PROJECT REPORT

BLUETOOTH INDOOR POSITIONING USING RASPBERRY PI

BOĞAZİÇİ UNIVERSITY
DEPARTMENT OF COMPUTER ENGINEERING, SOFTWARE ENGINEERING
By Taner Eşme

December 2018

Table of Contents

Introduction

This report gives you information about my project for SWE599. As you might remember, my project was an implementation for “Bluetooth indoor positioning using Raspberry Pi”. This report will first present preliminary findings that I have completed so far about the project. The report will consist of four main sections. You will read the project description provided by the lecturer and the other sections will constitute from the project’s requirements, design and implementation.

Project Description

Estimating the location of a Bluetooth capable device in an indoor environment has many practical purposes. Either for a smart warehouse with autonomous robots carrying the goods around, or for a classic multi-storey office with employees moving from one place to another, or for a vacation compound where crowded groups of tourists can go to swimming pools, restaurants, and other activity locations with a wearable computer (e.g. a smart bracelet, or a necklace), instantaneous information about where these people are at a certain time period, and where they are moving to, will definitely help organizational planning. Tracking employees on how much time they spend outside their workspace would be an important performance metric especially in production lines. In the case of tracking tourists, the hotels can adjust their security, and other services according to the movement of their guests. For the smart warehouse, location estimation has to be so precise that a robot should never place things in wrong shelves, nor it should look for items that do not exist at the current location of the robot.

Indoor positioning can be performed using a set of stationary Bluetooth access points.

Location estimation inside a building is harder than doing it with GPS. One of the major approaches in indoor positioning is to use a fingerprinting database, trained with multiple signal level measurements at specific locations, covering the entire area, with respect to each available access point. The other major approach is to use triangulation, which is based on coordinated location estimation using at least three access points, and signal arrival times. The arrival time can be calculated at the stationary access points, or the mobile device itself. This method can be roughly considered as an indoor alternative to the GPS solution.

In this project, we are going to implement a hybrid solution of fingerprinting and triangulation.

Requirements of the Project

I am going to use .NET technologies in the project to implement needed software that will run on Raspberry Pi. Using .NET technologies will be providing us with easiness in some of the topics and

also bringing a little burden of dependencies to the Microsoft technologies. To eliminate some of those dependencies, I will leverage the power of .NET Core, which is an open source framework promoted by Microsoft to support all other platforms such as Linus and Mac OS.

After a little preliminary research, I discovered that we could not run .NET Core on the first generation Raspberry Pis because of the unsupported CPU architecture (ARM v6). I also found something important related to .NET Core, we cannot do any implementations related to Bluetooth without using Windows UWP (Universal Windows Platform) because .NET Core, unfortunately, does not have a cross-platform implementation to communicate the peripheral devices with the capability of Bluetooth.

According to the descriptions above, I can make a list of the requirements like below:

- **Raspberry Pi 3 Model B+:** This is a third generation Raspberry Pi which is able to run .NET Core Runtime and Windows 10 IoT Core. It also has an onboard Bluetooth module to collect signals radiated from agents and the Wi-Fi module to transmit the collected data to the backend server(s).
- **A cheap version of beacon or a smartphone:** As an agent unit. It will be used when constituting the fingerprinting database and during tests.
- **.NET Core 2.1:** As a development framework.
- **Windows 10 IoT Core:** As an operating system to run on Raspberry Pi.
- **A server or a commodity computer:** To transmit the collected data, store fingerprinting database and execute the localization algorithm.
- **Visual Studio 2017:** As a development environment. There are a few other development tools for .NET Core, but when it comes to developing the applications associated with Bluetooth, we will have to use VS2017 because of the compatibility issues.
- **A PC installed Windows 10:** It is a must to have a PC that Windows 10 was installed on to develop applications that can run on the devices such as Raspberry Pi with Windows 10 IoT Core.

Design of the Project

In this project, we are going to make use of fingerprinting and triangulation to localize the agents. That's why we will need to have at least three gateways (access point, gateway, and Raspberry PI terms will be used interchangeably in the document) as Raspberry Pi implementation deployed around the field. After deployment, it simply will be looking like below.

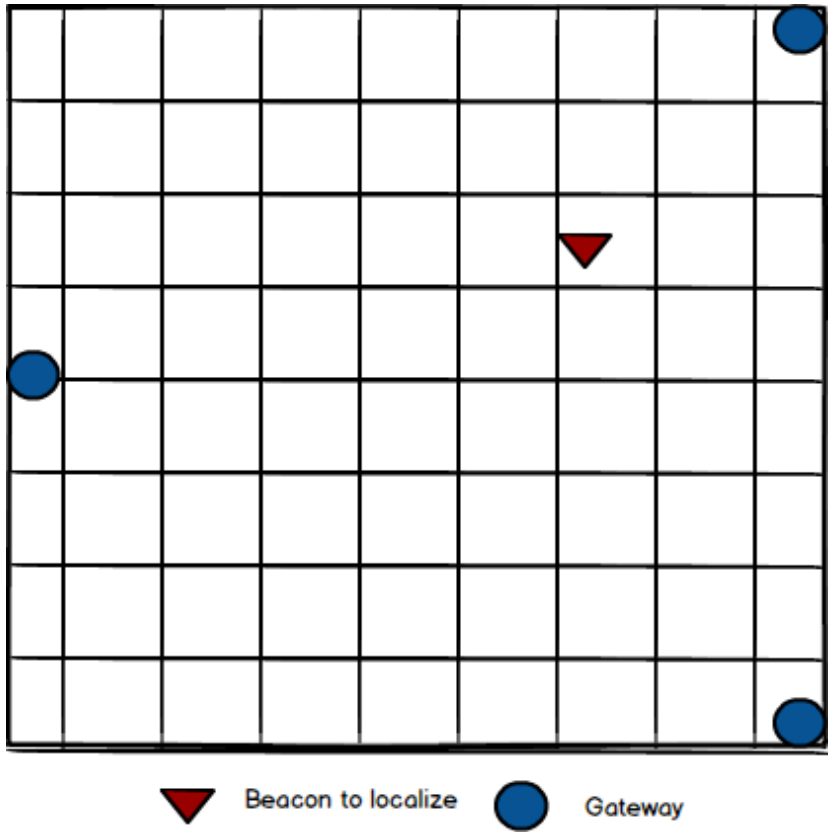


Figure 1 Organization of sensors in the field

The organization of the gateways can be changing according to the limitation of the field.

