Title of the project

Subtitle

Project Report

Group: SW805F20

Aalborg University
Department of Computer Science
Selma Lagerlöfs Vej 300
9220 Aalborg East, DK

Department of Computer Science
Aalborg University
Selma Lagerlöfs Vej 300
9220 Aalborg East, DK
www.cs.aau.dk

Title:
Title of the project

Theme:
Mobility

Project Period:
Spring Semester 2020

Project Group:
SW805F20

Participant(s):
Andreas Stenshøj
Daniel Moesgaard Andersen
Frederik Valdemar Schrøder
Jens Petur Tróndarson
Rasmus Bundgaard Eduardsen
Mathias Møller Lybech

Supervisor(s):
Brian Nielsen

Copies: 1

Page Numbers: 35

Date of Completion:
May 28th, 2020

Abstract:

This is the best abstract ever written

Contents

Terms and abbreviations

Terms and abbreviations used in the report:

Pozyx : The hardware used for positioning.
AR : Augmented reality.
UWB : Ultra-wide bandwidth.
TWR : Two-way-ranging.
OSI : Open systems interconnection.
TCP : Transmission Control Protocol.
UDP : User Datagram Protocol.

Chapter 1

Introduction

1.1 Project idea

The idea for this project is to create a team-building game using augmented reality (AR). Two teams will compete against each other to score the most goals using a ball. Each player will be equipped with a smartphone-based virtual reality headset, and these will display the playing field from a top-down 2D view. To achieve this, each player's position needs to be tracked as well as where the ball is located on the field. In the top-down view, each player needs to see the positions of the other players and the ball. They also need to see their position on the playing field and where the goals are. The players should be able to set a number of goals they need to score to win before beginning the game.

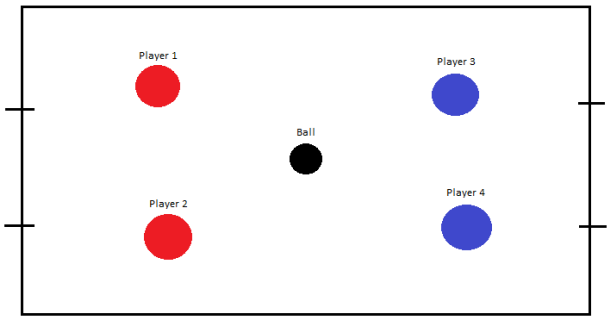

Figure 1.1: An illustration of the playing field

In Figure 1.1, an illustration of the playing field for the game is shown. There are goals at each end of the field, and the teams score goals by getting the ball between the goalposts. An alternative version of the game is suggested where, instead of goals at each end of the field, there would be virtual goal zones seen in the game which the teams need to bring the ball. These zones could even change locations as the game progressed.

1.1.1 Technical requirements for the project

Multiple pieces of hardware will be required to realize the vision of the game. First of all, each player must have a headset to hold their smartphone such that they can view a virtualized version of the playing field while playing. In order for the data to be synchronized between the players, they will also need to be equipped with a positioning device, which can transmit their location to the other players. This transfer of information will require a networking solution so that the virtualized playing field is synchronized between the players.

1.1.2 Problems to consider

The project idea proposes some problems that will need to be solved for the game to work. We will need to consider which technologies to use for the development of the visual aspect of the game which should show a top-down view for each player. As it is something we do not have experience with, it would be preferable if we do not have to build it from scratch. We will also need hardware that can track the positions of players and the ball. This must be accurate and update quickly such that the players do not run into each other, otherwise the game will not work. Another problem to consider is how the ball should be displayed in a 2D view. For the players to be able to find the ball on the field, it either has to be quite large to make it easier to find from the top-down view, or the game will need some metric to display how far the ball is from the ground. The game will also need to be able to track when the ball has crossed the goal line and then give feedback to the players. Another problem to solve is how to keep the positional data synchronized across all the players' devices, as it will be difficult to play the game without accurate data.

1.2 Essence

For the process of project development, we have chosen to work with Essence. Previous semesters we have worked with an agile approach inspired by Scrum, however, this semester we are attempting to apply the Essence approach. The basic idea of Essence is to encourage diverse thinking in the team, even though all members of our team share a similar background as bachelors in Software.

Essence uses two strategies to support value creation:

- *A systematic use of diverse viewpoints.* Values, views, and roles are used frequently in Essence. By using different views and roles to represent problems and solutions, Essence tries to facilitate a range of viewpoints on how a problem needs to be understood and solved.
- *A focus on idea maturation more than idea generation.* Essence applies the concept that ideas develop over time and tries to stimulate the team to evaluate and refine ideas [1].

1.2.1 Four variants of innovation

Essence tries to support innovation in software development, and hereby it defines four different variants of innovation, which are: [1]:

- *Product innovation* is new or radically changed software products or services.
- *Process innovation* is software solutions offering the user new or radically improved ways to produce products or services.
- *Project innovation* is fitting software solutions from earlier projects into new application domains
- *Paradigm innovation* is about software solutions coupled with changes in the mental model of what a business is, who the users are or what the market is.

1.2.2 Paradigms

There are two well-established software development paradigms: the document-oriented paradigm, which we know from the waterfall approach and agile paradigm which we know from for example extreme programming and Scrum. The author of Essence considers the new emerging paradigm called the pragmatic paradigm.

The document-oriented paradigm portrayed software developers as being document-oriented. The requirements are static and allow for a top-down waterfall approach to software development, which pays small attention to creativity and innovation.

The agile paradigm sees software development as user-oriented. Requirements are presumed dynamic as customers learn about options and constraints within the course of the project. This leads to incremental software development.

