# 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:** 101

**Date of Completion:**
May 28th, 2020



**Abstract:**

This is the best abstract ever written

# Contents

# Resources

The code for the project can be found in the following GitHub repositories:

## Host software

https://github.com/SW805F20/pozyx-location-tracker

## Client game

https://github.com/SW805F20/Unity

# 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.
**MVP** : Minimum viable product
**OSI** : Open systems interconnection.
**GDOP** : Geometric Dilution of Precision
**TCP** : Transmission Control Protocol.
**UDP** : User Datagram Protocol.
**Zeroconf** : Zero Configuration Networking.

# Chapter 1

# Introduction

## 1.1 Project idea

The idea for this project is to create a location-based competitive 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.

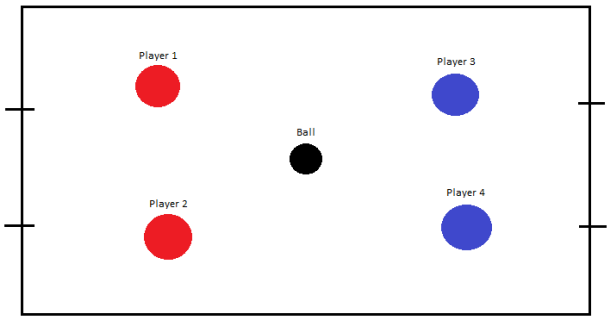


**Figure 1.1:** An illustration of the playing field

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 into. 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 it is not necessary to 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 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 Minimum viable product

To be able to evaluate the game with users as fast as possible, we establish a minimal viable product with functional and non-functional requirements.

### 1.2.1 Functional requirements

These functional features are essential to make the game work and are the core features of the game.

- Must be possible to start the game.
- Must be able to connect to the server from the clients.
- The users must be able to see their current positions.
- The users must be able to view the playing field, the goals, the players, the ball position, and the current score on their mobile device through VR goggles.

- A team must be able to win the game based on their score.
- The game must be dynamic, which includes:
  - Support an arbitrary number of players.
  - Size of the playing field can be set to a preferable size by the user.
  - Set how many goals needed to score to win.

### 1.2.2 Non-functional requirements

These requirements are more difficult to evaluate than the functional requirements and to test it a usability test must be conducted.

- The design of the field must be pleasant for the user to look at through a VR headset.
- The users should receive enough updates so that the positions of the players represent their real position.

## 1.3 Our process

This semester our process has been inspired by the `Essence` process taught in the Software Innovation course. We have been working with a ScrumBut approach during previous semesters, but we decided that we wanted to try to improve our process by taking inspiration from a different process this semester.
However, we have chosen to keep sprints, stand-up meetings and retrospective meetings from Scrum as these can complement Essence. To make the report fit this format, it has been split into 5 sprints that fit the length of the semester, with each sprint lasting three weeks, except for sprint 5 which will only last two weeks. The main aspect of Essence we drew inspiration from was the team organization.

### 1.3.1 Team organization in Essence

Within the team organization, in Essence, roles are used to create heterogeneity in teams, to ensure diverse points of view 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 try to ensure that the team understands the problem domain, and that it sees the potential solutions in the technology domain.

The roles are persistent as a rule of thumb, 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 in opposition to previous decisions. The rest of the team is not allowed to criticize the *Child*, but they are however allowed to ignore their suggestions. The *Child* is one of the main sources of ideas and other perspectives on the project.

*Responders* are the developers in the team, and 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.

The *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.

### 1.3.2 Roles in practice

We have divided the roles described in subsection 1.3.1 between the members of the group. One member has the *Challenger* role, meaning that they are accountable for prioritizing the tasks in the backlog. The process of prioritizing tasks is further described in the following section. Another member has the *Anchor* role, and is responsible for changes to the process as well as in charge of leading the stand up meetings, retrospectives and evaluations of the process. The rest of the group functions as *Responders*, which is the role for the developers of the project. The *Child* role fluctuates between members of the group. Everyone can add suggestions to improve an idea and give other perspectives on the project.
Due to our team size, the challenger and anchor will also work as developers during the duration of the project.

### 1.3.3 Prioritizing tasks

Our backlog is saved as a board on Jira, which can be seen on Figure 1.2. The leftmost column is the *Suggested* column. Everyone can make suggestions for tasks

that they find useful for the project. After each stand-up meeting, the challenger will present new suggestions that seem relevant to work on in the near future. This presentation will include the definition of done, and all members of the group will then vote on how valuable it is for the project and how time-consuming it is. The priority of a task is then calculated as $reward - time$, which is an arbitrary number to indicate how important it is.

The challenger then chooses the most important features from the *Discussed* column, often based on the highest priority, as *Chosen for Development*. Responders then have the opportunity to take tasks from this column and move it to the next column *In progress* when they start working on it. When the task has been completed, reviewed and merged into the develop branch, it is automatically moved to the column *Done*.

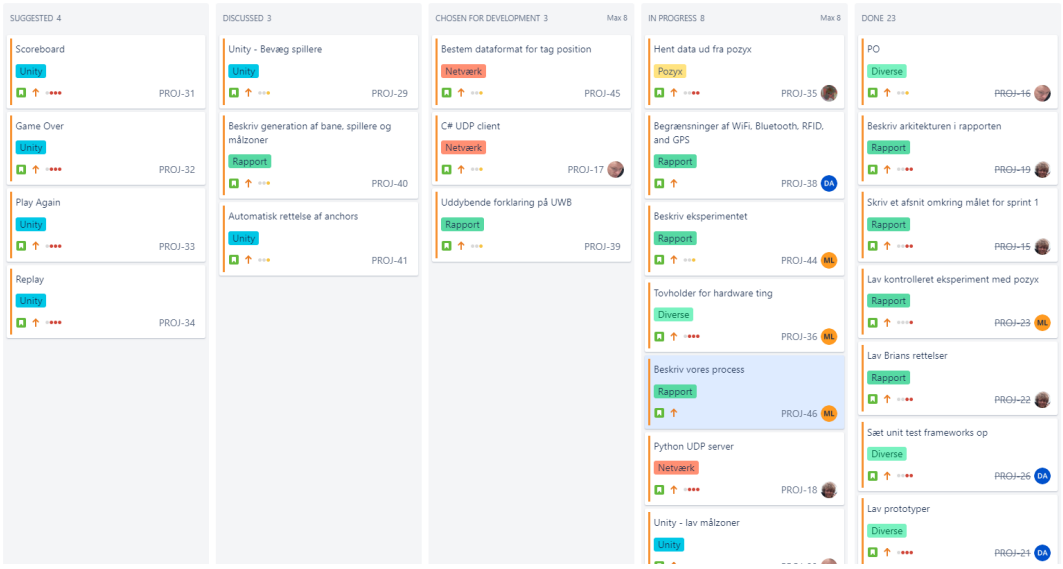


**Figure 1.2:** The board for tasks.

### 1.3.4 Pair reviews

Everything undergoes a formal review before it is added to the project in order to ensure quality. This involves ensuring that the code is functional, implements the definition of done, is readable and makes sense, as well as accompanied by tests for parts of the system that might need it. For report tasks, two reviewers are assigned to review and approve it before it can be merged. For coding tasks, two people are likewise assigned to review it, but the review has to be done as a pair, meaning that they have to physically sit together and go through the code on a shared screen. These pair reviews are a good way to share knowledge about the implementation through the group, as people have to understand it to be able to discuss it. During previous semesters people have reviewed the code separately, which led to fewer

comments as the code was not discussed between reviewers.

# Chapter 2

# Sprint 1

## 2.1 Sprint goal and introduction

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 visualizing the game for the users so they can see themselves in the game as well as the playing field.
We will look into using Pozyx to define the corners of the playing field as well as get the players' positions in an accurate way with fast updates, and to determine whether or not Pozyx is even viable for use in the context of this project an experiment will be conducted where we look into the accuracy of the positional precision of the tags.
As this is the first sprint of the project, the main focus is to gain more knowledge about the various aspects of the project and to gain a shared vision amongst the group members as to how the game should work. This initial 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

The first step in the project is to arrive at a common vision about the layout and functionality of the system. To accomplish this, a series of wireframe prototypes were created. A wireframe is a lo-fi prototyping technique where you create a grey box schematic that outlines the primary features of the user interface, unlike hi-fi prototypes which generally have a higher amount of detail and are used to test the interaction with the application. 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.

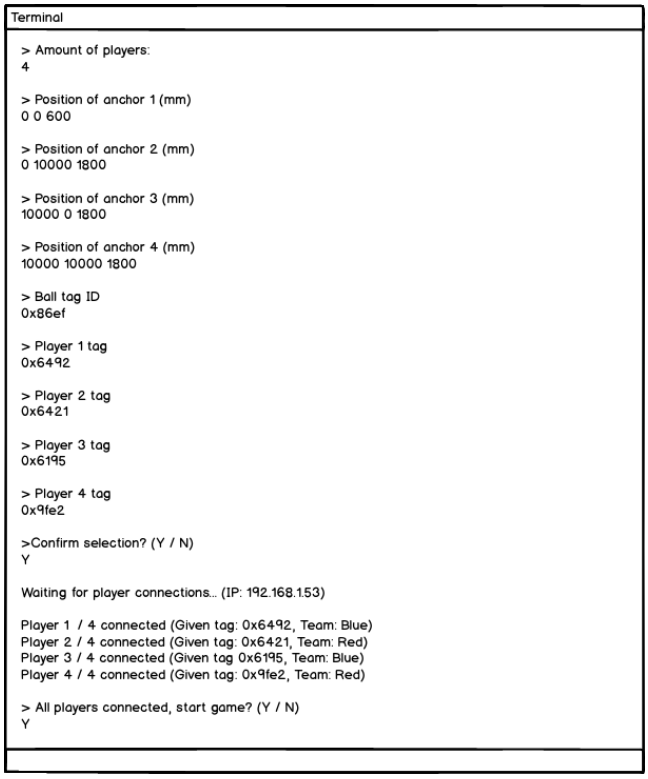


**Figure 2.1:** Prototype of hosting interface

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.

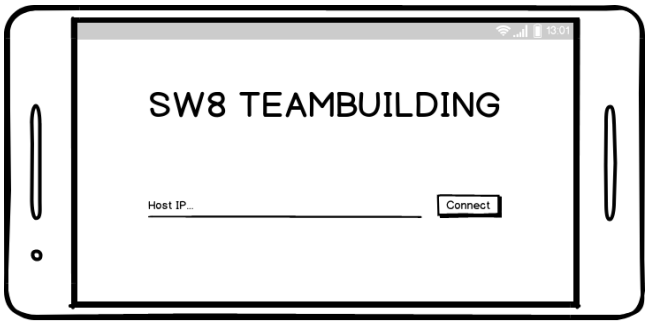


**Figure 2.2:** Prototype of game menu

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.

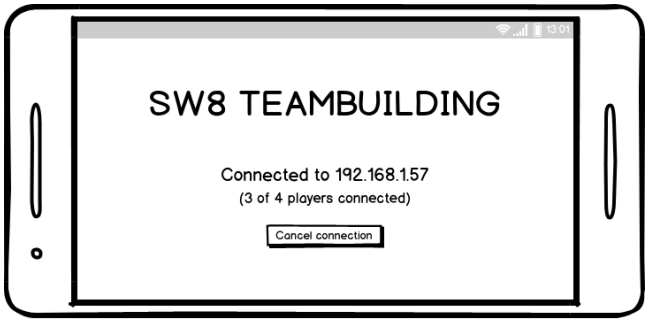


**Figure 2.3:** Prototype of screen where a user has connected to the host

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 players' icons are represented by small squares and the goals are represented by larger rectangles and have the same color as the players they belong to.
The current player's icon is highlighted by having a solid color, whereas the other players are just shown as outlines. The players are divided into two teams which are represented by the color of the player icons. In the middle of the screen is the ball in a designated starting area to make the game fair for both teams.

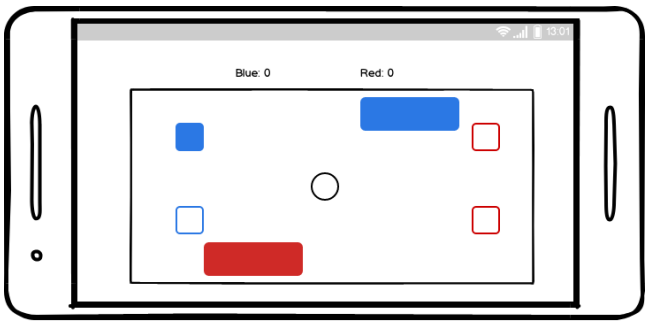


**Figure 2.4:** Prototype of in-game screen

## 2.3 Architecture of the game

In addition to the prototypes, which served as a broad overview of how the flow in the application should be, we will look at the overall architecture of the different components involved in the game from a more technical perspective.

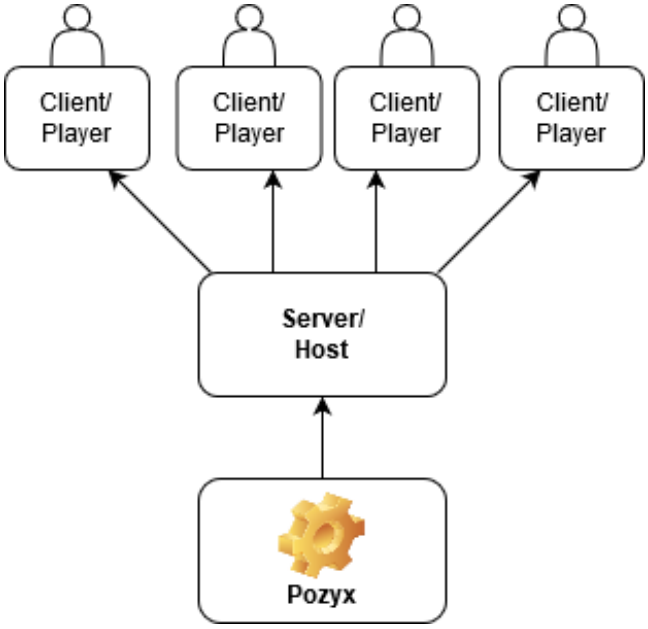


**Figure 2.5:** The different components of the game

In Figure 2.5 we see an illustration of the initial idea of how the different components of the game are going to work together. 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, this will be described in further depth in section 2.4. 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, such that it can 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

Since positioning is a major part of the project, it is important to have accurate positioning information. Pozyx is a hardware/software solution that is used to provide positioning with an accuracy of down to 10 cm [8]. It makes use of ultra-wide band (UWB) 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 [8]. 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.

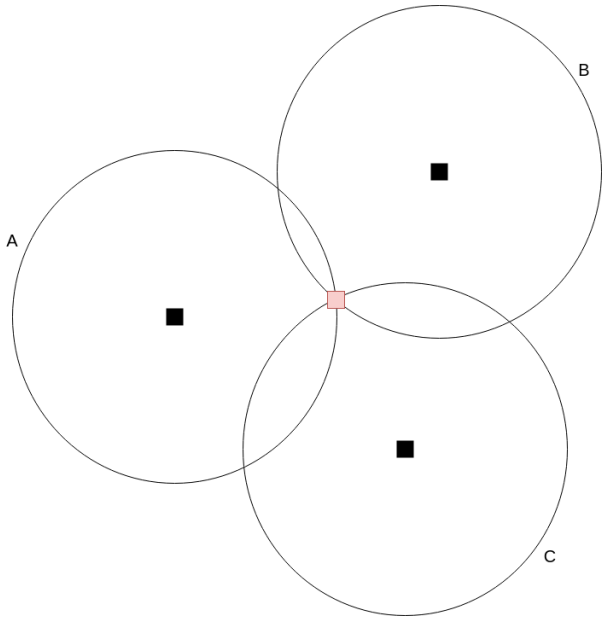


**Figure 2.6:** An example of trilateration using three anchors

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, 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 UWB [10]. UWB is a technology for transmitting data over a wide bandwidth usually wider than 500 MHz. The Pozyx sensors use exactly 500 MHz which means they just qualify as a UWB system. Because the bandwidth is this wide it makes the wavelength very short, and by combining multiple sinusoidal signals with slightly different frequencies the Pozyx tags can create a pulse with a peak which is very narrow. Measuring a narrow peak results in a more accurate timing, allowing the Pozyx to be much more precise than other technologies like Bluetooth and WiFi. 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[9]. To reduce the amount of interference the regulations say that UWB systems may only transmit at very low power. This means that a UWB transmission can not travel very far and therefore the chance of it interfering with another system is very low. Because of this we are not able to build huge playing fields with the limited amount of anchors we have if we also want it to be precise or even playable. Building a playing field that is too big could cause some tags to not reach all anchors and therefore not calculate a position. This could be combated somewhat by having more anchors, but as will be explained in 2.4.4 the slave tags might not be able to communicate with the master tag because the distance between them is too large.

### 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 [8].

#### 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 [18]. 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 [11].

#### 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 [12]. 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.

#### 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 [13].
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.

#### Conclusion

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 [19].
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 location-based 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 [23]. 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 [20].

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` [21].
Unity was chosen for the development of the game aspect of this project since 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 from the Pozyx tags to the Unity applications used to visualize 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 [22]. 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 [6].

The advantage of using a library that is built for Unity such as UNet or Photon is that it is easy to set up with the Unity engine compared to a solution built from scratch. UNet provides a higher level of abstraction with functions to control the networked state of the game, send and receive messages between server and clients and much more.
UNet and Photon also allow for "client-hosted" games that act like lobbies. So any client can host a game that other clients can connect to. This allows the clients to send and receive data between each other [22]. A disadvantage of these libraries is that they are generalized and thus would not be able to achieve the same efficiency as a custom solution tailored to the specific needs of the game.

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 [28]. 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 [29]. 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 [5].

### 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. The criteria that were considered when evaluating which solution to use were:

- Customizability
  - How much we can customize the solution to our specific use case. If the customizability is low the project needs to be built around the networking solution, whereas if the customizability is high the networking solution can be customized to our needs.
- Requirements
  - How much knowledge about the subject is needed to use the solution.
- Optimization
  - How much optimization are we able to do ourselves if we use this solution.
- Learning reward
  - How much will we learn if we use this solution.

| | **Pre-existing Unity-based** | **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 |
| **Learning reward** | Most of it is already implemented, so the learning reward is low | Learning reward is high because we have to implement everything ourselves | Have to customize some of it ourselves, so we will have to get familiar with the subject |

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

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 two major factors: the lack of transparency in a pre-existing solution as well as the need for fast communication and the opportunity to learn more about networking at a low level. 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. Additionally, we have not previously worked with networking at a low level, but wanted to learn more about

it, and decided that working without a framework would provide the best learning experience.

### 2.6.3 Introduction to sockets

In order to construct a network solution, a familiarity with the layers of a network is needed 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.

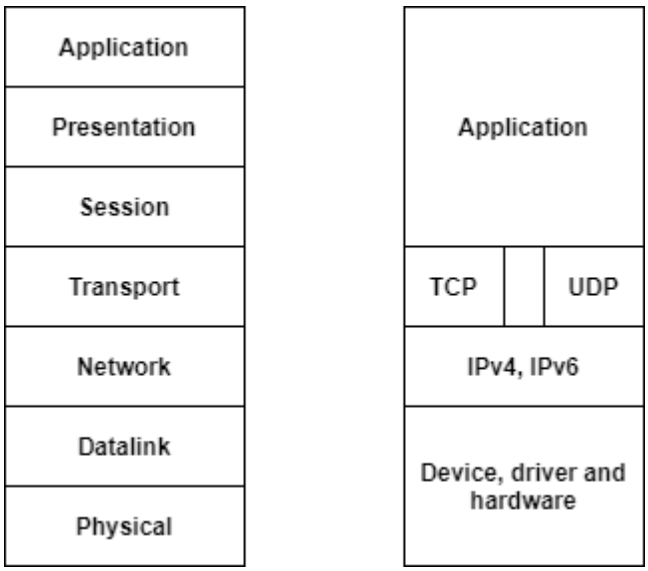


**Figure 2.7:** An illustration of the OSI-layer, and the corresponding technology for each layer.

As shown, the relevant layers for the purpose of creating a networking solution through 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, which is described in subsection 2.6.4.
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 [5]. 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 [5]. 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 [5].

