I/O components that are supported the framework gets closer to realizing the five interaction

models that were presented in section 2.3. Distinctive from the first case is the need to support different devices. The only restriction set by the framework that the device needs to be able to act as an MQTT Client, either directly or through another device that is able communicate on behalf of the device, similar to the setup used with the Arduino and the RFduinos. In order to be able to support different devices and components several standard for identification is proposed for the initial implementation of the framework. The goal of these standards is to provide developers with a simple, comprehensible way of connecting their custom IoT devices without creating a large overhead in the prototyping process. More details about these standards are presented in section 7.2.

Game logic outside

Moving the game logic from the IoT devices to a game engine like Unity means that the need to write extensive code for the IoT devices themselves is reduced. Even though newer IoT devices are powerful enough to run complex code, not every device supports the same programming languages, meaning that if developers want to have an as wide as possible support for different IoT devices they would need to develop the game logic for several different programming languages. This could be a fairly time consuming activity compared to developing the game logic in only one language. Although moving the game code to an game engine would reduce the amount of code produced for specific IoT devices, it still doesn't keep the developers from writing some code for each of the devices as the devices have to act according to the actions initiated by the game, i.e. turning on an LED or reading the current temperature in the room. But these kinds of I/O interactions are usually well documented for most IoT devices and the level of development skill required to implement this type of code is quite low, meaning most developers with basic programming knowledge should be able to write the implementations without too much effort.

Another perk of moving the game logic to a game engine is the ability to build game clients that could be run on almost any platform. Whether it be a smart phone or a game console, this gives the developers the option to build complete applications that are able to be run outside the game engine, while keeping all the development within the engine. This type of flexibility eases the step from prototype, which might be run from the engine itself, to a commercial product that could be distributed across different platforms.

Scalability

The introduction of a centralized game logic removes the need for the IoT devices to have any knowledge about the existence of other devices that are connected to the game. Having the game engine or game application be the only device that needs to

keep track of the connected devices means that the support for scaling the application is far better as the traffic caused by transferring device information between every device is removed. This is a crucial feature in cases where the developers want to develop large scale games as the amount of data that needs to be handled by the MQTT broker is greatly reduced.

7.2 Implementation

In order to design and implement the software used for the three cases the Design Science Process Model (DSR Cycle), introduced in chapter 3, was used in order to follow the iterative approach of the design and creation process. The development was done through several cycles as the earliest cycles within each case identified issues with the initial proposals that needed to be addressed. All of the cases were developed in turn, from Case 1 to Case 3, and were treated as separate cases, however any useful knowledge that would benefit the later cases was taken into consideration throughout the development. The following subsections will present the implementation part of the research, where the initial design proposals presented previously were realized through three separate artifacts.

Case 1

As presented earlier in this chapter and in the case description in chapter 4 Case 1 is built purely on the RFduino device with an LED and a push button as the chosen I/O for the user to interface with. The initial proposal for the implementation was built on the idea of having the main game logic be run of the RFduino Host and have the RFduino Devices be used as the interactive game objects. This meant that the only task that should be performed by the RFduino Devices were displaying whether or not they are in possession of the red dot and registering user input. Figure 7.6 displays the schematics for the hardware setup of the RFduino Devices with the LED and the push button interface.

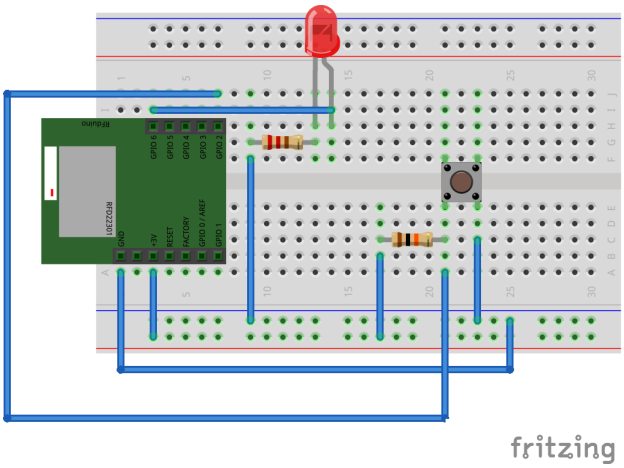

Figure 7.6: Schematics for the RFduino Device

On the software side of the RFduino Devices the logic was built around a boolean variable which was used to indicate if the device was in possession of the red dot or not. When the device doesn't possess the red dot the only action it performs is polling for messages on RFduino Host by sending NULL messages every 100 millisecond. Upon receiving the red dot from the Host the LED is turned on to indicate that it now has the red dot and can be interacted with. While in possession of the red dot stops sending polling messages to Host and waits for the player to press the push button. Once the player presses the button the LED is turned of and a message indicating that the button was pressed is sent to the Host, and the Device returns back to its initial state.

The RFduino Host is in charge of delegating the red dot to the Devices and does so by utilizing a Random Number Generator (RNG) function based on the number of Devices that are currently connected to the Host. This delegation is done upon game start, which is controlled by pressing a push button attached to the Host, and upon receiving a message indicating that the button has been pressed on the Device that had the red dot. The only restriction on the delegation is that the dot should not be passed on to the same Device two times in a row, which is handled by recursively calling the RNG function until a Device that is not the last the Device to have the dot is selected.

During the evaluation step of the initial proposal it was quickly discovered that the implementation was faulty and game breaking errors were experienced during game play. What happened was that when the push button was pressed on a device that was in possession of the red dot it would sometimes fail to reset the boolean value that indicated the possession of the red dot, and also sometimes fail to turn its LED off. The following code segment indicates the code that was run when a button is pressed on the Device that is in possession of the red dot:

```
1     if (hasTheRedDot) {
2       if (button.pressed()){
3         hasTheRedDot = false;
```

```
4           digitalWrite(ledPin, LOW);
5           RFduinoGZLL.sendToHost("#p");
6         }
7       }
```

In the code segment the Device is supposed to set its boolean value back to false (line 3), indicating that it no longer has the red dot, turn off its LED (line 4), and finally send a message back to the Host indicating that the button was pressed (line 5). During testing there was no consistent pattern to indicate what made the code break. In every attempt when the "`hasTheRedDot`" variable was true the Device would send the "`#p`" message to the Host and the Host would successfully receive it, however sometimes the boolean value wouldn't be set to false or the LED would remain on, and even in some cases neither of those lines would trigger. The code was tested on several different RFduino devices with different alterations made to the code as well, but this approach remained too inconsistent to do any proper testing.

The inconsistency observed in the first iteration lead to a rework of the initial idea of the implementation and the proposal of developing the code on the RFduino Host and Device to be closer to how it was envisioned to be in when used with the final framework. This meant that instead of having the Devices act based on if they had the red dot or not, they would now act on instructions sent by the Host, and only the Host would be aware of which Device had the dot or not. The code on the Devices was built to provide interaction with its I/O, both for the players and the developers, meaning that the only thing the Device would do is to turn on and off its LED based on messages from the Host, and send notification to the Host about its button being pressed. On the Host the code was structured to have all the tracking and handling of the red dot based on the interactions with the Devices.

Overall this approach isn't any more complex than the one in the initial proposal and if anything it is closer to the approach sought for in the final case. The RFduinos now send three different messages through the GZLL messaging protocol, in addition to the NULL message used for polling, two being from the Host to the Device and one being from the Device to the Host. Table 7.6 displays the different messages and their functionality.

Table 7.6: Messages used for Case 1

Message	**Sent from**	**Sent to**	**Functionality**
`"#LED:ON"`	Host	Device	Enables the LED on the Device
`"#LED:OFF"`	Host	Device	Disables the LED on the Device
`"#BUTTON:P"`	Device	Host	Indicates that the button has been pressed

Case 2

Case 2 introduces new hardware and moves the application from a local to a global playground. Based on observations made in the development of Case 1 some changes to the initial design of case 2, presented in section 7.1, were made. Instead of having the RFduino Hosts delegate the red dot in the same manner as initially implemented in Case 1, the logic used in the final implementation of Case 1 is used for this case. This means that code for the Devices remains unchanged, while the code for the RFduino Host receives alterations in order to address the added functionality of the beacon and the serial communication with the Raspberry Pi.

The addition of another Host to interact with leads to the inclusion of a beacon, which is used to indicate which of the two Hosts should be delegating the red dot to its Devices. This beacon is simply represented by an ID character which is sent between the two Hosts. If a Host receives a beacon matching its pre-set ID it proceeds to delegate the red dot to one of its devices in the same manner as done in Case 1. In order to determine if the beacon should be passed on to the other Host a RNG function is used, and since the dot cannot be passed back to the same Device two times in a row and both the Hosts have a similar amount of Devices connected to them, the chance is always higher for it to be passed back. In the implementation six RFduinos, meaning that each of the Hosts have two Devices each connected to them.

By adding the Raspberry Pis for global communication there is also added some code in order to support serial communication between the RFduino Hosts and the Raspberry Pis. In the initial design the messages that were transfered were simple one character messages, which during testing proved to cause minor issues in regards to reliability as serial communication is known to be prone to errors caused by static. This meant that the serial messaging had to be reiterated and the messages were made longer, and less-than (<) and greater-than (>) symbols were added to the beginning and end of the message to ease message parsing on both ends.

Table 7.7: Topics and messages used for Case 2

Topic : Message	**Functionality**
`"FtRD/RPI1":"<#BEACON>"`	Indicates that the Host connected to the Raspberry Pi with ID 1 now has the beacon.
`"FtRD/RPI2":"<#BEACON>"`	Indicates that the Host connected to the Raspberry Pi with ID 2 now has the beacon.

Support for MQTT and serial messaging on the Raspberry Pi was implemented in Python 3.5 and runs of a single script. Given that the pool of messages being transfered to and from the Raspberry Pi was very limited the MQTT Clients only published and subscribed to a single topic as this is sufficient. The Raspberry Pi's role is simply to be a communication tool, which means that it will publish received serial messages

to the MQTT topic and send any received message on the subscribed MQTT topic to its connected Host through the serial communication. Table 7.7 displays how the combination of topics and messages was used to handle the beacon between the two Hosts.

In regards to the hardware setup the RFduino Devices were setup identical to Case 1, while the RFduino Host now is connected to the Raspberry Pi through the use of a USB shield and the USB port on the Pi, as shown in figure 7.7. Several tries were made on using the connection between the digital pins on the Raspberry Pi and the RFduino Host to support the serial communication, but it proved to be prone to errors as the communication link was sensitive to static in the environment and the USB ports provided a more stable environment with less errors as it supports built in error detection and retry functionality.