The pragmatic paradigm is problem-oriented. Systems are becoming more complex and it is more difficult to separate systems from each other. The amount of data, software libraries and hardware components available is steadily increasing, leading to hypercomplex software projects. Hypercomplexity is a degree of complexity that makes it impossible to make rational decisions within a reasonable time constraint. The most important features of this paradigm are that requirements are not completely known when the projects start. Ideas evolve in the process of the project, and during the project the requirements for the project are negotiable [1].

1.2.3 Core concerns

All software projects involve these four core concerns:

- Do we know what to build?
- Do we understand the solution?

- Do we understand the problem?
- Should we pivot or persevere?

1.2.4 Team organization

Within the team organization, in Essence, roles are used to create heterogeneity in teams, to ensure diverse points on views and to ensure cohesion despite diversity. The focus of these roles is to increase learning with personal interaction by sharing insights and experiences. The roles also ensure that the team understands the problem domain, and see potentials in the technology domain.

As a rule of thumb, the roles are persistent meaning that a member will have the same role for the duration of the project. The roles in Essence are compatible with agile software development, making it possible to combine Essence with other processes like Scrum.

There are four roles in essence:

- Child
- Responder
- Challenger
- Anchor

The role of *Child* can ask any questions and make propositions that are opposite of previous decisions. The rest of the team is not allowed to criticize the child, but they are however allowed to ignore the person's suggestions. The child is one of the main sources of ideas and other perspectives on the project. Outsiders are also allowed to take this role.

Responders are the developers in the team, and they are usually the majority within the team. Responders work closely together with the *Challenger*, so that the most important features are developed first.

Challenger is the customer or customer representative. The challenger can be compared to the *Product Owner* in Scrum. This role formulates and explains the Challenge, prioritizes features and accepts the solutions. There can be more than one Challenger, but if there are they must agree on the product vision.

Anchor is the one responsible for leading evaluations but does not decide the consequences. If necessary, the anchor can intervene and remove threats to the team's ability to develop ambitious responses. A potential threat could be something that results in productivity issues.

Chapter 2

Sprint 1

2.1 Sprint goal

The goal for this sprint is to explore the project idea and look for possible solutions to some of the problems that the project idea introduces. In this sprint, we want to learn more about how we can visualize the game for the users so they can see themselves in the game as well as the playing field. We want to look at how we can use Pozyx to get the playing field as well as the players' positions in an accurate way with fast updates. To test if Pozyx is viable for use in this project we want to conduct an experiment where we test the accuracy of the positioning. As this is the first sprint of the project, the main focus is to gain more knowledge about the different aspects of the project and to gain a shared vision amongst the group members of how the game is going to work. The shared vision will be achieved by making prototypes of the game as well as architectural diagrams of how different components in the game will work together.

2.2 Prototypes

To come to a common vision about the layout and functionality of the system, a series of prototypes were created. The primary focus of these prototypes is not to be used for implementation, but rather for comparing opinions about how the flow in the system should be created.

Since the user interface is mostly focused on the mobile devices that the players will be wearing, it was decided that the host computer should simply have a text-based interface, as seen on Figure 2.1.

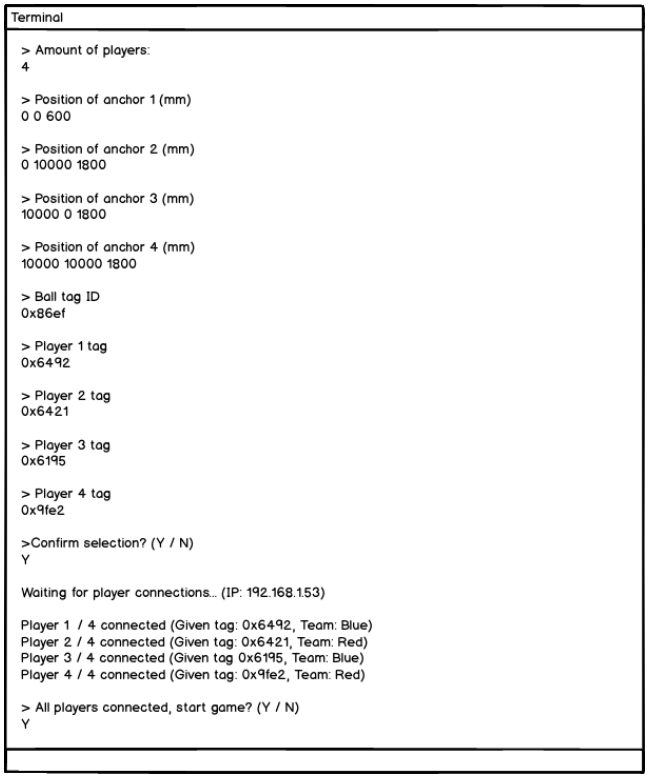

Figure 2.1: Prototype of hosting interface

When a user starts the application, they will be greeted with an input field, where they will specify the IP address of the host machine, as seen on Figure 2.2.

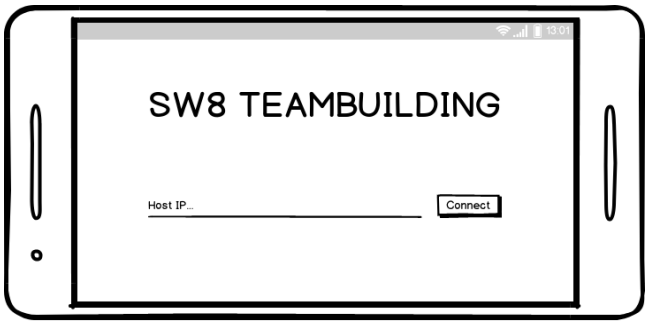

Figure 2.2: Prototype of game menu

After inputting an IP and confirming, they will be redirected to a page where they can see how many users have connected to the host, as seen on Figure 2.3.