Logical Design

The gateways receive signals radiated by the agent through Bluetooth protocol and send them to the consumer server(s) that will store and analyze the data. Physical design for the project will look like below:

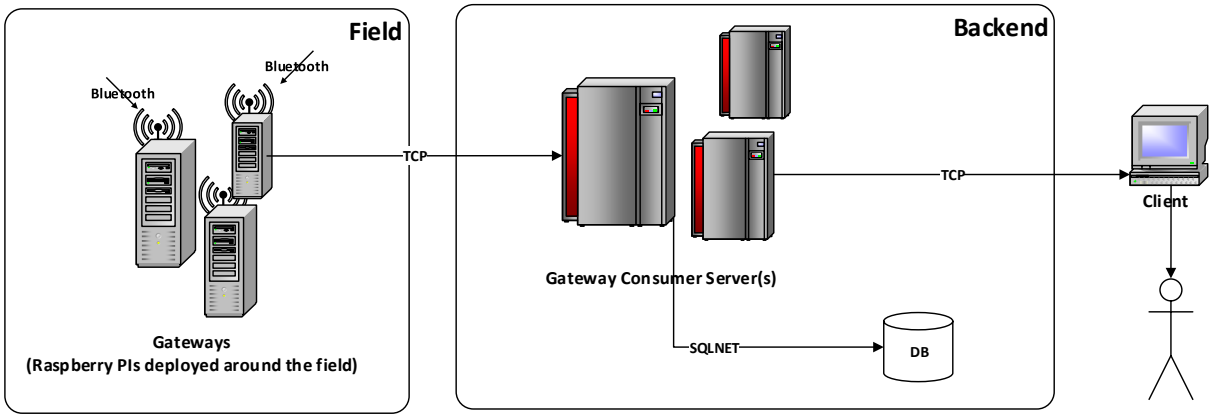


Figure 2 Logical Design

- **Gateway:** The modules receiving the signals through Bluetooth which are Raspberry PIs.
- **Gateway Consumer Server(s):** The servers that access points will be sending the data they collected via HTTP or HTTPS. There will be API for access points to connect. These servers also will have a UI application for users to view the locations of the agents.
- **DB:** The database will store the signal levels and fingerprinting data.

Software Modules

The software has two essential parts, first is the application running on Raspberry PIs and the second one to run at the backend.

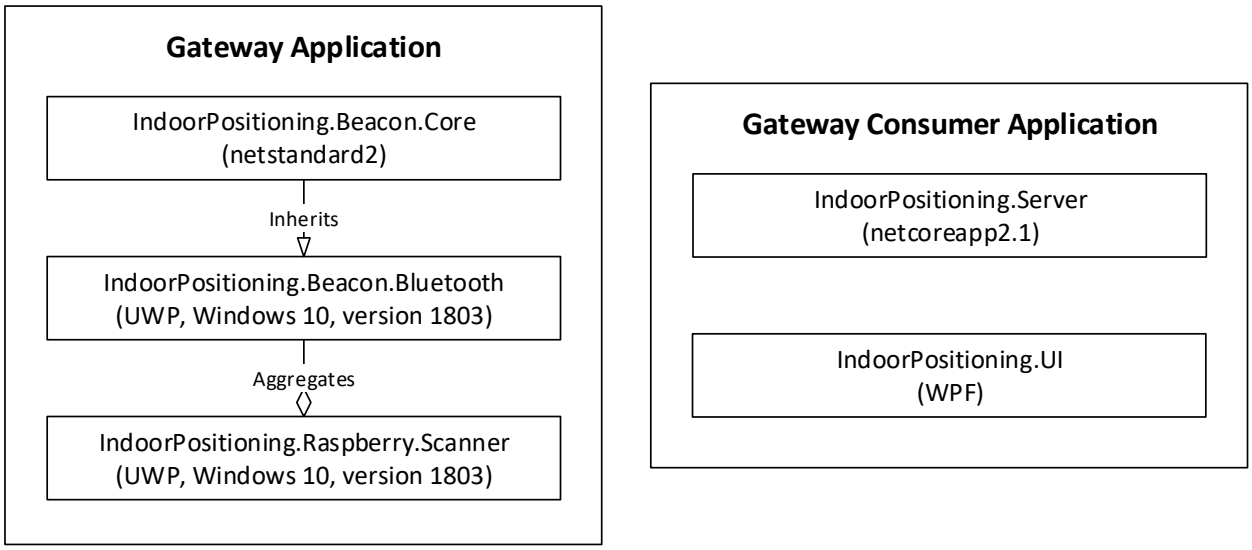


Figure 3 Software Modules

- **IndoorPositioning.Beacon.Core:** It is a .NET Standard 2 library project to constitute a standard for the devices that can be used by agents.
- **IndoorPositioning.Beacon.Bluetooth:** It is a .NET Standard 2 library project to implement the standard that is constituted by IndoorPositioning.Beacon.Core for Bluetooth.
- **IndoorPositioning.Raspberry.Scanner:** It is a .NET Core 2.1 console application project using the IndoorPositioning.Beacon.Bluetooth library to scan the signals of the Bluetooth capable devices.
- **IndoorPositioning.Server:** It is a .NET Core 2.1 Console application project for the gateways (Raspberry PIs) to send their data collected.
- **IndoorPositioning.UI:** It is a WPF (Windows Presentation Foundation) project for the users to position the agents and it also allows them to manage the overall of the environments the gateways running. This application is where the localization algorithms are implemented and to be run.

Implementation

The implementation of the project strictly relies on the technologies promoted by Microsoft. From the operating system running on Raspberry PI to the programming languages, all are the technologies supported by Microsoft.

When starting the project, we decided to develop it in .NET Core, because reportedly .NET Core provides a cross-platform development environment to allow the developers to build and run their application anywhere. However, in the context of our project, implementing one and running anywhere is not a way applicable because the peripheral libraries that we are going to leverage for the integration with Bluetooth devices are not cross-platform. They are “Windows” libraries and we are not able to use those libraries of .NET Core on any another platform other than a new platform announced by Microsoft in 2015, UWP (Universal Windows Platform). UWP is an open source API to help the developers develop applications that can be run on any Windows based platforms such as Xbox and HoloLens.

Prerequisites for Implementation

UWP is relatively new and developing. That’s why it brings some constraints onto the developers that want to develop applications that can be run on UWP platforms.

UWP applications can be developed by making use of Visual Studio 2017 – an IDE (integrated development environment) used to develop software with the programming languages such as C#, VB.NET. But you cannot develop UWP applications with Visual Studio 2017 on a platform other than Windows 10 because it requires the UWP functionalities on your computer. Therefore, you should have at least Windows 10 installed on your computer to develop UWP applications.