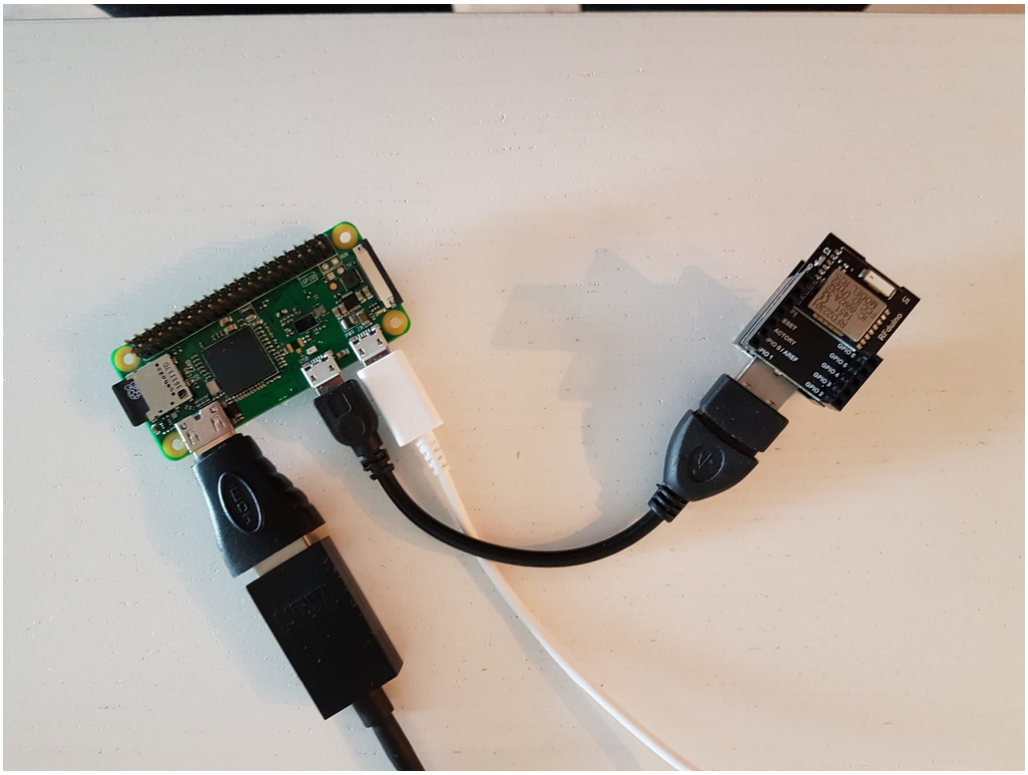

Figure 7.7: Hardware setup of the Raspberry Pi Zero W and the RFduino Host.

Case 3

In addition to an implementation of ”Follow the Red Dot”, Case 3 also includes the implementation of the proposed framework for prototyping IoT-based pervasive games. The framework itself is supposed to support features that reduces the overall implementation cost, including time usage and other monetary costs, of prototyping IoT-based pervasive games. The main features are the ability to support Digital Twins, having the game logic centered at one location, and support any IoT device that is able to support MQTT communication. This is done by implementing features that support

an intuitive API allows the developers to engage in the prototyping process quicker, removing the overhead of developing code to connect, track, and communicate with the IoT devices. More details about the implementation of the framework will be presented in the next subsections before the implementation of the instance of ”Follow the Red Dot” is presented.

The framework

The framework code is based around two main object types, the MQTT handler and the IoT Device. The MQTT handler is responsible for handling any communication happening through the MQTT Client, which includes publishing and subscribing to a preset of MQTT topics, message parsing, and message distribution. The IoT Device is the virtual representation of the Digital Twin which contains the information about the IoT Device and provides ways for developers to interact with the virtual device. Any interaction that happens on the virtual side is mirrored on the physical side by transferring information about state changes to the physical device.

The MQTT handler is implemented as a singleton and hides most of its functionality for the developers as most of it involves message handling, parsing, and distribution which the developer shouldn't necessary need to deal with. The only functionality that the MQTT handler exposes to the developer is related to the connection of new devices and how the developer wants it to be handled. The handler has the option to connect new devices automatically, which leads to the handler adding the virtual part of the Digital Twin making a connection request, or having the developer implement custom handling of connection request by subscribing to the `OnConnectRequest` delegate function.

The underlying functionality of the MQTTHandler is built on eight different MQTT message types that are used to communicate with the physical IoT devices. Each of these types have a specific purpose and is used to communicate different actions and attributes which in turn is used to handle the connection of new devices and mirroring the Digital Twins. Table 7.8 shows the different message types, the structure of the message being transfered, and the associated sender and receiver. The structure of the JSON-objects sent on the `State` and `Interface` topics are presented in Appendix A.4. Table 7.9 describes the purpose of each of the topic types.

Since most of the messages being sent from the Unity MQTT Client contains information specific for only one physical IoT device there is a need to add a way to have the information reach that specific device as the underlying publish and subscribe structure of the MQTT message protocol allows any MQTT client to subscribe to any specific topic. The way this is done in the framework is by adding additional levels to each topic in order to be able to specify the receiver of a message. Table 7.10 shows the basic structure of the MQTT topics used and examples from the implementation used in ”Follow the Red Dot”. By using such a structure it adds the support to send messages

Table 7.8: MQTT topic types and message structure standards for the framework.

Topic	Message structure	Sender	Receiver
`Event`	"LED:OFF"	Unity MQTT Client	Physical IoT Device
`Action`	"BUTTON:P"	Physical IoT Device	Unity MQTT Client
`StateReq`	"REQ"	Unity MQTT Client	Physical IoT Device
`State`	JSON object	Physical IoT Device	Unity MQTT Client
`InterfaceReq`	"REQ"	Unity MQTT Client	Physical IoT Device
`Interface`	JSON object	Physical IoT Device	Unity MQTT Client
`ConnectReq`	"REQ"	Physical IoT Device	Unity MQTT Client
`Connect`	boolean	Unity MQTT Client	Physical IoT Device

Table 7.9: MQTT topic type description.

Topic	Description
`Event`	Sends a message indicating that a component of the virtual twin has changed state, and it should be mirrored on the physical twin.
`Action`	Sends a message indicating that a component of the physical twin has changed state, and it should be mirrored on the virtual twin.
`StateReq`	Sends a request to the physical twin asking the device to send the state information of its I/O components.
`State`	Sends a JSON object containing state information for each of the I/O components connected to the device.
`InterfaceReq`	Sends a request to the physical twin asking the device to send information about which I/O components are connected and the different states they support.
`Interface`	Sends a JSON object containing information about all the I/O components currently connected to the device
`ConnectReq`	Sends a connection request requesting permission to connect to the game.
`Connect`	Sends a boolean value indicating whether or not the device is permitted to connect and take part in the game.

to one specific device, i.e. the `DEVICE4` RFduino connected to the Raspberry Pi with id 2 in the first example, or a subset of devices, i.e. all of the devices within the `House` environment in the second example.

The additional layers, `Environment`, `IoTDevice`, and `Child`, is matched with variables set on the IoT devices and the combination of all three effectively acts their identifier. For each of the layers it is vital that values of the subsequent layer are unique, i.e. within the `House` environment there shouldn't exist any `IoTDevice` with matching IDs, but there could exist a device with the same ID in the `Apartment` environment. The topic layer structure used in the framework could be altered by the developers to better suit their device structure, however this would lead to a bit of extra work as the identification of the IoT devices would need to be altered to match the new topic layer structure.

The second major object type that the framework is built around is the IoT Device which is the virtual representation of the Digital Twin. The IoT Device is represented as

Table 7.10: MQTT topic structure with an example topic from ”Follow the Red Dot”.

Type/Environment/IoTDevice/Child/
`Action/House/RPI2/Dev4`
`Action/House/#`
`Action/House/+/Dev1`

a game object and has separate IoT components that can be attached to the device to mirror the I/O modules on the physical device. These components adds the ability for the developers to add virtual interactions that can be triggered from the game engine and be mirrored on the physical twin. The IoT Device object doesn't offer any ways to interact with the physical device, but serves as a hub for storing information about the device itself, i.e. ID, children, and attached components, and mirroring incoming information from the physical twin. All of the IoT Devices are managed by a Device handler which offers functionality for keeping track of each of the connected devices, adding new ones, and removing already connected devices from the game.

By having the interaction based around the IoT components rather than the IoT device means that developers can add game logic for custom components, i.e. an ultrasonic sensor, that would work for any type of IoT device that has that component attached rather than having to write specific code for each different combination of I/O modules on the different IoT devices. The framework offers the base `IoTComponent.cs` class that can be used to create customized sub-classes that match the developer's intended use and potentially support all of the interaction models presented in Chapter 2. Additionally with all of the information about the state of each the components attached to the device the developers are free to add custom rendering scripts to represent the virtual devices on screen and add ways to players to interact with the virtual device, i.e. through keyboard shortcuts, mouse interaction, or even speech input. The framework only comes with a simple render that displays the device information textually on screen and has a single mouse click interaction that can be attached to the interaction function of any component attached to the IoT device.

Follow the Red Dot

Implementing the instance of ”Follow the Red Dot” that was built using the framework had major implications on the development of the framework itself as the need to add, remove, or modify the functionality of the framework emerged throughout the development. The core of the framwork was initially developed in the first iterative cycle and additional functionality was added in the second iteration. The main issue with the first iteration was that it was too focused to fit the development of ”Follow the Red Dot” and left out key features that would be useful for other types of pervasive games, i.e. the ability to add custom rendering and ways to interact with each components. In

the first iteration the interaction functionality was performed on the IoT device, rather than on the components, leading to limitations when adding custom components with custom interactions. During the second iteration changes were made to the framework to match the functionality presented in the previous section and the implementation of ”Follow the Red Dot” was altered to match the new changes in the framework.

For the ”Follow the Red Dot” instance there was created two custom IoT components, one for the push button and one for the LED on the RFduino Devices. Since there is only one input component the interaction on the virtual twin is done through a simple mouse click on the base textual representation of the device. The output component is enabled and disabled based on the game logic presented earlier. Functionality for handling the connection of new devices during runtime was also implemented. Figure 7.8 shows the base visual representation of the virtual twin in Unity.

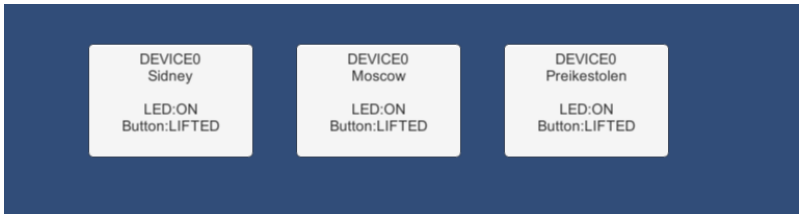

Figure 7.8: The base visual representation of the IoT devices.

There was also a need to add additional functionality on the RFduino Host and the Raspberry, as they needed to have functionality that supported the different topics and messages used in the MQTT messaging. In the code used for Case 2 the message sent between the Raspberry Pi and the RFduino was on the format `"<#Beacon>"` as the MQTT messages were transfered two different topics between the two Raspberry Pis. With the addition of several different topics in Case 3, there was a need to add the topic to the serial string message in order for the Raspberry Pi to identify which topic to publish to, and for the RFduino Host to trigger the correct functionality to match the incoming message. This produced the following base message structure: `<"topic:message">`, where the format in the `message` varies and is parsed based on the topic. The only code that wasn't altered was the code on the RFduino Devices as the software and hardware used in the previous cases suited the implementation in Case 3.

Chapter 8

Results and Evaluation

This chapter presents the results related to each of the four research questions established in Chapter 1. The results for RQ1 are based on a literature review and presents a small set of pervasive games that utilize IoT technology in their implementation. For RQ2 a quick summarization of the results is presented as the main results are presented in Chapter 5 and used for as a tool for answering the remaining research questions. The results for RQ3 are presented in detail in Chapter 6 and are summarized in this chapter. Finally a evaluation related to RQ4 is presented. All of the results presented in this chapter are discussed in the next chapter.

8.1 RQ1

Even though the topic of IoT and ubiquitous computing isn't a new one in terms of research, both are still fairly new in terms of the commercial aspects and the use for these technologies are still being explored. This also relates to the use of IoT in pervasive games as it is a topic that has gained some interest only within the last decade. For this reason there exists fairly few pervasive games that utilize IoT technology and they are almost exclusively used for research purposes. Even though its a fairly new approach to pervasive games there is little doubt that the increasing accessibility and deployment of IoT technology will provide for exiting ways for developers to create new and unique pervasive games. In this section three of the most cited pervasive games that utilize IoT technology are presented and a description of their research purpose and how they utilize different IoT technology is provided.

8.1.1 Treasure

Treasure (Guo et al. (2012)) is a pervasive treasure hunt game built in order to analyze the concept of "Design-in-play" introduced in the same paper. Like any treasure hunt game the goal of the game is to locate some sort of treasure, but this instance utilizes IoT technology to bring pervasive elements to an in-house treasure hunting game. The

game is divided into to stages, one game authoring stage where one group of game authors reconfigure the settings of the game to fit their preferences, and one game play stage where a group of players play out the game set by the game authors with the goal of locating the treasure hidden by the game authors. The game supports a co-location mode, where the game authors and players are within the same room, and a networking mode, where the users are playing from different locations.