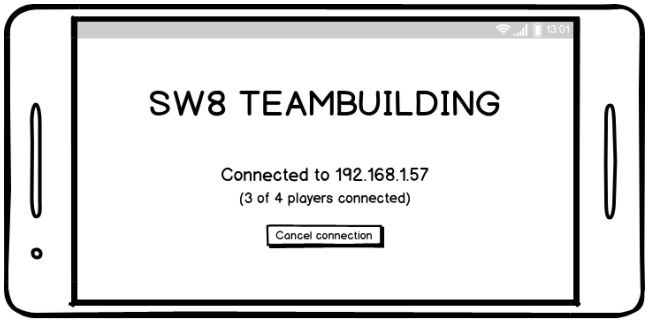

Figure 2.3: Prototype of screen where a user has connected to the host

When all users have connected, the host can start the game, and they will now see the virtual game field, as seen on Figure 2.4. In this prototype, the player's icon is highlighted by having a solid color, whereas the other players are just shown as outlines. In the middle of the screen is the ball in a designated starting area to make the game fair for both teams.

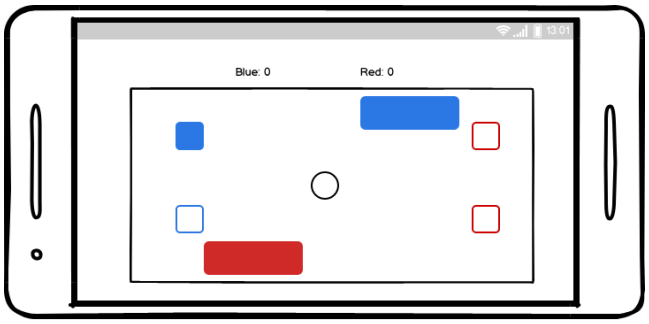

Figure 2.4: Prototype of in-game screen

2.3 Architecture of the game

In this section, we will look at the overall architecture of the different components involved in the game.

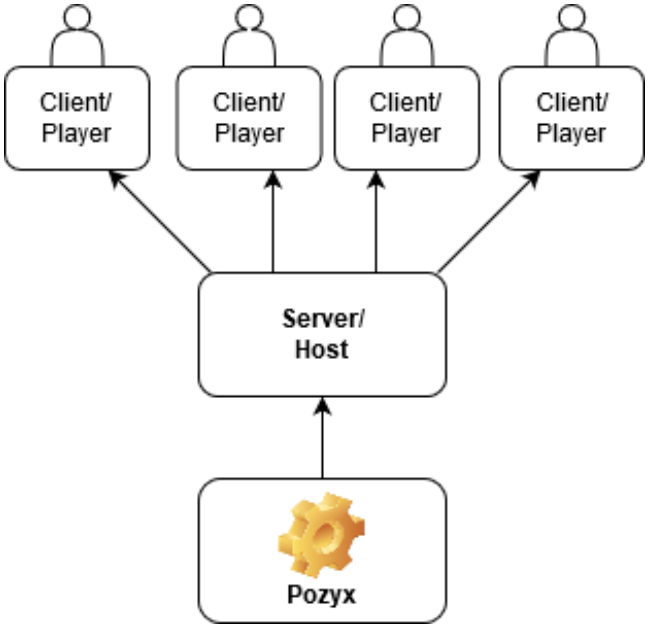

Figure 2.5: The different components of the game

In Figure 2.5 an illustration of how we imagine the different components of the game are going to work together is shown. The arrows in the diagram show the flow of data in the game. The Pozyx component is responsible for providing the positional data to the host. Each client/player is equipped with one of the tags that track their positions on the field. The playing field's dimensions will be set to match the distances between the anchors. The host has the positional data for each client, but it will also need to know which of the tags each client is equipped with. The host will then continuously transfer positional data to the clients. This information will also include which of the tag ids belong to which clients. Each client will be running an instance of the game, and with the use of the positional data, it will render the playing field, the ball and each of the players. The host is responsible for checking if the ball has crossed the goal line and it should then provide this information to the players. In general, the clients' instances of the game should only be responsible for rendering. Any game logic should happen on the server-side to make sure that the game is synchronized for each player.

2.4 Pozyx

Pozyx is a hardware/software solution that is used to provide positioning with an accuracy of down to 10 cm [5]. It makes use of ultra-wideband in combination with machine learning for positioning, which according to their documentation is more precise and efficient than traditional positioning systems such as WiFi, Bluetooth, RFID, and GPS.

Since the two major requirements for positioning in this project are precision and a high update rate to ensure that the players can have reliable data available, the Pozyx system seems like a good place to start.
The Pozyx tags support update rates of up to 125 Hz for a single tag [5]. The Creator system from Pozyx is sold with 4 anchors and 5 locatable tags. An anchor is a stationary sensor used by the moveable tags to get their exact position.