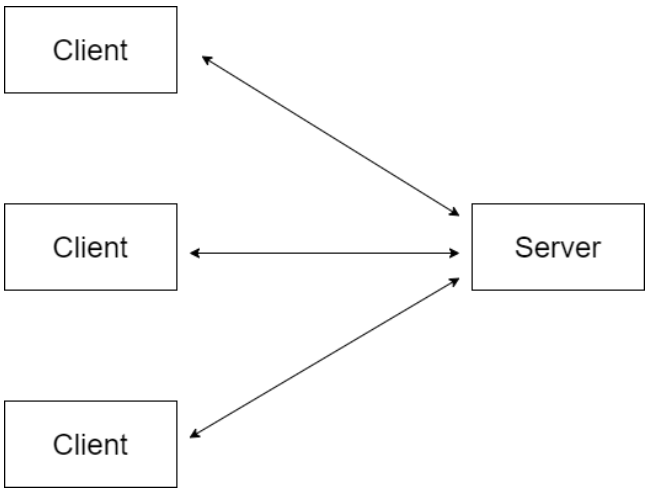


**Figure 2.8:** An illustration of the client-server architecture with multiple clients

### 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 by making sure that every package that is sent is recieved by the reciever. When TCP sends data, it requires an acknowledgment from the receiver that the data has been received. If it does not receive such an acknowledgment, TCP automatically retransmits the data and then waits for an acknowledgment for the retransmission. After a certain number of retransmissions, TCP gives up.
Based on the implementation TCP will typically attempt to send data for 4-10 minutes. This does not guarantee that 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 acknowledgment from the receiver. To know how long to wait for acknowledgments, 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 increases as the receiver reads data from its buffer.

### UDP

UDP is a simple transport-layer protocol [5]. 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 [7].
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 an acknowledgment 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 a fast UDP sender can 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 acknowledgments 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 [5]. For this project, the most important type of message is the player location data. This message will always be

formatted in the same way, and as such, a maximum size can be agreed to mean the flow control aspect of TCP is not needed. Another aspect of TCP 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 to attempt to retransmit a message containing position data that is more and more likely to be outdated.
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 acknowledgment 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 TCP requires about 10 packets [5], 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-trip time + server processing time, and the minimum time for TCP is 2× round-trip time + server processing time [5].

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 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 afterward 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.

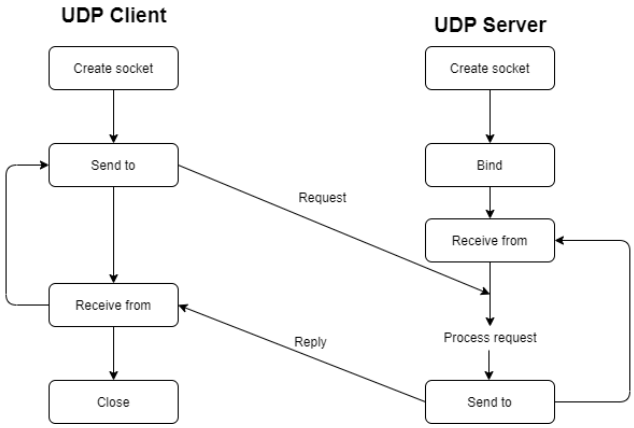


**Figure 2.9:** An illustration of the client-server architecture with UDP

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 acknowledgment, 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.

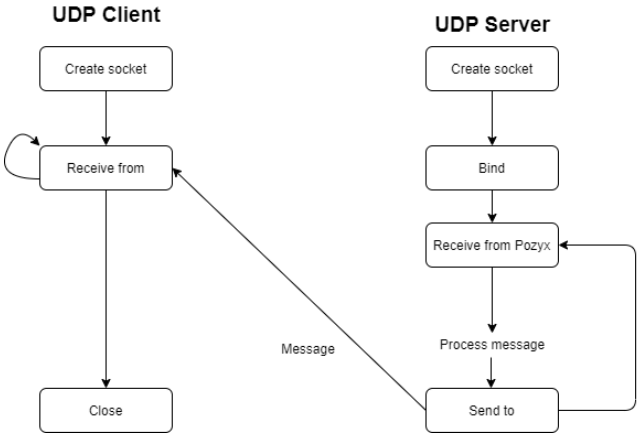


**Figure 2.10:** An illustration of a publisher-subscriber architecture with UDP

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 [4]. For this project, however, the major drawback of unicast is that 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 receive 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 number 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 [4]. This one-to-many approach seems suitable for this project since it would intuitively lead to better bandwidth utilization 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 [5]. 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 represent the address type. IPv6 addresses are usually written as eight 16-bit hexadecimal numbers [5]. The higher-order bits of the 128-address imply the type. For this project, this should not have a significant impact, and either can be used.

### 2.6.7 Format of data

As it is necessary to send data over the network, the format of the data must be decided upon. The network format for player position and goal position is `0xyyyyxxxxiisstt`, where `t` is the type, `s` is the timestamp, `i` is the ID, `xxxx` is

the `x` position and `yyyy` is the `y` position. The format can be seen for the player positions on Table 2.2. The format is little-endian, as the most significant byte is the one related to type, which is placed at the end. The abstract syntax for these messages is:

- Type of the message
- Timestamp
- ID associated with the message
- X position
- Y Position

The format for both is identical, the only difference between them is the type of the message and the context of the IDs. The abstract syntax for field anchor position is nearly identical, as the same information is required, but the field anchor format does not require a timestamp. For player position and field anchor position messages the `x` and `y` position is the location of the player or anchor. For the goal position format the `x` and `y` position is the center of the goal, where the size of the goal is set to have a length equal to 20% of the shortest side of the playing field.

The IDs for each syntax represent different things. The IDs for Table 2.2 are for each anchor, whereas ID for players is either the player tag or the ball tag, and the ID for the goal position message identifies the team. The type is represented at the right hand side of the hexadecimal so that it is faster to decode and to switch on which type it is. After the type the timestamp is placed, which is used to detect outdated data as a sequence number. The timestamp is an integer value ranging between 0 and 255 such that when it hits 255 it resets to 0 to be able to continuously send timestamps, regardless of game length. The downside to this approach is since UDP does not guarantee delivery of messages, it may not go straight from 255 to 0 every time. To counter this, 0 to 20 is considered as newer sequence numbers than 235 to 255 so that the system is allowed to miss up to 20 messages near the end, to ensure that it does not end up being locked until the values reach the high range again.

It was decided to use a size of respectively 16 and 8 bits so that it is straightforward to work with in the `C#` language, because it is the same size as some variable types in `C#` and hereby easy to type cast. Given the highest unsigned number that can be in two bytes is $2^{16} - 1$, the maximum value for the `x` and `y` positions is 65.535.

The goals scored has a different format of `0x1100sstt`, where `t` is type, `s` is timestamp, `00` is the score of team 0 and `11` is the score of team 1. This format can be seen on Table 2.3.

| POSY (y) (0-65.535) | POSX (x) (0-65.535) | ID (i) (0-3) | TIMESTAMP (s) (0-255) | TYPE (t) (0) |
|---|---|---|---|---|
| 16 bits | 16 bits | 8 bits | 8 bits | 8 bits |

**Table 2.2:** Format for player positions

| TEAM1SCORE (1) (0-100) | TEAM0SCORE (0) (0-100) | TIMESTAMP (s) (0-255) | TYPE (t) (3) |
|---|---|---|---|
| 8 bits | 8 bits | 8 bits | 8 bits |

**Table 2.3:** Format for goals scored.

## 2.7 Experiment with Pozyx

To determine the accuracy of the Pozyx tags an experiment was conducted. The primary goal of the experiment was to test the accuracy, but a secondary goal of the experiment was to determine the frequencies of updates for each tag.

### 2.7.1 Purpose

According to research regarding latency in VR [26] and interactive systems [16], the number of movement errors increase as the latency increases. An example of this is when a tag is moved, and the tag first sends a new position after 3 seconds, then the movement error could be high as the user does not know their exact position for these 3 seconds. According to experiments with VR, the motor performance and sense of body ownership start to decrease at latencies above 75 ms.
This means that the optimal results from these experiments would indicate that the Pozyx system can send position updates fast enough which allows the system to operate with a latency of less than 75 ms between their physical movement and the in-game reflection of this movement.

### 2.7.2 Setup

The tags were set to transmit with the highest bitrate with the longest preamble length and with the ranging mode set to precise [3].
According to the documentation, these values lower the tags update rate to about 9hz which is then divided by the number of tags used in the system as described in subsection 2.4.4. As the main goal of the experiment was to test the accuracy these settings were chosen to gives the most accurate positioning. The settings can be changed at a later point to try to increase the update rate to a level where the latency for the users is acceptable.
The experiment was set up as shown on Figure 2.11. The experiment was conducted indoors on our campus in the building Novi 9. The anchors `0x632b` and `0x676e` were

mounted on a wall 240 centimeters apart, and the remaining anchors `0x6738` and `0x676c` were mounted on a bulletin board.
The number of centimeters accompanying the hexadecimal number of each anchor is the height at which the anchor was mounted during the experiment. Different heights were chosen as Pozyx documentation suggests that not all anchors should have the same height [27]. The reason for this is the principle of geometric dilution of precision (GDOP), which can cause the error on range measurements to be amplified.

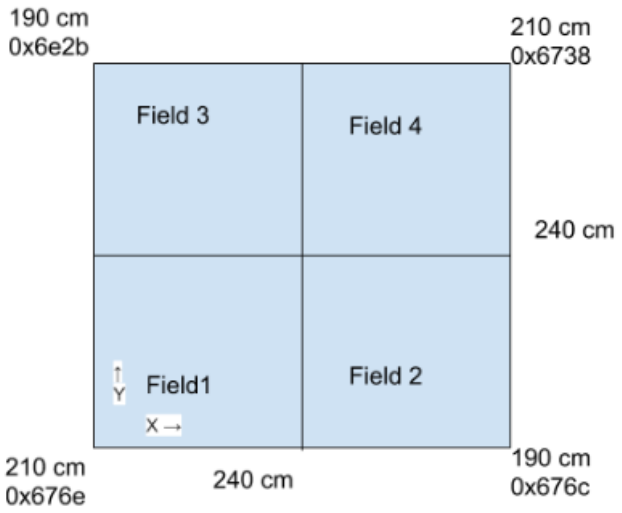


**Figure 2.11:** The setup of the experiment with the anchors and the height at which they were placed in the corners. The hexadecimal number is the anchor, and above that is the height at which the anchors were placed.

The fields were created by a whiteboard on which lines were drawn every 10 centimeters to know the actual position as seen on Figure 2.12. This whiteboard was moved intermittently to act as respectively fields 1, 2, 3, and 4.

**Figure 2.12:** The whiteboard with the drawn positions.

The procedure for the experiment was based on this whiteboard. The whiteboard would be placed in one of the fields defined in Figure 2.11, and the tags would be placed in certain positions on the board and record the accuracy with which the position was reported. The tags would be placed in a position, remain there for five seconds, then be moved to the next position over the next five seconds and remain in this position for five seconds before being moved again. This procedure was repeated until a satisfactory number of measurements had been made. This amount was usually 5 measurements. Once one field had been tested, the whiteboard was moved to the next field, as seen on Figure 2.11, and the process was repeated.

### 2.7.3 Analyzing the data

In section 7.1 the data for the experiment can be seen. For each test the average grid was calculated, and the minimum and maximum values for $x$, $y$ and $z$ were found. The min and max values were noted, as they may be useful for future analysis. The $z$ values were very inconsistent throughout the entire test, and due to this and the $z$-axis being largely irrelevant to the game, we will not focus on analyzing them. Because of this, it was decided to only calculate the deviance between the actual position and the measured position in the $xy$-plane.

#### Precision with 1 tag

When testing with only one tag the average of all deviations was 15.20 cm.

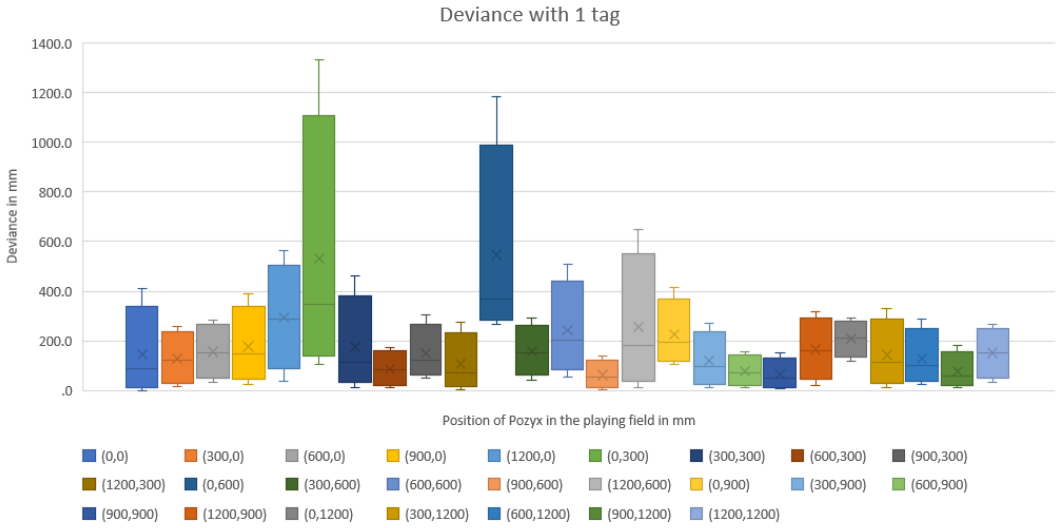


**Figure 2.13:** A box plot showing the dispersion of measured positions with 1 tag

As can be seen on Figure 2.13, some measurements can be quite far off the actual position. On the $x$-axis you can see the actual position of the tag in mm compared to the playing field. The $y$-axis shows the amount that the measurements were off the actual position in mm. Each measurement set has its own box and whiskers. The box shows the measurements within the first quartile and third quartile. Within the box the $X$ denotes the second quartile which is equal to the median and the flat line denotes the average value of all the measurements. Extending from the box is the lower and upper whiskers where the lower whisker shows the measurements below the first quartile and the upper whisker shows the measurements above the third quartile. The end of the lower whisker is the lowest deviance measured in the set, likewise the end of the upper whisker is the highest deviance measured.

Interestingly it is the measurements closest to the $x$-axis that are the most inaccurate, we will discuss why in subsection 2.7.4. The measurement with the highest deviance was recorded when the tag was in position `(0,3000)` where the measured position was 1.3 meters off. The average deviance of all the measurements from that position was only 36.5 cm, however. The highest average deviance of all the positions was 39.5 cm in position `(0,6000)`. When running with one tag the average amount of measurements were 12.368 per second.

### Precision with 3 tag

For this part of the experiment, the accuracy was tested with three tags. The tags were positioned at three different x-coordinates. 1400, 1800 and 2200 were chosen as the $x$ coordinates and the $y$ coordinate was moved up by 300 mm each time starting from 0 and going up to 1200. The average amount of updates from all the tags per second were 1.83, less than 10 percent of the amount of updates when only one tag

was used. The amount of updates were very different between the different tags, as the one with the highest update rate had an average of 3.4 per second and the lowest only 0.92.

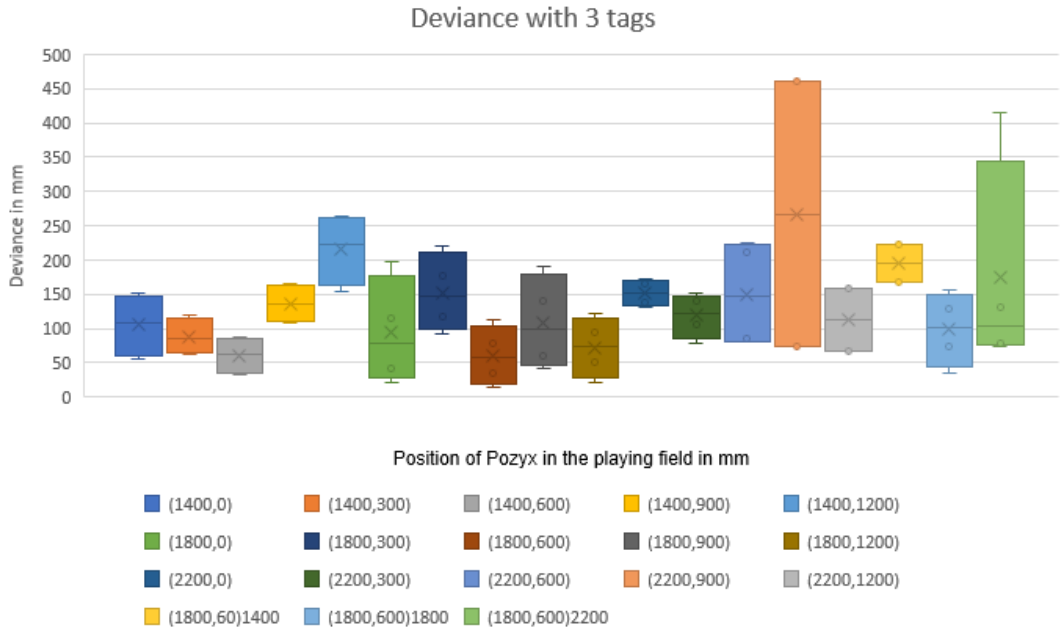


**Figure 2.14:** A box plot showing the dispersion of measured positions with 3 tags

On Figure 2.14 all the measurements from the 3 tag experiment can be seen. The 3 last sets of measurements are from when all the tags were placed close to each other at the same point. The trailing number after the coordinate is which tag the set of measurements belong to, `(1800,600)1400` denotes the tag used for the measurments along the line where $x$ equals 1400.

This was done to see if they would affect the precision of each other. Based on the data it seems they might have a small effect on each other but not enough for it to be a problem. The biggest deviance in the test was only 46 cm off compared to 1.3 meters in the 1 tag test which seemed interesting, but could be because the measurements in the 3 tag experiment were made in the middle of the playing field compared to at the edge in the 1 tag experiment.

**Tag 26467** This tag was positioned 1400 mm along the $x$-axis. The overall average deviation for this tag was 11.9 cm and it made 1.16 measurements per second. No measurements were further away than 26.3 cm from the actual position and the biggest average deviance for a set of measurements was only 22.26 cm.

**Tag 26895** This tag was positioned 1800 mm along the $x$-axis. This tag was only 9.2 cm off on average and had by far the highest average update rate at 3.4 Hz. Like

the previous tag it also did not have any measurements with large deviances. The furthest measurement was 22.1 cm off and the worst average deviance for a set was 15.09 cm. This was by far the best performing tag in this experiment regarding both precision and update rate.

**Tag 24622** This tag was positioned 2200 mm along the $x$-axis. The overall average deviation for this tag was 14.48 cm and an update rate at 0.92 hz. For the first three points the results seemed fairly accurate, however, the grid `(2200, 900)` had a large deviation of 22.43 cm. That set also had the measurement with highest deviance with 46.0 cm.

### Precision with 5 tag

The tags were positioned at five different x-coordinates. 1400, 1600, 1800, 2000 and 2200 were chosen as the $x$ coordinates and the $y$ coordinate was moved up by 200 mm each time starting from 0 to 1200 mm.

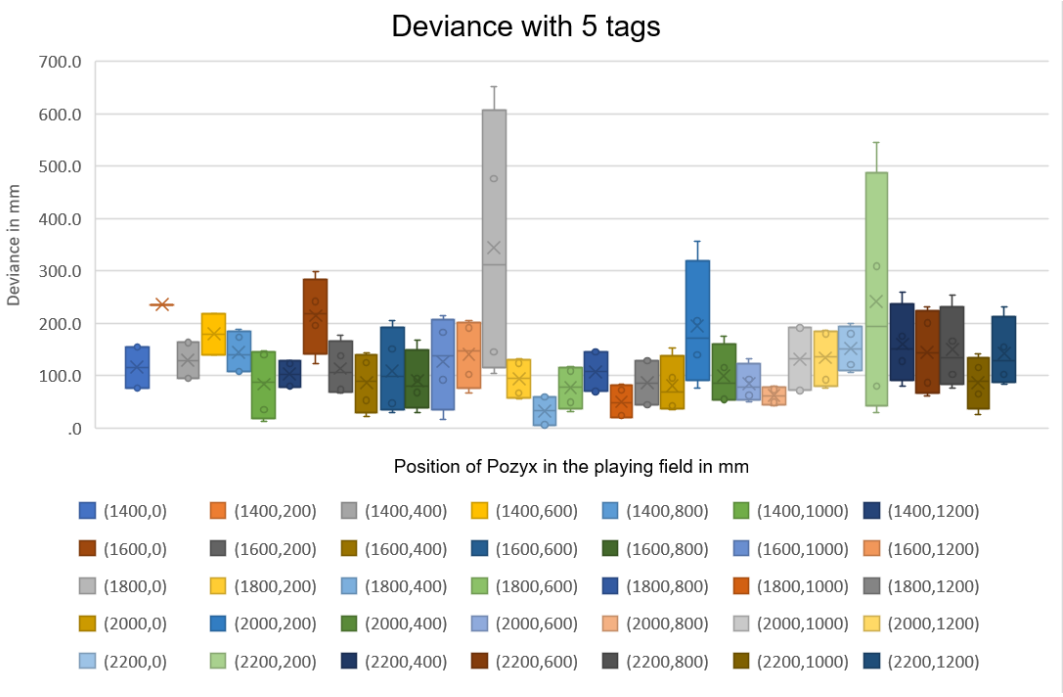


**Figure 2.15:** A box plot showing the dispersion of measured positions with 5 tags

Like the experiment with 3 tags, seen on Figure 2.14, there were no huge outliers like in the first experiment where some measurements were off by more than a meter. But two instances were still seen as further off than the average.