By utilizing a projection device called Prot which is a combination of a projector, an ultrasonic speaker, a webcam, and a 3-DOFs rotating base. In combination with ultrasonic 3D tags placed on the hidden objects the Prot reveals information about the objects and clues for where to find the treasure in the form of animations or images projected on the wall that the player is currently facing. The game also uses MOTE sensors to monitor the orientation and environmental changes around objects placed by the game authors. Both the location information gathered by the ultrasonic 3D tags and the MOTE sensors is used by the players both in co-location mode and in networking mode as the data can also be displayed to the remote users.

8.1.2 Area Control Game

MacDowell and Endler (2015) introduces an Area Control Game based on the analog playground game know as capture the flag, much like the pervasive game CTF introduced in Sreekumar et al. (2006). The Area Control Game is played out by a certain number of people in a specific area, time, and duration. The players utilize smart devices supporting Bluetooth Smart technology that has the ability to interact with mobile sensor objects (IoT devices such as the Texas Instruments Sensor Tag) that acts as the flags. The game was developed as a part of a case study for an architectural analysis of Pervasive IoT games and is played out by to or three teams, each with at least one flag of their own. When the game starts the flags are placed within areas belonging to each team and then the goal of the game is to capture the other teams flags by bringing them into your own zone. Players are only allowed to take unguarded flags or steam them from other players carrying them. If players from opposite teams meet they will engage in a battle and where the victor will have the ability to steal a flag from the opposing player or avoid loosing the that he is carrying. The game is won when one team has captured all the enemies flag into their own zone.

8.1.3 Barbarossa

Barbarossa (Kasapakis et al. (2013, 2015)) is another example of a pervasive game that utilizes IoT technology. The game classifies itself as a two-phase trans-reality role playing game, where the first phase was available world wide through the application "The Conqueror" and the second phase was restricted to invited higher rank players. The game was initially created to display how and open and portable game design archi-

tecture could be used for developing pervasive games. Later it was used to explore how the use of a preparatory game phase could be used to discover the most appropriate evaluators for a pervasive game compared other more common ways of gathering evaluators during the development phase, like e-mail invitations, personal contacts, or recruitment of colleagues/organization employees.

In the first phase of the game players try to free the conquered city of Mytilene which has been captured by the Barbarossa pirate brothers. The game is played out through a custom Android application that utilizes the location-aware technology within Android smartphones. During the second phase three players utilize custom Android applications in order to solve assigned missions that will lead them to the location of a locked chest which can be opened by two lock combinations that is also discovered through the missions. For these missions the developers have made use of a wide array of technology, including the use of IoT sensor devices (SunSPOTs), that provide data used to solve the missions.

8.2 RQ2

Table 5.2 in Chapter 5 presents the six different requirements for IoT technology that makes it easy to prototype pervasive games. These requirements are listed from most to least important, with the most important requirement listed first. The requirements are based on the basic principles of ubiquitous computing, the IoT vision, and the base interaction models introduced by MacDowell and Endler (2015) in Chapter 2.

C2.1, C2.2 and C2.4 is mainly based on the interactions models presented in Chapter 2 as being able to uniquely address IoT devices that have the ability to be wireless and support different I/O modules enables developers to easier prototype pervasive games that make use of IoT technology. C2.2, C2.3, C2.5, and C2.6 are key features of ubiquitous computing and realizing the IoT vision, and thus they are also key attributes for IoT technology that will be used when prototyping pervasive games.

8.3 RQ3

Chapter 6 presents a set of available technology that can be used to realize a technology stack used in a prototyping environment suited for pervasive games. The architecture of the stack is based on the five-layer model presented by Al-Fuqaha et al. (2015) and uses four of the five layers presented in the paper. The reason for not including the Business layer is because this layer would be made up of the prototype created by the prototyping developer which should be left up the developer itself. The four layers that make up the stack are: Application layer, Service Management layer, Object Abstraction layer, and the Objects layer.

In addition to presenting relevant technologies for each layer, Chapter 6 also presents a promising stack for doing a proof-of-concept evaluation of the framework developed for prototyping IoT-based pervasive games. The proposed stack is made up of a combination of Arduino and Raspberry Pi technology in the Objects layer, BLE and Wi-Fi in the Object Abstraction layer, MQTT in the Service Management layer, and finally Unity Engine in the Application layer. There exists several different combinations of technologies across the different layers that would be able to make up a stack that could be used for this proof-of-concept evaluation. The following section evaluates the proposed stack in detail against the each of the evaluation criteria established in table 5.3 in Chapter 5. It is important to note that the evaluation in this section is done for the proposed technologies for the stack based on the available documentation and their advertised features, and not based on personal experiences with the technologies, this will be done in section 8.4.

Evaluating the proposed stack

Being able to give the framework support for unique addressability is the most important requirement for the technology stack. The proposed stack is able to realize this through the chosen technology in the Object Abstraction layer where BLE and Wi-Fi both support unique identification of each of the devices that are connected. Even though MQTT is able to abstract the devices that publish and subscribe to the broker, its topic architecture allows the topics to be used to address specific devices that are connected to the broker. The abstraction of devices using the MQTT protocol also provides a high level of interoperability, as the need for devices to be aware of the existence or configuration of other devices is removed.

The stack also offers great support for the framework to handle connecting new devices. This can be done in two different ways, either by adding new RFduinos nodes to the RFduino BLE networks, or by connecting new Raspberry Pi gateways with associated RFduino network. The stack even supports connecting other types of IoT devices that support MQTT communication, meaning that the framework isn't restricted to the technologies chosen in the Objects layer. The usage of the Raspberry Pis as gateways serves as a simple way to take the local area networks created by the RFudinos and connect them to the IoT. By using the Unity Engine to build applications that handle the game logic and deploying these applications to mobile platforms also adds support for developers to deploy their game at any chosen location as long as the mobile device has Internet connectivity.

With the RFduino being built on the Arduino platform and the code developed in the Arduino IDE means that it comes with support for a wide array of different I/O modules. These I/O modules are able to be reproduced, mirrored, and displayed virtually within the Unity Engine by implementing the I/O modules as separate components that can be attached to game objects representing IoT devices. With MQTT being

supported in C# it is possible to connect the IoT devices to the Unity Engine or any application made in the engine, which gives the developers the opportunity to develop their game logic within the game engine and have the engine interact with and react to the I/O modules on the IoT devices.

The final requirement is for the technology to support scalability and this is realized by using the MQTT protocol in the Service Management layer and the combination of the RFduino WLAN and the Raspberry Pi Wi-Fi gateway. Different MQTT broker implementations are supported by load balancing and traffic distribution functionality that is able to support several thousand devices communicating on a single broker. One issue with scaling with the suggested stack is the cost, as there exists devices that would be less expensive compared to the Raspberry Pi and RFduino setup.

8.4 RQ4

After having utilized the technologies from the proposed stack to develop the three instances of ”Follow the Red Dot” and the prototyping framework the insights gained from the iterative development has been used to provide a full evaluation of the stack. The evaluation criteria defined in table 5.3 are also used in this evaluation to provide an comparison of the advertised and experienced features. The evaluation is presented on a layer basis in table 8.1 where each layer is evaluated either as passing the evaluation criteria (+), passing the criteria with some remarks (+*), and failing the criteria (-). The remarks are presented along with the discussion of the evaluation in the upcoming chapter.

Table 8.1: Evaluation of the proposed stack. + : pass, +* : pass with remark, - : fail

Criteria/Layer	**Objects**	**Object Abstraction**	**Service Man.**	**Application**
C3.1 - Addressability	+*	+	+*	+
C3.2 - Interoperability	+	+	+	+
C3.3 - Connecting new devices	+*	+	+	+
C3.4 - Distributed vs Local use	+	+	+	+
C3.5 - IoT Flexibility	+	+	+	+
C3.6 - Game logic centralized	+	+	+	+
C3.7 - Scalability	+*	+*	+	+

Chapter 9

Discussion

9.1 RQ1

In the previous chapter three different pervasive games were presented to display some of the pervasive games that utilizing IoT technology. As mentioned in the introduction of RQ1 the state of the research is in its infancy and there aren't a lot of games out there that utilize IoT technology yet and the ones that are doing so focus on having their own, custom built Things that are created with the only purpose of being used in one specific game. As the wave of IoT is sweeping throughout the world and the amount of connected devices is predicted to increase in the near future there is very little research aimed at how all of these new sensors could be used for pervasive games. The most common uses are to create a tangible user interface (TUI) for pervasive games that previously have utilized positioning technology in order to provide interaction between the virtual and the physical worlds, i.e. the Area Control Game (MacDowell and Endler (2015)) and CTF (Sreekumar et al. (2006)).

Treasure (Guo et al. (2012)) does a good job of displaying how IoT technology can be used to create intuitive TUIs that without breaking the immersion of real-world play. Here the use of ultrasonic 3D tags and the MOTE sensors are applied to common household items which in turn is hidden, providing the same experience that one would get by playing a traditional treasure hunt game without the pervasive element. This displays how the traditional games can be augmented both spatially and immersively by providing the option for remote users to play together and by giving the option of displaying custom actions through the Prot projector. However, Treasure also illustrates some of the issues related to using IoT technology to create TUIs. The authors in Kasapakis et al. (2015) addresses the issue of the orchestration required before the games are played out when IoT technology has to be set up by some of the users or game organizers beforehand. Both games try to make the orchestration a part of their game by having dedicated phases of the game where the orchestration takes place. This solution breaks down some of the frustrations that players feel when having to spend time setting up a game before its played, but it restrains the temporal aspect of

the pervasive games, as it could be difficult to join the game during play and in some cases would force players to sit out until a new round is started.

9.2 RQ2

With pervasive games being such a broad game genre that includes so many different sub-genres, being able to set a final set of requirements that would help ease the prototyping process of these games could easily become a subject to subjective opinions. As for this task, it easy to let the structure and simplicity of ”Follow the Red Dot” color the results of the research and it is important to do a proper evaluation that makes use of existing literature and implementations that make use of the same technology. In addition to this it has been important to look at IoT itself as it is a technology that is quickly evolving, and the best way to try and predict the future of IoT-based pervasive games is to look at the prediction of how IoT will be evolving in the future and how that could effect future IoT-based pervasive games.

The set of requirements should be used as a guideline rather than a list of strict requirements for choosing the IoT technology that is suitable for pervasive games as the architecture and functionality of pervasive games varies. For certain applications just rearranging the importance of the requirements could lead to a more suitable set of requirements for that specific application. It is also important to note that the requirements are suggested for prototyping IoT-based pervasive games rather than developing the final implementation of a IoT-based pervasive game. It could be argued that security should be a requirement as there could be applications that require authentication and encryption to avoid users with malicious intents to tamper with the game. The reason for not including this would be that prototypes in most cases are deployed in controlled environments to better be observed by the developers, making the risk of having attackers target the applications small enough to leave this requirement out of the list.

9.3 RQ3

The enormous attention that IoT receives from the scientific community and from commercial actors has lead the development of a massive amount of different technologies within each of the layers used in the proposed stack. This research presents some of the available technologies that exists, but due to the limitations of the research the amount of technologies presented only scratches the surface of what is available. However, the technologies presented within each layer are among the most popular and well documented technologies that could be found frequently used in prototyping, developing, and implementing IoT solutions and traditional pervasive games.