2.4.1 Finding the location of anchors

A trilateration method is used for finding the position of a given tag using the anchors. This method uses basic geometry to estimate the position by measuring the distance to the anchors of which we know the position. With this distance estimate, it is possible to draw a circle with a given radius. If we use two anchors, we will have two intersection points which are the possible positions of the tag. This means that to find a two-dimensional location, we will need at least three anchors, which will lead to only a single point where all three circles intersect, as seen on Figure 2.6

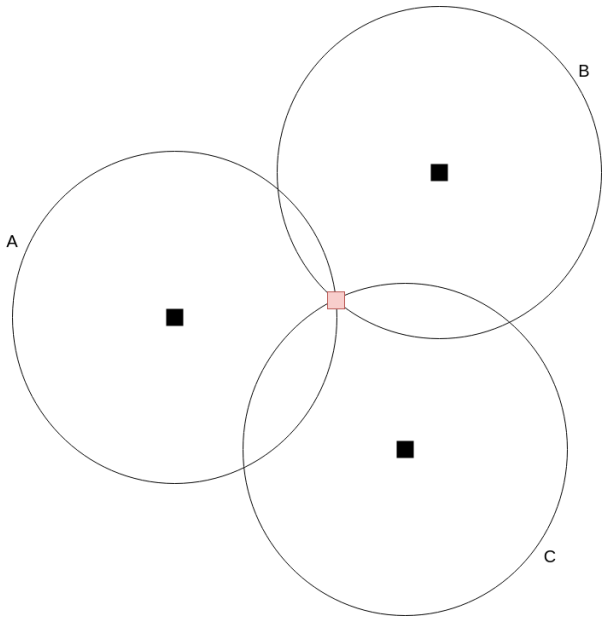

Figure 2.6: An example of trilateration using three anchors

The issue with this approach is that the measurements are not perfect, which might cause the circles to not intersect at exactly one point which will make the positional data seem to jitter.

2.4.2 Using UWB

To find the position of the tags Pozyx makes use of radio waves. Radio waves travel at the speed of light in vacuum, so by dividing the time of travel between anchors

with the speed of light, the distance between them can be found. Because the speed of light is so fast the time measure needs to be very accurate to get the correct distance. To achieve this the anchors make use of ultra-wide bandwidth (UWB) [6]. The ultra-wideband signals that the Pozyx devices utilize has a bandwidth of 500 MHz. This makes the wavelength very short and by detecting the peak of a narrow "pulse", an accurate time can be found. High bandwidth means faster data transfer, which most people would prefer, but if everyone were to use the same frequency the signals would interfere with each other, therefore the use of high-frequency signals is tightly regulated. Pozyx can use 500 MHz because it transmits at very low power.

2.4.3 Alternatives to UWB

Having a high accuracy is essential for the users' experience since a low accuracy could result in the in-game information being incorrect or just too imprecise to make it a joyful experience. While Pozyx uses UWB for positioning, it is interesting to take a look at the primary alternatives to indoor positioning and to see if any of them can compete with Pozyx' high accuracy **??**.

GPS

The obvious alternative for positioning data is GPS, which is frequently used for outdoors positioning. In optimal outdoor situations, GPS can provide accuracy within 4.9 meters **??**. However, the GPS positioning accuracy can be degraded due to buildings, bridges, and trees, making it a sub-optimal choice for indoor positioning.

Wi-Fi

Since the project already utilizes networking for data transfer, Wi-Fi may be worth considering to decrease the number of technologies needed. However, Wi-Fi only provides an accuracy of between 5 and 15 meters depending on the hardware chosen for access points and clients. In addition to this, iOS devices are blocked from using Wi-Fi for indoor navigation purposes, since it usually makes use of a technology called fingerprinting, which only functions with Android devices due to technical restrictions [7].

Bluetooth

Using Bluetooth for indoor location is similar to how Pozyx works with anchors and tags. Instead of anchors, Bluetooth beacons are able to send out signals with a range of up to 30 meters [8]. With Bluetooth, you get a significantly higher accuracy compared to GPS and Wi-Fi. In the optimal settings, this solution can provide an accuracy of up to one meter. Unlike Wi-Fi, this solution would work on both iOS and Android devices but has a lower range than the Pozyx solution.

RFID

Finally, we have RFID which uses radio waves to wirelessly transmit the identity of an object. Unlike the other solutions, RFID offers high accuracy, but a very limited range of less than a meter [9]. For a game like what we are building, it would be possible to use RFID for the location of the ball to check which player is currently holding it, but due to the limited range, it would not make sense to use it for localizing the players.

Generally, using Pozyx with their UWB technology for tracking the position of the players and the ball appears to be the most optimal solution.

2.4.4 Two-way-ranging

We are using the Pozyx Creator Kit Lite which uses the Two-way-ranging (TWR) protocol for positioning [10]. A tag calculates its position by communicating with the anchors one by one, getting the distance from the anchor to itself. Once it has the distance from 3 anchors it can compute its position utilizing trilateration.
If multiple tags are being used at once, one tag is made the master tag and the other tags become the slave tags. The master tag instructs the slave tags to report their position to the master tag one by one. The master tag is then usually connected to a computer that can use the position data. This technique does not scale well as all the slave tags have to be within the radio range of the master tag so spreading them across huge areas is not possible. Instead of a tag being the master it is possible to use an anchor. This makes it easier to have a computer attached, as the anchors are stationary, unlike the tags.

2.5 Unity introduction

As defined in section 1.1, this project aims to create a team-building augmented reality game. This means the project has to have a game component - an application to display the objectives of the game, the play area, and the players. To create this, a game engine can be used, such as `Unity`. A game engine is a piece of software that provides creators with the necessary set of features to develop games quickly and efficiently [14]. This means that a game engine is a collection of reusable components, abstracted away from the game developer. This can include tools to help with, for example, graphics, physics, networking or audio. These tools would expose certain functionality to a developer to make use of, and hide the specific implementation details for that functionality, ensuring the developer can focus on more pressing issues. Unity supports the C# language for development [11].