To run an application, which leverages the windows Bluetooth libraries, developed in .NET Core on Raspberry PI, we also have to install a UWP based operating system on Raspberry PI which is Windows 10 IoT Core. Windows 10 IoT Core is a lightweight version of Windows 10 optimized for the smaller devices such as Raspberry PI, DragonBoard.

When installing Windows 10 IoT Core on Raspberry PI, there are a few ways to do that but the easiest way is to use an application named Windows 10 IoT Core Dashboard. This application downloads and installed Windows 10 IoT Core. However, you need Windows 10 at least as an operating system to use Windows 10 IoT Core Dashboard application.

As the last prerequisite, a .NET Core application can be run only with ARMv7 chip or a newer version because of .NET Core Runtime does not support the chipsets prior to them. But the first generation Raspberry PIs have ARMv6 chips. That’s why we have to use the Raspberry PI 2 or a newer version.

Functionalities Developed

The product emerged deals with the beacons, gateways and also has to manage the different environments apart from positioning. The other stuff to be considered was that how we are going to support the fingerprinting functionalities according to the different environments full of dirtiness of Bluetooth signals.

The most important part of developing the functionalities is how we will implement the positioning because there are lots and lots of Bluetooth capable devices around and which one is the one that we want to detect and localize?

To eliminate this kind of problems during the progress of the application, I decided to implement the application in a way to detect and use a selected beacon for fingerprinting and positioning.

Three different application developed in the project.

- The application running on Raspberry PIs (IndoorPositioning.Raspberry.Scanner)
- The application that Raspberry PIs connect and transmit the RSSI values they collect (IndoorPositioning.Server)
- The user interface to manage all of the functions of the system (IndoorPositioning.UI)

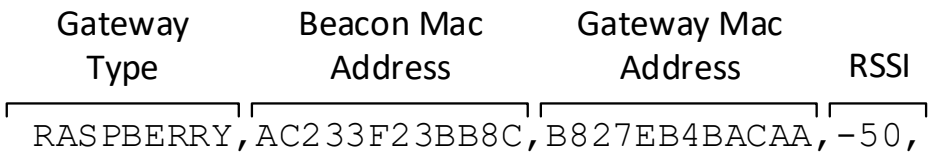
Scanner Application

It is a straightforward, lightweight background application developed based on UWP in .NET Core and runs on Raspberry PIs. It has 3 different components (see. Software Modules).

There is a predefined IP address and port for the server in the application to connect. Whenever the device boots it tries to connect this predefined IP address to transmit the data collected.

The application listens to the Bluetooth signals – It exploits the Windows Bluetooth libraries provided by .NET Core to integrate with the Bluetooth driver of Raspberry PI – regardless of where they are coming or who they are and reads the data radiated in the Bluetooth packages. After collecting and reading the data such as mac address of the device, RSSI (received signal strength indication), it transmits them to the server through TCP/IP protocol.

The sample data transmitted is:



The working sequence of the application is as follows:

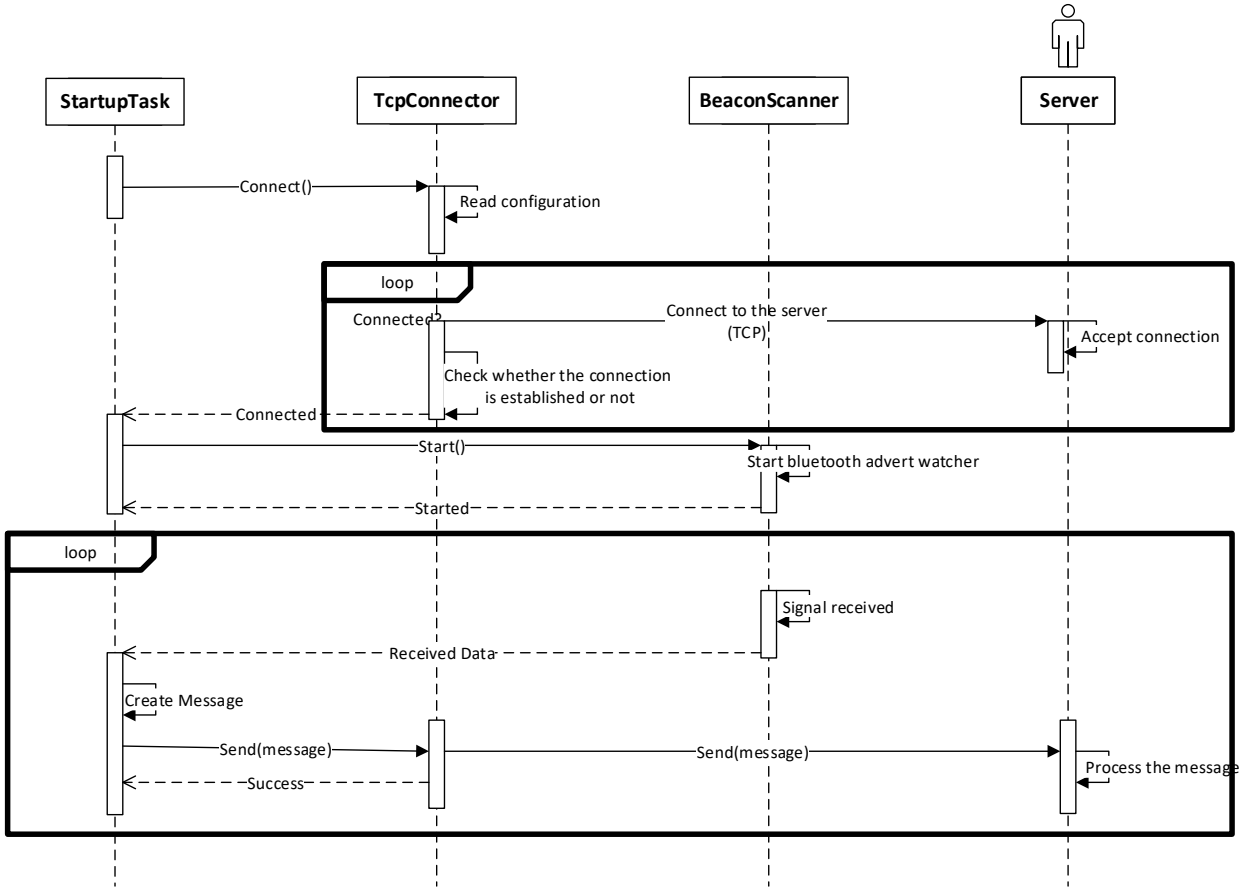


Figure 4 Sequence diagram of Scanner application

Server Application

This is the application that all Raspberry PIs deployed to connect and communicate. It listens to a specific port known by the gateways and a different port that is used for the service and managerial purposes.