**Tag 24622** This tag was positioned 1800 mm along the $x$-axis. The average deviation for this tag was 12.99 cm and the average update rate was 0.83 Hz. This was the tag that had the worst measurement on coordinates `(1800, 0)` which was off by 65.1 cm and the worst average deviance of a measurement set with 31.7 cm. Besides that the other sets were among the most accurate.

**Tag 26467** This tag was positioned 1400 mm along the $x$-axis. For this tag, the average deviation was 12.75 cm and the average amount of data points per second was 0.6. At `1400, 200` the tag only returned one measurement and at `(1400, 600)` there were only two measurements completed during the 5 second period the tag was in that position. The tag was overall very precise with all measurements within 23.6 cm of the actual position.

**Tag 26895** This tag was positioned 1600 mm along the $x$-axis. Tag 26895 had an average deviation of 12.39 cm and an update rate at 2.11 Hz. This update rate was much higher than tag 24622, 26467 and 27001, and equal to 26901 even though they all had the same settings.

**Tag 26901** This tag was positioned 2200 mm along the $x$-axis. The average deviation of tag 26901 was 14.74 cm, and it had an update rate of 2.11 Hz aswell. This tag had a few spikes when measuring on coordinate `(2200, 200)`, where the average deviance was 20.99 cm and the worst measurement was 54.6 cm off.

**Tag 27001** This tag was positioned 2000 mm along the $x$-axis. The average deviation of this tag was 10.46 cm and the average update rate was 1.23 Hz. This tag was pretty average with no big deviations. Its worst measurement set had an average deviation of 18.34 cm.

### 2.7.4 Possible influences on the test

One thing that could have affected the tags is that a whiteboard was used as a measure for positions. As there is metal in the whiteboard this could have affected the precision of the tags. Metals are conductors, which can lead to the signal having less power and reduced range, and the signal might spend extra time trying to get through the metal. Since Pozyx positioning relies on calculating the time of flight, having the signal spend extra time traveling reduces accuracy [24].

If we look at the $z$ coordinate in the test experiment data, it can be seen that $z$ fluctuates a lot more than the $x$ and $y$ coordinates. The max $z$ coordinate for multiple experiments is often more than 3 meters and the average $z$ coordinate is also often 1 meter higher than the actual coordinate. This could be due to the height difference between the anchors not being sufficiently large which can affect Pozyx' ability to measure the height of the tags.

During the test with 1 tag, the 0 value of the $x$ coordinate coincided with the wall. This could have been an influence on the positioning result in that it could increase uncertainty, as it was difficult to center the tag over the x coordinate since the coordinate collided with the wall. Also, the wall could have created interference with the signal and there might have been metal or wires going through it which could have affected the signal.

While the experiment was running the tags could throw errors instead of giving a position if something went wrong. Mainly during the multi-tag tests, some tags would report back with errors such as unable to get firmware version, flash memory corrupted or an error message saying there was no error. At that point in the experiment, it was not clear whether it was a hardware error with some of the tags or if it was code related. The tags would throw these errors randomly and then measure their position normally at the next update. This resulted in some of the tags having periods of update rates of less than 0.2 Hz. It was decided that further investigation into this issue should be conducted at a later point.

### 2.7.5 Conclusion on the experiment

Our conclusion of this experiment is that the precision is satisfactory, but the update rate is not optimal as that does not meet the standards of the refresh rate needed with the current settings.

We also need to consider the spikes in the coordinates, such that the user does not jump around on the screen, even though the user might not be moving. A good solution would be to use an algorithm to correct the data.

The results of this experiment shows that a three dimensional game is not viable with the errors that we had on the $z$ coordinate. This is most likely due to the height difference between the anchors only being 20 cm. If we, later on, choose to make use of the $z$ coordinate, then it is necessary to conduct another experiment with the Pozyx with different, larger height differences, but as it currently is only a two-dimensional game it is not necessary.

It also seemed like some tags were not able to send as many measurements as others. Consistently throughout the experiments tag 26895 was the best tag regarding update rate, sometimes having an update rate almost 3 times faster than the other tags. This was caused in part due to some tags seemingly having a slower update rate, but also because some of them would report errors instead of measurements randomly. This is something that will be investigated.

## 2.8 Sprint 1 conclusion

This section concludes the preceding chapter on sprint 1. It will conclude what knowledge was gained based on the sections of the chapter, and discuss a retrospective of the sprint as a whole, and what changes were made to the progress for upcoming sprints.

### 2.8.1 Overview of completed tasks

section 2.1 outlined the goals of the sprint. The sprint focused on exploring the project idea. Wireframes were constructed insection 2.2 in order to create a more uniform vision of the project. These generated a vision for how the users would interact with the game, and how the game aspect would look.

In section 2.3 a rudimentary overall architecture for the project was outlined. We defined an architecture in which data would be transferred from the Pozyx components to a host and then to the players of the game. section 2.4 gave an introduction to the Pozyx technology to be used for the tracking aspect of the project. Several alternatives for finding anchor locations were proposed, and UWB was defined as the best solution. section 2.5 gave an introduction to the game engine to be used for building the project. section 2.6 explored the possibilities for communication between the different parts of the architecture, settling on using a custom UDP based solution. Finally, section 2.7 defines an experiment conducted in order to determine the accuracy of the Pozyx system, concluding that the accuracy was acceptable for the use of this project.

### 2.8.2 Retrospective on the process

Initially, the project made use of a more scrum-inspired process, which consisted of smaller sub-sprints of one week composing the larger sprints defined as sections in this report. Each week had a backlog of tasks to be completed that week, evaluated on a story point-based system. During this sprint, this was changed to the process outlined in 1.3. The shorter sprints were phased out, and instead, a more constant backlog was kept, from which to choose tasks. We held a retrospective on this process at the end of the sprint. The developers liked not having the smaller sprints and decided to do reviews in pairs going forward. To ensure pull requests would be reviewed quickly, a rule to check for pull requests every morning was set in place. The waypoints were assigned to tasks in the process was questioned, as the way it was done lead to some bias towards tasks that took a short amount of time to complete. It was decided that creating a weight to influence the tasks in a way to avoid this bias was a good idea.

# Chapter 3

# Sprint 2

## 3.1 Sprint goal and introduction

The goal for this sprint is to explore the networking side of the project and implement initial versions of the UDP client and server based on what was learned in section 2.6. Additionally, the first version of the game should be created, which should allow players to see the game through a VR headset and show the players moving based on the positions of the tags.

## 3.2 Deployment Diagram

In section 2.3 the different components of the system were introduced in Figure 2.5. A deployment diagram is constructed in this section in order to further elaborate on the different components.

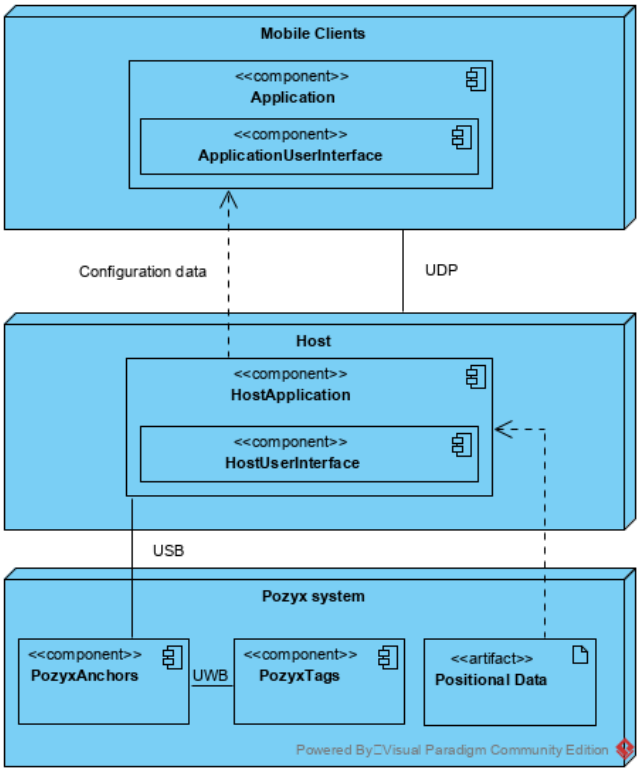


**Figure 3.1:** A deployment diagram for the system.

A deployed system will contain three nodes:

- Pozyx system
- Host
- Mobile Clients

Nodes are represented by cubes, and are entities that execute components. The Pozyx system node contains two components. These are the anchors and the tags needed to generate positional data with the Pozyx hardware. The system contains multiples of anchors and tags. These Pozyx components generate artifacts in terms of positional data for the location of the players and the ball. Anchors and tags are associated through a UWB connection in order to generate these artifacts. The host node is dependent on the positional data artifact, as the component receives the positional data and transforms it for communication. Positional data is transferred from the Pozyx system to the host through a USB association. One anchor is associated with the host application, and this anchor is responsible for collecting data from all tags. The host application component contains a user interface component, which is the interface that the person using the host component will interact with, in the form of a console application for the early version of the system. The host node is

associated with the clients through a UDP connection in order to communicate both the positional data and the configuration data to perform game setup. As such, the mobile client application component is dependent on the host application component, since it requires the configuration data. The mobile clients also contain user interface components, which is how the users interact with that part of the system. This user interface is the virtual playing field generated in Unity that the users view in the headset.

## 3.3 Accessing games on the network

Different methods of transmission were discussed in subsubsection 2.6.5, and multicasting was selected as the optimal solution for this project. In order to make proper use of this in an eventual deployment of the game, it would not make sense for the users to input the IP address to which they want to connect. As such, a way to access hosted games without this needs to be implemented. In order to support multiple games being played at the same time, the system also needs to properly make use of multicast groups based on the game being joined. Clients should only receive data from one specific host in their specific multicast group.

To do this, the game must include some form of LAN discoverability. One way of achieving this is to have clients broadcast a message to available hosts via the LAN when they are searching for a game. The hosts then reply, if they are available, with data for a multicast group, and the client can then join that group. Another method could be to make use of Internet Group Management Protocol (IGMP), which is a communications protocol used on IPv4 networks to establish multicast groups. The IP address 224.0.0.1 is a notable IPv4 address reserved for IP multicasting, which is the multicast group address for all hosts on the same network [14]. All hosts should join that group on start-up, and clients could then message the group to signal that they are looking for an available game.

Another way would be to have the host continually broadcast messages that it is available for players. This would, however, lead to the host needing to repeatedly send messages as long as the game it is hosting has available slots for players or has not started. This would likely be a less elegant solution than having players that are searching for a host sending the message. Having the players send a message to the multicast group address for all hosts is the preferable solution, as it will avoid unnecessary overhead caused by broadcast messages being sent to unrelated machines on the network.

Another option is to make use of the Zero Configuration Networking (Zeroconf) protocol. Zeroconf is a protocol that allows for browsing of available services. This can be used to facilitate IP multicasting in a way relevant for the game, where the host could make the service available and send the messages to all interested clients.

Zeroconf selects an IP address within a relevant range, and then claims that address if available. It will then create a name for the host device, such that it is not necessary to remember and type numerical addresses. Services can then be made available for discovery [2].

In order to select a game to join as a client, the game would need a game selection screen. Figure 2.2 and Figure 2.3 show the initial prototypes for joining a host based on their IP and waiting for the game to begin.
If a lobby were to be implemented, Figure 2.2 would need to be updated, and there would need to be an extra step before Figure 2.3 could be shown. Figure 3.2 and Figure 3.3 show new prototypes for this purpose. Choosing a lobby on Figure 3.3 would lead to Figure 2.3.

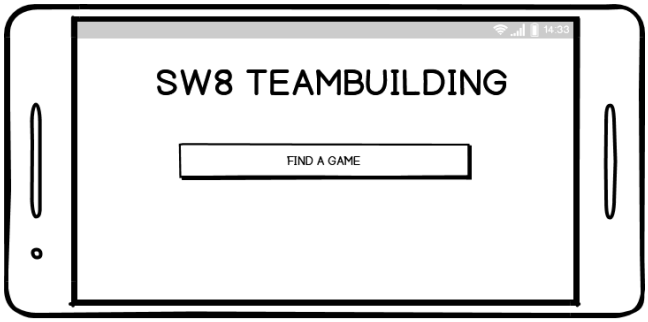


**Figure 3.2:** Prototype of the game menu when players can choose a game to join.

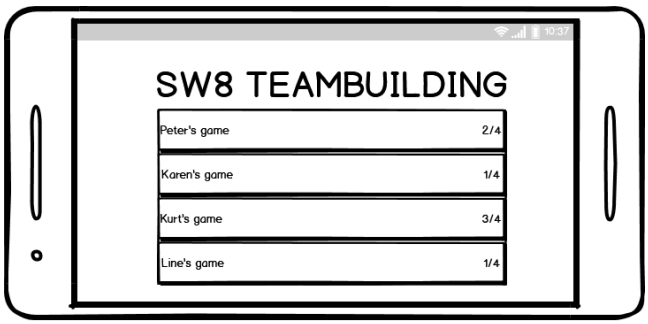


**Figure 3.3:** Prototype of a lobby menu. Available games are shown as well as the number of players. Players can choose one to join.

## 3.4 UDP implementation

In this section, we will delve into the implementation details of the UDP client and host. As described in section 3.2, the system will consist of a host computer which receives positional data from Pozyx and transmits it to the players using IP multicasting.

### Host

The main functionality of the host is to transmit positional and game state data from the Pozyx tags to the clients via multicasting. The first setup step of this is to connect the host to an IP address on the network dedicated to multicasting, as described in section 3.3. For the time being, this is hardcoded to be `224.3.29.71:10000`. After this initial setup, the host will continuously run through a loop where it updates the ball position, player positions and increments the timestamp. Each update consists of three steps: Getting the positional data from Pozyx, transforming it to fit the format described in subsection 2.6.7 and transmitting the data to the clients, as seen on Listing 3.1.

```
1 def update_ball_position(self):
2     """Broadcasts the updated ball position"""
3     ball_tag = self.setup.ball_tag
4     position =
          self.multi_tag_positioning.get_position(ball_tag)
5     message =
          self.formatter.format_player_position(self.time_stamp,
          0, position.x, position.y)
6     self.multicast_sender.send(message)
```

**Listing 3.1:** Updating ball position

### Client

Like the host, the client joins the multicast group on the specified IP. While the host continuously sends positional data, the client's job is to receive and react to the data that is being sent to the multicast group. This is done by creating an asynchronous callback, which ensures that the data receiver is called whenever data is being transmitted, which is seen on the first three lines of Listing 3.2. Once the `Receive` function is called, it saves the received bytes to an array and starts listening for new data before it starts working with the data. This was done to ensure that new messages would not be blocked if it takes too long to handle the data. In the current version, the receiver will simply log the data to the Unity console, but in future versions, it will be able to call the appropriate functions based on the type of data it has received.

```
1 public void StartListening()
2 {
3     uClient.BeginReceive(new AsyncCallback(Receive), null);
4 }
5
6 public void Receive(IAsyncResult res)
7 {
8     // Represents a network endpoint as IP address and port
          number.
```

```
9       IPEndPoint RemoteIpEndPoint = new
            IPEndPoint(IPAddress.Any, portNumber);
10
11      // Receives the message as an array of bytes, then ends
            communication with the remote endpoint.
12      Byte[] receiveBytes = uClient.EndReceive(res, ref
            RemoteIpEndPoint);
13
14      // Restarts communication again to receive a new datagram.
15      uClient.BeginReceive(new AsyncCallback(Receive), null);
16
17      // The bytes that were received are converted to a string,
            which is written to the unity debug log.
18      string returnData =
            System.Text.Encoding.ASCII.GetString(receiveBytes);
19      datagramMessage = returnData;
20      datagramSender = "Address: " +
            RemoteIpEndPoint.Address.ToString() + ", port: " +
            RemoteIpEndPoint.Port.ToString();
21  }
```

**Listing 3.2:** Receiving data from host

## 3.5 Sprint 2 conclusion

This section concludes the preceding chapter on sprint 2. It will conclude what knowledge was gained based on the sections of the chapter, and discuss a retrospective of the sprint as a whole, and what changes were made to the progress for upcoming sprints.

### 3.5.1 Current product

To give an overview of the progression of the project this section will provide an overview of what has been created during the sprint. This overview will be split into two categories, to reflect the structure of the project: Networking and game.

#### Networking

The network aspect has been a big focus this sprint and has led to a better understanding of how the data should flow through the system (section 3.2), and how the knowledge from sprint 1 about networking can be used to automatically find on-going games on the local network (section 3.3).

From an implementation perspective, this sprint has introduced an initial version of the UDP host which is capable of transmitting Pozyx location data from the host computer to the game clients. For the host, an initial console-based setup has been created, where the user can input the number of players, the location of the

anchors as well as specify which tags are used in the game. Additionally, the host can automatically re-order the list of entered anchors to ensure that they are sent to the clients in a clockwise manner, such that Unity can create a playing field mesh based on the coordinates.

#### Game

For the game aspect of the project, the most crucial parts have been implemented this sprint:
First of all, the game has been set up to support VR glasses by splitting the game view into two halves, one for each eye, as seen on Figure 3.4.

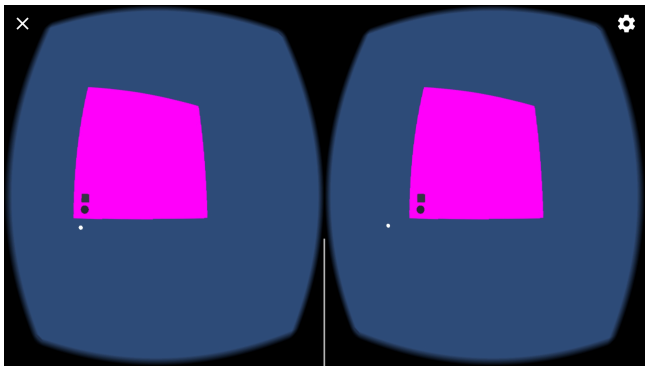

**Figure 3.4:** The first implementation of VR

The game supports moving the players upon receiving new information from the host, but the connection between the two was not coupled together in this sprint. Finally, an algorithm was implemented to generate the goal zones and ensure that they are placed fairly on the playing field.

### 3.5.2 Retrospective on the process

The process required some unexpected pivots in the second sprint due to the coronavirus, which lead to the university being locked down for a period.
The daily group work was moved from the physical group room to a virtual setting facilitated by Discord. One of the major differences was that it was previously possible to sit down and do the pair reviews in the morning, which meant that reviews could be completed quickly and easily. While we tried to still get the pair reviews done first thing in the morning, there was a noticeable delay before the new features were reviewed and merged. Another minor change was how we assigned points (cost, reward, priority) to tasks. This worked out fine online, as people could simultaneously post their chosen number in a chat, as opposed to showing a number of fingers while sitting around a table.

Since most of the project work is managed on Jira and completed by individual

group members, there was not a noticeable change in the amount of work getting done. A side effect of working with only voice chat was occasional difficulty in reaching the other members when their help was needed since they might be temporarily away from the computer, or too distracted or focused to be paying attention to the voice chat. This also resulted in some information not getting spread to the entire group, as not everyone may have been listening while a discussion was happening, which led to minor confusion in further discussions on the subject.

Like at the end of sprint 1 (section 2.8), a retrospective was conducted by the anchor to reflect upon how the process was working out. In addition to discussing the progress of the project, the following points were brought up:

### How does the process with Jira work?

Currently, it seems like the challenger is the only person adding suggestions to the backlog, after some discussion it turned out that multiple members thought that this was intentional, and did not know that they were also supposed to contribute with their suggestions for the project. This has been clarified, and all members are now aware that they are fully encouraged to add suggestions to the Jira when they come up with ideas for the project.

### Should we use pair programming more?

Right now, the only work done in pairs is the pair reviews. It was decided that utilizing pair programming for larger programming tasks would be beneficial to decrease the amount of time spent on it and allow for more perspectives on implementation. Whether or not a task should be done with pair programming will be decided as a part of the task discussions that take place after the daily standups if there are new tasks in the suggested column.