The proposed stack consisting of Arduino, Raspberry Pi, BLE, Wi-Fi, MQTT, and Unity makes up only one of several possible stacks that would satisfy the requirements set for a stack that would support prototyping pervasive games. The stacks technologies are easily interchangeable, meaning that developers could chose a different set of technologies within one layer and keep have the rest of the technologies within the other layers. This gives developers the flexibility to utilize technologies that are better suited for their specific solution. The proposed stack however, passes all the evaluation criteria that is set for the stack and offers a low entry bar for inexperienced developer by having well documented and easy to use technologies make up the stack.

A concern with the proposed stack is that it is strongly based in IoT-based pervasive games that utilize IoT devices as interactive game objects and doesn't address pervasive games that are more centered around utilizing sensor data, like weather data, to add to their game experiences. This would introduce a different way to handle the data flow, as the interactive game objects follow an event based approach, compared to the continuous data flow of sensor data where different technologies could be more suitable.

9.4 RQ4

The results presented in the previous chapter indicates that the stack passed every criteria that was proposed for a technology stack to be used to create a prototyping environment for IoT-based pervasive games and that it performed well as a stack. There were however a couple of remarks that are worth discussing.

Addressability is well supported by the stack as long as the developers are cautious when implementing their code. MQTTs publish/subscribe protocol is well suited for applications where object abstraction is usable, as devices are able to communicate without knowledge of the existence or configurations of other connected devices, but in order to support addressability MQTT requires developers to create topic structures that can be used to provide unique addressing. The framework that was developed supports a topic structure, presented in table 7.10, that delivers addressability to the developers as long as the devices are identified properly according the topic structure. In applications where addressability is a critical feature it could be argued for using technologies that natively support addressability, but it is important to note that this support often comes at the expense of other functionality, i.e. data rate, scalability, and message size restrictions.

During testing of the GZLL protocol used in the RFduino networks some issues related to reliability were experienced. In cases where the GZLL network were made up of more than four devices and a host the RFduinos experienced message loss and occasionally crashed. There were also issues related to the implementations of the serial communication between the RFduino host and the Raspberry Pis. Using serial com-

munication is known to be subject to issues related to static interference and developing a solution that provided a high level of reliability proved to be time consuming. These issues affected both the stacks ability to connect new devices and its scalability. Limiting the amount of devices that the GZLL networks are able to support reliably means that in order to scale up the application developers either have to accept the possible unreliability or connect more RFduino gateways, which would come at a higher cost.

The combination of the chosen technologies in the Objects layer and the use of a BLE based protocol in the Object Abstraction layer stand out as the weakest points of the stack and other combinations should be tested in future research. It is also possible that choosing a more reliable device instead of the RFduino could solve the issue, as the RFduino seemed to be prone to errors throughout the development. Based on the documentation of the RFduino and the technologies it supports it looked like a promising piece of technology, but as the research went on a lack of documentation and errors became apparent. On a positive note there exists a vast amount of different development boards that support the same technologies as the RFduino, however there are few that come at the size of the RFduino at the time of writing.

9.5 Research Methods

The literature review done for research question 1 revealed the limited amount of research done on IoT-based pervasive games. It is an emerging genre of games, but to this date there exists very few implementations that are documented in scientific literature.

In research question 2 the requirements presented were based heavily on features that are important for IoT and pervasive games, as there exists very little research done on the topic of IoT-based pervasive games. It would be useful to do a deeper research into what requirements are relevant and perform a deeper evaluation of the requirements. The same is also applicable to research question 3, as the amount of technologies that exists and offer useful features for prototyping IoT-based pervasive games is enormous. Performing a more detailed study and comparing the most relevant technologies would provide better knowledge on the topic and could provide additional insights into which technologies offer specific features for pervasive game development.

In the evaluation of proposed stack it could also be useful to have other researchers and developers test the stack in order to provide more feedback and give a more reliable evaluation of the stack. The same goes for the developed framework as doing a proof-of-concept evaluation doesn't necessarily provide a high enough level of validity to the research done. The topic of pervasive games is also very big and developing a stack and a framework that would be equally suitable for every type of IoT-based pervasive game is difficult as the genre contains so many different sub-genres. It could

have been useful to go more specific into certain types of pervasive games to assess what these types of pervasive games need it terms of supported features that would ease the prototyping phase of the development.

9.6 Limitations of the research

Several of the issues that has been brought forth in this section indicates that doing a more in-depth research related to RQ2 and RQ3 would be likely to provide useful insights into the research done in this paper. The limitations of the paper means that the findings of in the paper should be considered as initial proposals to the research questions asked and that the findings could serve as good incentives to do further research on the topic in order to help other researchers and developers.

It is also worth mentioning that even though the issues with the RFduino was observed during development, the time limitations on the research kept us from acquiring other technologies that could fill the space of the RFduino and possibly remove some of the issues related to the implementation.

Chapter 10

Conclusion

1: What existing pervasive games, documented in the scientific literature, utilize IoT technology?

Chapter 8 presents three of the most cited IoT-based pervasive games that are documented in the scientific literature as of now. These games are: Treasure, a pervasive treasure hunt game that makes use of IoT technology to provide information about the objects that are used in the game. Area Control Game, a pervasive outdoor game, based on the traditional capture the flag playground game, which makes use of IoT technology to provide interactive flags that provide additional virtual information on handheld smart devices used by the players. Finally the game Barbarossa, a two phased game where players play out the story of the Barbarossa pirates and utilize IoT technology in order to uncover clues about the hidden treasure in the final phase. All of these games provide useful insights in how IoT technology can be utilized to augment pervasive games even further, while also displaying some of the aspects that needs more research in order to improve the pervasive game experience.

2: What are the requirements for IoT technology that makes it easy to prototype pervasive games?

In table 5.2 a set of evaluation criteria are presented. These evaluation criteria reflect the requirements for IoT technology that makes it easy to prototype pervasive games. The requirements relate to important features of IoT in general as well as proposed features that make it easy to prototype pervasive games. The list of requirements include flexibility related to I/O modules, addressability, device-to-device communication, power management, local and distributed use, and scalability. The wide collection genres within pervasive games suggests that different features might be required for certain genres and that the list of requirements should be considered as more of a guideline than a final set of requirements to consider when choosing IoT technology to support pervasive games.

RQ3: What are the available tech stacks to realize such a prototyping environment and what is a promising stack for doing proof-of-concept evaluation?

Chapter 6 presents a suggested technology stack made up of the following four layers: Application, Service Management, Object Abstraction, and Objects. The same chapter also presents different technologies that exists within each of the layers and support features that are suitable for making up the technology stack used for prototyping IoT-based pervasive games. Unity Engine, MQTT, BLE, Wi-Fi, Arduino, and Raspberry Pi is suggested as promising technologies to make up the stack. The stack is evaluated in Chapter 8 and showed to support promising features for supporting prototyping IoT-based pervasive games. Chapter 9 discusses the need to perform a wider study related to identifying features from the different technologies that exists today that would be suitable for prototyping IoT-based pervasive games, as the limitations of this research and lack of other research on the topic suggests that a wider study could produce useful insights into the topic.

RQ4: How suited is the selected stack for creating such a prototyping environment?

The proposed stack from research question 3 was evaluated in Chapter 8 and proved to be suitable for supporting the framework developed to ease prototyping IoT-based pervasive games. The evaluation identified minor remarks related to the stacks support for addressability, connecting new devices, and scalability, however, none of these issues caused major concerns that kept the technology from realizing the three implementations of ”Follow the Red Dot”. Issues related to the RFduino and its implementation of the GZLL protocol caused some unreliability, and the discussion in Chapter 9 suggests that replacing the RFduino with other Arduino-based technology would provide a stack with similar capabilities to the one suggested.

Chapter 11

Recommendations for Further Work

The results of this research indicates that there are many possibilities for future work related to the topic of IoT-based pervasive games. In a short term perspective it could be useful to do a study that went deeper into each of the four layers to identify and bring forth technologies that are suitable for developing pervasive games that utilize IoT technology. Being able to address each layer specifically and classify features that the different technologies offer would make it easier for developers to chose technologies that best suit their genre of pervasive games, as it is difficult to argue that one specific set of technologies would be the best fit all types of pervasive games.

In the long term it could be useful to look at the applications of IoT-based pervasive games as a serious tool, i.e. for health promotion or rehabilitation. Research have already shown the positive outcomes of pervasive games as a serious tool and being able to utilize the future expansion of IoT to easier deploy serious pervasive games could increase the deployment rate and possibly help more users. It could also be interesting to expand the framework to add Cloud support. The Cloud support could make it easier for researchers to gather research data and explore usage patters among its users. With an expansion like this it is also necessary to address privacy concerns, what types of data that would be relevant to collect, and how the framework could support that in a best possible way.

Appendix A

Code

All of the code is available at: https://github.com/Bakkansen/MasterThesis

A.1 Case 1: Follow the Red Dot code

A.1.1 RFduino Host Code

```
1 #include <RFduinoGZLL.h>
2 #include "PLAB_PushButton.h"
3
4 device_t role = HOST;
5
6 const int ledPin = 6;
7 const int buttonPin = 2;
8
9 // Operation mode of the RFduino
10 int mode = 0; // 0 = Start, 1 = GameMode, 2 = Completed
11
12 // PushButton setup
13 PLab_PushButton button(buttonPin);
14
15 const int connectedDevices = 3; // identify the number
     of devices that is connected to the game
16
17 // List of connected devices
18 device_t devList[8] = {DEVICE0, DEVICE1, DEVICE2,
     DEVICE3, DEVICE4, DEVICE5, DEVICE6, DEVICE7};
19 device_t currentDotHolder = HOST;
20
21
```

```
22 void setup() {
23   // put your setup code here, to run once:
24   pinMode(ledPin, OUTPUT);
25   Serial.begin(115200);
26   RFduinoGZLL.begin(role);
27 }
28
29 void loop(){
30   button.update();
31   if (mode == 0) {
32     if (button.pressed()) {
33       mode = 1;
34       Serial.println("Starting game mode");
35       sendNewDot();
36     }
37   } else if (mode == 1) {
38     if (button.pressed()) {
39       resetGame();
40     }
41   }
42
43
44 }
45
46 void RFduinoGZLL_onReceive(device_t device, int rssi,
     char *data, int len) {
47   if (device == currentDotHolder) {
48     String str = data;
49     if (str.startsWith("#BUTTON:P")) {
50       RFduinoGZLL.sendToDevice(currentDotHolder, "#LED:
           OFF");
51       sendNewDot();
52     }
53   }
54 }
55
56 void resetGame() {
57   RFduinoGZLL.sendToDevice(currentDotHolder, "#LED:OFF")
       ;
58   currentDotHolder = HOST;
```

```
59   Serial.println("Game Reset, reconnect devices...");
60   mode = 0;
61 }
62
63
64
65 void sendNewDot() {
66   long randNumber = random(connectedDevices);
67   if (devList[randNumber] == currentDotHolder) {
68     sendNewDot();
69   } else {
70     currentDotHolder = devList[randNumber];
71     RFduinoGZLL.sendToDevice(currentDotHolder, "#LED:ON"
         );
72     Serial.print("Sent dot to: DEVICE");
73     Serial.println(randNumber);
74   }
75 }
```

A.1.2 RFduino Device code - used for all cases

```
1 #include <RFduinoGZLL.h>
2 #include "PLAB_PushButton.h"
3
4 device_t role = DEVICE2; // Set specific device ID here
5
6 const int ledPin = 6;
7 const int buttonPin = 2;
8
9 unsigned long pollTime = 0;
10
11 // PushButton setup
12 PLab_PushButton button(buttonPin);
13
14
15 boolean pollInterval(unsigned long &since, unsigned long
       interval) {
16   unsigned long currentmillis = millis();
17   if (currentmillis - since >= interval) {
18     since = currentmillis;
19     return true;
```

```
20   }
21   return false;
22 }
23
24
25 void setup() {
26   // put your setup code here, to run once:
27   pinMode(ledPin, OUTPUT);
28   Serial.begin(9600);
29   RFduinoGZLL.begin(role);
30 }
31
32 void loop(){
33   button.update();
34   if (button.pressed()){
35     RFduinoGZLL.sendToHost("#BUTTON:P");
36   }
37   if (pollInterval(pollTime, 100)) {
38     RFduinoGZLL.sendToHost(NULL, 0);
39   }
40 }
41
42 void RFduinoGZLL_onReceive(device_t device, int rssi,
     char *data, int len) {
43   String str = data;
44   if (str.startsWith("#LED:ON")) {
45     digitalWrite(ledPin, HIGH);
46   }
47   if (str.startsWith("#LED:OFF")) {
48     digitalWrite(ledPin, LOW);
49   }
50 }
```