The server application processes the messages coming from the gateways like below:

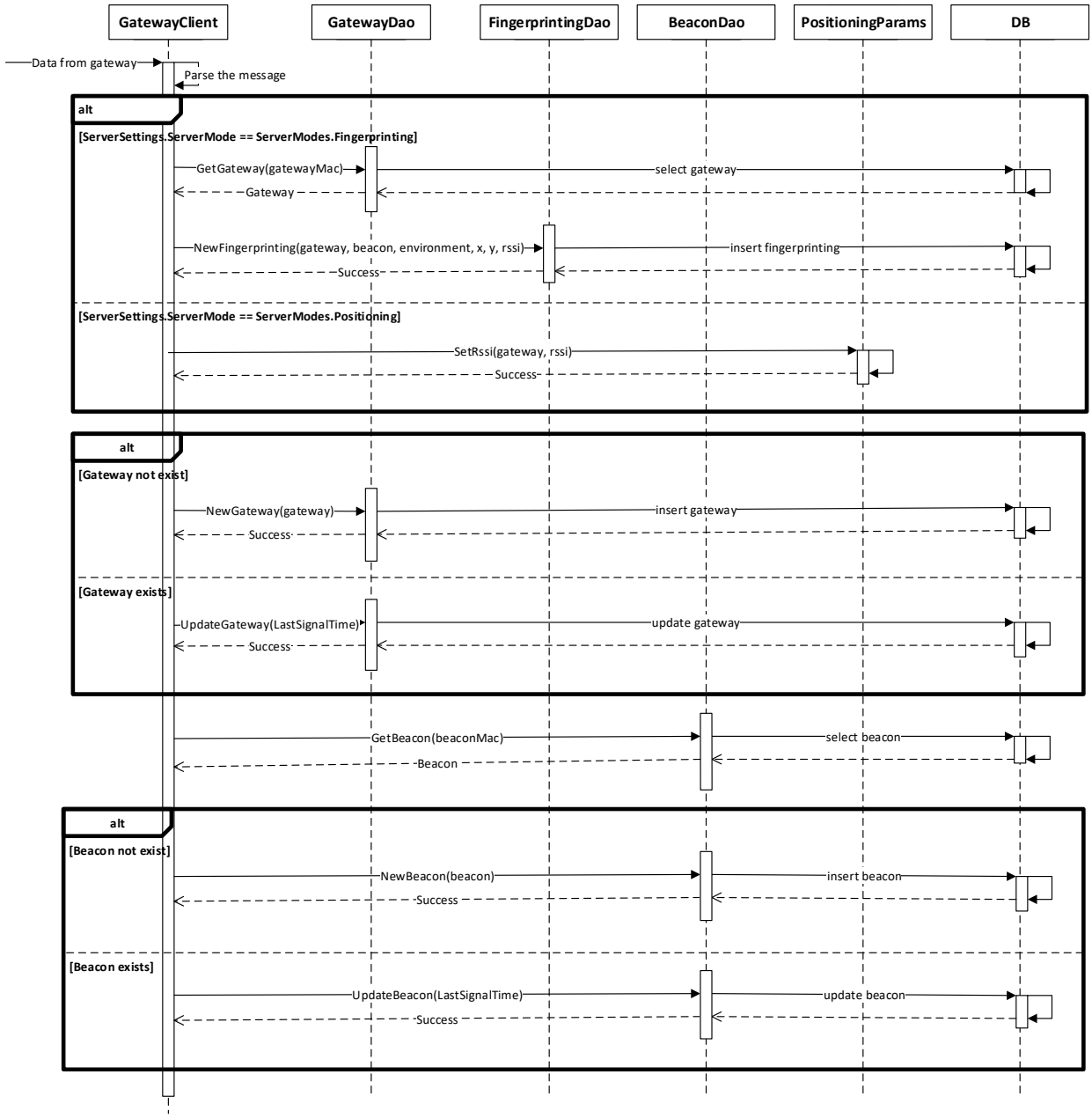


Figure 5 Sequence diagram of Server's gateway integration

Besides handling the gateways' messages, the server also provides the needed information to the user interface. A tailored command based integration protocol upon TCP/IP is developed to allow the user interface to fetch and post the necessary information for the purposes such as management, fingerprinting and positioning.

Server Command Set

`echo` returns all the command back.

`get beacons` returns all the beacons stored on DB as JSON.

`get beacon –id <digit>` returns the beacon corresponding of `<digit>` as JSON.

`get gateways` returns all the gateways stored on DB as JSON.

`get gateway –id <digit>` returns the gateway corresponding of `<digit>` as JSON.

`get mode` returns the server's current mode as a string. Server's mode can be one of idle, fingerprinting and positioning

`get environments` returns all the environments stored on DB as JSON.

`get fingerprinting –env <digit>` returns all the fingerprint values corresponding environment id of `<digit>` stored on DB.

`get rssi –count <digit>` returns the RSSI values received from 3 gateways. If the provided count does not match the existing gateway counts an exception will occur. This command can be used when the server mode is positioning. In other cases, the server does not care about the RSSI values it received.

`update beacon <json>` tries to update the beacon provided by `<json>` as JSON.

`update gateway <json>` tries to update the gateway provided by `<json>` as JSON.

`update environment <json>` tries to update the environment provided by `<json>` as JSON.

`set mode idle` sets server's mode as idle. In idle case, the server still accepts the data coming from gateways and only executes the process depicted in (Figure 5 Sequence diagram of Server's gateway integration).

`set mode positioning –beacon <digit>` sets the server's mode as positioning. In positioning case, the server collects only the data coming from the gateways for the beacon id `<digit>` and keeps them in memory, other data coming for the other beacons in the range is ignored.

`set mode fingerprinting –env <digit_1> –beacon <digit_2> -x <digit_3> -y <digit_4>` sets the server's mode as fingerprinting with the provided parameters. In fingerprinting case, the server collects only the data coming from the gateways for the beacon id `<digit_2>` and saves them in DB with the environment-id provided as `<digit_1>`. The values provided by `<digit_3>` and `<digit_4>` indicate the coordinates that the collected RSSI values will be corresponded.

`delete beacon <json>` tries to delete the beacon provided by `<json>` as JSON from DB.

`delete gateway <json>` tries to delete the gateway provided by `<json>` as JSON from DB.

`delete environment <json>` tries to delete the environment provided by `<json>` as JSON object from DB.

`add environment <json>` inserts the environment provided by `<json>` as JSON in DB.

User Interface

The user interface has 2 major roles in the project.