### How do we give better estimates about when a task is done in daily standups?

Generally, the biggest reason that it is difficult to estimate how much remains of a given task, is that the definitions of done are not precise enough. To combat this, it was decided to remove the prioritization of tasks where we would assign a reward, cost, and priority. Instead, the discussion will be about what a good and specific DoD is for the given task to ensure that everyone is on the same page. The hope with this approach is that discussing the DoD will give new perspectives to a given task, and shift the focus from how long it will take to complete the task to exactly what needs to be done. A possible by-product of this shift of focus is that the in-depth discussion will result in discovering new tasks that need to be completed.

### How do we feel about daily standups?

Everyone is generally pleased with the daily standups, but there is a tendency to focus more on what has already been done rather than what is currently being worked on. To prevent this, a new format for the meetings has been proposed:

- What have I been doing? (short)
- What am I working on now?
- When will my task be completed?
- What is my current challenge?
- Do I need help or reviewers?

After each member has presented these four points, we will go through the new tasks in the suggested column and define a DoD for them. Finally, we went through all tasks that had been completed in the sprint to ensure that everything that had been implemented was also documented in the report, to ensure that the knowledge is shared between all members of the group.

# Chapter 4

# Sprint 3

## 4.1 Sprint goal and introduction

The main goal for sprint 3 in terms of implementation is to achieve a minimum viable product(MVP). To do this, the game should be able to receive positional data from the Pozyx system. This data should then be formatted and used to update the positions of the players. On top of this, the host should be able to calculate when a player has entered the opposition's goal with the ball, and notify the players that a goal was scored. Finally, a win condition should be implemented such that a game can end, such as one team reaching three goals scored.

## 4.2 Uppaal modelling

To illustrate how the UDP protocol should work in detail a Uppaal model was constructed consisting of two templates: The host and an arbitrary amount of clients. If you are not familiar with Uppaal, please refer to section 7.4 as a quick start guide on how to read the models. This first iteration of the model is not intended for heavy model checking, but rather to take a higher-level look at how the protocol is intended to work. The two templates can be seen on Figures 4.1 and 4.2.

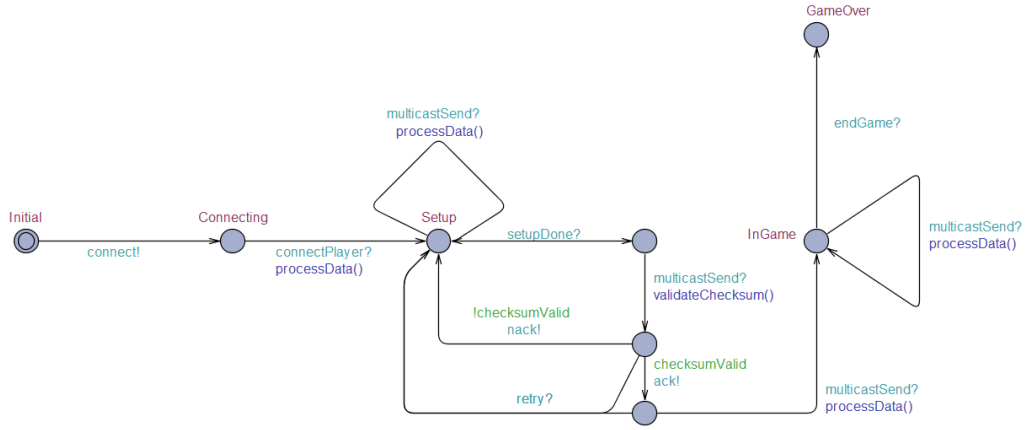


**Figure 4.1:** First iteration of the UPPAAL client template.

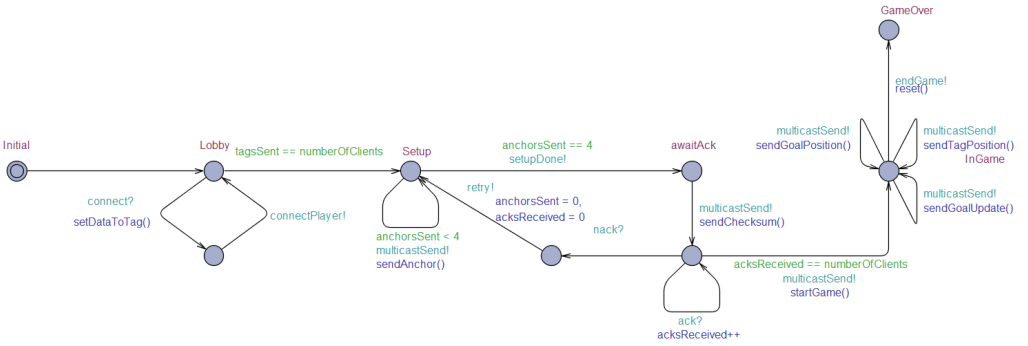


**Figure 4.2:** First iteration of the UPPAAL host template.

### 4.2.1 Walkthrough of the model

The model for the client is fairly straight forward, it starts in an `initial` location which indicates that the client is not yet trying to connect to the host. The connection happens on the edge from the initial location to the `connecting` location. In the model, this is represented with the client utilizing the channel `connect`, and waiting for the host to synchronize by listening to the same channel. When the synchronization happens, the host will execute the *`setDataToTag()`* function, which updates the data it is transmitting, and uses the channel `connectPlayer` to inform the client that connection is now ready. The client will call a function *`processData()`* to read the content of the buffer and react accordingly. These steps will happen until the amount of `tagsSent` is equal to the number of clients specified in the configuration.

### Setup

The next location for both the host and client is `Setup`, which is the point where the host will transmit the anchor positions and ensure that all clients have consistent data. The host will attempt to send the anchors one at a time with the *`sendAnchor()`* function until it has sent all 4 anchors to the clients. Unlike the `connect` and `connectPlayer` channels, this happens on a broadcast channel, which allows multiple clients to receive the data at once to simulate the use of multicasting. Since there is a chance that not all clients have received the four anchor coordinates correctly, the host will broadcast a checksum and have the clients either acknowledge or negative-acknowledge that the checksum is equal to the checksum of the data they have received. In case any client returns a negative-acknowledge (using the channel `nack`), all clients will go back to the setup and receive the anchor coordinates again until all clients send acknowledgements.

### InGame

Finally, both the client and host will go to the `InGame` location to imply that the game is now happening. To simulate asynchronous tasks, the host will continuously take a non-deterministic choice between sending a goal position, a tag position, a goal update or ending the game. Meanwhile, the client only has two choices: Process data received on the multicast or wait for the game to end.

### Code aspect

Behind the model, there are a series of variables and functions to make it all work. In the global declarations the number of clients is specified, as well as an integer array called data, which is used as a buffer to transmit data between the host and clients, to simulate the internet connection in the real implementation. The array has space for 5 elements to conform to the data format specified in subsection 2.6.7, such that the first element in the array will always be the type, which allows the client to act based on that when *`processData()`* is called. The concrete implementation of *`processData()`* can be seen on Listing 4.1. For this iteration of the UPPAAL model, updating player and goal positions are not implemented, since it did not seem like a significant detail to the model.

```
void processData(){
    int type = data[0];
    if(type == 0){
        int anchorId = data[1];
        anchorsX[anchorId] = data[2];
        anchorsY[anchorId] = data[3];
        anchorsReceived++;
    } else if(type == 1){
        // Player position
```

```
    } else if(type == 2){
        // Goal position
    } else if(type == 3){
        // Score update
    } else if(type == 4){
        // Tag received
        tag = data[1];
        playerId = data[2];
    } else if(type == 5){
        // Setup checksum
    } else if(type == 6){
        // Start game
    }
}
```

**Listing 4.1:** Processing Data in UPPAAL model

Likewise, when the host has to send data, it will update the first element of the buffer and fill in the relevant data in the other elements.

### Checksum

A checksum will be calculated on the client for checking whether or not the client has the correct setup information, and based on the checksum that the host broadcasts, the players will return either an acknowledge or negative-acknowledge.
This checksum is simply calculated as the sum of all four anchor coordinates, as seen in Listing 4.2.

```
void sendChecksum(){
    int checksum = 0;
    int i;
    for(i = 0; i < 4; i++){
        checksum += anchorsX[i];
        checksum += anchorsY[i];
    }
    data[0] = 5;
    data[1] = checksum;
}
```

**Listing 4.2:** Calculating checksum in UPPAAL model

## 4.2.2 Updates to the network protocol

A good side-effect of generating the UPPAAL model was that it required some reflection upon the network data format specified in subsection 2.6.7. This resulted in

some minor refactoring of the format.
First of all, the format for sending a field anchor position had a timestamp as a part of the package. Since the anchors will only be sent once, it does not make sense to timestamp it so this has been removed. The updated version of the data format can be found in section 7.3.
Additionally, three new packages have been specified: Sending a player tag, sending an acknowledgement or sending the signal to start the game.

| PLAYER_ID (p) (1-4) | TAG_ID (i) (0-65.535) | TYPE (t) |
|---|---|---|
| 8 bit | 16 bit | 8 bit |

**Table 4.1:** Format sending player tag.

| ACK (a) (0-1) |
|---|
| 8 bit |

**Table 4.2:** Acknowledge or negative-acknowledge.

| TYPE (t) 6 |
|---|
| 8 bit |

**Table 4.3:** Game start.

The reason that Table 4.2 does not have a type is that it is the only package that will be sent from the client to the host, so it will only need to contain a signal for the host to know whether the checksum is correct or not.

## 4.3 Dead reckoning

The accuracy of the representation of the players as well as the ball is important for the game to function well. Since these positions are sent over the network, they might not always be the most recent positions, since there could be delays or packages could be lost.

Dead reckoning is a technique that is used to predict where a game object is at a given time based on its last known position, velocity and acceleration. It is often used for networking games where packages about the object's kinematic state are continuously being sent from the server to the clients. The kinematic state of an object includes its position, velocity, acceleration, orientation and angular velocity [17]. If the client misses a package or they are not being sent often enough to have a

new update for each frame, the representation of the object will jerk across the screen instead of having smooth and consistent movement. That is where dead reckoning can be used to predict an objects movement to make it appear more believable. For the game we are building, it is important that each player's and the ball's positions are very accurate for the game to function properly.

It is not possible to have a new update for each frame, zero packet losses or zero latency. Therefore dead reckoning is needed to achieve a believable representation of the ball and the players' movement [17].

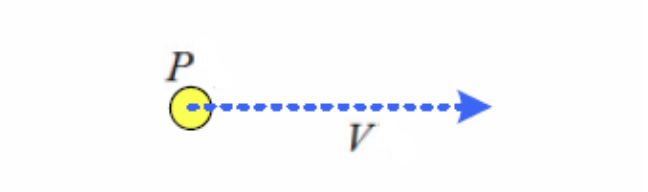


**Figure 4.3:** Linear player movement.

Figure 4.3 shows an example of a player's movement in the game. This movement proceeds from an initial position $P$ along a vector $V$. In this example, dead reckoning would be a linear problem where the position, velocity and acceleration can be used to predict where the player will move to in the future. The dead reckoned position for a specific time $Q_t$ in this example can be calculated by:

$$Q_t = P + VT + \frac{1}{2}AT^2$$

where $P$ is the position, $V$ is the velocity, $A$ is the acceleration and $T$ is the time [17].

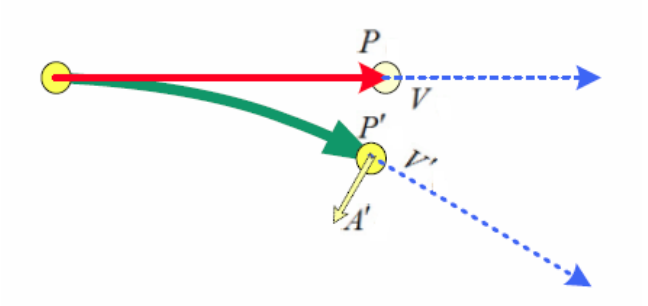


**Figure 4.4:** A new update about the player's state is received from the server.

In Figure 4.4 a new update about the player's kinematic state is received. Initially the player position was calculated through dead reckoning based on $P$ and $V$ as following the red arrow. However, the new update shows a right turn, along the green arrow,

denoted by $P'$, $V'$, and an acceleration in the direction of the turn, $A'$. This results in conflicting realities when compared to the dead reckoned position calculated from the previous example. In the new update the player has turned right and a new dead reckoned position must be found, but this time it becomes a lot trickier since a believable curve must be created from where the player was thought to be and where the player is estimated to be based on the new information. Representing the new position $P'$ immediately would result in the player warping across the screen which would not be ideal. Instead, the player is represented at $P$ where the player was thought to be, and the player should then move towards a new dead reckoned position, calculated with the information from the new update [17].

### 4.3.1 Projective Velocity Blending

Calculating the curve that the player's movement needs to follow requires a good algorithm that is not too CPU intensive. The algorithm must work well for a segment of a curve that passes through two points, those being the player's current position $P$ and the estimated future location $P'$. The recommended approach for this problem is projective velocity blending [17].

Projections are created for the current kinematic state and the last known kinematic state and then these are blended together.

$$P_t = P + VT_t + \frac{1}{2}A'T_t^2 \quad \text{(Projecting from last dead reckoned position)}$$

$$P'_t = P' + V'T_t + \frac{1}{2}A'T_t^2 \quad \text{(Projecting from last known)}$$

$$Q_t = P_t + (P'_t - P_t)\hat{T} \quad \text{(Combination of the two)}$$

Here, $T_t$ is how much time has elapsed since $T_0$ which is the time at the last known position, which would be the beginning of the curve. $T_1$ is the time when the player reaches the end of the curve. $\hat{T}$ is a normalized time value where $0.0 \leq \hat{T} \leq 1.0$ and should represent how far along the curve the player is.

From these calculations we get the dead reckoned location $Q_t$ at the specified time. The reason that $A$ is used in both projections is that it will converge to the player's true path much faster and it reduces oscillation, compared to when using $A$ when calculating $P_t$. However, these calculations will still give inadequate results since the player's movement will have bad oscillations. These are caused by the changes in velocity, $V$ and $V$, when new updates are received from the server. To account for this, a linear interpolation between the old velocity and the last known velocity is computed, which creates a blended velocity $V_b$. This velocity is used in the projection from where the player was, when a new update is received. This is what is known as

*projective velocity blending* [17].

$$V_b = V + (V' - V)\hat{T} \quad \text{(Velocity blending)}$$
$$P_t = P + V_b T_t + \frac{1}{2}A'T_t^2 \quad \text{(Projecting from last dead reckoned position)}$$
$$P'_t = P' + V'T_t + \frac{1}{2}A'T_t^2 \quad \text{(Projecting from last known position)}$$
$$Q_t = P_t + (P'_t - P_t)\hat{T} \quad \text{(Combination of the two)}$$

This should reduce the oscillations in the player's movement significantly.

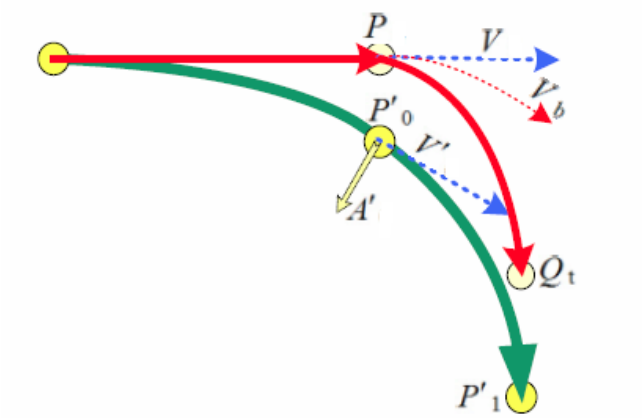


**Figure 4.5:** Shows the the curve for the dead reckoned position $Q_t$ when making use of projective velocity blending.

Figure 4.5 shows the red curve that the player should follow to reach the dead reckoned position $Q_t$. $P$ is where the player is currently represented. $P'_0$ is a recent position received from the server, and $P'_1$ is the most recent position received from the server. This illustrates how the dead reckoned position $Q_t$ converges on the actual positions, shown through two different updates of the position. The green curve is the one that the player actually follows, while red is the dead reckoned curve.

This technique would be ideal to implement in the game if we find that the representation of the players' movement is inconsistent and in need of improvement.

## 4.4 Goal generation

The first implementation of goal generation was done entirely in Unity. This meant that the goal generation and detection was handled on the client-side. The idea was to limit the amount of data that needed to be sent across the network. It was then decided that new goal zones should be generated in random locations after a goal

had been scored, and by generating it on the client-side it could cause the game state to be different between different players. Issues could be caused by players not receiving all the necessary ball positions to observe a goal or by generating goal zones in different places. One way to solve this was by giving all clients the same seed at the start of the match to generate random positions so that the goal zones would be in the same positions across all clients. The concern about this solution was that the goal detection could still have issues if some of the positional data for the ball would not arrive at the same time or at all. In the end, it was decided to move all the goal zone generation and goal detection to the server-side and utilizing TCP so that the data would be consistent across all clients.

### 4.4.1 Server-side

On the server-side, the size of the goal is calculated based on a percentage of the shortest edge of the playing field. This size is used to calculate where the center points of the goal zones can be placed as the corners of the goal zones should not exceed the playing field edges. The goal zone will always be a square where all edge lengths are of equal length and the playing field should always closely resemble a rectangle. The first goal zones are always set at a certain point on the playing field, after which the goal zone positions are randomly generated. The randomly generated goal zones will always mirror each other and will not spawn too close to the middle of the playing field. The goal zones will also never change sides so if a goal zone is initially generated on the left of the center of the playing field it will stay on that side for the duration of the game. Whenever a goal is scored, new center points are generated for both goal zones and then sent from the server to all connected clients.

### 4.4.2 Client-side

On the client-side, the new center points are received along with the goal size. Based on this data, the corners of the goal zones can be calculated. The corners are calculated in clockwise order. This is done to make it easier to apply the mesh which is the visual component in Unity that renders the goal zone so that the players can actually see it. The mesh can only be rendered in triangles and the mesh is only visible from one direction, meaning if you see the mesh from the back it will be invisible. Because of this, each goal zone has to be split up in two triangles that together create a square, and the corners of the triangles need to be in the correct order or the goal zone can not be seen from the players' perspective.

### 4.4.3 Goal zone package

To support goal zone generation on server-side we introduced a new package that can be seen on Table 4.4. To create the goal zone on the client side the center position is needed, which is the `POSY` and `POSX`. We need this to be 16 bit, as 8 bit would only allow us to give a coordinate ranging from 0 to $2^8 - 1$, whereas 16 bit will increase

the range to 0 to $2^{16} - 1$. The offset has been chosen to be 8 bit instead of 16 bit. With 8 bit for the goal zone offsets, goals can not be larger than 255 cm x 255 cm, but we decided that it is a reasonable size. Alternatively, the size could be increased to 16 bit, resulting in the goal zones having a maximum size of 655.36 m x 655.36 m, which could give the user room for customization to have much larger goal zones, but we deemed goal zones larger than 255 cm x 255 cm unnecessary for the time being.

| Offset (o) | POSY (y) | POSX (x) | TEAM_ID (i) | TYPE (t)<br>5 |
|---|---|---|---|---|
| 8 bit | 16 bit | 16 bit | 8 bit | 8 bit |

**Table 4.4:** Format for goal zones.

## 4.5 Sliding window protocol

In subsection 2.6.7 it was decided that the format of timestamps would be an integer from 0-255, where 0 to 20 would be considered as newer timestamps than 235 to 255. However, if all the timestamps from 0 to 20 were lost, there would not be any updates until it has iterated through all other timestamps and back to 0. To find a better solution to this problem the sliding window protocol was researched.

Sliding window protocols are often used where reliable data is required, which means making sure that all data points are received. This protocol uses window sized buffer space, where the window is a fixed number and the sender will send this number of the packets or bytes to the receiver [15]. Whenever the sender has sent the same number of packets or bytes as the size of the window, it waits for acknowledgements that the receiver has received the packets or bytes before sending additional ones. Each packet or byte gets a sequence number and the acknowledgement that is sent back also contains the sequence number. By doing this the receiver can keep track of the packets, remove duplicates, identify missing packets and know which order is the correct one.

On Figure 4.6 there is an illustration of a sliding window, where the window size is three. In the first frame, three packets are sent to the receiver, in the second frame the receiver has acknowledged all three packets, but the first acknowledgement fails. The sliding window for the receiver has moved its window, but as the sender only has received acknowledgements from the second and third packet, it is unable to move its window and instead sends the first packet again in the third frame. In the fourth frame the first packet has been acknowledged, and therefore the sliding window is moved for the sender and the sender is now able to send new packets.