A.2 Case 2: Follow the Red Dot code

RFduino Host Code

```
1 #include <RFduinoGZLL.h>
2
3 device_t role = HOST;
```

```
4
5
6 // Serial parsing variables
7 const char startChar = '<';
8 const char stopChar = '>';
9
10 unsigned int index_pos = 0;
11
12 const unsigned int MAX_DEV_INPUT = 5;
13 const unsigned int MAX_MSG_INPUT = 30;
14
15 char msg[MAX_MSG_INPUT];    // Array to keep the
     component name
16
17 enum state_t {
18   None,
19   Msg
20 };
21 extern  state_t state = None;
22
23 // FtRD variables
24 const int connectedDevices = 3; // identify the number
     of devices that is connected to the Host
25
26 int mode = 0; // 0 = Starter host (must receive button
     press from DEV0 to start), 1 = Other hosts (all others
      need to have set to 1)
27
28 device_t devList[8] = {DEVICE0, DEVICE1, DEVICE2,
     DEVICE3, DEVICE4, DEVICE5, DEVICE6, DEVICE7};
29 device_t currentDotHolder = HOST;
30
31
32 void setup() {
33   state = None;
34   Serial.begin(115200);
35   RFduinoGZLL.begin(role);
36 }
37
38 void RFduinoGZLL_onReceive(device_t device, int rssi,
```

```
     char *data, int len) {
39   if (device == DEVICE0 && mode == 0) {
40     String str = data;
41     if (str.startsWith("#BUTTON:P")) {
42       mode = 1;
43       delegateBeacon();
44     }
45   } else if (device == currentDotHolder) {
46     String str = data;
47     if (str.startsWith("#BUTTON:P")) {
48       RFduinoGZLL.sendToDevice(currentDotHolder, "#LED:
           OFF");
49       delegateBeacon();
50     }
51   }
52 }
53
54 void delegateBeacon() {
55   long randNumber = random(connectedDevices*2);
56   if (randNumber < connectedDevices || currentDotHolder
       == HOST) {
57       sendNewDot();
58   } else {
59     currentDotHolder = HOST;
60     Serial.println("#Beacon");
61   }
62 }
63
64 void sendNewDot() {
65   long randNumber = random(connectedDevices);
66   if (devList[randNumber] == currentDotHolder) {
67     sendNewDot();
68   } else {
69     currentDotHolder = devList[randNumber];
70     RFduinoGZLL.sendToDevice(currentDotHolder, "#LED:ON"
         );
71   }
72 }
73
74
```

```
// Serial Reading Code


void handleIncomingData(const byte inByte) {
  switch (state) {
    case Msg:
      if (index_pos < (MAX_MSG_INPUT - 1)) {
        msg[index_pos++] = inByte;
      }
      break;
  }
}

void handleStopCharReceived() {
  String str = msg;
  Serial.println(str);
  if (str.startsWith("#Beacon") && currentDotHolder ==
     HOST && mode == 1) {
    sendNewDot();
  }
  state = None;
}

void processIncomingByte(const byte inByte) {

  switch (inByte) {

    case startChar:
      state = Msg;
      index_pos = 0;
      break;

    case stopChar:
      handleStopCharReceived();
      break;

    default:
      handleIncomingData(inByte);
      break;
  }
```

```
114 }
115
116 void loop() {
117   while (Serial.available() > 0) {
118     processIncomingByte(Serial.read());
119   }
120 }
```

Raspberry Pi Gateway Python Code

```
1  import paho.mqtt.client as mqtt
2  import paho.mqtt.publish as publish
3  import time
4  import serial
5
6  #RPI Information - Switch these for the other RPI
       Gateway: RPIID = otherRPID and otehrRPID = RPIID
7  RPIID = "RPI1"
8  otherRPIID = "RPI2"
9
10 #MQTT Information
11 #Topic stucture: FtRD/RPIID
12 mqttBrokerAddress = "YOUR_BROKER_ADDRESS" # Add your
       broker address here
13 subTopic = "FtRD/" + otherRPIID
14 pubTopic = "FtRD/" + RPIID
15
16
17 def on_connect(client, userdata, flags, rc):
18     print("Connected with result code " + str(rc))
19     client.subscribe(listenTopic)
20
21
22 def on_message(client, userdata, msg):
23   try:
24       print("Received data from " + msg.topic + ": " +
           msg.payload)
25       HandleMQTTMessage(msg)
26   except (UnicodeDecodeError):
27     print("Received faulty msg")
28
```

```
29
30 def HandleMQTTMessage(msg):
31   topicSplit = msg.topic.split('/')
32   if (len(topicSplit) > 1):
33     if (topicSplit[1] == otherRPIID):
34       text = msg.payload.rstrip()
35       if (text.startsWith("#Beacon")):
36         SendSerialMsg("#Beacon")
37
38
39 def HandleSerialMessage(msg):
40   msg.rstrip('\r\n')
41   if (msg.startsWith("#Beacon")):
42     PublishMQTTMsg("#Beacon");
43
44
45 def SendSerialMsg(serialMsg):
46   sendMsg = "<" + serialMsg + ">"
47   sendMsg = sendMsg.encode()
48   ser.write(sendMsg)
49   print("Sent serial message: [" + sendMsg + "]")
50
51
52 def PublishMQTTMsg(payload):
53   print("Published mqtt message: [" + pubTopic + "]: " +
       "[" + payload + "]")
54   publish.single(pubTopic, payload, hostname=
       mqttBrokerAddress)
55
56
57 def CleanUp():
58   print("Ending and cleaning up")
59   ser.close()
60   client.disconnect()
61
62
63 try:
64   #Serial Information
65   print("Connecting Serial port")
66   ser = serial.Serial(
```

```
67      port='/dev/ttyUSB0',
68      baudrate = 115200,
69      parity=serial.PARITY_NONE,
70      stopbits=serial.STOPBITS_ONE,
71      bytesize=serial.EIGHTBITS,
72      timeout=1
73    )
74
75  except:
76    print("Failed to connect serial")
77    raise SystemExit
78
79  try:
80    client = mqtt.Client()
81    client.on_connect = on_connect
82    client.on_message = on_message
83
84    client.connect(mqttBrokerAddress, 1883, 60)
85
86    client.loop_start()
87    print("MQTT client connected!")
88
89
90    while True:
91      read_serial = ser.readline()
92      HandleSerialMessage(read_serial.rstrip())
93
94  except (KeyboardInterrupt, SystemExit):
95    print("Interrupt received")
96    CleanUp()
97
98  except (RuntimeError):
99    print("Run-Time Error")
100   CleanUp()
```

A.3 Case 3: Follow the Red Dot code

RFDuino Host Code

```
#include <RFduinoGZLL.h>

device_t role = HOST;

const char startChar = '<';
const char stopChar = '>';
const char delimiter = ':';

unsigned int index_pos = 0;

const unsigned int MAX_DEV_INPUT = 5;
const unsigned int MAX_MSG_INPUT = 30;

char dev[MAX_DEV_INPUT];       // Array to keep the
    device info received
char msg[MAX_MSG_INPUT];     // Array to keep the
    component name

enum state_t {
  None,
  Devi,
  Msg
};
extern  state_t state = None;

void setup() {
  state = None;
  Serial.begin(115200);
  RFduinoGZLL.begin(role);
}

void RFduinoGZLL_onReceive(device_t device, int rssi,
    char *data, int len) {
  String str = data;
  if (len > 1) {
    String sender = "Action:" + getDeviceFromDevice_t(
        device) + str;
    Serial.println(sender);
  }
}
```

```
37
38 void handleDelimiter() {
39   if (state == Devi) {
40     index_pos = 0;
41     state = Msg;
42   } else if (state == Msg) {
43     if (index_pos < (MAX_MSG_INPUT - 1)) {
44       msg[index_pos++] = ':';
45     }
46   }
47 }
48
49 void handleIncomingData(const byte inByte) {
50   switch (state) {
51     case Devi:
52       if (index_pos < (MAX_DEV_INPUT - 1)) {
53         dev[index_pos++] = inByte;
54       }
55       break;
56
57     case Msg:
58       if (index_pos < (MAX_MSG_INPUT - 1)) {
59         msg[index_pos++] = inByte;
60       }
61       break;
62
63   }
64 }
65
66 void handleStopCharReceived() {
67   device_t receiver = getDevice();
68
69   String m = msg;
70   if (m.startsWith("REQ")) {
71     String d = dev;
72     String sender = "State:" + d + "LED:OFF:BUTTON:
          Lifted";
73     Serial.println(sender);
74   } else if (receiver != HOST) {
75     RFduinoGZLL.sendToDevice(receiver, msg);
```

```
76   }
77   state = None;
78 }
79
80 device_t getDevice() {
81   device_t receiver = DEVICE0;
82   int deviceInt = -1;
83   if (isDigit(dev[3])) {
84     deviceInt = dev[3] - '0';
85   }
86
87   switch (deviceInt)  {
88     case 0:
89       receiver = DEVICE0;
90       break;
91
92     case 1:
93       receiver = DEVICE1;
94       break;
95
96     case 2:
97       receiver = DEVICE2;
98       break;
99
100    case 3:
101      receiver = DEVICE3;
102      break;
103
104    case 4:
105      receiver = DEVICE4;
106      break;
107
108    case 5:
109      receiver = DEVICE5;
110      break;
111
112    case 6:
113      receiver = DEVICE6;
114      break;
115
```

```
116     case 7:
117       receiver = DEVICE7;
118       break;
119   }
120   return receiver;
121 }
122
123 String getDeviceFromDevice_t(device_t dev) {
124   String receiver = "DEV0";
125
126   switch (dev)  {
127     case 0:
128       receiver = "DEV0";
129       break;
130
131     case 1:
132       receiver = "DEV1";
133       break;
134
135     case 2:
136       receiver = "DEV2";
137       break;
138
139     case 3:
140       receiver = "DEV3";
141       break;
142
143     case 4:
144       receiver = "DEV4";
145       break;
146
147     case 5:
148       receiver = "DEV5";
149       break;
150
151     case 6:
152       receiver = "DEV6";
153       break;
154
155     case 7:
```

```
156       receiver = "DEV7";
157       break;
158   }
159   return receiver;
160 }
161
162
163 void processIncomingByte(const byte inByte) {
164
165   switch (inByte) {
166
167     case startChar:
168       state = Devi;
169       index_pos = 0;
170       break;
171
172     case stopChar:
173       handleStopCharReceived();
174       break;
175
176     case delimiter:
177       handleDelimiter();
178       break;
179
180     default:
181       handleIncomingData(inByte);
182       break;
183   }
184 }
185
186 void loop() {
187   while (Serial.available() > 0) {
188     processIncomingByte(Serial.read());
189   }
190 }
```