1. To use the server commands (see. Server Command Set) in an easy manner.
2. To run the positioning algorithms.

The server collects lots and lots of Bluetooth data from the gateways, but it cannot determine which data will be consumed for the positioning purposes. That's why this decision left to the user interface and turned into a demand-triggered approach.

Environments Screen

The application designed to manage more than one environment. Therefore, the first thing to do before starting positioning and other things is to determine where we are going to make positioning. That's why we define a new environment with the necessary information needed by the system.

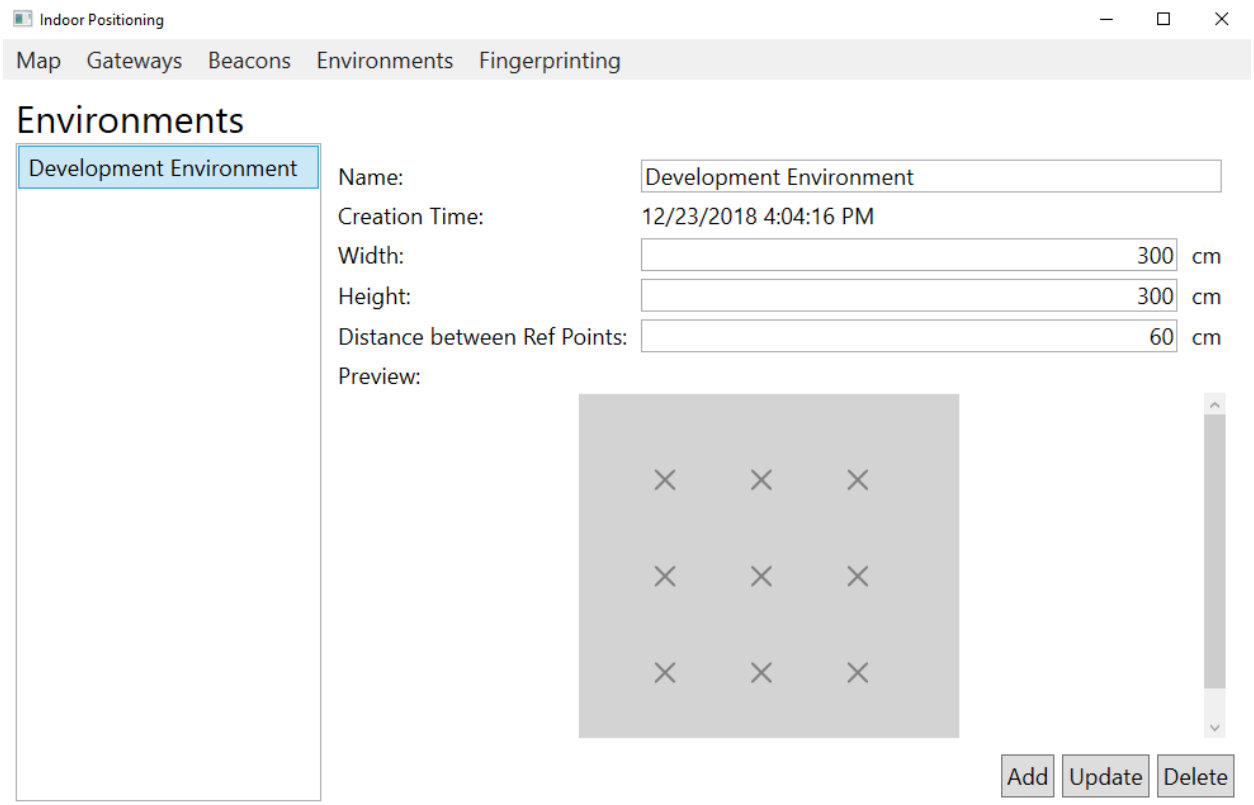


Figure 6 Environments Screen

In the concept of this project, the environment is very ad hoc. You can only define a rectangle-environment, the other shapes are not supported by the system.

The properties of the environments are quite obvious, but the property of “distance between ref points” is important because it indicates the distance between the reference points that will be employed during the fingerprinting process. When a numeric value is provided the application automatically puts the reference point on the component in the preview part like seeing in the screenshot above.

Gateways Screen

After creating the environment, we need the gateways that will be deployed in the area.

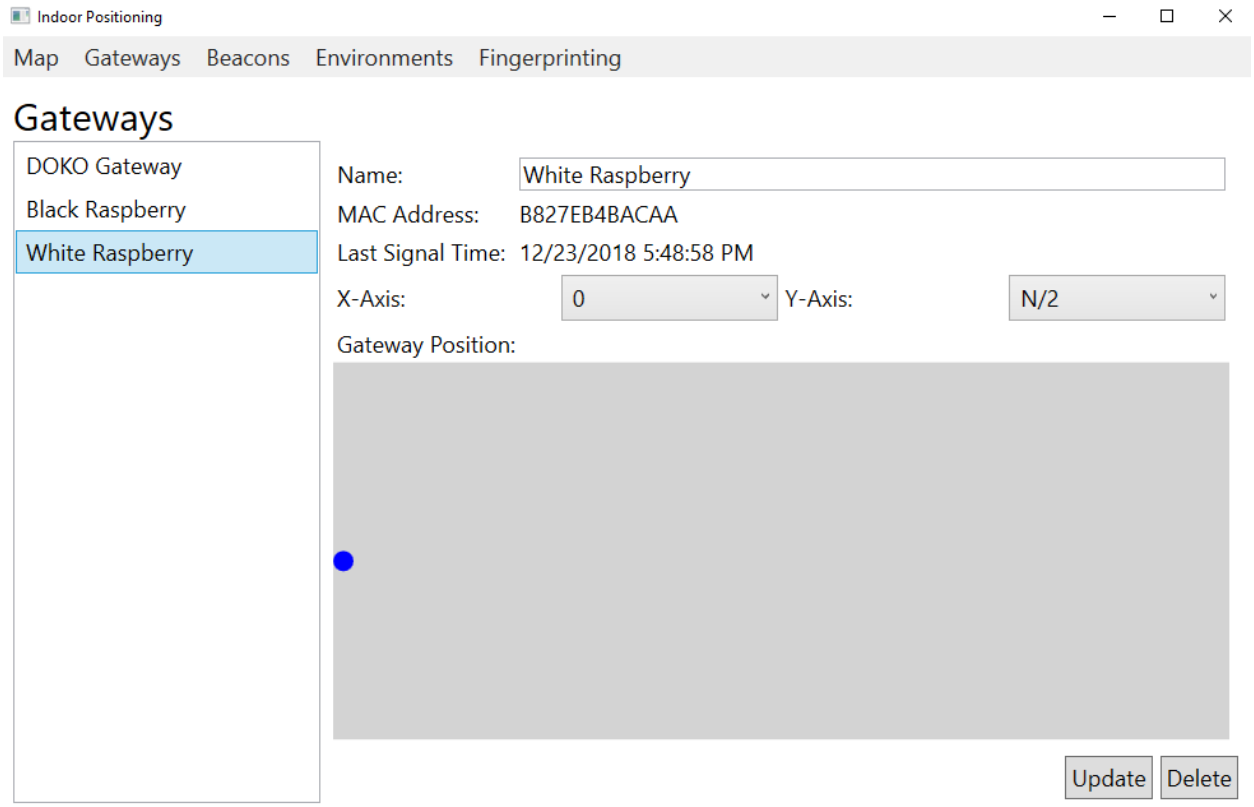


Figure 7 Gateways Screen

All of the gateways connected to a server through TCP saved in DB automatically. In the list of gateways above, all the gateways connected to the server in any time were displayed on the screen and it will let you delete and update the name and the position of the gateways.