The Unity game engine supports development for different game platforms. Of particular interest to this project is the support for both `Android` and `iOS` devices, as well as `Google Cardboard` [12]. We chose to use Unity for the development of the game aspect of this project. This facilitates that a greater amount of time can be

spent on the other aspects of the project rather than the low-level details of game development, and it allows for easier inclusion of multiple platforms.

2.6 Networking

The following section will examine the different possibilities regarding transmitting player position data to the Unity applications used to play the game.

2.6.1 Possible networking solutions

Unity will be used for the creation of the game aspect of this project as described in section 2.5. Unity includes a proprietary networking solution known as UNet [13]. This solution allows developers to use a high-level API, giving access to commands that cover many common requirements for multiplayer games, without worrying about the low-level details. Since the solution is developed alongside the actual game engine, it has a higher level of integration with the Unity Editor and Engine, which allows for certain components and visual aids to aid the building of the game. As of the beginning of this project, the UNet solution has been deprecated for a while, and the Unity developers are actively working to create a new system to replace it. The current UNet iteration is usable but will be removed in the future. Other third-party solutions for Unity-based games also exist, such as Photon Engine. Photon provides functionality for the developers to make use of to create multiplayer games in the same way as UNet, exposing higher-level functionality. Photon supports multiple platforms outside of just Unity, with both Android and iOS support [4].

ZeroMQ is also a possible solution. ZeroMQ is an asynchronous messaging library. It can carry messages across various transport formats and is available in many different programming languages [15]. It aims to be a high-performance library to be used in distributed or concurrent applications that are reliable. According to the getting started guide provided by ZeroMQ, certain issues tend to arise when developers attempt to create a networking solution using sockets [16]. These are:

- How to handle I/O?
- How are dynamic components handled? What happens if a component disappears temporarily?
- How are messages represented? Different sizes and different content can change representations
- How are messages that cannot be delivered immediately handled?
- Where should message queues be stored?
- How are lost messages handled?

- What if the network transport changes, for example, TCP to UDP?
- How do messages get routed? Can the same message be sent to multiple peers?
- How to write an API for another language?
- How to represent data such that it can be read between different architectures? How much of this should be the messaging system's job?
- How do network errors get handled?

These issues are mostly applicable to general solutions that need to accommodate changing requirements or be reusable. However, for this project, not all of these issues are relevant. In terms of problems to overcome, this project should only be concerned with handling dynamic components, handling lost messages, routing messages and handling network errors. If a player closes the game application it can lead to dynamic component issues. A message can be lost during the playing of the game. Messages should be delivered to all players to ensure that they all have the same information. Finally, a player might suddenly disconnect from the network.

The alternative to making use of a pre-existing solution is creating a custom solution. A custom solution entails a need to establish a familiarity with the required knowledge to construct such a solution. A custom solution would involve sockets, which are a network API that allows programs to communicate with each other [3].

2.6.2 Choosing a solution

There are certain pros and cons associated with both approaches of using either a pre-existing solution or a custom solution. Table 2.1 shows some of the considerations made when deciding an approach for this project. Based on these considerations, it was decided that a custom solution should be created to handle networking in this project. This choice was based on the lack of transparency in a pre-existing solution as well as the need for fast communication. For the game to be playable and enjoyable, the location data collected by the Pozyx system needs to be transmitted to all the clients as quickly as possible such that they always have an up to date view of the positions of the players. To achieve this, it would be preferable to build a solution capable of performing the minimum amount of work as quickly as possible. Pre-existing solutions cannot be guaranteed to do the minimum amount of work as lower-level details are obscured from the developers. With a custom solution, the data sent across the network can be guaranteed to be exactly what is needed. ZeroMQ was also a possible choice based on the performance needs, but its generalized approach concerning itself with reusability and issues unlikely to be a big factor in this project meant it was dismissed, in favor of a custom solution in which the problems defined in the previous section are handled.

	Pre-existing Unity	Custom	ZeroMQ
Customizability	Consists of a set of pre-defined functionalities	Can have any functionality implemented	Has pre-defined functionalities, but these are lower level than a pre-existing solution
Requirements	Familiarity with the solution	Familiarity with the knowledge required to implement a usable solution	Needs familiarity with a mix of pre-existing and custom solutions
Optimization	Lower-level details are obscured, optimized for general use	Lower-level details are freely available, can be optimized for a specific purpose	Focuses on performance, but the solution is general

Table 2.1: A comparison of the pros and cons of the possible solutions

2.6.3 Introduction to sockets

In order to construct a network solution, a familiarity with the layers of a network is needed in order to gain an intuition of what sockets are. A common way to describe these layers is through the *open systems interconnection* (OSI) model for communication. This model is illustrated in Figure 2.7, along with approximate mappings of the technologies used for each layer.

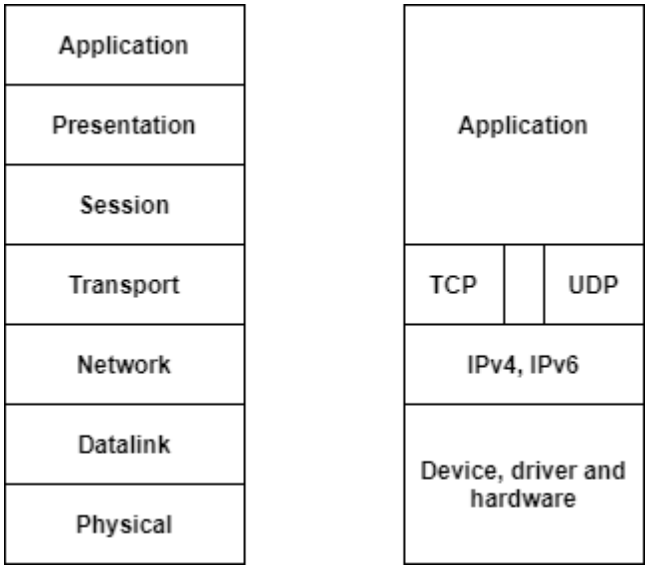

Figure 2.7: An illustration of the OSI-layer, and the corresponding technology for each layer.