Raspberry Pi Gateway Python Code

```
1 import paho.mqtt.client as mqtt
2 import paho.mqtt.publish as publish
3 import time
```

```
4 import serial
5 import json
6
7
8 devList = { "DEV0": [ "LED", "OFF", "BUTTON", "LIFTED"],
      "DEV1": [ "LED", "OFF", "BUTTON", "LIFTED"], "DEV2":
     [ "LED", "OFF", "BUTTON", "LIFTED"]}
9
10 interfaceJSON = {
11   "components" : [
12     {"componentID":"LED", "componentType":"LED", "states
         ":["ON", "OFF", "DISABLED"]},
13     {"componentID":"BUTTON", "componentType":"BUTTON", "
         states":["LIFTED", "PRESSED", "DISABLED"]}
14   ]
15 }
16
17 #MQTT Information
18 #Topic stucture: Pettjo/Type/Environment/RPI/Dev ex:
     Pettjo/Action/House/RPI1/Dev0
19 mqttBrokerAddress = "YOUR_BROKER_ADDRESS" # Add your
     broker address here
20 listenTopic = "FtRD/#"
21
22
23 #RPI Information
24 rpiID = "RPI1"
25 Environment = "House"
26
27
28 # MQTT handling
29
30 def on_connect(client, userdata, flags, rc):
31     print("Connected with result code " + str(rc))
32     client.subscribe(listenTopic)
33
34
35 def on_message(client, userdata, msg):
36   try:
```

```
37        print("Received data from " + msg.topic + ": " +
              msg.payload)
38        HandleMQTTMessage(msg)
39    except (UnicodeDecodeError):
40      print("Received faulty msg")
41
42
43  def HandleMQTTMessage(msg):
44    topicSplit = msg.topic.split('/')
45    msgType = "#"
46    msgEnvironment = "#"
47    msgRPI = "#"
48    msgDev = "#"
49    if (len(topicSplit) > 1):
50      msgType = topicSplit[1]
51    if (len(topicSplit) > 2):
52      msgEnvironment = topicSplit[2]
53    if (len(topicSplit) > 3):
54      msgRPI = topicSplit[3]
55    if (len(topicSplit) > 4):
56      msgDev = topicSplit[4].upper()
57
58    if (msgRPI != rpiID and msgRPI != "#"):
59      return
60
61    if (msgType == "Event"):
62      if (msgDev == "#"):
63        for dev in devList.keys():
64          EventMsgRcvd(dev, msg.payload.rstrip())
65      elif (msgDev in devList.keys()):
66        EventMsgRcvd(msgDev, msg.payload.rstrip())
67
68    elif (msgType == "StateReq"):
69      if (msgDev == "#"):
70        for dev in devList.keys():
71          StateReqMsgRcvd(dev, msg.payload.rstrip())
72      elif (msgDev in devList.keys()):
73        StateReqMsgRcvd(msgDev, msg.payload.rstrip())
74
75    elif (msgType == "InterfaceReq"):
```

```
76      if (msgDev == "#"):
77        for dev in devList.keys():
78          InterfaceReqMsgRcvd(dev, msg.payload.rstrip())
79      elif (msgDev in devList.keys()):
80        InterfaceReqMsgRcvd(msgDev, msg.payload.rstrip())
81
82
83  def EventMsgRcvd(dev, msg):
84    # MQTT msg format: "Comp:CompState"
85    print("Recvd MQTTmsg: [" + dev + "] : [" + msg + "]")
86    parsedMsg = msg.split(':')
87    # Checks if device has component
88    if (parsedMsg[0].upper() in devList[dev]):
89      sendString = dev + ":#" + parsedMsg[0] + ":" +
          parsedMsg[1]
90      SendSerialMsg(sendString)
91
92  def StateReqMsgRcvd(dev, msg):
93    print("Recvd MQTTmsg: [" + dev + "] : [" + msg + "]")
94    if (msg == "REQ"):
95      sendString = dev + ":" + msg;
96      SendSerialMsg(sendString)
97
98
99  def InterfaceReqMsgRcvd(dev, msg):
100   print("Recvd MQTTmsg: [" + dev + "] : [" + msg + "]")
101   if (msg == "REQ"):
102     PublishMQTTMsg("InterFace", json.dumps(interfaceJSON
          ), dev)
103
104
105 # Serial handling
106
107 def ActionSerialRcvd(dev, payload):
108   sendString = payload[0] + ":" payload[1]
109   PublishMQTTMsg("Action", sendString, dev)
110
111 def StateSerialRcvd(dev, payload):
112   stateJSON = {
113     "connected":1,
```

```
114     "components": [
115         {"componentID":payload[0], "currentState":
               payload[1]},
116         {"componentID":payload[2], "currentState":
               payload[3]}
117         ]
118     }
119     PublishMQTTMsg("State", json.dumps(stateJSON), dev)
120
121
122 def HandleSerialMessage(msg):
123   parsedMsg = msg.split(':')
124   if (len(parsedMsg) != 3):
125     return
126
127   print("Received Serial message: [" + msg + "]")
128   topic = parsedMsg[0]
129   dev = parsedMsg[1].upper()
130   payload = parsedMsg[2:]
131
132   if (topic == "Action"):
133     ActionSerialRcvd(dev, payload)
134
135   elif (topic == "State"):
136     StateSerialRcvd(dev, payload)
137
138
139 def SendSerialMsg(serialMsg):
140   #print("Sent serial message: [" + serialMsg + "]")
141   sendMsg = "<" + serialMsg + ">"
142   sendMsg = sendMsg.encode()
143   ser.write(sendMsg)
144   print("Sent serial message: [" + sendMsg + "]")
145
146
147 def PublishMQTTMsg(msgType, payload, dev):
148   if (dev in devList.keys()):
149     topic = "FtRD/" + msgType + "/" + Environment + "/"
          + rpiID + "/" + dev
```

```
150     print("Published mqtt message: [" + topic + "]: " +
          "[" + payload + "]")
151     publish.single(topic, payload, hostname=
          mqttBrokerAddress)
152
153
154
155
156 # System setup
157
158 def CleanUp():
159   print("Ending and cleaning up")
160   ser.close()
161   client.disconnect()
162
163 try:
164   #Serial Information
165   print("Connecting Serial port")
166   ser = serial.Serial(
167     port='/dev/ttyUSB0',
168     baudrate = 9600,
169     parity=serial.PARITY_NONE,
170     stopbits=serial.STOPBITS_ONE,
171     bytesize=serial.EIGHTBITS,
172     timeout=1
173   )
174
175 except:
176   print("Failed to connect serial")
177   raise SystemExit
178
179 try:
180   client = mqtt.Client()
181   client.on_connect = on_connect
182   client.on_message = on_message
183
184   client.connect(mqttBrokerAddress, 1883, 60)
185
186   client.loop_start()
187   print("MQTT client connected!")
```

```
188
189
190   while True:
191     read_serial = ser.readline()
192     HandleSerialMessage(read_serial.rstrip('\r\n'))
193
194 except (KeyboardInterrupt, SystemExit):
195   print("Interrupt received")
196   CleanUp()
197
198 except (RuntimeError):
199   print("Run-Time Error")
200   CleanUp()
```

Unity code: IoTDevice.cs

```
1  using IoTPlatform.IoTComponents;
2  using IoTPlatform.Events;
3  using System.Collections.Generic;
4  using UnityEngine;
5  using IoTPlatform.Master.Utility;
6
7  public class IoTDevice : MonoBehaviour
8  {
9
10
11     public string Environment;
12     public string RPI;
13     public string ID;
14     private List<IoTComponent> components = new List<IoTComponent>()
           ;
15
16     // Use this for initialization
17     void Start()
18     {
19         IoTComponent[] comps = GetComponents<IoTComponent>();
20         foreach (IoTComponent i in comps)
21         {
22             components.Add(i);
23             if (i.ComponentID == "LED")
24             {
25                 i.SetCurrentState("OFF");
26             }
27             if (i.ComponentID == "BUTTON")
28             {
```

```
29                 i.SetCurrentState("Lifted");
30             }
31         }
32     }
33
34     private void OnEnable()
35     {
36         IoTEventHandler.Instance.AddListener<ActionEvent>(
               ActionEventReceived);
37         IoTEventHandler.Instance.AddListener<InterfaceEvent>(
               InterfaceEventReceived);
38         IoTEventHandler.Instance.AddListener<StateEvent>(
               StateEventRecieved);
39     }
40
41     private void OnDisable()
42     {
43         IoTEventHandler.Instance.RemoveListener<ActionEvent>(
               ActionEventReceived);
44         IoTEventHandler.Instance.RemoveListener<InterfaceEvent>(
               InterfaceEventReceived);
45         IoTEventHandler.Instance.RemoveListener<StateEvent>(
               StateEventRecieved);
46     }
47
48     private void ActionEventReceived(ActionEvent e)
49     {
50         if (IsItMe(e.Environment, e.RPI, e.Device))
51         {
52             string[] msg = e.Msg.Split(':');
53             if (msg.Length != 2)
54             {
55                 Debug.Log("Invalid msg format: [" + msg + "]");
56                 return;
57             }
58             if (components.Count == 0)
59             {
60                 Debug.Log("No compoents attached to Gameobject: [" +
                       gameObject.name + "]");
61                 return;
62             }
63             foreach (IoTComponent comp in components)
64             {
65                 if (comp.ComponentID == msg[0])
66                 {
67                     comp.SetCurrentState(msg[1]);
68                     Debug.Log("Set state: [" + ID + "]" + ":" + msg
                           [1]);
```

```
69                      return;
70                  }
71              }
72          }
73      }
74
75
76      private void StateEventRecieved(StateEvent e)
77      {
78          if (IsItMe(e.Environment, e.RPI, e.Device))
79          {
80              foreach (ComponentState c in e.components)
81              {
82                  foreach (IoTComponent comp in components)
83                  {
84                      if (comp.ComponentID == c.componentID)
85                      {
86                          comp.SetCurrentState(c.currentState);
87                      }
88                  }
89              }
90          }
91      }
92
93
94      private void InterfaceEventReceived(InterfaceEvent e)
95      {
96          if (IsItMe(e.Environment, e.RPI, e.Device))
97          {
98              if (components.Count > 0)
99              {
100                 foreach (IoTComponent c in components)
101                 {
102                     Destroy(c);
103                 }
104             }
105             foreach (ComponentInterface c in e.components)
106             {
107                 if (c.componentID == "LED")
108                 {
109                     gameObject.AddComponent<LED>();
110                 }
111                 if (c.componentID == "BUTTON")
112                 {
113                     gameObject.AddComponent<IoTPlatform.
                            IoTComponents.Button>();
114                 }
115             }
```

```
116             MQTTHandler.Instance.MqttPublishMsg(MQTTHandler.
                    MQTTMsgType.State_Req, MQTTHandler.
                    MQTTMsgEnvironment.House, e.RPI, e.Device, "REQ");
117
118
119         }
120     }
121
122     private bool IsItMe(string environment, string rpi, string id)
123     {
124         return (this.Environment == environment && this.RPI == rpi
                && (this.ID == id || id == "#"));
125     }
126
127
128 }
```