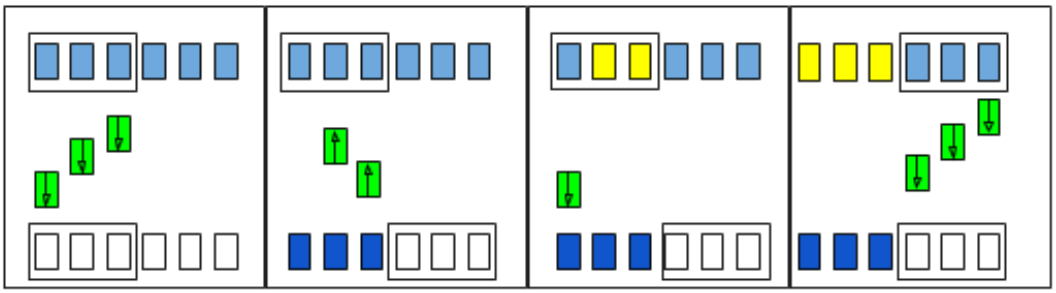

**Figure 4.6:** Sliding window protocol illustration where the top is the sender and the bottom is the receiver.

Reliable data in the format of goal zones and goal score is important for this project, however the reliability of player positions are not important. The reasoning for this is explained in subsubsection 2.6.4. With the `Sliding Window` protocol a player position would be retransmitted if it previously failed, and this position would now be outdated because the player might have moved since then. This means that a sliding window protocol could be used for the game critical aspects transmitted over a UDP connection, such as the format of goal zones and goal scores.

## 4.6 Sprint 3 conclusion

This section concludes the preceding chapter on sprint 3. It will recapitulate the knowledge that was obtained based on the sections of the chapter, and discuss the retrospective meeting that was held at the end of the sprint.

### 4.6.1 Current product

To give an overview of the progression of the project this section will provide an overview of what has been created during the sprint. This overview will be split into two categories to reflect the structure of the project: Networking and game.

#### Networking

The major focus for the network part this sprint was to generate an initial Uppaal model to visualize the flow of data. This initial model is based on the assumption that packages are not lost and timeouts will not happen for simplicity. Creation of this model lead to some revision to the network protocol as described in subsection 4.2.2. In addition to the Uppaal model, the ideas behind dead reckoning were examined in relation to this project. The technique was deemed relevant to implement if the representation of the players' movement is inconsistent, which will be delved further into when the game is connected with the actual Pozyx location data.

### Game

The biggest change regarding the game this sprint was the addition of goal zones. The goal zones were previously generated on the client, but to ensure a consistent game state across all clients, this has been moved to the host and will be transmitted over the network instead. This meant that the major focus for the game in this sprint was refactoring how goals were displayed, as described in section 4.4.

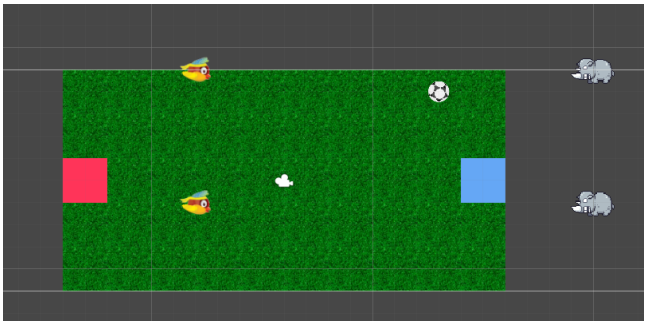

**Figure 4.7:** The current game status with goals and textures. This is a work in progress photo taken in Unity. The green rectangle is the playing field, and the smaller squares on either side are the goal fields. The birds and rhinos represent players, and the ball represents the ball being used for play. The white camera in the center is a Unity representation of where the camera is placed.

Additionally, some temporary textures have been added to the game to get rid of the pink playing field and square players. These new textures can be seen on Figure 4.7.

## 4.6.2 Retrospective on the process

The ideal outcome of this sprint was an MVP of the product, but unfortunately, this was not achieved due to various problems. First of all, there was a series of problems with merge conflicts in auto-generated files in Unity, which lead to the agreement that only one pair of group members should work on Unity at a time to avoid unsolvable merge conflicts. This lead to a major delay in the Unity development.
Additionally, due to the COVID-19 lockdown, we did not have an opportunity to test the system properly with external users to get their feedback on the current product, as well as what to focus on for the last two sprints. In addition to discussing the progress of the process, the following points were brought up:

### How does the process with Jira work?

The process has generally been improved since the last retrospective, not having to spend time assigning priority points to every task is nice. The challenger expressed that he would appreciate if more people contributed to the backlog suggestions since it can be difficult to continuously come up with new tasks.

### How has pair programming been working?

Implementing the pair programming practice has been good for productivity. However, it has been hard to coordinate at times, since it requires two people to be free at the same time for a pair programming task to properly start. In the future, a single person should start with the development and when someone is available to join at a later point they can do so.

### How is the daily stand-up working?

While the new format proposed in the last sprint's retrospective was effective for a few days, it was quickly forgotten. To keep the stand-ups effective, the anchor should be better at enforcing the format to make sure that everyone knows who needs help, and what people are currently doing.

### How are the reviews going?

The pair review format for code reviews does not seem to be working too well since it is usually procrastinated until the next morning because one of the two reviewers is busy with something else.
The single-person reviews are also lacking at the moment, where people forget to check what they are assigned to. It is suggested that people regularly go to the front page of GitHub and check the "Recent Activity" which shows the pull requests you are assigned to. It is also suggested that it is possible to ask people to review your code or text right away if it is important that it gets merged. This is mostly if the task is blocking other tasks and thus is of great importance.

### Closing comments

Finally, we went through all the tasks that had been completed in the sprint to make sure that everything that was deemed important was described in the report. It worked well since it helped create an overview of the status of the project and figure out what to do next.

# Chapter 5

# Sprint 4

## 5.1 Sprint goal and introduction

As a fully functional MVP was not achieved for sprint 3, the main goal for sprint 4 in terms of implementation is to finish such an MVP. The main lacking feature from the sprint 3 iteration is a lack of functionality to move the player as Pozyx data is received, as well as scoring goals and winning the game. These functionalities form the basis of the sprint 4 implementation goals. Another goal for sprint 4 is to perform an initial test of the system in a real setting, in order to discover potential issues that can revised in the final sprint.

## 5.2 Updating the networking protocol

During implementation of the networking protocol outlined in section 4.2, problems were encountered when attempting to implement acknowledgements in the UDP based connection for game critical data. Section 4.2 defines the way the protocol was envisioned to work based on the choice made in section 2.6, where UDP sockets were selected as the type to be used for this project. With the unreliability of UDP, and the importance of certain data being transmitted to players, it would be necessary to implement an acknowledgment from clients that they had received crucial data relating to the setup of the game, as defined in section 4.2. During the implementation of this, issues were encountered when sending the acknowledgment message from the client to the host. As the sockets should make use of multicasting, as defined in subsubsection 2.6.5, it seemed as though the clients should just be able to send messages to the same multicast group, and have the host receive that message, with a type that specified that the message was an acknowledgment that could be either positive or negative, structured in the same way as described in subsection 2.6.7. However, the host had issues receiving the messages. This prompted a discussion regarding how to remedy this issue, as a new approach seemed necessary. Three main alternatives were considered:

- Split the networking into two distinct phases - setup and in-game data
- Expand the messages being sent to include the crucial data on all messages sent
- Introduce a TCP element to make use of its reliability for game critical data

### Splitting networking into two distinct phases

This approach aims to create a clear delineation between the crucial data that requires acknowledgment from clients and the data for which acknowledgments are unnecessary. This was roughly the original approach, as shown on Figure 4.2, which illustrates how the host would wait in a setup state until acknowledgments had been received from all clients, and then the game would start. This presents issues in terms of the multicast grouping, in which the host was unable to read messages sent through the group. This could be remedied by having a separate multicast group in which clients could send acknowledgments, to avoid issues related to multicasting. While the host was sending out the information, a client could check if it had received the correct information, and if not send out a negative acknowledgment for a while before timing out, to attempt to receive the information again. Eventually all the clients would be sending positive acknowledgments.

However, an issue exists with this approach. When scoring a goal, the goalzones should be moved. This would take place during the phase when the game is played, and receiving the information about the location of the new goalzone is crucial, such that all payers know where to score. As such, the phase split loses some of its purpose, as an acknowledgment would need to be sent from clients during the playing phase.

### Expanding messages

In addition to the data defined in section 7.3, crucial data could be appended to all messages sent during play. This would mean that whenever any message was received, it would include to required information, and no acknowledgment would be necessary as the clients are guaranteed to receive some messages if they are continuously sent. This would have the detriment of increasing the size of each individual package. While the packages are currently not large enough to where this is likely to cause a problem, creating minimal packages is still a worthwhile goal in order to increase the speed of transmission and decoding.

### Introducing TCP

On top of the currently described UDP approach to sending messages to ensure recent data is used for positioning while the game is played, a TCP socket connection between the host and clients could be implemented. As described in section 2.6, TCP

includes reliability. This reliability ensures that every message sent will be received by the client, through built in acknowledgments. This would remove the need for clients sending UDP acknowledgments for game crucial data, and also make the implementation of a sliding window as described in section 4.5 for UDP unnecessary. Because of the reliability, and needing established connections, TCP messages could potentially be slower to transmit. This would be the case if the TCP connection had to retransmit a message. Retransmitting a message involves waiting for a timeout, which incurs the larger overhead in comparison to UDP. As one of the focuses for the project is creating a networking solution capable of continuously updating positional data as quickly as possible, this is not an ideal solution for transmitting the data when the game is being played. We performed a PMI(Plus Minus Interesting) analysis on

| | Plus | Minus | Interesting |
|---|---|---|---|
| TCP | Reliable<br>Built in acknowledgements<br>Two way communication | Slower than UDP<br>More overhead because of reliability and two way communication | For critical messages that concern the state of the game TCP ensures that all players receive the update. |
| UDP | Faster than TCP<br>Less overhead because it is a one way communication | Not as reliable<br>No built in acknowledgements | Because UDP does not ensure that the clients have received the correct packages it can send updates faster than TCP. |

**Table 5.1:** PMI analysis of TCP and UDP

both protocols to outline the strengths and weaknesses of TCP and UDP which can be seen in Table 5.1. As can be seen on the table, both approaches have specific use cases where they are useful and in our case we have use cases for both TCP and UDP. As such, a combination is proposed - TCP for crucial data, and UDP for positional data. The issue described above relating to goalzones can also be remedied through a combination. Rather than send location for the new goalzones while the game was being played and waiting for an acknowledgment, this data could be sent on the TCP connection as well, to ensure it arrives.

### Final choice of updating

Based on the three alternatives to remedy the issues encountered, introducing a separate TCP connection between the hosts and the clients to handle crucial data that requires acknowledgments seems like the ideal choice. It avoids the confusion inherent in the unreliability of UDP requiring manual handling of acknowledgments, and avoids communication back and forth, while allowing fast update times for positional data through the UDP connection. Facilitating these changes would entail a slight change in the deployment diagram shown in Figure 3.1. An association would have to be added between the host and client nodes indicating the new TCP connection, which is shown on Figure 5.1. The dependency between the host and client applica-

tion has also been updated to indicate that the client depends on both configuration and positional data.

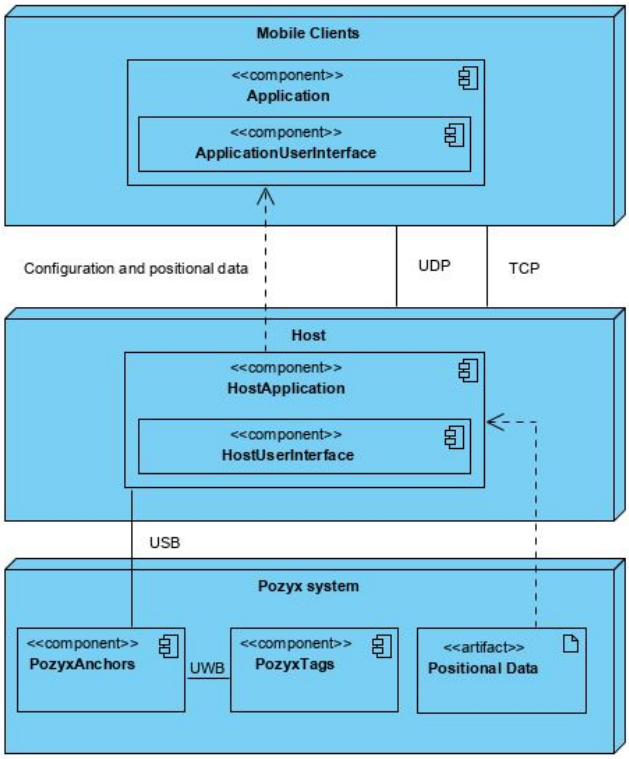


**Figure 5.1:** A slightly revised deployment diagram for the system.

## 5.3 Receiving the information

With the networking protocol in place, let us take a look at how the data is received on the mobile clients. The `in-game` scene in Unity has an `empty object` which handles receiving data from the UDP and TCP clients. Having two separate clients allows them to run concurrently, so the UDP client does not block the TCP client or vice-versa.

### UDP client

Handling the datagrams on the UDP client is simple as it can currently only receive a message with type 0, which is to a message containing a position for a Pozyx tag. The datagram is received in the format `0xYYYYXXXXIISSTT` where 0x indicates hex and the rest of the identifiers are:

- Y : Y position of the tag
- X : X position of the tag
- I : Id of player
- S : Timestamp
- T : Package type

When this datagram is received, we start by parsing the hexadecimal to a long datatype. Since one hexadecimal corresponds to four bits we can get the type of the message, which is the last two hex values of the datagram by typecasting it to a byte as seen on line 13 in Listing 5.1.

```
1  private void DatagramHandler(string datagramMessage)
2  {
3      // Remove 0x from string before parsing
4      if(datagramMessage.ToLower().StartsWith("0x"))
5      {
6          datagramMessage = datagramMessage.Remove(0, 2);
7      }
8
9      long data;
10     byte type;
11     if (long.TryParse(datagramMessage,
           System.Globalization.NumberStyles.HexNumber,
           System.Globalization.CultureInfo.InvariantCulture, out
           data))
12     {
13         type = (byte)data;
14         switch (type)
15         {
16             case 0:
17                 UpdatePlayerData(data);
18                 break;
19
20         }
21     }
22     else
23     {
24         Debug.LogError("Network data could not be parsed");
25     }
26 }
```

**Listing 5.1:** Processing datagrams in UDP client

The next step in processing the data is calling the appropriate function as seen in the switch, in this case we will take a look at the `UpdatePlayerData`.

```
1  private void UpdatePlayerData(long data)
2  {
3      // bitshifting the hex string and typecasting to byte to
           get the values.
4      // see network format in the report for more detail
5      byte time = (byte)(data >> 8);
6      byte id = (byte)(data >> 16);
7      ushort x = (ushort)(data >> 24);
8      ushort y = (ushort)(data >> 40);
9
10
11     if (CheckTimestamp(time))
12     {
13         if (id == 0)
14         {
15             gameStateHandler.ballPosition.x = x;
16             gameStateHandler.ballPosition.y = y;
17         }
18         else
19         {
20             // Player id starts at 1 while the playerposition
                   array is 0 indexed. Decrementing id so that
                   they line up.
21             id--;
22             gameStateHandler.playerPositions[id].x = x;
23             gameStateHandler.playerPositions[id].y = y;
24         }
25     }
26
27 }
```

**Listing 5.2:** Updating player data in UDP client

The next step in processing the data is to read the actual content of the message sent. This is done by utilizing the fact that all parts of the message have a size corresponding to a type in C#. The datagram is read from right to left using the right bitshift operation. When a hexadecimal is right bit shifted 4 bits every decimal in the hexadecimal is moved one decimal to the right as illustrated on Figure 5.2.

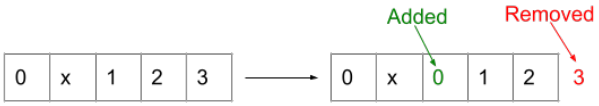


**Figure 5.2:** Right bit shifting a hexadecimal by 4 bits.

This can be utilized to read the different parts of the message that the client received. The first part that is read from the message is timestamp. In the message

the decimals representing the timestamp are followed by two decimals representing the type of the message as seen earlier. Because one decimal corresponds to four bits, the message has to be right bit shifted by 8 bits to move the timestamp to the back of the hexadecimal as illustrated on Figure 5.3.

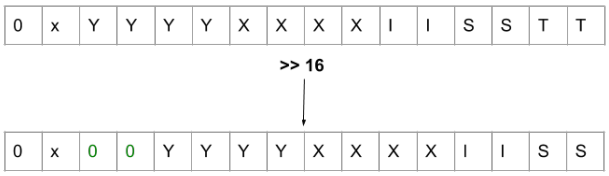


**Figure 5.3:** Right bitshifting the message by 8 bits to get the timestamp to the back.

The message can then be typecasted to a `byte` to extract the timestamp from the package. The next part that is read from the message is the player id. Decimals representing the player id are followed by four decimals representing the timestamp and the type of the package. This means that to get the player id to the back of the hexadecimal the message has to be right bit shifted by 16 bits.

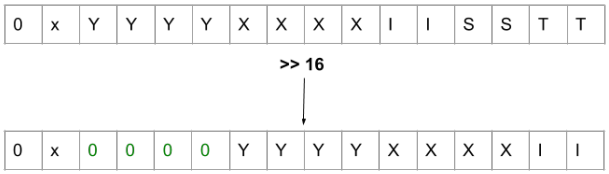


**Figure 5.4:** Right bitshifting the message by 16 bits to get the player id to the back.

Because the player id is represented as two decimals it can be decoded from the message by typecasting the message to the `byte` datatype in C#. This is done for each part of the message where the message is right bitshifted by the combined size of the previously read parts and then typecasted to a datatype that has the same size as the part of the message that is being read.

### TCP client

The principle behind the TCP client is much like the UDP client, where the type is read as a byte and then a switch statement ensures that the data is handled by the correct function. However, since TCP is sent as a stream, all packages have been prefixed with a numerical values to indicate the length of each package. This means that the client cannot simply wait for a new message, read it and act according to the type. Instead, the client will read the first two bytes of the stream and then read the incoming amount of bytes indicated by this numerical value, allowing the program to handle packages of varying length.

## 5.4 Updating the Uppaal model

As mentioned in section 5.2 the approach to handling the network data has changed from being purely UDP to a combination of UDP and TCP. This led to the previous Uppaal model specified in section 4.2 being outdated, and that it had to be updated to fit the new requirements.
Additionally, the previous model focused mostly on the flow of the program and did not properly take the network part of the system into account. In this new model, the system will consist of four parts: The host, which is meant to represent the Python program that the game facilitator uses, an arbitrary number of clients who have connected to the game, and the UDP and TCP connection between those.

### 5.4.1 System declarations

```
system UDP_host, TCP_host, TCP_client, Host, Client;
```

**Listing 5.3:** System declarations

The project consists of five templates: `Host`, `Client`, `TCP_client`, `TCP_host`, `UDP_host`. As mentioned above, the `Host` and `Client` templates are meant to represent the interface that users interact with when using the system. In this new version of the model, the inclusion of `TCP_client`s, `TCP_host`s, and a `UDP_host` allows us to model the behavior of the network aspect of the system.