Beacons Screen

Beacons are the other important stuff of our system.

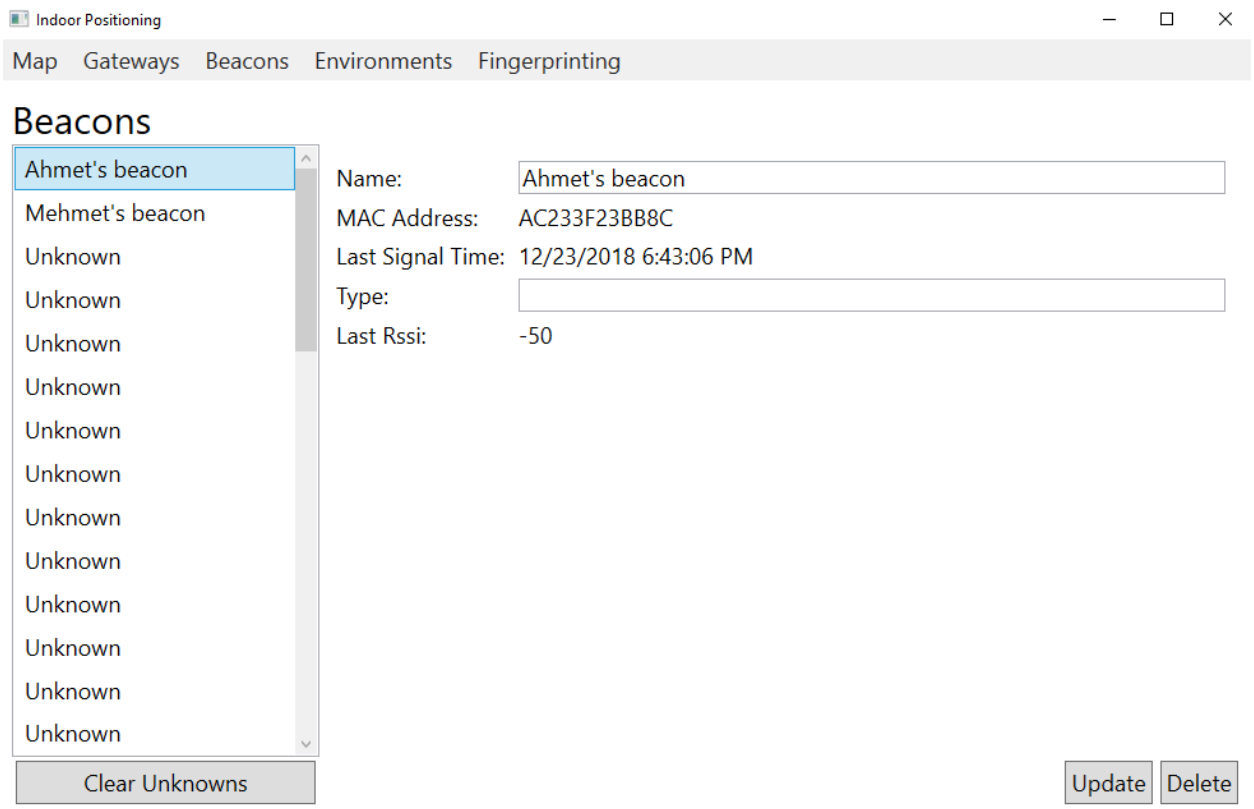


Figure 8 Beacons Screen

The server collects all beacons' data from the gateways and stores them on DB. If it receives a data of a beacon that it does not know, it will store them as "unknown" and by using the screen above, these "unknown" beacons can be turned into the known ones.

Fingerprinting Screen

This is one of the most important screens of the project because, without fingerprinting data, the system cannot do positioning.

To activate the fingerprinting mode on the server, the steps to be taken are as follows:

1. Select a beacon: The server will only care the data received for this particular beacon to create fingerprinting data. The data for other beacons will be ignored by the server.
2. Select an environment: Where we will fingerprint for.
3. Enable fingerprinting on UI: This functionality designed to allow the users to create a more accurate fingerprinting dataset.
4. Select a reference point from the map: Once we click on one of the "X" signs on the map, UI will send a command to activate fingerprinting on server-side. When any other reference point selected or the same reference point clicked on, fingerprinting will be deactivated to prevent the system from dirty fingerprinting data.