As shown, the relevant layers for the purpose of creating a networking solution though sockets is the third and fourth layers, transport and network. The network layer is handled by the IPv4 and IPv6 protocols, which will be discussed in subsection 2.6.6, and the transport layer is handled by either TCP or UDP. The reason for the gap between TCP and UDP is to illustrate that it is possible to bypass this layer and use IPv4 or IPv6 directly [3]. Sockets provide the interface from the upper application layers to the transport layer. The upper layer handles details about the application, and the lower layers handle details relating to communication.

Programs that communicate across a computer network need an agreement on how those programs will communicate. This is known as a protocol. Generally, before defining the design details of the protocol, a decision should be made as to which program is expected to initiate communication. One way of defining this is through the client server architecture illustrated in Figure 2.8. This split is used by most network-aware applications [3]. The most common method of initiating communication when using the client-server architecture is to have the client initiate requests. This tends to simplify the protocol and the programs themselves [3].

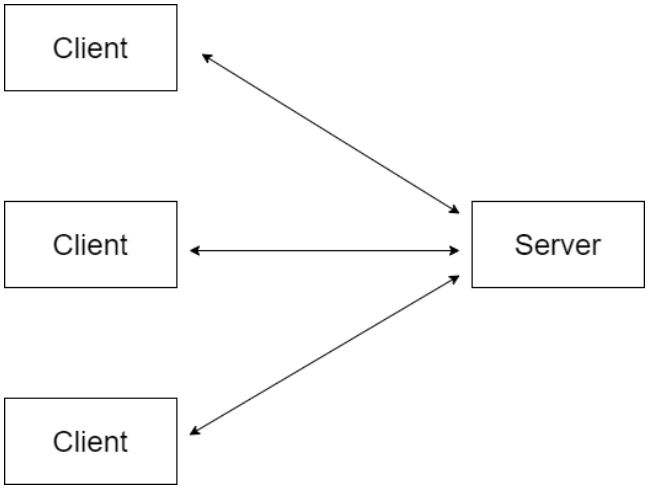

Figure 2.8: An illustration of the client-server architecture with multiple clients

2.6.4 TCP and UDP

The following section introduces two different protocols for the transport layer - Transmission Control Protocol (TCP) and User Datagram Protocol (UDP). Both of these protocols use a network-layer protocol known as IP, which can be either the protocol IPv4 or IPv6.

TCP

TCP provides connections between clients and servers. A TCP client establishes a connection with a given server, then it receives or sends data to that server across the network and closes the connection. TCP provides reliability. When TCP sends data, it requires an acknowledgement from the receiver that the data has been received. If it does not receive such an acknowledgement, TCP automatically retransmits the data and then waits for an acknowledgement for the retransmission. After a certain number of retransmissions TCP gives up. Based on implementation TCP will typically attempt to send data for 4-10 minutes. This does not guarantee that the the receiver will receive the data, but the guarantee is that it will deliver the data if possible, or notify the user that the connection has been broken without an acknowledgement from the receiver. In order to know how long to wait for acknowledgements, TCP contains algorithms to estimate the *round-trip time* between the client and server dynamically. It also performs these estimations continuously, as the result can be affected by variations in the network traffic. TCP sequences data by associating bytes and sequence numbers. For example, if an application writes 2048 bytes to a TCP socket, it would be sent in two segments, with the first containing data with the sequence number 1-1024, and the second containing data with the sequence number 1025-2048. If they arrive in the wrong order, the receiving TCP reorders the segments based on the sequence numbers before passing the data to the

receiving application. If the receiver receives duplicate data this can also be detected through the sequence numbers, and the duplicate data can be deleted. TCP provides flow control by telling clients how many bytes of data can be accepted at any time, known as the window. The size of the window decreases as data is received, and decreases as the receiver reads data from its buffer.

UDP

UDP is a simple transport-layer protocol [3]. An application writes a message to a UDP socket, which gets encapsulated in a UDP datagram, which further gets encapsulated in an IP datagram and then sent to the destination. A datagram is a self-contained entity of data carrying information to be routed from the source to the destination nodes without reliance on earlier exchanges between the nodes and the transporting network [**hpbrowsernetwork**]. A UDP datagram is not guaranteed to reach its final destination, nor is it guaranteed that order will be preserved across the network, or that datagrams arrive only once. This means that the UDP protocol is unreliable. If a datagram is lost on the network and not delivered to the UDP socket, it will not be automatically retransmitted. UDP also does not provide acknowledgement that datagrams were received, sequence numbers to ensure data can be ordered, *round-trip time* estimation or timeouts. UDP has no notion of flow control, meaning it is possible for a fast UDP sender to transmit data at a rate the receiver is unable to keep up with. As such, it does not provide the same reliability as TCP. If reliability is a requirement, it has to be built through features such as timeouts, retransmissions and adding acknowledgements from the receiving end. A UDP datagram has a length, which is passed to the receiving application along with the data. UDP is considered connectionless, as there does not need to be a long-term relationship between a UDP client and the server. A UDP client can create a socket and send a datagram to a server, and then immediately send another datagram on the same socket to a different server. A UDP server can receive several datagrams on a single UDP socket, each from different clients.