Unity code: MQTTHandler.cs

```
1  using System.Collections;
2  using System.Net;
3  using UnityEngine;
4  using uPLibrary.Networking.M2Mqtt;
5  using uPLibrary.Networking.M2Mqtt.Messages;
6  using uPLibrary.Networking.M2Mqtt.Utility;
7  using uPLibrary.Networking.M2Mqtt.Exceptions;
8  using IoTPlatform.Events;
9  using IoTPlatform.Master.Utility;
10
11 using System;
12
13
14 public class MQTTHandler : MonoBehaviour
15 {
16
17     // Singleton
18     static MQTTHandler instanceInternal = null;
19     public static MQTTHandler Instance
20     {
21         get { return instanceInternal; }
22     }
23
24     public delegate void OnConnectRequest();
25
26     public OnConnectRequest onConnectRequest;
27
28
29     public bool AutoConnectNewDevices = true;
```

```
30
31      private MqttClient client;
32      private string mqttBrokerAddress = "mqtt.idi.ntnu.no";
33      private int mqttBrokerPort = 1883;
34      private string eventSubTopic = "FtRD/Event/House/#";
35      private string actionSubTopic = "FtRD/Action/House/#";
36      private string stateReqSubTopic = "FtRD/State_Req/House/#";
37      private string stateSubTopic = "FtRD/State/House/#";
38      private string interfaceReqSubTopic = "FtRD/Interface_Req/House
            /#";
39      private string interfaceSubTopic = "FtRD/Interface/House/#";
40      private string connectReqSubTopic = "FtRD/ConnectReq/House/#";
41      private string connectSubTopic = "FtRD/Connect/House/#";
42      private string AllTopicLabel = "#";
43      private string WildcardLabel = "+";
44
45      public enum MQTTMsgType { Event, Action, State_Req, State,
            Interface_Req, Interface, ConnectReq, Connect, ALL };
46      public enum MQTTMsgEnvironment { House, ALL };
47
48
49      private void Awake()
50      {
51          if (instanceInternal != null && instanceInternal != this)
52          {
53              Destroy(gameObject);
54          }
55          instanceInternal = this;
56          DontDestroyOnLoad(gameObject);
57      }
58
59      void Start()
60      {
61          client = new MqttClient(mqttBrokerAddress, mqttBrokerPort,
                false, null);
62
63          client.MqttMsgPublishReceived += MqttMsgPublishReceived;
64
65          string clientId = Guid.NewGuid().ToString();
66          client.Connect(clientId);
67
68          client.Subscribe(new string[] { actionSubTopic }, new byte[]
                { MqttMsgBase.QOS_LEVEL_EXACTLY_ONCE });
69          Debug.Log("Subscribed to: [" + actionSubTopic + "]");
70
71          client.Subscribe(new string[] { stateSubTopic }, new byte[]
                { MqttMsgBase.QOS_LEVEL_EXACTLY_ONCE });
72          Debug.Log("Subscribed to: [" + stateSubTopic + "]");
```

```
73
74          client.Subscribe(new string[] { interfaceSubTopic }, new
                byte[] { MqttMsgBase.QOS_LEVEL_EXACTLY_ONCE });
75          Debug.Log("Subscribed to: [" + interfaceSubTopic + "]");
76
77          client.Subscribe(new string[] { connectReqSubTopic }, new
                byte[] { MqttMsgBase.QOS_LEVEL_EXACTLY_ONCE });
78          Debug.Log("Subscribed to: [" + connectReqSubTopic + "]");
79
80          Debug.Log("Start Called " + client.ToString());
81      }
82
83      internal void MqttPublishMsg(MQTTMsgType action,
            MQTTMsgEnvironment house, object rPI, object iD, string v)
84      {
85          throw new NotImplementedException();
86      }
87
88      void MqttMsgPublishReceived(object sender,
            MqttMsgPublishEventArgs e)
89      {
90          string[] topics = e.Topic.Split('/');
91          if (!Enum.IsDefined(typeof(MQTTMsgType), topics[1]))
92          {
93              Debug.Log("Received message from different topic: " + e.
                    Topic);
94              return;
95          }
96
97          MQTTMsgType topic = (MQTTMsgType)Enum.Parse(typeof(
                MQTTMsgType), topics[1]);
98          switch (topic)
99          {
100             case MQTTMsgType.Event:
101                 ActionPerformed(e);
102                 break;
103             case MQTTMsgType.Interface:
104                 InterfaceReceived(e);
105                 break;
106             case MQTTMsgType.State:
107                 StateReceived(e);
108                 break;
109             case MQTTMsgType.ConnectReq:
110                 ConnectReqReceived(e);
111                 break;
112         }
113
114
```

```
115     }
116
117     private void ConnectReqReceived(MqttMsgPublishEventArgs e)
118     {
119         string[] topics = e.Topic.Split('/');
120         string env = "#";
121         string rpi = "#";
122         string dev = "#";
123         if (topics.Length > 2)
124         {
125             env = topics[2];
126         }
127         if (topics.Length > 3)
128         {
129             rpi = topics[3];
130         }
131         if (topics.Length > 4)
132         {
133             dev = topics[4];
134         }
135
136         if (AutoConnectNewDevices)
137         {
138             DevHandler.Instance.AddNewDevice(env, rpi, dev);
139             Debug.Log("Topic to publish to: " + interfaceReqSubTopic
                    + " : ['REQ']");
140             Byte[] bytes = System.Text.Encoding.UTF8.GetBytes("REQ")
                ;
141             client.Publish(interfaceReqSubTopic, bytes);
142         } else
143         {
144             onConnectRequest();
145         }
146     }
147
148     private void StateReceived(MqttMsgPublishEventArgs e)
149     {
150         string[] topics = e.Topic.Split('/');
151         string env = "#";
152         string rpi = "#";
153         string dev = "#";
154         if (topics.Length > 2)
155         {
156             env = topics[2];
157         }
158         if (topics.Length > 3)
159         {
160             rpi = topics[3];
```

```
161         }
162         if (topics.Length > 4)
163         {
164             dev = topics[4];
165         }
166 
167         Debug.Log("[State]: " + env + "/" + rpi + "/" + dev);
168         IoTState iotState = JsonUtility.FromJson<IoTState>(System.
                Text.Encoding.UTF8.GetString(e.Message));
169         IoTEventHandler.Instance.Raise(new StateEvent()
170         {
171             Device = dev,
172             RPI = rpi,
173             Environment = env,
174             isConnected = iotState.connected,
175             components = iotState.components
176         });
177     }
178 
179     private void InterfaceReceived(MqttMsgPublishEventArgs e)
180     {
181         string[] topics = e.Topic.Split('/');
182         string env = "#";
183         string rpi = "#";
184         string dev = "#";
185         if (topics.Length > 2)
186         {
187             env = topics[2];
188         }
189         if (topics.Length > 3)
190         {
191             rpi = topics[3];
192         }
193         if (topics.Length > 4)
194         {
195             dev = topics[4];
196         }
197 
198         Debug.Log("[Interface]: " + env + "/" + rpi + "/" + dev);
199         IoTInterface ioTInterface = JsonUtility.FromJson<
                IoTInterface>(System.Text.Encoding.UTF8.GetString(e.
                Message));
200         IoTEventHandler.Instance.Raise(new InterfaceEvent()
201         {
202             Device = dev,
203             RPI = rpi,
204             Environment = env,
205             components = ioTInterface.components
```

```
206         });
207     }
208
209
210     private void ActionPerformed(MqttMsgPublishEventArgs e)
211     {
212         string[] topics = e.Topic.Split('/');
213         string env = "#";
214         string rpi = "#";
215         string dev = "#";
216         if (topics.Length > 2)
217         {
218             env = topics[2];
219         }
220         if (topics.Length > 3)
221         {
222             rpi = topics[3];
223         }
224         if (topics.Length > 4)
225         {
226             dev = topics[4];
227         }
228         Debug.Log("[Action] " + env + "/" + rpi + "/" + dev + ": " +
                System.Text.Encoding.UTF8.GetString(e.Message));
229
230         IoTEventHandler.Instance.Raise(new ActionEvent
231         {
232             Environment = env,
233             RPI = rpi,
234             Device = dev,
235             Msg = System.Text.Encoding.UTF8.GetString(e.Message)
236         });
237     }
238
239     public void MqttPublishMsg(MQTTMsgType type, MQTTMsgEnvironment
            area, string RPI, string device, string msg)
240     {
241         string topic = "FtRD/";
242         if (type == MQTTMsgType.ALL)
243         {
244             topic += "#";
245         } else
246         {
247             topic += type.ToString() + "/";
248
249             if (area == MQTTMsgEnvironment.ALL)
250             {
251                 topic += "#";
```

```
252             } else
253             {
254                 topic += area.ToString() + "/";
255 
256                 if (RPI == AllTopicLabel)
257                 {
258                     topic += AllTopicLabel;
259                 } else
260                 {
261                     topic += RPI + "/";
262 
263                     if (device == AllTopicLabel)
264                     {
265                         topic += AllTopicLabel;
266                     } else
267                     {
268                         topic += device;
269                     }
270                 }
271             }
272         }
273         Debug.Log("Topic to publish to: " + topic + " : [" + msg + "
                ]");
274         Byte[] bytes = System.Text.Encoding.UTF8.GetBytes(msg);
275         client.Publish(topic, bytes);
276     }
277 
278     private void OnDestroy()
279     {
280         client.Disconnect();
281     }
282 }
```

A.4 MQTT JSON objects

```
1 { State: {
2 "connected" : boolean,
3 "components" : [
4   { "componentID" : "idA", "currentState" : "ON" },
5     { "componentID" : "idB", "currentState" : "LIFTED" }
6     ]
7 }}
```

```
1 { Interface: {
2   "components" : [
```

```
3    { "componentID" : "idA", "componentType" : "LED", "
       states" : [ "ON", "OFF", "DISABLED"] },
4      { "componentID" : "idB", "componentType" : "button",
         "states" : [ "LIFTED", "PRESSED", "DISABLED"] }
5      ]
6  }}
```

Appendix B

Search terms

B.1 Search Engines and libraries

The most commonly used search engines and libraries that were used to explore the literature are included in the list below:

Google Scholar (https://scholar.google.no/)

ResearchGate (https://www.researchgate.net/)

ACM Digital Library (https://dl.acm.org/)

Oria (http://oria.no/)

Semantic Scholar (https://www.semanticscholar.org/)

B.2 Search terms

Some of the search terms that were used during the research is listed below. The terms were also combined at times to create more specific searches. I have also used other search terms, but these are among the ones that provided mos relevant results.

Games

Pervasive games

Serious games

Exergames

Pervasive game architecture

Ubiquitous computing

Internet of Things (IoT)

Cloud

IoT-based pervasive games

Game development

IoT Architecture

IoT Technology

Machine-to-machine communication

Activity promoting games

Active video game

Motion-based games

Bibliography

(2008). Internet of Things in 2020: Roadmap for the future, Version 1.1.

Al-Fuqaha, A., Guizani, M., Mohammadi, M., Aledhari, M., and Ayyash, M. (2015). Internet of things: A survey on enabling technologies, protocols, and applications. *IEEE Communications Surveys & Tutorials*, 17(4):2347–2376.

Alankus, G., Proffitt, R., Kelleher, C., and Engsberg, J. (2011). Stroke therapy through motion-based games: a case study. *ACM Transactions on Accessible Computing (TACCESS)*, 4(1):3.

Ashton, K. (2009). That 'internet of things' thing.

Atzori, L., Iera, A., and Morabito, G. (2010). The internet of things: A survey. *Comput. Netw.*, 54(15):2787–2805.

Babu, S. M., Lakshmi, A. J., and Rao, B. T. (2015). A study on cloud based internet of things: Cloudiot. In *Communication Technologies (GCCT), 2015 Global Conference on*, pages 60–65. IEEE.

Baranowski, T., Buday, R., Thompson, D. I., and Baranowski, J. (2008). Playing for real: video games and stories for health-related behavior change. *American journal of preventive medicine*, 34(1):74–82.

Benford, S., Magerkurth, C., and Ljungstrand, P. (2005). Bridging the physical and digital in pervasive gaming. *Commun. ACM*, 48(3):54–57.

Biswas, A. R. and Giaffreda, R. (2014). Iot and cloud convergence: Opportunities and challenges. *2014 IEEE World Forum on Internet of Things (WF-IoT)*, pages 375–376.

Bonomi, F., Milito, R., Zhu, J., and Addepalli, S. (2012). Fog computing and its role in the internet of things. In *Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing*, MCC '12, pages 13–16, New York, NY, USA. ACM.

Bormann, C., Castellani, A. P., and Shelby, Z. (2012). Coap: An application protocol for billions of tiny internet nodes. *IEEE Internet Computing*, 16(2):62–67.