### 5.4.2 Global declarations

The global declarations (seen on Listing 5.4) for the system have two purposes: Setting game and simulation specific options, such as how many clients should be instantiated, or how long the TCP client should wait for a response before declaring the connection as timed out.
To facilitate data transfer between the templates, the model makes use of global buffers. Each place in the `TCP_buffer` array corresponds to a buffer related to a `Client` or `Host` process so that (Client(0)) will read from index 0 of the array, (Client(1)) will read from index 1 and so on. The `host` variable is saved as an alias to know which space in the buffers is reserved for messages from clients to the host. Since the `UDP_buffer` is supposed to correspond to a multicast, it is simply saved as an integer instead, which will hold the information that any client listening to the multicast will receive. Finally, a series of channels are defined. The channels are used to synchronize between the different processes and will be further explained when going into details about each template.

```
const int number_of_clients = 4;
const int time_limit = 2;
const int host = number_of_clients;
```

```
typedef int[0, number_of_clients - 1] id_t;

// Each TCP instance has their own entry in the TCP buffer.  The
    clients have the entry at index = their id and the host has the last
    index.
int TCP_buffer[number_of_clients + 1];
int UDP_buffer;
broadcast chan read_UDP_buffer;
chan tcp_sync[number_of_clients], ack_sync[number_of_clients],
    ack[number_of_clients], send_message[number_of_clients],
    timeout[number_of_clients];

chan TCP_send[number_of_clients], TCP_connect[number_of_clients],
    TCP_connected[number_of_clients], TCP_client_connected,
    read_TCP_buffer[number_of_clients];

chan UDP_start, UDP_send;
```

**Listing 5.4:** Global declarations

### 5.4.3 Host template

The `Host` has three variables that are declared in Listing 5.5 which are: `connected_clients` which it uses to keep track of how many clients have connected, `local_TCP_buffer` which is where the contents of the global TCP buffer is copied into and `i` which it uses to keep track of how many messages it has sent. The template seen in Figure 5.5 can be divided into three sections. In the first section the host waits for the clients to connect through TCP. Every time a `Client` has connected the `Host` receives a `TCP_client_connected?` synchronization and `connected_client` is incremented. When all of the `Clients` have connected the `Host` moves on the next section. In this section the `Host` uses the `TCP_send!` synchronization to notify each `Client` that the game is starting through TCP. Finally the `Host` is in-game and continuously sends messages to the `Clients`. This is done through either `TCP_send!` or `UDP_send!` which are the TCP and UDP channels respectively.

```
int connected_clients = 0;
int local_TCP_buffer;
int i = 0;
```

**Listing 5.5:** local Host declarations

### 5.4.4 Client template

Like the `Host` template, the `Client` has a local declaration for saving the data it reads from the global buffer. Since it is desirable to have a variable amount of instances,

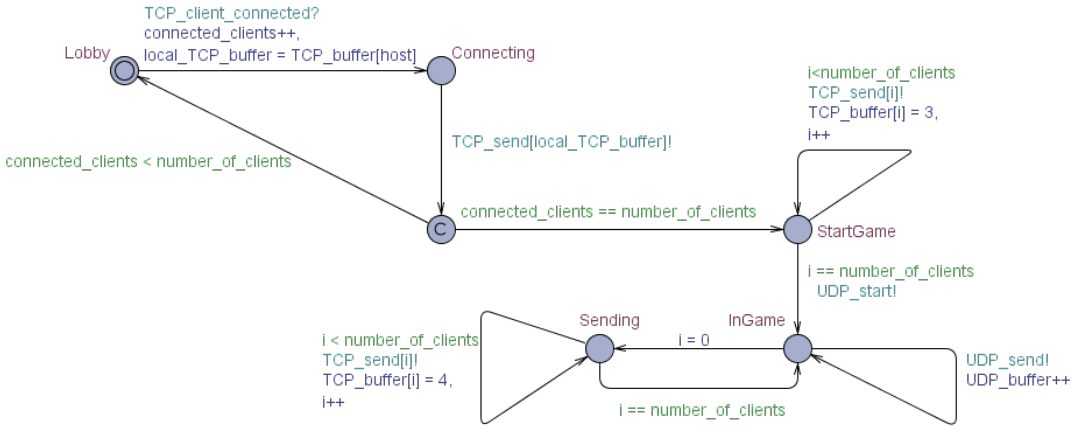


**Figure 5.5:** The Host template of the Uppaal model.

the template has an `id_t` parameter called `id`, to allow instantiating multiple client processes. The `Client` has four named locations, as seen on Figure 5.6, which correspond to the states that the actual implementation can be in. First off, the process starts in an `Initial` location, which then uses the `TCP_connect[id]!` to make the `TCP_client(id)` start connecting through TCP and transition to `Connecting`. The `Client(id)` receives `TCP_connected[id]?` synchronization when the `TCP_client(i)` has connected and it then transitions to the `Lobby` location. Everytime it receives a `read_TCP_buffer[id]?` it means that the `Host` sent a message. It then transitions to an in-between location and then either transition back to `lobby` or to `InGame` depending on what message is received. The in-between location is a committed location. This is done so that the `Client(id)` has to transition from it immediately instead of waiting. Finally, there is the `InGame` location, which is where the client will be located most of the time during the simulation. This indicates that the client is now in a game, and can continuously receive information about the game state through the `read_UPD_buffer?` and `read_TCP_buffer[id]?` synchronizations.

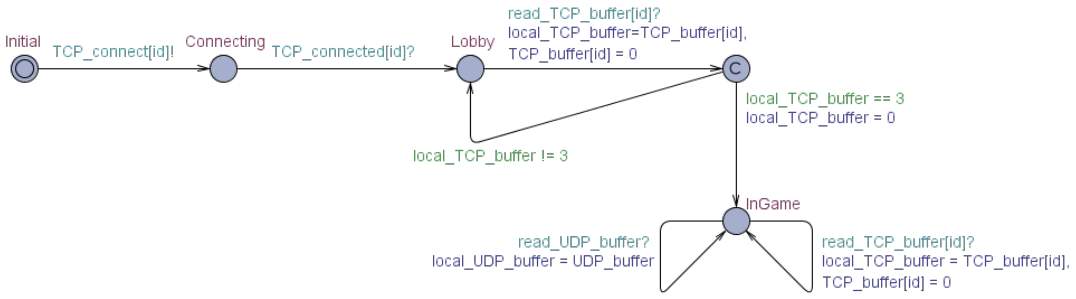


**Figure 5.6:** The Client template of the Uppaal model.

### 5.4.5 TCP host and client templates

The `TCP_Host` and `TCP_Client` were designed such that there will be a single process of both respectively for each client process that exists. This was designed to model the socket connection that is established between the host and a client. Like in the `Client`, this happens by having an `clientId` as a parameter for when the process is instantiated. As seen on Figure 5.7 and Figure 5.8, the TCP templates will stay in a `Start` location until `TCP_client` receive an `TCP_connect[id]?` synchronization from a `Client`. The `TCP_Host` and `TCP_Client` will then begin to initialize a 3-way handshake to ensure that both parties are ready for transferring data between them. Once the handshake has been completed, the `TCP_Host` will send a message to the `Host` to inform that a new player has been connected via the `TCP_client_connected!` synchronization, and then enter an `Idle` location to indicate that it is ready to send messages. Likewise, in the `TCP_Client`, after the handshake is completed it will inform the `Client` that a connection has been successfully established with the `TCP_connected[id]!` synchronization, and transition to the `Idle` location where it is ready to receive messages.

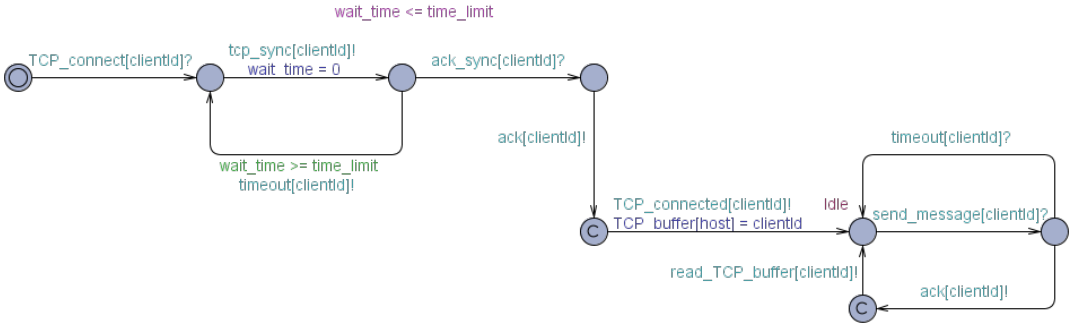


**Figure 5.7:** The TCP_client template of the Uppaal model.

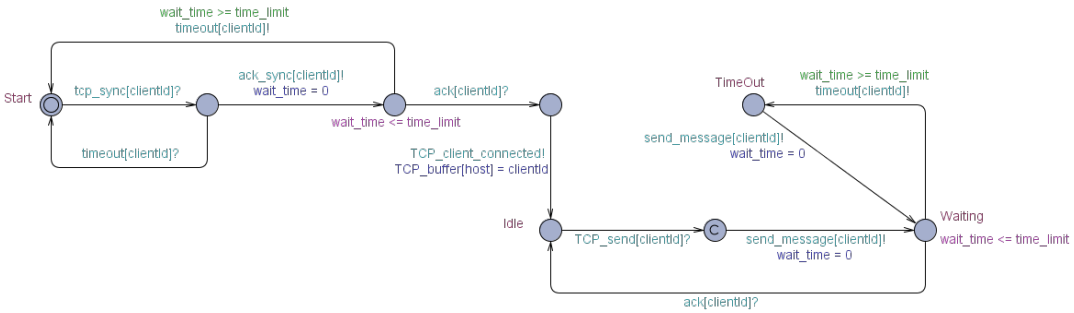


**Figure 5.8:** The TCP_host template of the Uppaal model.

### 5.4.6 UDP host templates

The final template seen in Figure 5.9 is the `UDP_host`. It is very simple as it waits until the `Host!` communicates that the games has started through `UDP_start?`. When the `UDP_host` has started it waits for a `UDP_send?` synchronization from the host and then notifies the `Clients` through the `read_UDP_buffer!` synchronization that they should read the global buffer.

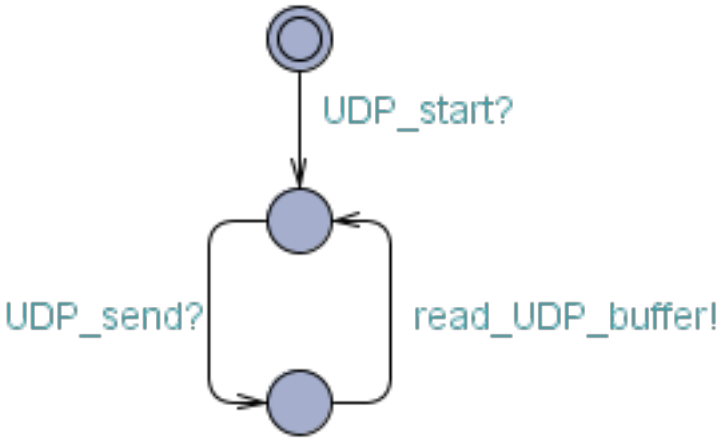


**Figure 5.9:** The UPD_host template of the Uppaal model.

### 5.4.7 Perfect scenario simulation trace

In Uppaal it is possible to generate traces of simulations of the model. This is useful when verifying different properties of the model or when getting an overview of how the model works. A simulation trace can be seen on Figure 5.10. At the top of the trace you see all of the processes in the system. You can then see the trace of each process which is the line that goes from the process and to the bottom of the diagram. The boxes on the trace indicate which Location the process is in at any given time. The red arrows between the traces symbolize synchronizations that happen between processes, and the label on the red arrow indicates which channel it happens through. As seen on the trace `Client(0)` first connects through the threeway handshake without any timeouts. `Client(1)` then also connects through a three-way handshake without any timeouts. Finally, after both of the clients are in the `Lobby`, the `Host` transitions to `InGame`.

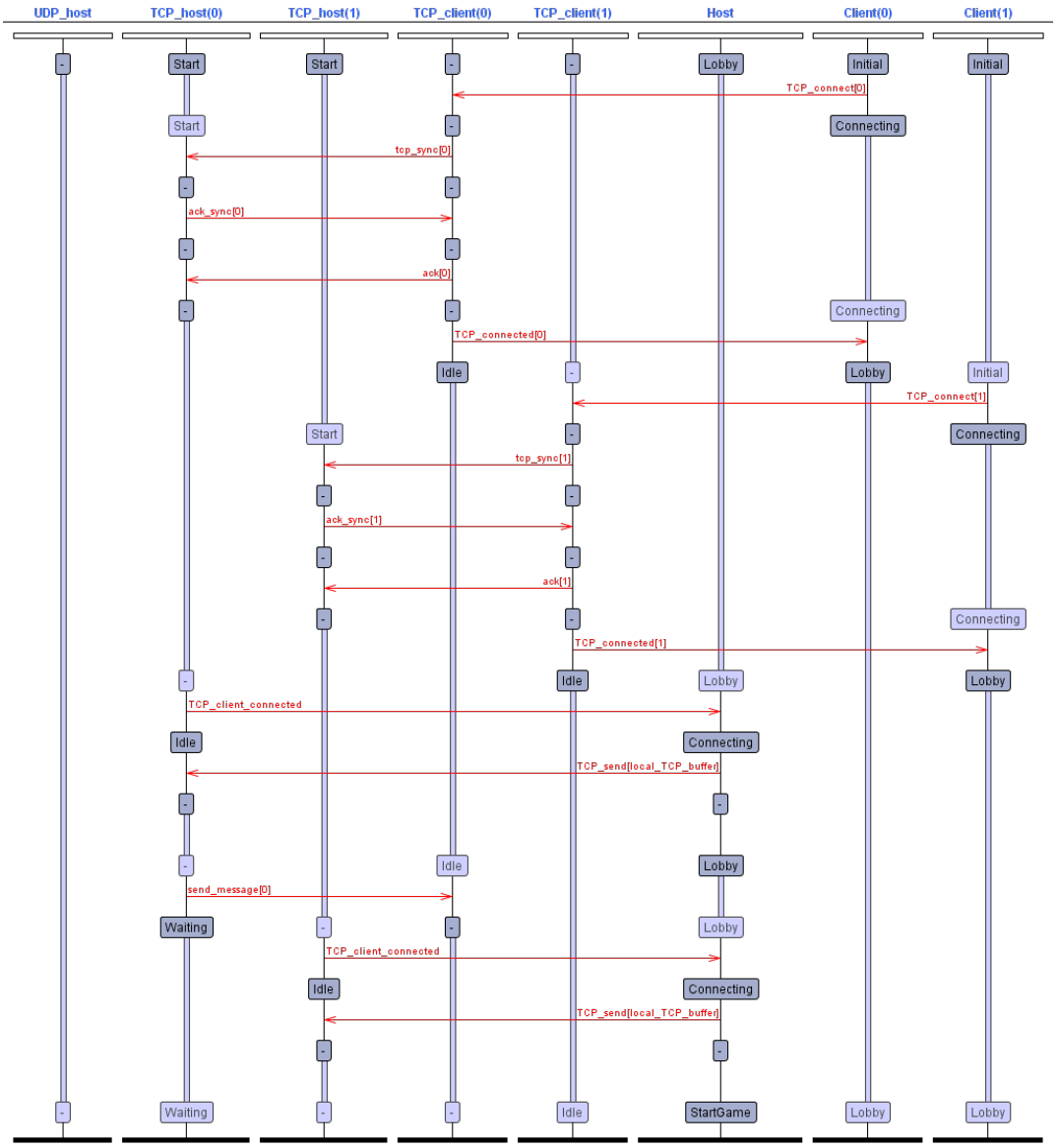


**Figure 5.10:** Example of a trace where both `Client` processes connect without any problems.

### 5.4.8 Trace with a timeout

A trace where it times out once while connecting can be seen on Figure 5.11. `Client(1)` connects without any issues, but `Client(0)` times out once while connecting. This can be seen on the trace of `TCP_client(0)`. When it sends the `tcp_sync[0]!` it times out because it does not get a response from the `TCP_host(1)`. It then resends the message successfully and continues with connecting.

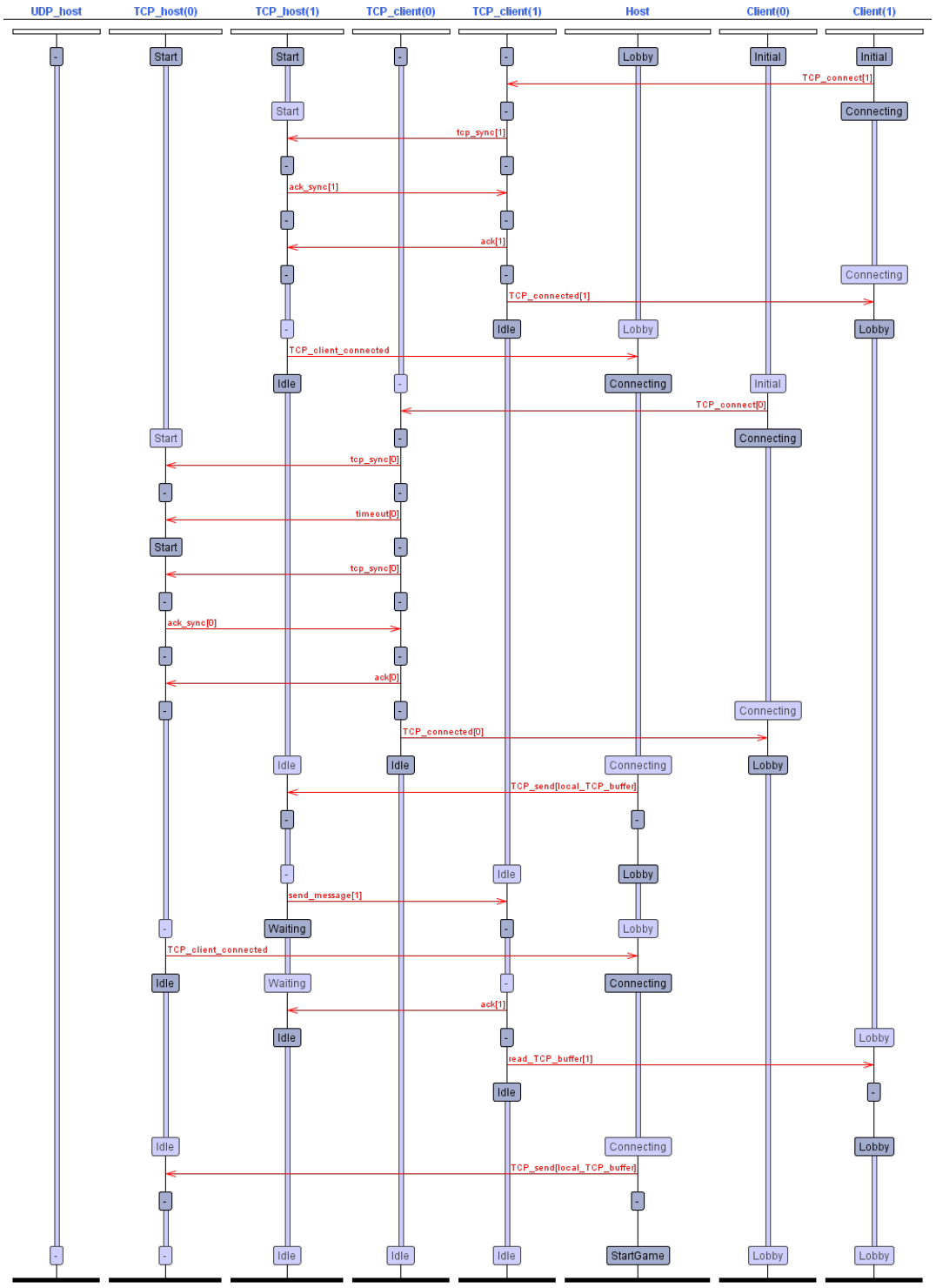


**Figure 5.11:** Example of a trace where a `TCP_host` times out while connecting.

## 5.5 Camera handling

There were some complications with handling the camera in Unity. Since the playing field can have a varying size the camera needs to be able to fit the whole playing field on the screen such that it can be rendered. Another problem is the usage of VR headsets that allow the player to manipulate the camera's view by looking in different directions. This can cause the playing field to go out of the player's view, which is a problem since they can then no longer see their position.

The way this is solved is by always having the playing field and its attached game objects face the direction of the camera. This is achieved by giving the playing field the same rotation as the camera. By doing this, the player will always see the playing field in the same position in the 3D space no matter which direction they look. But this solution is not enough, since the playing field will not have the same position in the 3D space as the camera. It must be a certain distance away from the camera, based on its width and height, to allow it to be rendered. Because of this, a new position must be calculated for the playing field in relation to the direction of the camera.

```
1    newPosition = playingFieldTransform.transform.position +
        playingFieldTransform.transform.forward *
        distanceFromCamera;
2    transform.position = newPosition;
3    transform.rotation = mainCamera.transform.localRotation;
```

**Listing 5.6:** Calculating and setting the position and rotation of the playing field

As seen on Listing 5.6 the new position is calculated by adding the playing field's initial position relative to the camera, which is the `playingFieldTransform.transform.position` variable, with an offset vector. The offset vector is calculated by multiplying `playingFieldTransform.transf` which is a normalized vector pointing in the camera's looking direction, with a certain distance defined through the `distanceFromCamera` variable, which is how far the field should be from the camera. These calculations are done in the `Update` method which is called for every frame rendered in the game.

For the `distanceFromCamera` calculation, the camera needs to be far enough away from the field to fit it in its far clipping view. The far clipping view is the maximum distance from the camera that 3D objects will be rendered.

```
1    distanceFromCamera = 1.2f * size * 0.5f /
        Mathf.Tan(mainCamera.fieldOfView * 0.5f *
        Mathf.Deg2Rad);
```

**Listing 5.7:** Calculation of the distance between the camera and field

This is a simple trigonometry problem where a triangle is formed between the camera, the center of the field and the top of the field. The height of the field and the angle of

**Figure 5.12:** A view of the room used for the test. Two anchors can be seen on the sides.

the camera's field of view are the known values, and the distance between the camera and the center of the field can be found. Half of the side opposite to the camera is divided with the tangent function of half of the field of view angle converted to radians. The calculation is seen on Listing 5.7. An additional arbitrary value of `1.2f` is multiplied with the distance to allow the whole field to be seen in VR with some additional free space around the edges of the field.