Choosing between UDP or TCP

Windowed flow control might not be necessary for transactions where both ends agree on the maximum size of a request or a reply [3]. For this project only one type of message will be sent - the player location data. This message will always be formatted in the same way, and as such, a maximum size can be agreed meaning the flow control aspect of TCP is not needed. Another aspect of TDP that is not needed for this project is the automatic retransmission of messages. While this can provide reliability, it is of no importance for the game. The players of the game are only concerned with the most recent updates of their position. As such, if a message were to not be received, it would not make sense to continually delay subsequent messages in order to attempt to retransmit a message containing position data that is more and more likely to be outdated. In order for the position data

to be as recent as possible, the messages should be sent as frequently as possible. It is also not necessary to provide an acknowledgement that the message has been received. The receiving applications should simply update their locations to comply with the most recently received data. The sender should not be concerned that a message was received, it should just continue to send the next message, which is likely to be more recent. Duplicate messages also do not pose much of an issue. If the applications were to receive the same location data multiple times, it would not impact the overall functionality of the program, rather just the speed at which the next updates would be received. UDP has no connection setup or teardown costs. UDP only requires two packets to exchange a request and a reply, whereas TDP requires about 10 packets [3], if a new TCP connection is established for each exchange. In terms of transaction time, the minimum time for a UDP request-reply is the $round - triptime + serverprocessingtime$, and the minimum time for TDP is $2 \times round - triptime + serverprocessingtime$ [3].

Because of the limited scope and uniformity of what is going to be transmitted via the networking solution, a lot of the features included in TCP are unnecessary. UDP is slightly faster because of its lack of reliability and other benefits, but might require some extra work in order to implement some of the functionality that is missing when compared to TCP, if this were to become necessary. Because of the reasons discussed, UDP seems to fit the needs of this project more than TCP, and is the protocol chosen for the networking solution featured in this project. An issue with the choice of UDP could present itself in that messages are not guaranteed to arrive in order. It could pose a problem if a player in the game received a message with recent location data, and then another message afterwards with outdated data. This could cause the player objects in the game to be at positions in which they were in the past, but not currently in the present.

2.6.5 Introduction to UDP sockets

UDP is a connectionless, unreliable datagram protocol. Figure 2.9 shows an illustration of the client-server architecture using UDP. The client side creates a socket, and sends a request to the server as illustrated. Once the request has been sent the client transitions to a state of awaiting a reply. Once the reply is received the client can send another request, or the socket can be closed. The server side also creates a socket, and then binds the socket to a port. Once bound, the server can await a request from the client. When a request is received, the server processes it, and then sends it to the client after which it can return to awaiting requests.

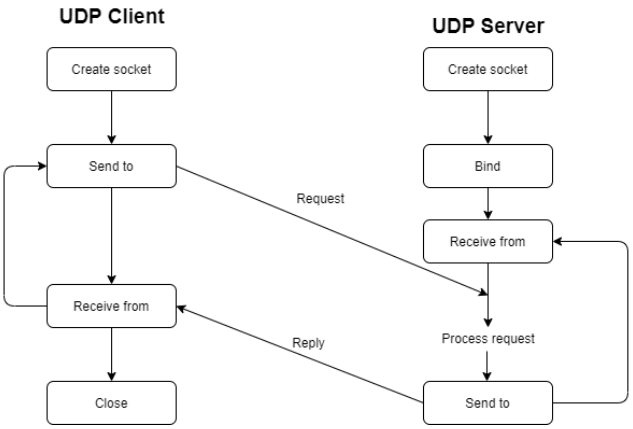

Figure 2.9: An illustration of the client-server architecture with UDP

For the purpose of this project, it might not be optimal to have client request the server. The server does not need information from the client or acknowledgement, meaning the clients do not have to send messages. As such, it might be better to use a publisher-subscriber approach. The server would then act as a publisher, constantly sending messages to the clients that would be subscribed to the publisher. An illustration of this concept using UDP can be seen in Figure 2.10.

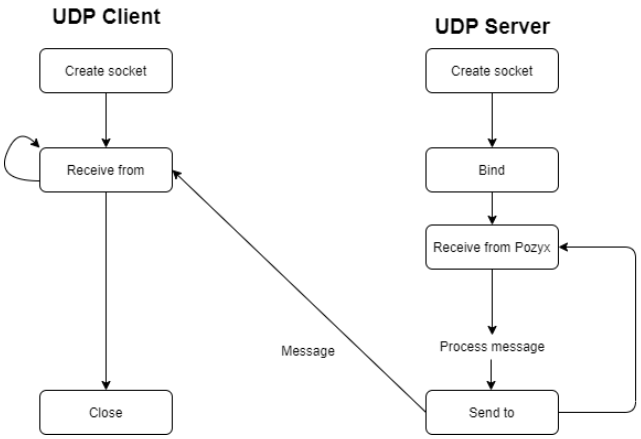

Figure 2.10: An illustration of a publisher-subscriber architecture with UDP

Possible issues with using this architecture is that there would need to be a way to ensure all clients receive a sufficient amount of messages, and that the receivers are not overloaded with messages.