Brauner, P., Calero Valdez, A., Schroeder, U., and Ziefle, M. (2013). Increase physical fitness and create health awareness through exergames and gamification. In Holzinger, A., Ziefle, M., Hitz, M., and Debevc, M., editors, *Human Factors in Computing and Informatics*, pages 349–362, Berlin, Heidelberg. Springer Berlin Heidelberg.

Broll, G. and Benford, S. (2005). *Seamful Design for Location-Based Mobile Games*, pages 155–166. Springer Berlin Heidelberg, Berlin, Heidelberg.

Brown, G. A., Holoubeck, M., Nylander, B., Watanabe, N., Janulewicz, P., Costello, M., Heelan, K. A., and Abbey, B. (2008). Energy costs of physically active video gaming: wii boxing, wii tennis, and dance dance revolution. *Medicine & Science in Sports & Exercise*, 40(5):S460.

Chamberlin, B. and Gallagher, R. (2008). Using video games to promote physical activity.

Cheok, A. D., Goh, K. H., Liu, W., Farbiz, F., Fong, S. W., Teo, S. L., Li, Y., and Yang, X. (2004). Human pacman: A mobile, wide-area entertainment system based on physical, social, and ubiquitous computing. *Personal Ubiquitous Comput.*, 8(2):71–81.

Cisco (2015). Fog computing and the internet of things: Extend the cloud to where the things are. Technical report.

Csikszentmihalyi, M. (1975). *Beyond Boredom and Anxiety*. The Jossey-Bass behavioral science series. Jossey-Bass Publishers.

Culler, D. and Chakrabarti, S. (2009). 6lowpan: Incorporating ieee 802.15. 4 into the ip architecture. *IPSO Alliance, White paper*.

Dash, S. K., Mohapatra, S., and Pattnaik, P. K. (2010). A survey on applications of wireless sensor network using cloud computing. *International Journal of Computer science & Engineering Technologies (E-ISSN: 2044-6004)*, 1(4):50–55.

de Vries, S. I., Simons, M., and Jongert, T. W. (2008). Energy expenditure of active computer-games. *Medicine & Science in Sports & Exercise*, 40(5):S198.

Dickinson, A., Lochrie, M., and Egglestone, P. (2015). Ukko: Enriching persuasive location based games with environmental sensor data. In *Proceedings of the 2015 Annual Symposium on Computer-Human Interaction in Play*, CHI PLAY '15, pages 493–498, New York, NY, USA. ACM.

DiRico, E., Davis, K. A., Washington, C., Galvanin, E., Otto, R. M., and Wygand, J. W. (2009). The metabolic cost of an interactive video game: 559. *Medicine & Science in Sports & Exercise*, 41(5):11.

Dynastream Innovations Inc. Ant message protocol and usage.

Epstein, L. H., Beecher, M. D., Graf, J. L., and Roemmich, J. N. (2007). Choice of interactive dance and bicycle games in overweight and nonoverweight youth. *Annals of Behavioral Medicine*, 33(2):124–131.

Esposito, C., Russo, S., and Di Crescenzo, D. (2008). Performance assessment of omg compliant data distribution middleware. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium On*, pages 1–8. IEEE.

Fouladi, B. and Ghanoun, S. (2013). Security evaluation of the z-wave wireless protocol. *Black hat USA*, 24:1–2.

Gao, Z. and Chen, S. (2014). Are field-based exergames useful in preventing childhood obesity? a systematic review. *Obesity reviews*, 15(8):676–691.

Gershenfeld, N., Krikorian, R., and Cohen, D. (2004). The Internet of Things. *Scientific American*, 291(4):76–81.

Glaessgen, E. and Stargel, D. (2012). The digital twin paradigm for future nasa and us air force vehicles. In *53rd AIAA/ASME/ASCE/AHS/ASC Structures, Structural Dynamics and Materials Conference 20th AIAA/ASME/AHS Adaptive Structures Conference 14th AIAA*, page 1818.

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

Graves, L., Stratton, G., Ridgers, N. D., and Cable, N. T. (2007). Comparison of energy expenditure in adolescents when playing new generation and sedentary computer games: cross sectional study. *BMJ*, 335(7633):1282–1284.

Grieves, M. and Vickers, J. (2017). Digital twin: Mitigating unpredictable, undesirable emergent behavior in complex systems. In *Transdisciplinary Perspectives on Complex Systems*, pages 85–113. Springer.

Guo, B., Fujimura, R., Zhang, D., and Imai, M. (2012). Design-in-play: improving the variability of indoor pervasive games. *Multimedia Tools and Applications*, 59(1):259–277.

Huitzinga, J. (1944). *Homo Ludens: A study of the play element in culture*. Routledge and Kegan Paul.

Jegers, K. (2009). *Pervasive GameFlow: Identifying and exploring the mechanisms of player enjoyment in pervasive games*. PhD thesis, Umeå Universitet, Inst för Informatik.

Kasapakis, V. and Gavalas, D. (2015). Pervasive gaming: Status, trends and design principles. *Journal of Network and Computer Applications*, 55:213–236.

Kasapakis, V., Gavalas, D., and Bubaris, N. (2013). Addressing openness and portability in outdoor pervasive role-playing games. In *Communications and Information Technology (ICCIT), 2013 Third International Conference on*, pages 93–97. IEEE.

Kasapakis, V., Gavalas, D., and Chatzidimitris, T. (2015). *Evaluation of Pervasive Games: Recruitment of Qualified Participants Through Preparatory Game Phases*, pages 118–124. Springer International Publishing, Cham.

Khssibi, S., Idoudi, H., Van Den Bossche, A., Val, T., and Saidane, L. A. (2013). Presentation and analysis of a new technology for low-power wireless sensor network. *International Journal of Digital Information and Wireless Communications*, 3(1):pp–75.

Lanningham-Foster, L., Jensen, T. B., Foster, R. C., Redmond, A. B., Walker, B. A., Heinz, D., and Levine, J. A. (2006). Energy expenditure of sedentary screen time compared with active screen time for children. *Pediatrics*, 118(6):e1831–e1835.

Lee, J.-S., Su, Y.-W., and Shen, C.-C. (2007). A comparative study of wireless protocols: Bluetooth, uwb, zigbee, and wi-fi. In *Industrial Electronics Society, 2007. IECON 2007. 33rd Annual Conference of the IEEE*, pages 46–51. Ieee.

Lee, S., Kim, W., Park, T., and Peng, W. (2017). The psychological effects of playing exergames: A systematic review. *Cyberpsychology, Behavior, and Social Networking*, 20(9):513–532.

Luzuriaga, J. E., Perez, M., Boronat, P., Cano, J. C., Calafate, C., and Manzoni, P. (2015). A comparative evaluation of amqp and mqtt protocols over unstable and mobile networks. In *Consumer Communications and Networking Conference (CCNC), 2015 12th Annual IEEE*, pages 931–936. IEEE.

Lwin, M. O. and Malik, S. (2012). The efficacy of exergames-incorporated physical education lessons in influencing drivers of physical activity: a comparison of children and pre-adolescents. *Psychology of Sport and Exercise*, 13(6):756–760.

MacDowell, A. and Endler, M. (2015). *Internet of Things Based Multiplayer Pervasive Games: An Architectural Analysis*, pages 125–138. Springer International Publishing, Cham.

McCarthy, J. (1980). Circumscription—a form of non-monotonic reasoning. *Artificial Intelligence*, 13:27–39.

Mehmood, N. Q. and Culmone, R. (2015). An ant+ protocol based health care system. In *Advanced Information Networking and Applications Workshops (WAINA), 2015 IEEE 29th International Conference on*, pages 193–198. IEEE.

Montola, M. (2005). Exploring the edge of the magic circle: Defining pervasive games. In *CD-ROM Proceedings of Digital Arts and Culture. Copenhagen*, pages 1–3.

Nordic Semiconductor. Things you should know about Bluetooth range. Accessed: 15. May 2017.

Oates, B. J. (2005). *Researching information systems and computing*. Sage.

Oh, Y. and Yang, S. (2010). Defining exergames & exergaming. *Proceedings of Meaningful Play*, pages 1–17.

Olli, S. (2002). All the world&#39s a botfighter stage: Notes on location-based multiuser gaming. In *Computer Games and Digital Cultures Conference Proceedings*. Tampere University Press.

Park, T., Hwang, I., Lee, U., Lee, S. I., Yoo, C., Lee, Y., Jang, H., Choe, S. P., Park, S., and Song, J. (2012). Exerlink: enabling pervasive social exergames with heterogeneous exercise devices. In *Proceedings of the 10th international conference on Mobile systems, applications, and services*, pages 15–28. ACM.

Paw, M. J. C. A., Jacobs, W. M., Vaessen, E. P., Titze, S., and van Mechelen, W. (2008). The motivation of children to play an active video game. *Journal of Science and Medicine in Sport*, 11(2):163–166.

Rao, B. P., Saluia, P., Sharma, N., Mittal, A., and Sharma, S. V. (2012). Cloud computing for internet of things & sensing based applications. In *Sensing Technology (ICST), 2012 Sixth International Conference on*, pages 374–380. IEEE.

Reiter, G. (2014). Wireless connectivity for the internet of things. *Europe*, 433:868MHz.

RFID Journal. Study Forecasts 350 Percent Rise in IoT in Retail by 2021. Accessed: 27. May 2017.

Saint-Andre, P. (2011). Extensible messaging and presence protocol (xmpp): Core.

Sakamura, K. (2006). Challenges in the age of ubiquitous computing: A case study of t-engine, an open development platform for embedded systems. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 713–720, New York, NY, USA. ACM.

Scalagent. Benchmark MQTT servers.

Semiconductor, N. (2017). Gazell link layer user guide.

Sinclair, J., Hingston, P., and Masek, M. (2007). Considerations for the design of exergames. In *Proceedings of the 5th international conference on Computer graphics and interactive techniques in Australia and Southeast Asia*, pages 289–295. ACM.

Sreekumar, A., Cheok, A. D., Thang, L. N., and Lei, C. (2006). Capture the flag: Mixed-reality social gaming with smart phones. *IEEE Pervasive Computing*, 5:62–69.

Staiano, A. E. and Calvert, S. L. (2011). Exergames for physical education courses: Physical, social, and cognitive benefits. *Child development perspectives*, 5(2):93–98.

Sterling, B. (2005). *Shaping Things*. The MIT Press.

Toma, I., Simperl, E., and Hench, G. (2009). A joint roadmap for semantic technologies and the internet of things.

Tomporowski, P. D., Lambourne, K., and Okumura, M. S. (2011). Physical activity interventions and children's mental function: an introduction and overview. *Preventive medicine*, 52:S3–S9.

Tuegel, E. J., Ingraffea, A. R., Eason, T. G., and Spottswood, S. M. (2011). Reengineering aircraft structural life prediction using a digital twin. *International Journal of Aerospace Engineering*, 2011.

Union, I. T. (2012). Overview of the internet of things. *ITU-T Y.2060*.

Uzor, S. and Baillie, L. (2014). Investigating the long-term use of exergames in the home with elderly fallers. In *Proceedings of the 32Nd Annual ACM Conference on Human Factors in Computing Systems*, CHI '14, pages 2813–2822, New York, NY, USA. ACM.

Vaishnavi, V. and Kuechler, W. (2004). Design research in information systems.

Warburton, D. E., Bredin, S. S., Horita, L. T., Zbogar, D., Scott, J. M., Esch, B. T., and Rhodes, R. E. (2007). The health benefits of interactive video game exercise. *Applied Physiology, Nutrition, and Metabolism*, 32(4):655–663.

Wiberg, M. and Jegers, K. (2006). Pervasive gaming in the everyday world. *IEEE Pervasive Computing*, 5:78–85.

Zaslavsky, A., Perera, C., and Georgakopoulos, D. (2013). Sensing as a service and big data. *arXiv preprint arXiv:1301.0159*.