## 5.6 Initial test on May 7th

On May 7th the group met up to perform the first real test of the system, aiming to try out the full experience of the game, using the VR goggles to view the game and using the Pozyx system to generate positional data based on the movement of the players. This test was conducted with the goal of exposing issues with the state of the system, as well as simply test whether or not it would be playable, meaning the player characters would move in a regular fashion without jerking around the screen. A testing area was constructed in order to conduct the test. One part of this area can be seen on Figure 5.12, while Figure 5.13 shows how we mounted the anchors onto a wall for the setup. The final area was documented, with the details shown on Figure 5.14. The documentation of distance is necessary for the host to know how

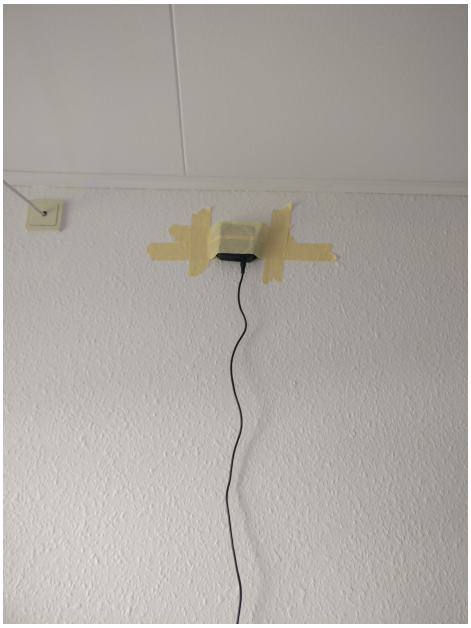

**Figure 5.13:** A single Pozyx anchor has been attached to a wall.

to input the anchor positions in the setup of the game, and the anchors are placed high on the walls, to increase the chance of receiving a good signal [27]. In order to view the game, the test made use of four sets of VR goggles, shown on Figure 5.15

### The results of the test

The overall goal of this test was to play the game in a two versus two fashion. This test, however, did not quite accomplish that goal. Leading up to the test, members of the group had communicated regarding a smaller scale test, only involving a single person, that would test that the Pozyx system generated data and passed it along to Unity. This seemed to work, so the actual test was carried out with more people. There had been a miscommunication however, the smaller scale test was carried out on test data, making use of predefined data that had been used during development. When the day of the test came, it turned out this would be the first test with real data based on an actual game setup. This led to the encountering of several bugs.

The first, and most time consuming, was caused by some firmware issues with the Pozyx anchors, delaying the test until they had been updated. Then issues were encountered when trying to send the host configuration data to the players. This was caused by some unintended type conversions in the Python code. Another issue related to the input from the host. There was no specific information regarding the unit which should be used when inputting information, such as centimeters, millimeters or meters, and this caused some confusion. Finally, once four players were able

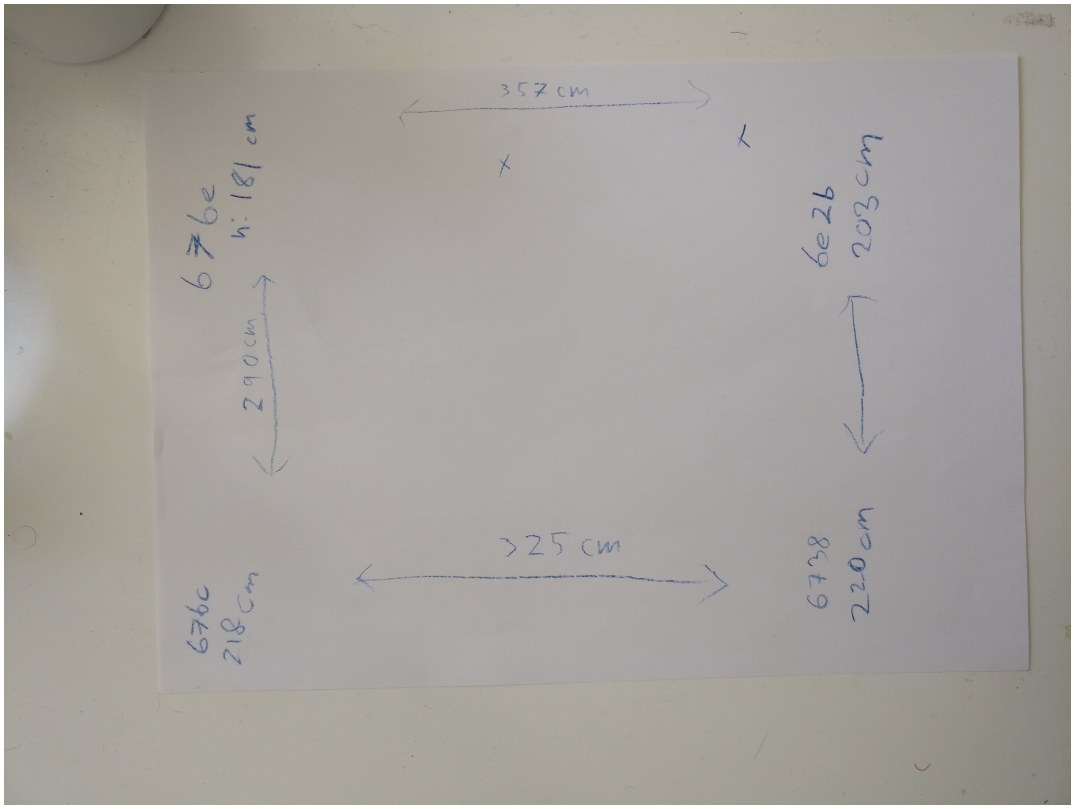


**Figure 5.14:** A paper defining the setup, noting height of anchors, anchor tags and distance between anchors.

**Figure 5.15:** The VR goggles used to hold and view the phone.

to connect and receive configuration data from the host, the game launched for the players, but they were immobile, not moving on the field based on the Pozyx data. Another issue encountered during testing was related to the blue goal zone. This goal zone specifically could flicker when the player moved his head, which the red zone did not. As such, while the test did not achieve the original goal, it still exposed certain issues to fix for the upcoming sprint.

# Chapter 6

# Sprint 5

## 6.1 Sprint goal and introduction

With the test results from the previous sprint, the main goal for this final sprint is first and foremost to correct the bugs that were discovered. Since this is the last sprint, and it is a week shorter than the previous sprints due to time constraints, the focus is shifted from implementing major new features to polishing the currently existing features and prepare the system for a final test.

## 6.2 Limiting the amount of UDP transmissions

As the game was being tested we encountered a problem with the amount of UDP packages that were being transmitted. Initially there were no limitations to the number of packages sent, and the server would transmit as many as it possibly could. This would overburden the internet and could cause it to crash. A limit had to be set for how many packages were able to be transmitted per second. Since Pozyx can send updates on tags at a rate of approximately 60 per second, it would be preferable to have the limit at close to that amount [3]. The limit could even be a bit higher than 60, since the updates are being transmitted with UDP which means that there is no assurance that each package arrives. Eventually, a limit of 70 packages per second was chosen based on the Pozyx limit and the chance of packages not arriving.

```
1       self.time_now = time.time()
2       if(self.time_prev != None):
3           self.bytesAheadOfSchedule -=
                self.ConvertSecondsToBytes(self.time_now -
                self.time_prev)
4       self.time_prev = self.time_now
5
6       self.bytesAheadOfSchedule += 7
```

```
7       if(self.bytesAheadOfSchedule > 0):
8           time.sleep(self.ConvertBytesToSeconds(self.bytesAheadOfSchedule))
9
10      # Send message to all clients listening on the
            multicast_group
11      self.sendto(message, self.multicast_group)
```

**Listing 6.1:** Implementaion of the limit on the amount of packages that can be sent per second

The code from Listing 6.1 is from the `send` function of the UDP socket. This code is run each time a UDP package is sent. Initially the current time is stored, and the time difference between the current time and the time when the previous package was sent is converted to bytes. The variable `bytesAheadOfSchedule` keeps track of whether or not too many bytes are being sent per second. Since the size of the packages sent with UDP is always the same size, being 7 bytes, `bytesAheadOfSchedule` is incremented with 7 for each package sent section 2.6. If `bytesAheadOfSchedule` is above 0, too many packages are transmitted and the server should wait a bit before sending the next package using `time.sleep()`. By doing this for each call to the `send` function, the number of packages sent per second can be limited to 70 per second.

```
1       def ConvertSecondsToBytes(self, numSeconds):
2           return numSeconds*self.maxSendRateBytesPerSecond
3
4       def ConvertBytesToSeconds(self, numBytes):
5           return float(numBytes)/self.maxSendRateBytesPerSecond
```

**Listing 6.2:** function for converting seconds to bytes and bytes to seconds

The conversion functions between bytes and seconds are seen in Listing 6.2. `maxSendRateBytesPerSec` is defined as the the maximum number of packages that can be sent per second, 70 in this case, times the size of the package in bytes which is 7. The conversion is then simply a case of either dividing the number of bytes provided with `maxSendRateBytesPerSecond` which gives the amount of seconds the system should sleep. `ConverSecondsToBytes` is used for figuring out the number of bytes `bytesAheadOfSchedule` should be decremented with. The parameter passed to this function is the number of seconds that passed between the current and previous call to the `send` function.

# Chapter 7

# Appendix

## 7.1 Results for experiment

### 7.1.1 Experiment with one tag

| Actual grid | Average grid (x,y,z) | x min | x max | y min | y max | z min | z max |
|---|---|---|---|---|---|---|---|
| (0,0) | (8.8, 0.2, 113.5) | (0.0) | (34.0) | (-23.1) | (11.0) | (0.0) | (346.5) |
| (30,0) | (37.0, 9.1, 180.8) | (26.9) | (51.2) | (0.6) | (21.3) | (74.7) | (336.3) |
| (60,0) | (70.8, 9.9, 221.1) | (55.7) | (84.3) | (-3.6) | (28.1) | (84.0) | (325.4) |
| (90,0) | (100.2, 5.7, 263.1) | (83.4) | (108.9) | (-37.3) | (21.5) | (79.3) | (322.6) |
| (120,0) | (135.6, -22.6, 270.3) | (116.1) | (172.1) | (-56.3) | (14.0) | (48.2) | (305.0) |
| (0,30) | (24.2, 36.1, 199.0) | (11.6) | (57.0) | (25.6) | (67.7) | (28.4) | (334.1) |
| (30,30) | (66.9, 30.7, 193.5) | (56.8) | (77.1) | (19.3) | (40.2) | (66.0) | (324.3) |
| (60,30) | (100.6, 24.6, 292.2) | (83.1) | (109.5) | (0.2) | (33.0) | (88.2) | (318.8) |
| (90,30) | (121.7, 35.5, 291.4) | (104.8) | (136.8) | (14.1) | (53.3) | (64.3) | (313.3) |
| (120,30) | (31.2, 22.9, 242.8) | (9.0) | (107.1) | (-49.2) | (57.0) | (24.5) | (357.6) |
| (0,60) | (29.6, 39.1, 128.7) | (-11.9) | (105.7) | (6.3) | (113.2) | (-23.1) | (356.6) |
| (30,60) | (39.2, 60.1, 116.8) | (5.7) | (53.0) | (49.7) | (83.8) | (69.9) | (334.9) |
| (60,60) | (64.9, 77.5, 255.2) | (47.0) | (89.0) | (52.4) | (101.9) | (45.9) | (339.3) |
| (90,60) | (91.2, 64.0, 120.4) | (80.7) | (98.8) | (56.8) | (70.7) | (70.6) | (320.3) |
| (120,60) | (110.3, 52.3, 127.4) | (74.3) | (143.9) | (13.8) | (76.8) | (62.2) | (314.4) |
| (0,90) | (17.0, 90.3, 122.4) | (4.9) | (30.1) | (73.8) | (122.9) | (47.9) | (334.1) |
| (30,90) | (37.7, 90.2, 242.5) | (19.4) | (54.0) | (69.7) | (102.2) | (61.4) | (330.6) |
| (60,90) | (65.6, 89.7, 105.4) | (54.1) | (75.3) | (83.4) | (98.8) | (82.7) | (317.6) |
| (90,90) | (88.0, 91.8, 92.2) | (80.9) | (95.7) | (75.2) | (99.2) | (82.1) | (304.4) |
| (120,90) | (107.5, 84.8, 90.8) | (88.1) | (126.9) | (73.2) | (95.0) | (65.0) | (314.8) |
| (0,120) | (20.8, 119.4, 67.5) | (11.6) | (28.9) | (112.5) | (131.4) | (59.5) | (74.4) |
| (30,120) | (33.7, 114.9, 213.4) | (15.9) | (53.2) | (87.8) | (138.6) | (52.7) | (333.1) |
| (60,120) | (66.0, 127.4, 281.4) | (45.1) | (83.2) | (111.4) | (143.1) | (82.2) | (310.9) |
| (90,120) | (89.4, 120.7, 92.8) | (76.3) | (98.3) | (111.9) | (132.0) | (76.5) | (314.2) |
| (120,120) | (134.1, 121.8, 86.3) | (112.2) | (145.4) | (100.2) | (133.4) | (61.9) | (337.4) |

**Table 7.1:** Table with grids for experiment with 1 tag. Tag: 26895

| Position | Amount of data points | Average deviation |
|---|---|---|
| (0, 0) | 58 | 9.92 cm |
| (30, 0) | 65 | 12.29 cm |
| (60, 0) | 54 | 15.76 cm |
| (90, 0) | 65 | 14.74 cm |
| (120, 0) | 62 | 28.85 cm |
| (0, 30) | 59 | 36.56 cm |
| (30, 30) | 64 | 11.65 cm |
| (60, 30) | 61 | 8.40 cm |
| (90, 30) | 67 | 12.82 cm |
| (120, 30) | 53 | 7.95 cm |
| (0, 60) | 65 | 39.56 cm |
| (30, 60) | 61 | 15.32 cm |
| (60, 60) | 60 | 20.34 cm |
| (90, 60) | 65 | 5.39 cm |
| (120, 60) | 66 | 18.90 cm |
| (0, 90) | 65 | 20.10 cm |
| (30, 90) | 63 | 10.28 cm |
| (60, 90) | 59 | 7.23 cm |
| (90, 90) | 59 | 5.39 cm |
| (120, 90) | 65 | 15.23 cm |
| (0, 120) | 65 | 21.12 cm |
| (30, 120) | 57 | 11.71 cm |
| (60, 120) | 59 | 11.10 cm |
| (90, 120) | 67 | 5.87 cm |
| (120, 120) | 62 | 15.26 cm |

**Table 7.2:** Average deviation for each position with tag 26895

### 7.1.2 Experiment with three tags

#### Tag 26467

| Actual grid | Average grid (x,y,z) | x min | x max | y min | y max | z min | z max |
|---|---|---|---|---|---|---|---|
| (140,0) | (141.7, -8.4, 268.2) | (134.5) | (147.7) | (-15.1) | (1.6) | (79.1) | (325.5) |
| (140,30) | (140.1, 31.4, 124.0) | (134.1) | (150.4) | (23.1) | (39.1) | (77.5) | (306.6) |
| (140,60) | (133.7, 59.1, 162.7) | (131.3) | (137.0) | (57.3) | (61.2) | (87.5) | (310.0) |
| (140,90) | (126.5, 88.5, 90.8) | (123.8) | (129.2) | (86.7) | (91.9) | (87.8) | (96.1) |
| (140,120) | (118.6, 117.6, 86.1) | (113.7) | (128.9) | (109.3) | (121.0) | (80.2) | (91.7) |

**Table 7.3:** Table with grids for experiment with three tags. Tag: 26467

| Position | Amount of data points updates | Average deviation |
|---|---|---|
| (140, 0) | 7 | 10.8 cm |
| (140, 30) | 6 | 8.79 cm |
| (140, 60) | 6 | 6.47 cm |
| (140, 90) | 5 | 13.72 cm |
| (140, 120) | 5 | 22.26 cm |

**Table 7.4:** Average deviation for tag 26467

### Tag 26895

| Actual grid | Average grid (x,y,z) | x min | x max | y min | y max | z min | z max |
|---|---|---|---|---|---|---|---|
| (180,0) | (184.0, -2.0, 239.3) | (172.6) | (192.7) | (-10.7) | (18.2) | (86.6) | (304.9) |
| (180,30) | (184.9, 18.4, 165.5) | (174.5) | (202.1) | (9.3) | (29.1) | (79.7) | (298.1) |
| (180,60) | (179.9, 57.8, 155.3) | (173.7) | (187.1) | (49.6) | (67.5) | (84.9) | (307.2) |
| (180,90) | (174.6, 95.1, 198.1) | (167.0) | (187.0) | (85.8) | (104.6) | (85.7) | (320.5) |
| (180,120) | (175.4, 116.8, 82.6) | (168.6) | (180.5) | (108.5) | (122.7) | (74.0) | (94.5) |

**Table 7.5:** Table with grids for experiment with three tags. Tag: 26895

| Position | Amount of data points updates | Average deviation |
|---|---|---|
| (180, 0) | 17 | 8.46 cm |
| (180, 30) | 17 | 15.0 cm |
| (180, 60) | 17 | 5.45 cm |
| (180, 90) | 17 | 9.91 cm |
| (180, 120) | 17 | 7.09 cm |

**Table 7.6:** Average deviation for tag 26895

### Tag 24622

| Actual grid | Average grid (x,y,z) | x min | x max | y min | y max | z min | z max |
|---|---|---|---|---|---|---|---|
| (220,0) | (234.1, 5.4, 324.5) | (232.7) | (236.1) | (2.8) | (8.2) | (323.9) | (325.4) |
| (220,30) | (229.5, 28.2, 264.3) | (216.7) | (234.4) | (22.3) | (37.1) | (90.9) | (298.2) |
| (220,60) | (223.6, 46.1, 214.6) | (215.5) | (231.3) | (40.5) | (53.4) | (105.9) | (288.1) |
| (220,90) | (203.4, 78.6, 237.7) | (190.0) | (213.4) | (55.1) | (94.2) | (67.2) | (329.2) |
| (220,120) | (216.4, 119.1, 162.7) | (204.2) | (226.4) | (113.5) | (124.5) | (80.8) | (322.4) |

**Table 7.7:** Table with grids for experiment with three tags. Tag: 24622

| Position | Amount of data points updates | Average deviation |
|---|---|---|
| (220, 0) | 5 | 15.18 cm |
| (220, 30) | 7 | 11.76 cm |
| (220, 60) | 5 | 15.30 cm |
| (220, 90) | 3 | 22.43 cm |
| (220, 120) | 3 | 10.08 cm |

**Table 7.8:** Average deviation for tag 24622

## 7.2 Experiment with five tags

### Tag 24622

| Actual grid | Average grid (x,y,z) | x min | x max | y min | y max | z min | z max |
|---|---|---|---|---|---|---|---|
| (180,0) | (173.8, -30.0, 125.8) | (168.3) | (180.1) | (-65.1) | (0.0) | (93.7) | (277.2) |
| (180,20) | (182.1, 25.4, 160.4) | (175.2) | (189.9) | (16.9) | (32.3) | (79.4) | (309.8) |
| (180,40) | (183.5, 41.3, 90.9) | (179.6) | (185.5) | (39.6) | (42.2) | (84.4) | (97.9) |
| (180,60) | (186.6, 63.4, 97.5) | (181.2) | (190.2) | (62.2) | (65.7) | (92.7) | (100.3) |
| (180,80) | (188.4, 74.2, 229.4) | (186.7) | (190.8) | (70.3) | (81.9) | (98.3) | (295.3) |
| (180,100) | (183.8, 101.9, 146.4) | (181.7) | (186.9) | (99.9) | (104.8) | (89.3) | (314.0) |
| (180,120) | (176.9, 125.5, 172.2) | (169.3) | (183.3) | (123.9) | (127.1) | (96.1) | (317.3) |

**Table 7.9:** Table with grids for experiment with five tags. Tag 24622

| Position | Amount of data points updates | Average deviation |
|---|---|---|
| (180, 0) | 7 | 31.70 cm |
| (180, 20) | 6 | 9.23 cm |
| (180, 40) | 3 | 4.09 cm |
| (180, 60) | 5 | 7.61 cm |
| (180, 80) | 3 | 11.28 cm |
| (180, 100) | 4 | 4.38 cm |
| (180, 120) | 3 | 7.90 cm |