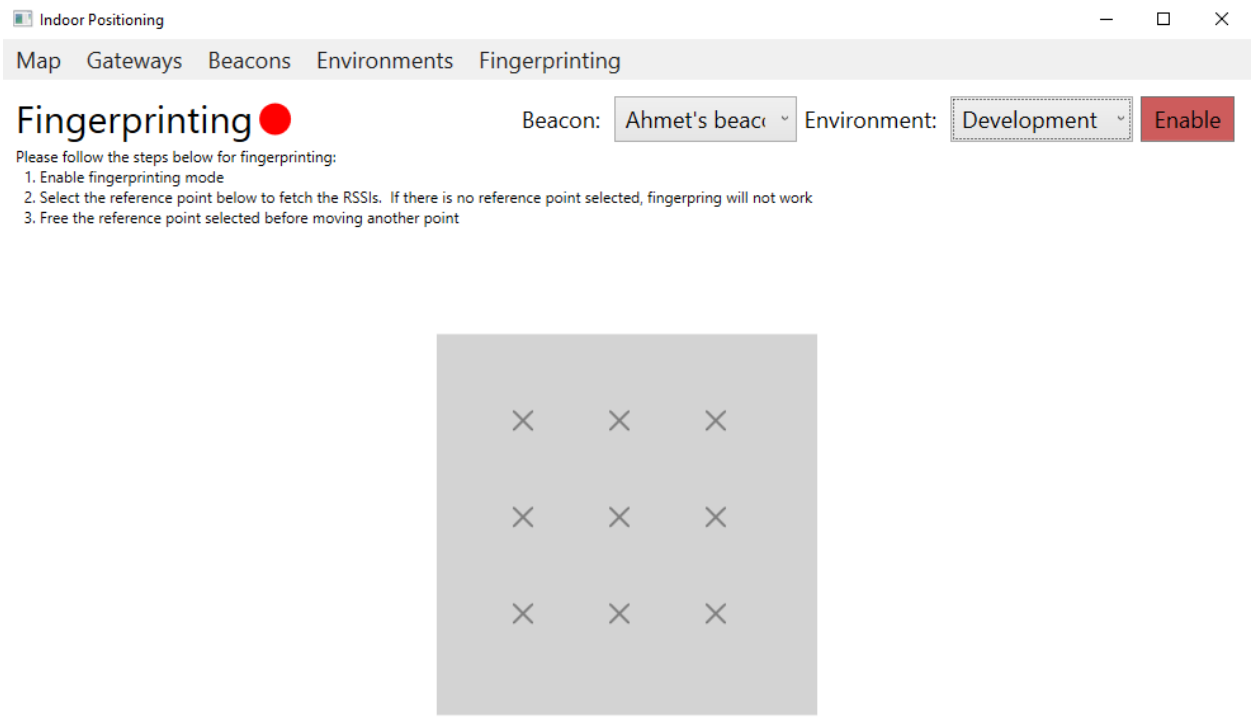


Figure 9 Fingerprinting Screen

Map Screen

After completing all of the steps, it is time to do positioning. Map Screen is a straightforward and easy-to-use screen, but positioning can be done for only one beacon and in one environment. This approach prevents us from the distractions.

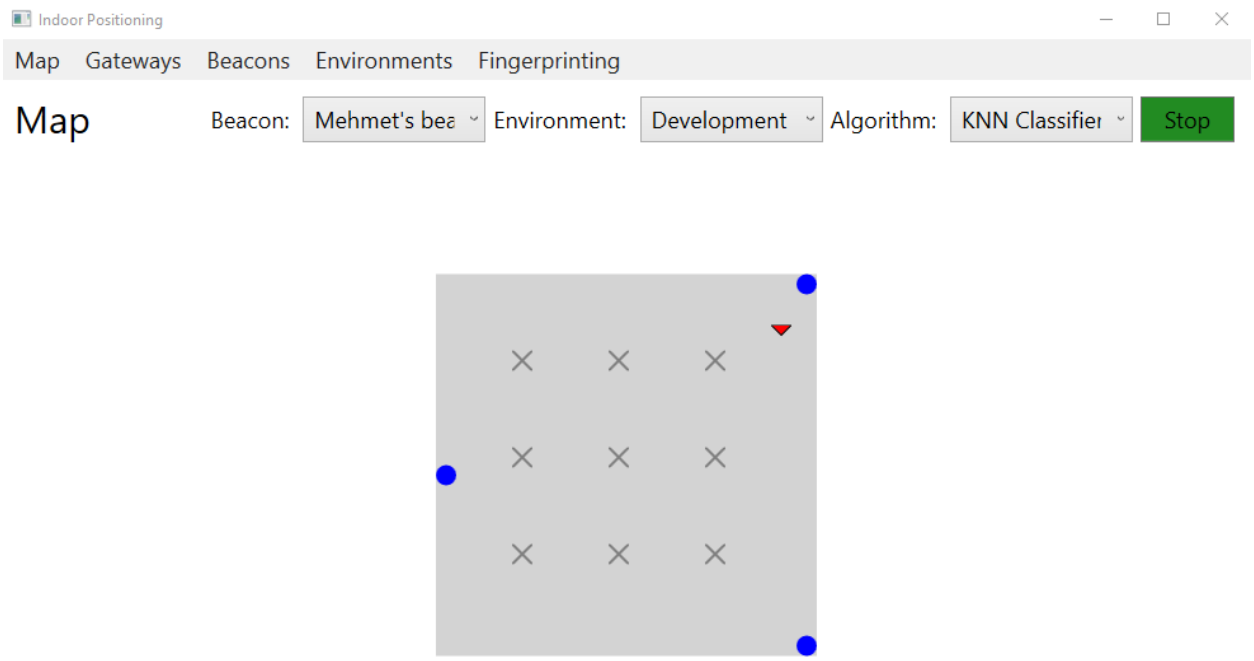


Figure 10 Map Screen

There is three algorithms implemented:

- KNN Classifier
- KNN Proximity
- RSSI Proximity

KNN Classifier

K-Nearest neighbors classifier algorithm trained by fingerprinting data tries to classify the collected RSSI values from the gateways to one of the reference points not to coordinate.

k-Nearest Neighbor
Classify $(\mathbf{X}, \mathbf{Y}, x)$ // $\mathbf{X}$: training data, $\mathbf{Y}$: class labels of $\mathbf{X}$, x: unknown sample
for $i = 1$ **to** m **do**
  Compute distance $d(\mathbf{X}_i, x)$
end for
Compute set I containing indices for the k smallest distances $d(\mathbf{X}_i, x)$.
return majority label for $\{\mathbf{Y}_i$ where $i \in I\}$

Figure 11 Pseudocode for KNN

KNN Proximity

In this way, we make use of KNN again but not for classifying, for only searching for the nearest neighbors. After getting the nearest points, we will have the coordinates of the nearest points and the distances of those points to the point we look for.

Considering all of these, if the coordinate we are looking for is x_1 and x_2 then we can draw a schema in the case of that k is 3 and the coordinates of the nearest points are (a_1, a_2), (b_1, b_2), (c_1, c_2) and the distances are d_a, d_b, d_c.