Transmission in UDP

When working with UDP, there are three types of possible transmissions: Unicast, multicast or broadcast. Unicast is a one-to-one communication with a single source

sending information to a single receiver, while a broadcast transmits the information to all nodes on a network. The benefit of unicast is that it has been in use for a long time and utilizes well-established protocols, it is known from applications like http, ftp and telnet [2]. For this project, however, the major drawback of unicast is that in order to transmit the message to multiple nodes, it has to send multiple unicasts messages addressed to each receiver, which also requires us to know the exact IP address of each destination device. Looking at broadcast, we can instead transmit the information to all nodes on a network, which ensures that all nodes on the network receives the message. This could be useful if we only had the players of the game on our network, but since they might be connected to a larger network with a large amount of clients, this could lead to the data being sent to clients that are not a part of the ongoing game session. Luckily, we have multicast in the middle of the two extremes, where you do not send from one client to another or one client to all others. Instead the data is sent to as many destinations as express an interest in receiving it [2]. This one-to-many approach seems suitable for this project, since it would intuitively lead to better bandwidth utilisation and does not require the receivers' addresses to be known.

2.6.6 IPv4 and IPv6

Both IPv4 and IPv6 provide packet delivery service for TCP and UDP. The major difference in IPv4 and IPv6 is the addresses they use. IPv4 uses 32-bit addresses, whereas IPv6, being a newer version, uses larger addresses of 128 bits [3]. IPv4 addresses are usually written as four decimal numbers separated by ".". This is known as *dotted-decimal notation*. Each decimal number represents one of the four bytes of the 32-bit address. The first of the four numbers represents the address type. IPv6 addresses are usually written as eight 16-bit hexadecimal numbers [3]. The higher-order bits of the 128-address imply the type. For the purposes of this project this should not have a significant impact, and either can be used.

Chapter 3

Sprint 2

Chapter 4

Sprint 3

Chapter 5

Sprint 4

Chapter 6

Appendix

6.1 Test plan

6.1.1 Documentation

All functions should be documented. This is done by writing comments in the code following the standards of the language.

For C#:
https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/language-specification/documentation-comments

For Python:
https://www.geeksforgeeks.org/python-docstrings/

These comments should explain what the parameters of the function are and a short summary of what the function does.

If the function that is being implemented deals with more complex concepts, additional comments should be written in the code to explain what is happening in the function.

6.1.2 Testing

What to test

All code that can influence the user's experience should be tested. This means that every function that would influence the program negatively if it had an incorrect calculation should be tested to try to avoid this.
All public functions should always be tested.

How to test

- Unit tests
  - Functions are tested in isolation.
  - If the function needs data from outside the function it should be manually generated using mocks.
  - More information about unit testing
- Integration tests
  - Testing if different modules integrate correctly.
  - An example of this is when we are fetching data from the python server into the C# client.
  - More information about integration testing

Unit testing in unity

There are two ways of running unit tests in Unity: play mode and editor mode. Play mode requires the framework to load the scenes before testing code while editor mode can test the scripts in the project without loading the scenes. When the scenes are loaded in play mode, the objects are instantiated with the predefined data in the scene.
Since this makes it more difficult to control the state of the program, the focus will be on testing the functions isolated in editor mode. This has the consequence that some code that utilizes the Unity API and functions on presentational objects in Unity can not be tested without using the play mode.
To make it easier to unit test your code you should try to separate logic that depends on Unity features and logic that does not into different functions.

How much to test

100% code coverage is not necessary, but if there are no unit tests for a part of the system it is not enough. The focus should be on the quality of tests rather than the number of tests.
All functionality in a function that can affect the output of the function should be tested. If a bug is fixed and it took more than a couple of minutes to fix it, unit tests should be created for this bug to prevent it from being reintroduced later on.

Bibliography

[1] Ivan Aaen. *Essence. Problem Based Digital Innovation*. 2020.

[2] Finjan Cybersecurity. URL: https://blog.finjan.com/unicast-broadcast-multicast-data-transmissions/ (visited on 03/01/2020).

[3] Bill Fenner, Andrew M. Rudoff, and Richards W. Stevens. *UNIX Network Programming, Volume 1. The Sockets Networking API*. 3. edition. Addison-Wesley Professional, 2003.

[4] Exit Games. URL: https://www.photonengine.com/en-US/PUN (visited on 02/14/2019).

[5] *Home: Pozyx*. URL: https://www.pozyx.io/.

[6] *How does ultra-wideband work?* URL: https://www.pozyx.io/technology/how-does-uwb-work.

[7] infsoft. URL: https://www.infsoft.com/technology/positioning-technologies/wi-fi (visited on 03/02/2020).

[8] infsoft. URL: https://www.infsoft.com/blog/indoor-navigation-indoor-positioning-using-bluetooth (visited on 03/02/2020).

[9] infsoft. URL: https://www.infsoft.com/technology/positioning-technologies/rfid (visited on 03/02/2020).

[10] *Positioning protocols explained*. URL: https://www.pozyx.io/technology/positioning-protocols-explained.

[11] Unity Technologies. URL: https://unity.com/how-to/programming-unity (visited on 02/13/2019).

[12] Unity Technologies. URL: https://unity3d.com/unity/features/multiplatform (visited on 02/13/2019).

[13] Unity Technologies. URL: https://docs.unity3d.com/Manual/UNet.html (visited on 02/14/2019).

[14] Unity Technologies. *Game engines - how do they work?* URL: https://unity3d.com/what-is-a-game-engine (visited on 02/13/2019).

[15] ZeroMQ. URL: https://zeromq.org/get-started/ (visited on 02/18/2019).

[16] ZeroMQ. URL: https://zguide.zeromq.org/page:all (visited on 02/18/2019).

List of Figures

List of Tables

Listings