**Table 7.10:** Amount of data points with 5 seconds and the average deviation for tag 24622

### Tag 26467

| Actual grid | Average grid (x,y,z) | x min | x max | y min | y max | z min | z max |
|---|---|---|---|---|---|---|---|
| (140,0) | (142.0, -9.4, 229.3) | (132.7) | (147.1) | (-14.2) | (0.0) | (102.3) | (295.7) |
| (140,20) | (123.0, 3.7, 76.8) | (123.0) | (123.0) | (3.7) | (3.7) | (76.8) | (76.8) |
| (140,40) | (139.9, 26.8, 94.1) | (136.5) | (142.5) | (24.1) | (30.6) | (92.4) | (95.7) |
| (140,60) | (123.4, 65.0, 96.0) | (120.8) | (126.0) | (59.7) | (70.4) | (93.4) | (98.7) |
| (140,80) | (133.1, 70.4, 83.5) | (124.9) | (140.7) | (67.6) | (75.8) | (70.9) | (94.2) |
| (140,100) | (139.2, 90.6, 88.6) | (138.7) | (139.9) | (85.4) | (99.5) | (77.5) | (94.7) |
| (140,120) | (137.9, 110.4, 88.8) | (136.3) | (140.1) | (107.6) | (113.2) | (86.6) | (91.2) |

**Table 7.11:** Table with grids for experiment with five tags. Tag 26467

| Position | Amount of data points updates | Average deviation |
|---|---|---|
| (140, 0) | 3 | 12.32 cm |
| (140, 20) | 1 | 23.55 cm |
| (140, 40) | 3 | 13.44 cm |
| (140, 60) | 2 | 17.91 cm |
| (140, 80) | 4 | 13.30 cm |
| (140, 100) | 4 | 9.55 cm |
| (140, 120) | 4 | 9.98 cm |

**Table 7.12:** Amount of data points with 5 seconds intervals and the average deviation for tag 26467

### Tag 26895

| Actual grid | Average grid (x,y,z) | x min | x max | y min | y max | z min | z max |
|---|---|---|---|---|---|---|---|
| (160,0) | (174.4, -13.1, 249.4) | (153.6) | (181.6) | (-29.1) | (8.9) | (83.1) | (319.9) |
| (160,20) | (169.6, 20.8, 87.3) | (165.1) | (177.6) | (11.2) | (29.5) | (77.5) | (98.0) |
| (160,40) | (164.4, 35.4, 130.4) | (154.3) | (172.9) | (26.8) | (45.8) | (84.4) | (313.3) |
| (160,60) | (168.2, 62.4, 131.8) | (161.8) | (180.5) | (55.3) | (69.6) | (91.8) | (297.5) |
| (160,80) | (163.7, 86.9, 96.3) | (158.9) | (166.8) | (79.0) | (96.2) | (90.5) | (105.9) |
| (160,100) | (166.8, 95.3, 222.9) | (155.1) | (177.5) | (79.1) | (117.9) | (68.7) | (323.2) |
| (160,120) | (162.1, 133.1, 153.3) | (155.1) | (166.1) | (124.5) | (140.0) | (74.3) | (315.9) |

**Table 7.13:** Table with grids for experiment with five tags. Tag 26895

| Position | Amount of data points updates | Average deviation |
|---|---|---|
| (160, 0) | 11 | 22.05 cm |
| (160, 20) | 11 | 10.74 cm |
| (160, 40) | 11 | 9.31 cm |
| (160, 60) | 11 | 9.51 cm |
| (160, 80) | 10 | 8.44 cm |
| (160, 100) | 10 | 12.75 cm |
| (160, 120) | 10 | 13.71 cm |

**Table 7.14:** Amount of data points with 5 seconds intervals and the average deviation for tag 26467

### Tag 26901

| Actual grid | Average grid (x,y,z) | x min | x max | y min | y max | z min | z max |
|---|---|---|---|---|---|---|---|
| (220, 0) | (229.9, 4.9, 233.6) | (214.6) | (246.5) | (-6.9) | (15.7) | (72.4) | (318.3) |
| (220, 20) | (227.5, 14.8, 151.4) | (221.7) | (233.4) | (3.0) | (30.8) | (87.7) | (296.1) |
| (220, 40) | (228.6, 42.4, 111.9) | (221.6) | (237.5) | (31.9) | (58.0) | (90.6) | (286.5) |
| (220, 60) | (227.5, 61.0, 132.9) | (220.5) | (241.1) | (43.7) | (75.2) | (87.1) | (293.6) |
| (220, 80) | (222.8, 77.6, 86.2) | (218.4) | (235.3) | (60.2) | (90.3) | (75.3) | (91.9) |
| (220, 100) | (220.2, 92.2, 165.6) | (180.5) | (228.8) | (51.3) | (115.2) | (22.4) | (342.8) |
| (220, 120) | (222.4, 108.4, 105.6) | (215.6) | (228.7) | (97.7) | (115.5) | (73.9) | (319.7) |

**Table 7.15:** Table with grids for experiment with five tags. Tag 26901

| Position | Amount of data points updates | Average deviation |
|---|---|---|
| (220, 0) | 11 | 14.79 cm |
| (220, 20) | 11 | 20.99 cm |
| (220, 40) | 11 | 14.71 cm |
| (220, 60) | 11 | 15.61 cm |
| (220, 80) | 10 | 13.90 cm |
| (220, 100) | 10 | 8.97 cm |
| (220, 120) | 10 | 13.54 cm |

**Table 7.16:** Amount of data points with 5 seconds intervals and the average deviation for tag 26467

### Tag 27001

| Actual grid | Average grid (x,y,z) | x min | x max | y min | y max | z min | z max |
|---|---|---|---|---|---|---|---|
| (200, 0) | (198.3, 0.3, 133.9) | (194.0) | (204.5) | (-5.8) | (3.8) | (92.7) | (289.2) |
| (200, 20) | (208.3, 11.7, 181.5) | (201.1) | (220.9) | (-1.7) | (24.2) | (102.7) | (294.6) |
| (200, 40) | (201.8, 31.8, 113.4) | (195.5) | (206.7) | (16.9) | (42.0) | (108.2) | (118.5) |
| (200, 60) | (205.6, 53.0, 140.8) | (200.3) | (213.5) | (37.7) | (60.0) | (112.4) | (272.8) |
| (200, 80) | (203.6, 71.5, 116.4) | (200.1) | (207.9) | (57.4) | (77.4) | (112.5) | (118.9) |
| (200, 100) | (201.6, 93.2, 112.0) | (197.5) | (205.8) | (90.1) | (96.8) | (109.7) | (115.5) |
| (200, 120) | (204.6, 110.8, 105.9) | (200.7) | (212.4) | (101.0) | (119.0) | (95.0) | (116.6) |

**Table 7.17:** Table with grids for experiment with five tags. Tag 27001

| Position | Amount of data points | Average deviation |
|---|---|---|
| (200, 0) | 6 | 7.32 cm |
| (200, 20) | 7 | 18.34 cm |
| (200, 40) | 7 | 9.69 cm |
| (200, 60) | 9 | 8.10 cm |
| (200, 80) | 7 | 6.21 cm |
| (200, 100) | 3 | 12.44 cm |
| (200, 120) | 4 | 14.03 cm |

**Table 7.18:** Amount of data points with 5 seconds intervals and the average deviation for tag 27001

## 7.3 Network packets

The final format of the network packages are described in this appendix. For all packages, it is assumed that the TCP messages are treated as unicast to a single recipient, and UDP messages are treated as multicast. All of these packets will only describe the actual data of the packet, and not contain the UDP/TCP headers. Each packet contains an ID identifying the type as well as the required information. Additionally, each TCP package will be prefixed with a number indicating the length of the current message, since it is transmitted as a stream. This ensures that the receiver knows when to stop reading the current packet and be ready to process the next one.

### 0: Send tag position (UDP)

| **POSY (y)** | **POSX (x)** | **ID (i)** | **TIMESTAMP (s)** | **TYPE (t) 0** |
|---|---|---|---|---|
| 16 bit | 16 bit | 8 bit | 8 bit | 8 bit |

Purpose: The position of a given tag will be continuously transmitted on the multicast channel for all players to receive. The positions are sent as values relative to the anchor positions sent during setup, such that a missed package will not impact the future game state. The ID (i) `0` is reserved for the ball tag, and all other tags are considered to be players. For a game with 4 players, each would have a unique ID ranging from 1-4.

### 1: Send anchor position (TCP)

| **POSY (y)** | **POSX (x)** | **ANCHOR_ID (i)** | **TYPE (t) 1** |
|---|---|---|---|
| 16 bit | 16 bit | 8 bit | 8 bit |

Purpose: Used as a part of the setup phase, where the host sends the position of one of the four anchors to the player. This packet will be sent multiple times to each client, but with various anchor IDs (i) until all clients have received the four positions.

### 2: Send player tag (TCP)

| **PLAYER_ID (p)** | **TAG_ID (i)** | **TYPE (t) 2** |
|---|---|---|
| 8 bit | 16 bit | 8 bit |

Purpose: Used as a part of the setup phase, where the host sends the related tag ID to the player, such that they can find the physical tag related to their client instance. This packet will be sent once to each client, but with various tag IDs (i) until all clients have received their tag ID.

### 3: Signal game start (TCP)

| **TYPE (t) 3** |
|---|
| 8 bit |

Purpose: Used to signal to the players that the game has been started.

#### 4: Send goal scored (TCP)

| TEAM2SCORE (2) | TEAM1SCORE (1) | TYPE (t) 4 |
|---|---|---|
| 8 bit | 8 bit | 8 bit |

Purpose: Used to signal the clients that a goal has been scored, and the scoreboard should be updated on their GUI. This triggers the host to generate new goal positions.

#### 5: Send new goal position (TCP)

| Offset (o) | POSY (y) | POSX (x) | TEAM_ID (i) | TYPE (t) 5 |
|---|---|---|---|---|
| 8 bit | 16 bit | 16 bit | 8 bit | 8 bit |

Purpose: Sends a new goal position to the clients. This message will be sent twice to each team when a goal is scored, such that they receive new positions for both teams' goals. The positions are the X and Y coordinates of the center of the goal as they have a pre-defined size based on the shortest edge of the playing field. It also contains the offset of goal zones (o), which is the distance from the center of the goal to the sides of the goal.

## 7.4 Understanding a Uppaal model

The goal of this appendix is not to give an extensive guide on how to use Uppaal, but rather to serve as a short introduction on how to read the models described in this report. For a more in-depth description of how Uppaal works, we recommend the official tutorial for Uppaal 4.0 by Gerd Behrmann, Alexandre David and Kim G. Larsen [1].

### 7.4.1 What is Uppaal?

Uppaal is an integrated tool environment for modeling and verification of timed automata. It is mainly used for simulating and verifying models of real-time systems [25] by means of property checking, where it can automatically generate a diagnostic trace to explain why a given property is, or is not, satisfied by the modeled system. This property checking is done using a reachability analysis, meaning that it will search through the state space of the system to find a state where a given property is true. Using property checking it is then possible to check for properties like whether or not the system can end up in a deadlock - or in the context of this system, it would be possible to check whether or not it is possible that one client is still in the lobby while the other clients are receiving in-game information such as positional data.

### 7.4.2 Format of a system in Uppaal

A very basic model in Uppaal consists of a set of locations and edges as seen on Figure 7.1. In this basic model, we have two locations: `A` and `B` and two transitions: From `A` to `B`, and from `B` to `A`. The location `A` has an extra ring, which indicates that it is an initial location, meaning that this is where the system will start when initialized.

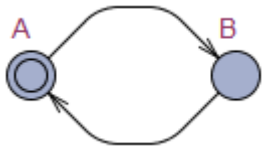


**Figure 7.1:** Locations in Uppaal.

This visual model combined with a set of local declarations related to the model is called a template.

In addition to the local declarations, which are scoped to the current template, it is also possible to have global declarations (first object in the Project folder on Figure 7.2), which are variables and functions that are shared across the entire system.

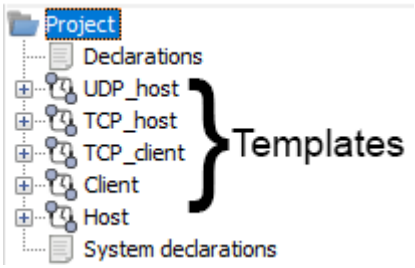


**Figure 7.2:** Content of a Uppaal project.

When the system is started in the simulator, these `templates` will be instantiated as `processes`, which start in their initial locations.

From here, the user can choose which transition to take next in the given process, as seen on Figures 7.3 and 7.4.

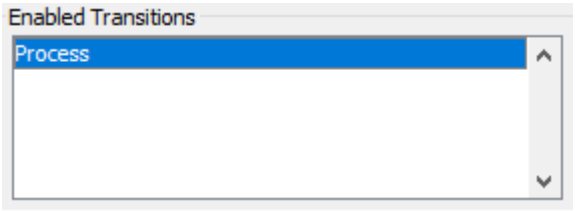


**Figure 7.3:** Enabled traces of a process.

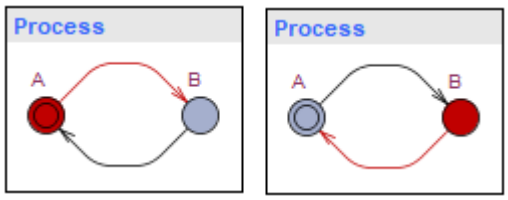


**Figure 7.4:** The process before and after taking the transition.

### 7.4.3 Synchronizations

As a UPPAAL system can consist of multiple processes, it is often useful to allow the different processes to communicate. One way of communicating is via the use of `synchronizations`. If a transition between two locations has a synchronization called sync on it, as seen in Figure 7.5, it can wait for another process to communicate with it before taking the transition.

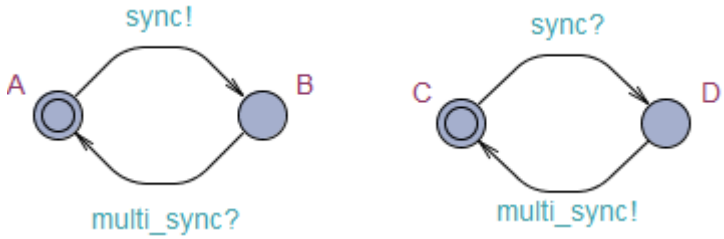


**Figure 7.5:** Model with synchronizations on transitions.

In the model on Figure 7.5, the **`sync`** has an exclamation mark after its name, and on the other a question mark. This means that A is the transmitter of a synchronization request and that it cannot take the transition before another process with a question mark (receiver) is ready to take a transition with a synchronization of the same name. Likewise, it is not possible to take a transition with a **`sync?`** until another process is transmitting a **`sync!`**. When this synchronization is ready to happen, both processes will take the transition.

In this basic example, it is easily seen that there is only one sender and one receiver for the synchronization. In a larger example, there might be several processes ready to receive the synchronization, but only one can actually receive the transition at a time.

This is due to the implementation of synchronizations, where we synchronize using a channel defined in the global declarations:

```
chan sync;
```

However, should we want to synchronize with multiple other processes instead, it is possible to declare the synchronization channel as a broadcast channel instead:

```
broadcast chan multi_sync;
```

Having the channel defined as a broadcast means that any process that is ready to synchronize when a synchronization output is transmitted must do so.

### 7.4.4 Clocks, guards, and selections

As mentioned previously, UPPAAL works with timed automata - so let us add some time to our sample model. To do this, we can add a **clock** to the local declarations of our template using the clock type:

```
clock time;
```

We can now use this clock to, for example, limit which transitions can be taken from a given state using guards.

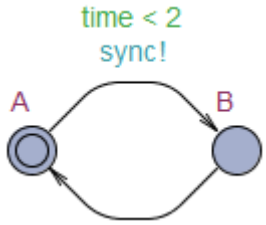


**Figure 7.6:** Model with a guard.

In the example on Figure 7.6, a **guard** has been added on the transition from A to B, specifying that at most two time units must have passed before this transition can be taken.

This quickly poses a problem for us, though, as the time will keep increasing, meaning that we may not be able to take the transition from A to B again, and will end up in a deadlock.

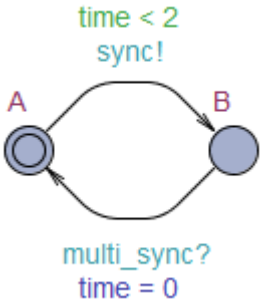


**Figure 7.7:** Model with a guard and selection.

To fix this, it is possible to use updates to update the value of the clock, as seen in Figure 7.7 where we reset the time to 0 when taking the transition from B to A.

### 7.4.5 Invariants

Much like guards limit the possibility of taking a transition until some conditions have met, invariants can be used to limit when it is possible to be in a given location. In Figure 7.8 we have added an invariant to `B`, specifying that less than 8 `time` units must have passed before it leaves the transition.

If the invariant conditions are not met, the process cannot enter the location or must leave it as soon as the conditions no longer hold.

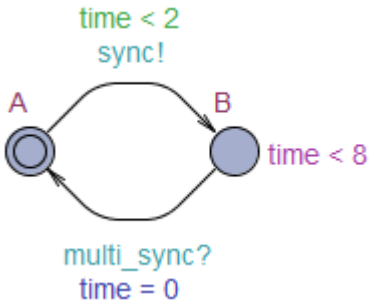


**Figure 7.8:** Model with an invariant on location B.

### 7.4.6 Committed and urgent locations

Urgent locations are locations where time is not able to pass. This means that if any of the processes are currently in an urgent location, all clocks in the system are paused until the process transitions out of that location. Just like in urgent locations, time cannot pass in committed locations. In addition to this, if any process is in a committed location, the next transition that happens in the system has to be from a process that is in a committed location. The main difference between these two is that while a process is in an urgent location, other processes are still able to take transitions while if a process is in a committed location, the only processes that are able to take a transition are processes that are in a committed location. Examples of how the syntax for these two looks can be seen on Figure 7.9.

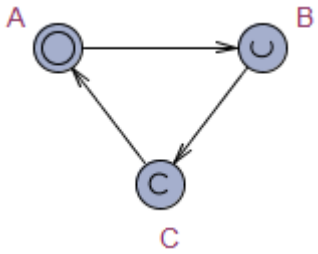


**Figure 7.9:** Model with committed and urgent locations.

Location `B` is an urgent location and location `C` is a committed location.

# Bibliography


[1] Gerd Behrmann, Alexandre David, and Kim G Larsen. "A tutorial on Uppaal 4.0". In: ().

[2] Stuart Cheshire and Daniel Steinberg. *Zero Configuration Networking: The Definitive Guide*. 1. edition. O'Reilly Media, 2005.

[3] *Creator System Performance*. URL: https://www.pozyx.io/products-and-services/creator/system-performance.

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

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

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

[7] Ilya Grigorik. *High Performance Browser Networking. What Every Develoepr Should Know About Networking And Web Performance*. 1. edition. O'Reilly Media, 2013.

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

[9] *Home: Pozyx*. URL: https://www.taitradioacademy.com/courses/basic-radio-awareness/.

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

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

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

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

[14] *IPv4 Multicast Address Space Registry*. URL: https://www.iana.org/assignments/multicast-addresses/multicast-addresses.xhtml#multicast-addresses-1.

[15] Gerard J. Holzmann. *Design and validation of computer protocols*. 1. edition. Prentice-Hall, 1991.

[16] I. Scott MacKenzie and Colin Ware. "Lag as a Determinant of Human Performance in Interactive Systems". In: *Proceedings of the INTERACT 93 and CHI 93 Conference on Human Factors in Computing Systems*. CHI 93. Amsterdam, The Netherlands: Association for Computing Machinery, 1993, pp. 488493. ISBN: 0897915755. DOI: 10.1145/169059.169431. URL: https://doi.org/10.1145/169059.169431.

[17] Curtiss Murphy. *Believable Dead Reckoning for Networked Games*. URL: https://www.researchgate.net/publication/293809946_Believable_Dead_Reckoning_for_Networked_Games (visited on 04/03/2020).

[18] US National Coordination Office for Space-based Positioning Navigation and Timing. URL: https://www.gps.gov/systems/gps/performance/accuracy/ (visited on 03/02/2020).

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

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

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

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

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

[24] *Ultra-wideband and obstacles*. URL: https://www.pozyx.io/technology/uwb-and-obstacles.

[25] AAU UUP. *Introduction to UPPAAL*. URL: https://www.it.uu.se/research/group/darts/uppaal/about.shtml (visited on 05/12/2020).

[26] Thomas Waltemate et al. "The impact of latency on perceptual judgments and motor performance in closed-loop interaction in virtual reality". eng. In: *Proceedings of the 22nd ACM Conference on virtual reality software and technology*. Vol. 02-04-. VRST '16. ACM, 2016, pp. 27–35. ISBN: 9781450344913.

[27] *Where to place the anchors*. URL: https://www.pozyx.io/technology/where-to-place-the-anchors.

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

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

# List of Figures

# List of Tables

# Listings