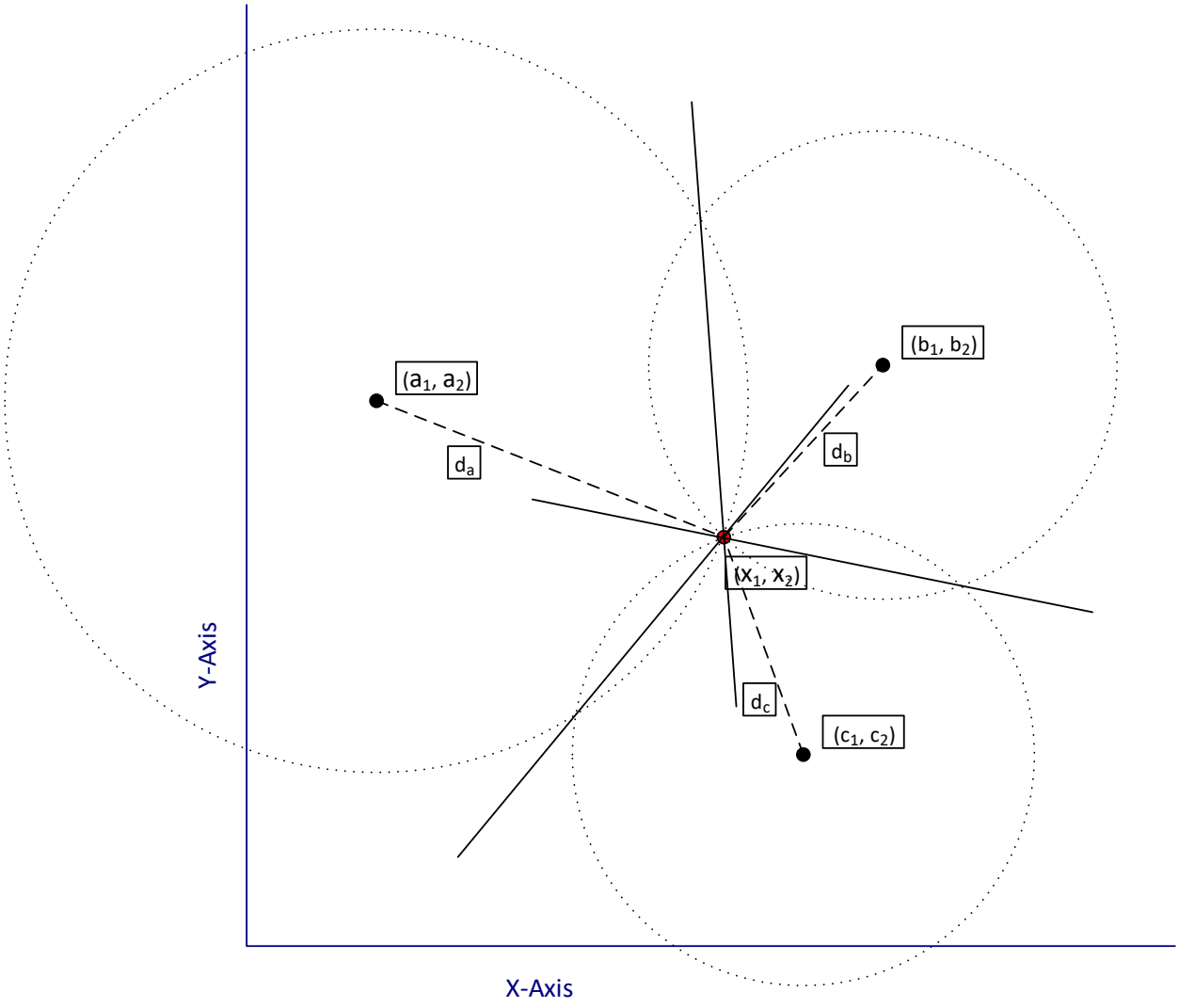


Figure 12 Coordinate calculation

Here, we know all the values of variables apart from x_1 and x_2 and we can calculate it by using this linear equation derived from Euclidean:

$$2\begin{bmatrix} a_1 - b_1 & a_2 - b_2 \\ a_1 - c_1 & a_2 - c_2 \\ b_1 - c_1 & b_2 - c_2 \end{bmatrix}\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} (\|a\|^2 - d_a^2) - (\|b\|^2 - d_b^2) \\ (\|a\|^2 - d_a^2) - (\|c\|^2 - d_c^2) \\ (\|b\|^2 - d_b^2) - (\|c\|^2 - d_c^2) \end{bmatrix}$$

Actually, we cannot use this equation directly, because there is no problem regarding 2 of them but if the third one does not prove the condition of distance, it will not work and this case has a huge probability because we are measuring the distances via KNN. That's why in the project, the equation is used by selecting 2 of 3 points for each of the points. Namely, the equation is solved for each of the pairs of two points and the average of the results is taken as a result.

RSSI Proximity

In this method, the same equation in KNN Proximity (see. KNN Proximity) is utilized without appealing to KNN and fingerprinting dataset. The RSSI values read from each of the gateways were considered as the distances of the gateways to the beacon and according to that, the equation solved.

Conclusion

In this project, I implemented an indoor position application functionally. During the development of the project, I experienced lots of issues regarding the development environment, technology and materials. After all, it looks like obvious that it is hard to localize the Bluetooth capable devices indoor because of the fluctuation, signal reflection, and weak data.

Fingerprinting is one of the ways that have the potential to increase the accuracy of the positioning and supporting it with machine learning or deep learning algorithms can be even more beneficial.

References

- *.NET Core Guide | Microsoft Docs.* (2018). Retrieved from https://docs.microsoft.com/en-us/dotnet/core/
- *Universal Windows Platform – Wikipedia.* (2018). Retrieved from https://en.wikipedia.org/wiki/Universal_Windows_Platform
- *Overview of Windows 10 IoT Core – Windows IoT | Microsoft Docs.* (2018). Retrieved from https://docs.microsoft.com/en-us/windows/iot-core/windows-iot-core
- *Windows 10 IoT Core Dashboard – Windows IoT | Microsoft Docs.* (2018). Retrieved from https://docs.microsoft.com/en-us/windows/iot-core/connect-your-device/iotdashboard
- *Building, Running, and Testing .NET Core and ASP.NET Core 2.1 in Docker on a Raspberry Pi (ARM32) - Scott Hanselman.* (2018). Retrieved from

https://www.hanselman.com/blog/BuildingRunningAndTestingNETCoreAndASPNETCore21InDockerOnARaspberryPiARM32.aspx

- *Test Run - Understanding k-NN Classification Using C#*. (2018). Retrieved from https://msdn.microsoft.com/en-us/magazine/mt814421.aspx
- Tay, Bunheang & Hyun, Jung Keun & Oh, Sejong. (2014). A Machine Learning Approach for Specification of Spinal Cord Injuries Using Fractional Anisotropy Values Obtained from Diffusion Tensor Images. Computational and mathematical methods in medicine. 2014. 276589. 10.1155/2014/276589.
- Bozkurt, Sinem & Elibol, Gulin & Gunal, Serkan & Yayan, Ugur. (2015). A comparative study on machine learning algorithms for indoor positioning. 1-8. 10.1109/INISTA.2015.7276725.
- Xiao, Linchen & Behboodi, Arash & Mathar, Rudolf. (2017). A deep learning approach to fingerprinting indoor localization solutions. 1-7. 10.1109/ATNAC.2017.8215428.